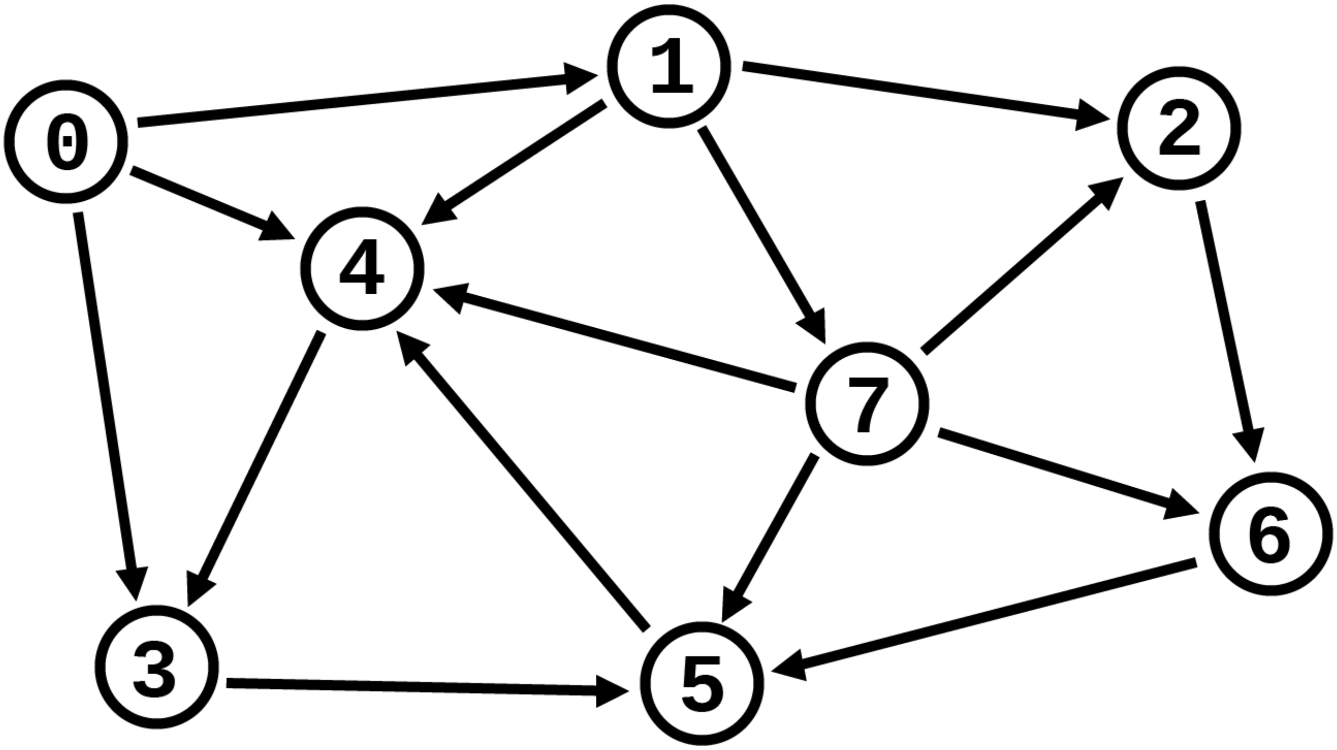
Tarea 7 – Análisis de algoritmos

Vargas Aldaco Alejandro

Ejercicio 1.




```
from collections import defaultdict


class Graph:

    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def isCyclicUtil(self, v, visited, recStack):
        vV = []
        vN = []
        # Marca el vertice actual como visitado
        visited[v] = True
        recStack[v] = True

        # Busca los vertices adyacentes no visitados
        for i in self.graph[v]:
            if not visited[i]:
                vN.insert(0, i)
            vV.insert(0, i)
```

```
        print("Vecinos de " + str(v) + " " + str(vV))
        print("No visitados " + str(vN))
        print("")

        # Recorre todos los vecinos del vertice actual
        # Si algún vecino ya fue visitado la gráfica es cíclica
        for neighbour in self.graph[v]:
            if not visited[neighbour]:
                print("Verificando ciclo en " + str(neighbour))
                if self.isCyclicUtil(neighbour, visited, recStack):
                    return True
            elif recStack[neighbour]:
                stack = []
                print("Se encontró un ciclo en " + str(v) + " -> " + str(neighbour))
                print()
                for j in range(self.V):
                    if recStack[j]:
                        stack.append(j)
                print("El stack se quedó con los elementos")
                print(stack)
                return True

        # Se limpia el stack antes de que termine la función
        recStack[v] = False
        return False

    # Regresa verdadero si la gráfica tiene un ciclo, falso en caso contrario
    def isCyclic(self):
        visited = [False] * self.V
        recStack = [False] * self.V

        print("Estructura de la gráfica ")
        print("")
        print(str(self.graph))

        for node in range(self.V):
            if not visited[node]:
                print("Inspeccionando en " + str(node))
                print("")
                if self.isCyclicUtil(node, visited, recStack):
                    return True
        return False


g = Graph(8)
g.addEdge(0, 1)
g.addEdge(0, 4)
g.addEdge(0, 3)
g.addEdge(1, 2)
g.addEdge(1, 7)
g.addEdge(1, 4)
g.addEdge(2, 6)
g.addEdge(3, 5)
g.addEdge(4, 3)
g.addEdge(5, 4)
g.addEdge(6, 5)
g.addEdge(7, 2)
g.addEdge(7, 4)
g.addEdge(7, 5)
g.addEdge(7, 6)

if g.isCyclic() == 1:
    print('La gráfica es cíclica')
else:
    print("La gráfica es acíclica")
```

Estructura de la gráfica

{0: [1, 4, 3], 1: [2, 7, 4], 2: [6], 3: [5], 4: [3], 5: [4], 6: [5], 7: [2, 4, 5, 6]})

Inspeccionando en 0

Vecinos de 0 [3, 4, 1]
No visitados [3, 4, 1]

Verificando ciclo en 1
Vecinos de 1 [4, 7, 2]
No visitados [4, 7, 2]

Verificando ciclo en 2
Vecinos de 2 [6]
No visitados [6]

Verificando ciclo en 6
Vecinos de 6 [5]
No visitados [5]

Verificando ciclo en 5
Vecinos de 5 [4]
No visitados [4]

Verificando ciclo en 4
Vecinos de 4 [3]
No visitados [3]

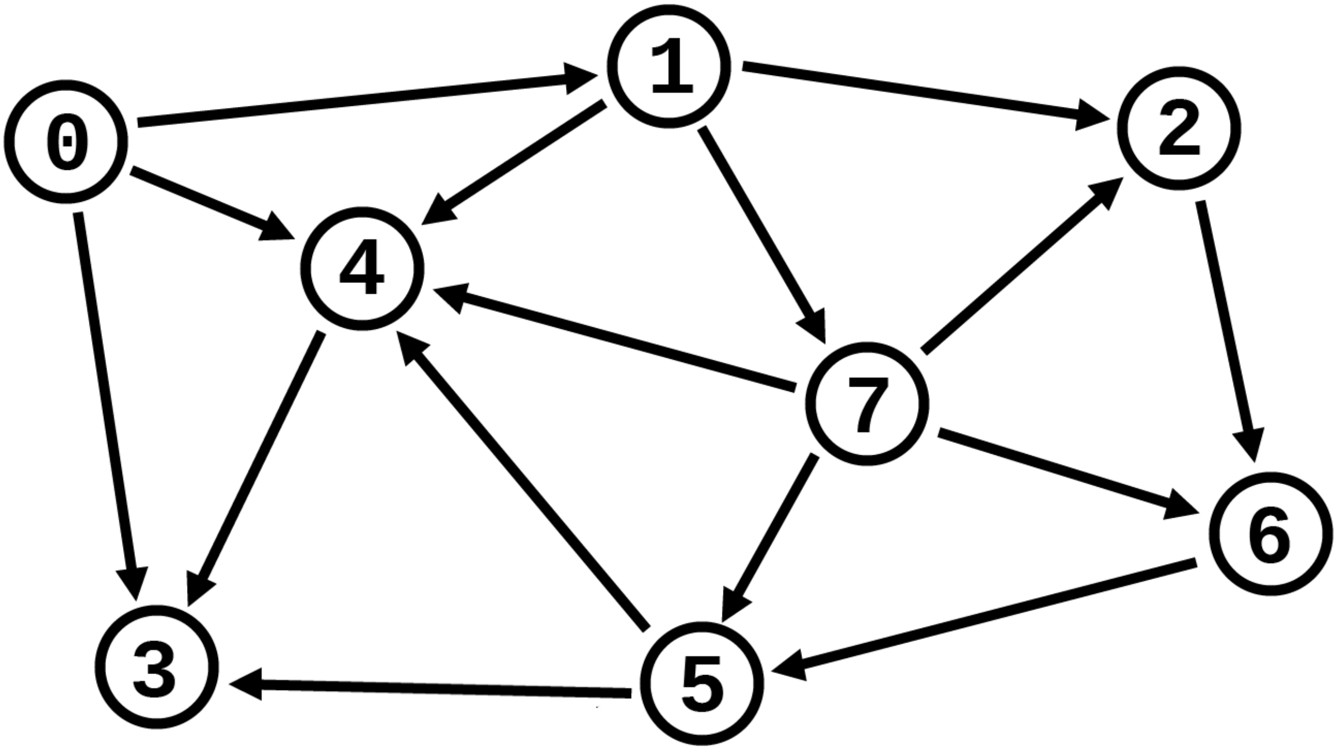
Verificando ciclo en 3
Vecinos de 3 [5]
No visitados []

Se encontró un ciclo en 3 -> 5

El stack se quedó con los elementos [0, 1, 2, 3, 4, 5, 6]

La gráfica es cíclica

Ejercicio 2.



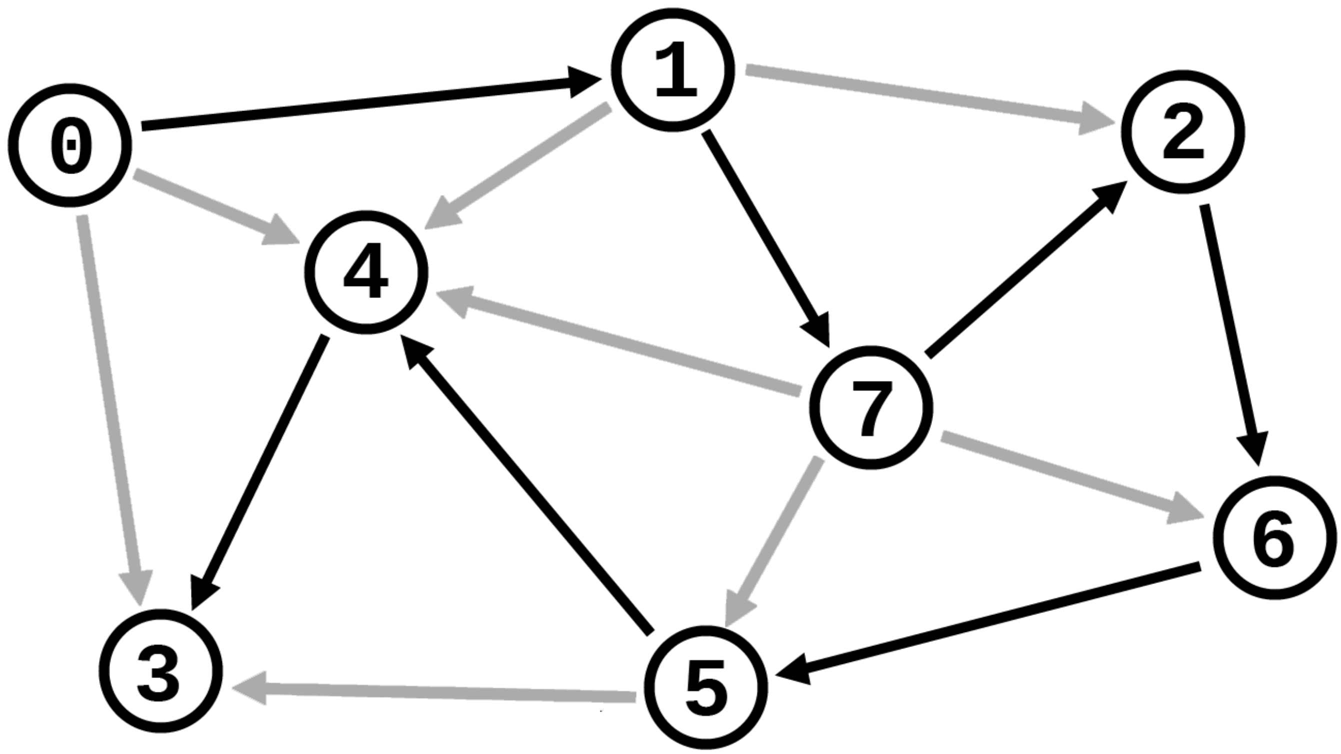

```
from collections import defaultdict


# Clase que representa la gráfica
class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)  # diccionario que guarda la adyacencia de los vértices
        self.V = vertices  # No. de vértices
        self.visitados = []
        self.index = 0

    # Función para agregar la adyacencia y dirección entre dos vértices de la gráfica
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # Función recursiva auxiliar para el ordenamiento topológico
    def topologicalSortUtil(self, v, visited, stack):

        self.index = self.index + 1
        vecinos = []

        # Marca el vertice actual como visitado
        visited[v] = True
        self.visitados.insert(0, v)
        print("Paso " + str(self.index))
        print("Visita elemento " + str(v))
        print("Elementos visitados " + str(self.visitados))

        # Busca los vertices adyacentes no visitados
        for i in self.graph[v]:
            if not visited[i]:
                vecinos.insert(0, i)

        print("Vecinos de " + str(v) + " no visitados " + str(vecinos))

        # Recursividad con los los vertices adyacentes
        for i in self.graph[v]:
            if not visited[i]:
                self.topologicalSortUtil(i, visited, stack)
```

```python
        # Inserta el vértice actual en el stack
        self.index = self.index + 1
        print("Paso " + str(self.index))
        stack.insert(0, v)
        print("Agrega al stack " + str(v))
        print("El stack " + str(stack))

    # La función que va a hacer la recursion es
    # topologicalSortUtil()
    def topologicalSort(self):
        # Marca todos los vértices como no visitados
        visited = [False] * self.V
        stack = []
        print("Estructura de la gráfica ")
        print(str(self.graph))
        print("El stack inicial :" + str(stack))
        print("Elementos visitados :" + str(self.visitados))
        # Busca los vertices que no dependen de nadie uno por uno
        # el siguiente después de haber hecho la recursión que generó el anterior
        for i in range(self.V):
            if not visited[i]:
                self.topologicalSortUtil(i, visited, stack)

    def printMatrix(self):
        rows = []
        trows = []
        for i in range(self.V):
            cols = []
            for j in range(self.V):
                cols.append(0)
            for j in self.graph[i]:
                cols[j] = 1
            rows.append(cols)
        for i in range(self.V):
            cols = []
            for j in range(self.V):
                element = rows[j]
                cols.append(element[i])
            trows.append(cols)
        for row in rows:
            print(row)
        print("transpuesta")
        for row in trows:
            print(row)


g = Graph(8)
g.addEdge(0, 1)
g.addEdge(0, 4)
g.addEdge(0, 3)
g.addEdge(1, 2)
g.addEdge(1, 7)
g.addEdge(1, 4)
g.addEdge(2, 6)
g.addEdge(4, 3)
g.addEdge(5, 3)
g.addEdge(5, 4)
g.addEdge(6, 5)
g.addEdge(7, 2)
g.addEdge(7, 4)
g.addEdge(7, 5)
g.addEdge(7, 6)

print("Ordenamiento topológico de la gráfica")

g.topologicalSort()

print("Matriz de la gráfica")

g.printMatrix()
```

Ordenamiento topológico de la gráfica

```
Estructura de la gráfica
       {0: [1, 4, 3], 1: [2, 7, 4], 2: [6], 4: [3], 5: [3, 4], 6: [5], 7: [2, 4, 5, 6]})
El stack inicial :[]
Elementos visitados :[]
Paso 1
       Visita elemento 0
       Elementos visitados [0]
       Vecinos de 0 no visitados [3, 4, 1]
Paso 2
       Visita elemento 1
       Elementos visitados [1, 0]
       Vecinos de 1 no visitados [4, 7, 2]
Paso 3
       Visita elemento 2
       Elementos visitados [2, 1, 0]
       Vecinos de 2 no visitados [6]
Paso 4
       Visita elemento 6
       Elementos visitados [6, 2, 1, 0]
       Vecinos de 6 no visitados [5]
Paso 5
       Visita elemento 5
       Elementos visitados [5, 6, 2, 1, 0]
       Vecinos de 5 no visitados [4, 3]
Paso 6
       Visita elemento 3
       Elementos visitados [3, 5, 6, 2, 1, 0]
       Vecinos de 3 no visitados []
Paso 7
       Agrega al stack 3 (Ya no hay más elementos por visitar en este camino)
       El stack [3]
Paso 8
       Visita elemento 4 (backtrack Paso 1)
       Elementos visitados [4, 3, 5, 6, 2, 1, 0]
       Vecinos de 4 no visitados []
Paso 9
       Agrega al stack 4
       El stack [4, 3]
Paso 10
       Agrega al stack 5 (backtrack Paso 5)
       El stack [5, 4, 3]
Paso 11
       Agrega al stack 6 (backtrack Paso 4)
       El stack [6, 5, 4, 3]
Paso 12
       Agrega al stack 2 (backtrack Paso 3)
       El stack [2, 6, 5, 4, 3]
Paso 13
       Visita elemento 7
       Elementos visitados [7, 4, 3, 5, 6, 2, 1, 0]
       Vecinos de 7 no visitados []
Paso 14
       Agrega al stack 7
       El stack [7, 2, 6, 5, 4, 3]
Paso 15
       Agrega al stack 1 (backtrack Paso 2)
       El stack [1, 7, 2, 6, 5, 4, 3]
Paso 16
       Agrega al stack 0 (backtrack Paso 1)
```

Finally El stack ordenado = [0, 1, 7, 2, 6, 5, 4, 3]



Ejercicio 3.

SeaV={v1...v|V|} el conjunto de vértices del grafo G, y E el conjunto de sus aristas. La matriz de adyacencia será una matriz de tamaño |V|×|V|, donde la entrada(i,j) será $\mathbf{a}_{i,j}$ = si existe una arista de $\mathbf{v}_i$ a $\mathbf{v}_j$ 1, en otro caso 0

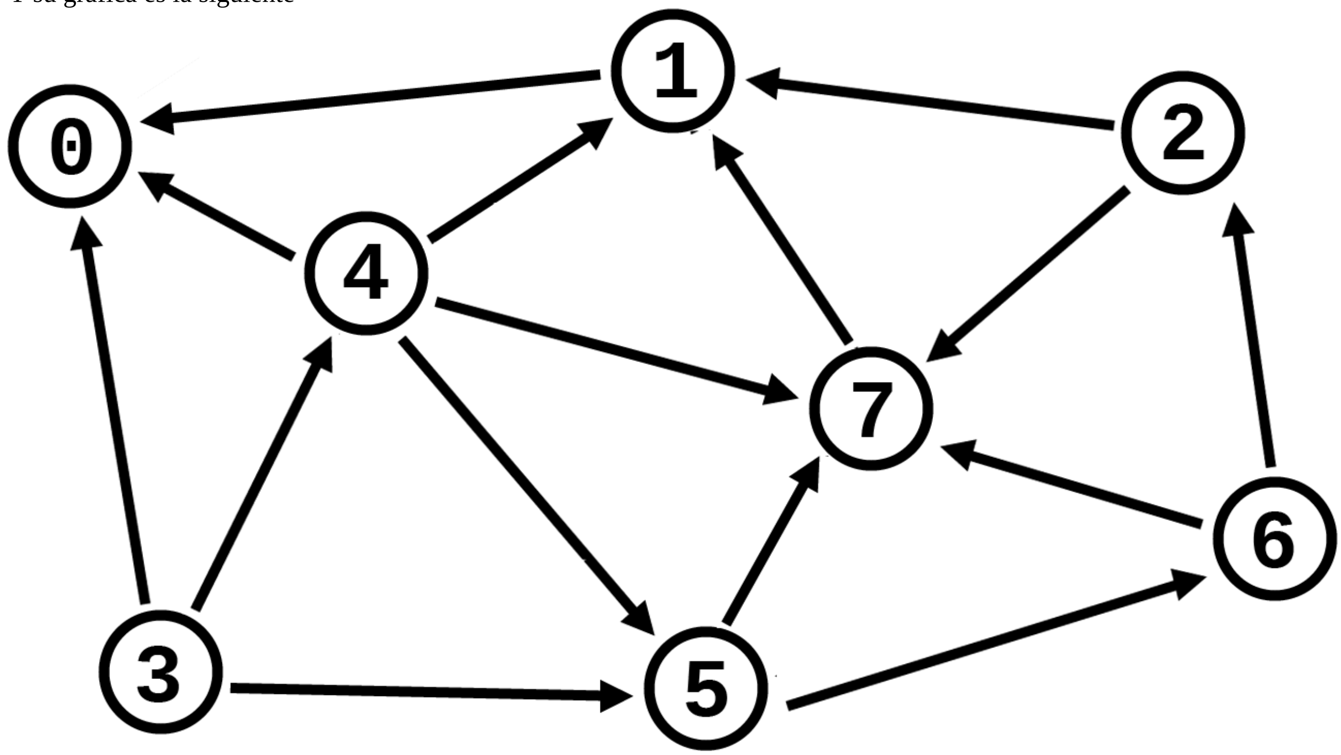
La matriz de adyacencia de la gráfica anterior esta dada de la siguiente manera

```
   0  1  2  3  4  5  6  7
0 [0, 1, 0, 1, 1, 0, 0, 0]
1 [0, 0, 1, 0, 1, 0, 0, 1]
2 [0, 0, 0, 0, 0, 0, 1, 0]
3 [0, 0, 0, 0, 0, 0, 0, 0]
4 [0, 0, 0, 1, 0, 0, 0, 0]
5 [0, 0, 0, 1, 1, 0, 0, 0]
6 [0, 0, 0, 0, 0, 1, 0, 0]
7 [0, 0, 1, 0, 1, 1, 1, 0]
```

Por lo tanto, la matriz transpuesta está dada de la siguiente manera

```
   0  1  2  3  4  5  6  7
0 [0, 0, 0, 0, 0, 0, 0, 0]
1 [1, 0, 0, 0, 0, 0, 0, 0]
2 [0, 1, 0, 0, 0, 0, 0, 1]
3 [1, 0, 0, 0, 1, 1, 0, 0]
4 [1, 1, 0, 0, 0, 1, 0, 1]
5 [0, 0, 0, 0, 0, 0, 1, 1]
6 [0, 0, 1, 0, 0, 0, 0, 1]
7 [0, 1, 0, 0, 0, 0, 0, 0]
```

Y su gráfica es la siguiente



Definición

Dado un grafo dirigido G = {N,A} un componente fuertemente conexo (CFC) es un conjunto U ⊆ N maximal tal que para todo u,v ∈ U existe u → v y v → u, donde a → b significa que existe un camino de a hacia b.

Sea D = {N,A} y U = N el componente maximal tal que para todo u,v ∈ U existe u → v y v → u. Es decir, tomamos toda la gráfica y verifico que se puede acceder a cualquier vértice desde cualquier otro. Por medio de DFS se determinará si D es fuertemente conexa.

```
from collections import defaultdict


# Clase que representa la gráfica
class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)  # diccionario que guarda la adyacencia de los vértices
        self.V = vertices  # No. de vértices
        self.index = 0

    # Función para agregar la adyacencia y dirección entre dos vértices de la gráfica
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # función recursiva usada por DFS
    def DFSUtil(self, v, visited):

        self.index = self.index + 1
        vecinos = []

        # Marca el vertice actual como visitado y lo imprime
        visited[v] = True
```

```
        print("Visitando " + str(v))

        # Busca los vertices adyacentes no visitados
        for k in self.graph[v]:
            if not visited[k]:
                vecinos.insert(0, k)

        print("Vecinos de " + str(v) + " no visitados " + str(vecinos))

        # Recursividad con los los vertices adyacentes
        for j in self.graph[v]:
            if not visited[j]:
                self.DFSUtil(j, visited)

    # La función que va a hacer la recursion es
    # recursive DFSUtil()
    def DFS(self, e):
        visited = [False] * self.V
        visitados = []
        j = -1
        print("Verificando vertices desde el elemento " + str(i))
        self.DFSUtil(e, visited)
        # imprime los elementos que fueron visitados
        for v in visited:
            j = j + 1
            a = j if v else - 1
            if a != -1:
                visitados.append(j)
        print(visitados)
        if j > visited.count(True) - 1:
            print("Desde el vértice " + str(e) + " no se pueden visitar todos los restantes")
            print("No cumple la definición")
        else:
            print("Desde este vértice " + str(e) + " si cumple la definición")


g = Graph(8)
g.addEdge(0, 1)
g.addEdge(0, 4)
g.addEdge(0, 3)
g.addEdge(1, 2)
g.addEdge(1, 7)
g.addEdge(1, 4)
g.addEdge(2, 6)
g.addEdge(4, 3)
g.addEdge(5, 3)
g.addEdge(5, 4)
g.addEdge(6, 5)
g.addEdge(7, 2)
g.addEdge(7, 4)
g.addEdge(7, 5)
g.addEdge(7, 6)

# Aplica la función comenzando por cada uno de los vértices
for i in range(8):
    g.DFS(i)
```

Verificando vértices desde el elemento 0

```
Visitando 0
Vecinos de 0 no visitados [3, 4, 1]
Visitando 1
Vecinos de 1 no visitados [4, 7, 2]
Visitando 2
Vecinos de 2 no visitados [6]
Visitando 6
Vecinos de 6 no visitados [5]
Visitando 5
Vecinos de 5 no visitados [4, 3]
Visitando 3
Vecinos de 3 no visitados []
Visitando 4
Vecinos de 4 no visitados []
```

```
Visitando 7
Vecinos de 7 no visitados []
```

Vértices visitados
```
[0, 1, 2, 3, 4, 5, 6, 7]
```

Desde el vértice 0 si cumple la definición

Verificando vértices desde el elemento 1

```
Visitando 1
Vecinos de 1 no visitados [4, 7, 2]
Visitando 2
Vecinos de 2 no visitados [6]
Visitando 6
Vecinos de 6 no visitados [5]
Visitando 5
Vecinos de 5 no visitados [4, 3]
Visitando 3
Vecinos de 3 no visitados []
Visitando 4
Vecinos de 4 no visitados []
Visitando 7
Vecinos de 7 no visitados []
```

Vértices visitados
```
[1, 2, 3, 4, 5, 6, 7]
```

Desde el vértice 1 no se pueden visitar todos los restantes
No cumple la definición

Verificando vértices desde el elemento 2

```
Visitando 2
Vecinos de 2 no visitados [6]
Visitando 6
Vecinos de 6 no visitados [5]
Visitando 5
Vecinos de 5 no visitados [4, 3]
Visitando 3
Vecinos de 3 no visitados []
Visitando 4
Vecinos de 4 no visitados []
```

Vértices visitados
```
[2, 3, 4, 5, 6]
```

Desde el vértice 2 no se pueden visitar todos los restantes
No cumple la definición

Verificando vértices desde el elemento 3

```
Visitando 3
Vecinos de 3 no visitados []
```

Vértices visitados
```
[3]
```

Desde el vértice 3 no se pueden visitar todos los restantes
No cumple la definición

Verificando vértices desde el elemento 4

```
Visitando 4
Vecinos de 4 no visitados [3]
Visitando 3
Vecinos de 3 no visitados []
```

Vértices visitados

```
[3, 4]

Desde el vértice 4 no se pueden visitar todos los restantes
No cumple la definición
```

Verificando vértices desde el elemento 5

```
Visitando 5
Vecinos de 5 no visitados [4, 3]
Visitando 3
Vecinos de 3 no visitados []
Visitando 4
Vecinos de 4 no visitados []

Vértices visitados
[3, 4, 5]

Desde el vértice 5 no se pueden visitar todos los restantes
No cumple la definición
```

Verificando vértices desde el elemento 6

```
Visitando 6
Vecinos de 6 no visitados [5]
Visitando 5
Vecinos de 5 no visitados [4, 3]
Visitando 3
Vecinos de 3 no visitados []
Visitando 4
Vecinos de 4 no visitados []

Vértices visitados
[3, 4, 5, 6]

Desde el vértice 6 no se pueden visitar todos los restantes
No cumple la definición
```

Verificando vértices desde el elemento 7

```
Visitando 7
Vecinos de 7 no visitados [6, 5, 4, 2]
Visitando 2
Vecinos de 2 no visitados [6]
Visitando 6
Vecinos de 6 no visitados [5]
Visitando 5
Vecinos de 5 no visitados [4, 3]
Visitando 3
Vecinos de 3 no visitados []
Visitando 4
Vecinos de 4 no visitados []

Vértices visitados
[2, 3, 4, 5, 6, 7]

Desde el vértice 7 no se pueden visitar todos los restantes
No cumple la definición
```

Se puede apreciar en la prueba que D no es fuertemente conexa debido a que el único vértice desde el que se puede llegar a todos los demás es el vértice `0`.