TAREA 3: Analisis y Diseño de Algoritmos

Andrey Arguedas Espinoza - 2020426569

October 25, 2022

1. Un ciclo de Euler para un grafo no dirigido es un camino que empieza y termina en el mismo vertice y que usa cada **arista** exactamente una vez. Un grafo G, conexo y no dirigido, tiene un ciclo de Euler si y solo si cada vértice tiene grado par (número de aristas). Ajustar el algoritmo de búsqueda en profundidad visto en clase para dar un algoritmo O(a) que encuentre un ciclo de Euler para un grafo con a aristas, siempre y cuando el ciclo exista.

   Sugerencias: encontrar un primer ciclo; marcar aristas y no vértices (los vértices se pueden repetir y si el grafo es conexo al empezar en cualquiera encuentro todos los demás).

Nuestro algoritmo consiste en los siguientes pasos (Ver pseudocodigo en la siguiente página):

1 Inicializar un array vacio para guardar los aristas visitadas.

2 Iniciar en el primer nodo de la lista de nodos.

3 En cada nodo si se cumple que sea de grado par realizamos los siguientes pasos, si no avanza al paso 4:

3.1 Para cada arista del nodo V la marcamos como visitada y entra al arreglo de aristas recorridas, si ya había sido visitada nos salimos del ciclo.

3.2 Avanzamos en dfs hacia el nodo destino de la arista recien recorrida.

4. Si el arreglo de aristas validas es igual al arreglo original de aristas y todos los nodos tenian grado para efectivamente existe el ciclo de euler.

```
#PseudoCode

function degree(node)
   return size of edges of node

# Suppose edge 1->2 is a pair [1,2]
function destiny(node, edge)
   if edge[0] == node:
        return edge[1]
    else
        return edge[0]

function dfsCustom(node, vistedEdges, validCycle)
    if degree(node) mod 2 == 0:
        for each edge of node and edge is not in vistedEdges:
            vistedEdges.add(edge)
            destiny = destiny(node, edge)
            dfsCustom(node, vistedEdges, validCycle)
    else
        validCycle = false

#G: <N,A>

function eulerCycle(G):
    vistedEdges = []
    validCycle = true
    startNode = N[0] #Starts in the first node of the list
    validCycle = dfsCustom(startNode, vistedEdges, validCycle)

    # Euler Cycle found if all nodes have even degree and vistedEdges equals t
    return validCycle == true and vistedEdges == A
```

2. Un nodo p de un grafo dirigido G=<N,A> se denomina sumidero si para todo nodo v∈N, v≠p, existe una arista (v,p), mientras que no existe la arista (p,v). Escribir un algoritmo que pueda detectar la presencia de un sumidero en G en un tiempo que esté en O(n), en donde n es el número de nodos que haya en el grafo. El algoritmo debe aceptar el grafo representado por su **matriz de adyacencia**.

Sugerencias:

- Si tiene un candidato a ser sumidero, ¿cuánto toma verificar si cumple o no?
- Considere el significado de las siguientes situaciones

  Si M[i,j] = 1, ¿qué dice eso sobre la posibilidad de que i sea sumidero o de que j sea sumidero?

  Si M[i,j] = 0, ¿qué dice eso sobre la posibilidad de que i sea sumidero o de que j sea sumidero?

Para este problema primero la idea que seguiremos es primero encontrar un candidato a ser sumidero, y luego verificar si realmente lo es, para esto realizamos las siguientes funciones:

```
function candidato(M)
    candidate = -1 #Not available candidate at the beggining
    i = 0
    j = 0
    matrix_size = length(M[0])
    while i < matrix_size and j < matrix_size:
        if M[i][j] == 1:
            i = i + 1
            candidate = i
        else:
            j = j + 1

    return candidate

function sumidero(M)
    matrix_size = length(M[0])
    candidate = candidato(M)
    if candidate != -1 then:
        for x from 0 to matrix_size:
            # Only 1s in column accepted except the position [candidate][cand
            if M[x][candidate] == 0 and x != candidate then:
                return False
            #If candidate has a 1 in the row is already discarted
            if M[candidate][x] == 1:
                return False
        # There is only 1s in the column of the candidate and only 0s in the r
        return True
    else:
        return false
```

Si tiene un candidato a ser sumidero, ¿cuánto toma verificar si cumple o no?

Verificarlo toma un tiempo de:

$$O(n) \tag{1}$$

Si M[i,j] = 1, ¿qué dice eso sobre la posibilidad de que i sea sumidero o de que j sea sumidero?

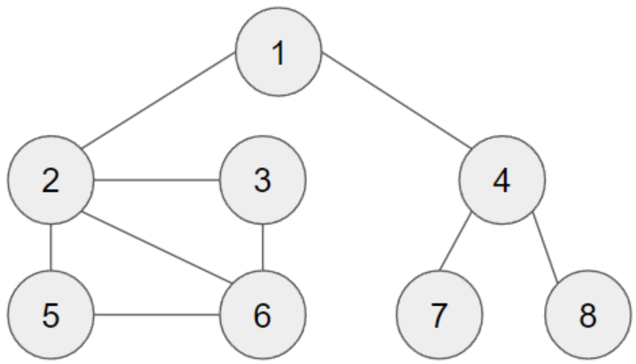
En este caso se descarta que i sea sumidero de j ya que el sumidero no tiene salidas, solo entradas y de j no nos asegura si es sumidero o no.

Si M[i,j] = 0, ¿qué dice eso sobre la posibilidad de que i sea sumidero o de que j sea sumidero?

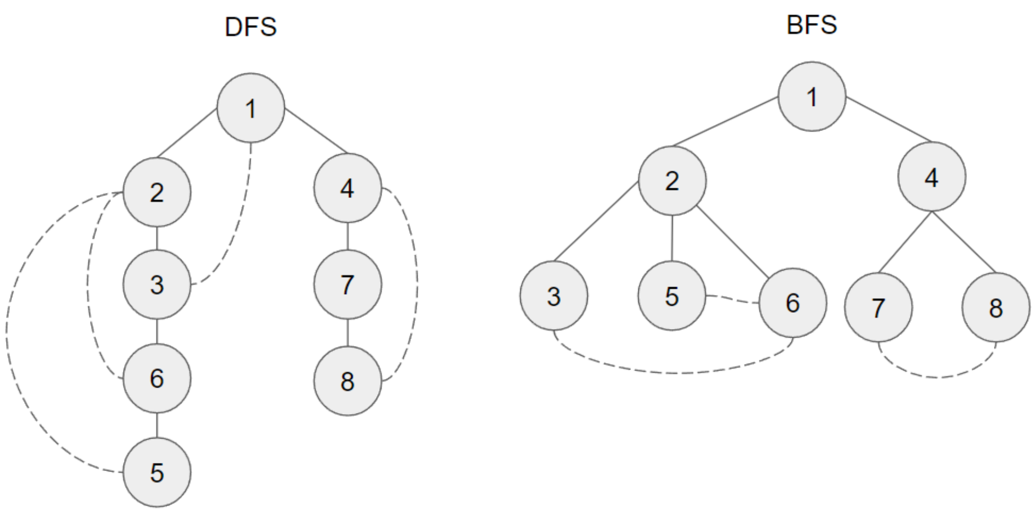
En este caso no se descarta la posibilidad de que i sea el sumidero, en cuanto a j nos da a saber que no tiene salidas hacia i.

3. Después de un recorrido por anchura de un grafo no dirigido, sea F el bosque de árboles generado. Si (u,v) es una arista de G que no posee una arista correspondiente en F (las representadas por líneas discontinuas), mostrar que los nodos u y v yacen en el mismo árbol de F, pero ni u ni v son ancestros uno del otro.

Para poder explicar por que no se pueden dar estos casos en un Spanning Tree generado por recorrido BFS realizamos lo siguiente:



A continuación veremos una comparativa del Spanning Tree del grafo anterior con reccorido DFS contra BFS



Por la forma de recorrido de BFS las lineas discontinuas no se van a formar a los niveles hacia abajo como si lo hace DFS, por lo que con BFS no podemos utilizar la misma tecnica para demostrar que un nodo u es ancestro de v.

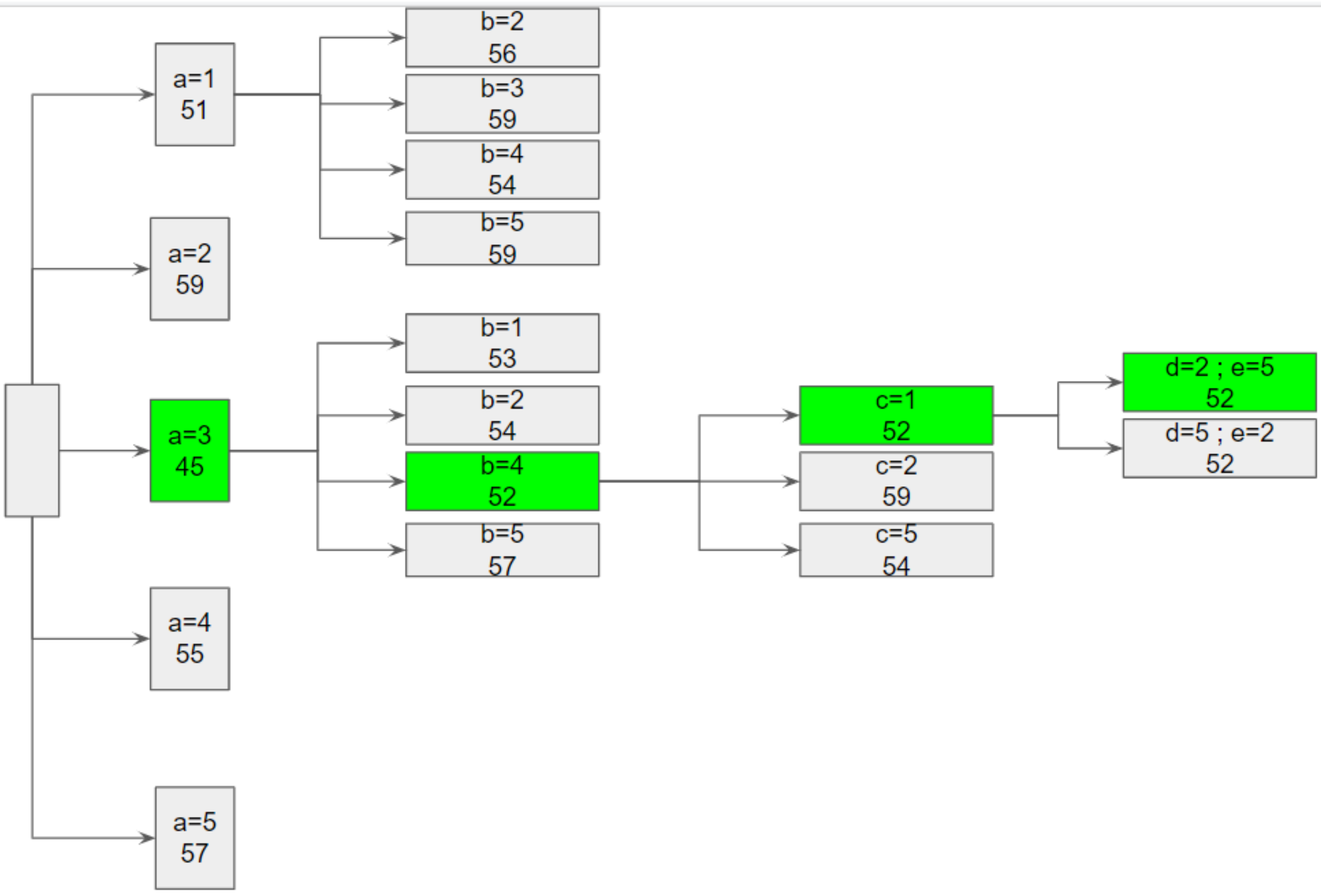
- Mostrar que tanto u como v pertenecen a F pero ni u ni v son ancestros uno del otro

Para demostrar esto imaginamos que **u** y **v** están en árboles separados, esto quiere decir que tanto su padre inmediato como la raíz del arbol son diferentes entre sí, y esto no es posible ya que como podemos ver el recorrido BFS uniría por lineas discontinuas solo a nodos que comparten el mismo padre. Por la misma razón no pueden ser ancestros uno del otro.

4. Utilizar un algoritmo de ramificación (branch-and-bound) y poda para resolver el problema de asignación que tiene asociada la siguiente matriz:

	1	2	3	4	5
a	11	17	8	16	20
b	9	7	12	6	15
c	13	16	15	12	16
d	21	10	12	11	15
e	14	10	12	11	15

Primero calculamos el valor de las diagonales:
Diagonal izquierda-derecha : 11+7+15+11+15 = 59
Diagonal derecha-izquierda : 14+10+15+6+15 = 60
- Escogemos $\boxed{59}$ como la cota superior.
Segundo calculamos los menores de cada fila y de cada columna:
Menores por fila : 8+6+12+10+10 = 46
Menores por columna : 9+7+8+6+15 = 45
- Escogemos $\boxed{45}$ como la cota inferior.
Cota inicial: [45...59]



Respuesta final: a=3, b=4, c=1, d=2, e=5 Valor=52