Análisis y Diseño de Algoritmos

Tarea 5

Daniel Rojo Mata
danielrojomata@gmail.com

Fecha de Entrega: 06 de octubre de 2024

Observación:

En esta tarea colaboré con:
Elizabeth Ríos Alvarado

Problema 1

Escribe un resumen de no más de una página del siguiente video: https://www.youtube.com/watch?v=nmgFG7PUHfo. En tu resumen explica la importancia del algoritmo de la transformada rápida de Fourier, porqué es un algoritmo divide y vencerás, y cómo es que logra bajar la complejidad de $O(n^2)$ del algoritmo simple a $O(n \log n)$ en el algoritmo divide y vencerás.

Resumen

La Transformada Rápida de Fourier (FFT) se emplea ampliamente en tecnologías actuales como radares, sonares, redes 5G, Wi-Fi y en el procesamiento de imágenes. Su capacidad para descomponer y analizar señales la convierte en una herramienta esencial para extraer información relevante de forma rápida y eficaz.
Su historia se remonta a los inicios de la Guerra Fría, cuando se requería detectar explosiones nucleares subterráneas para monitorear el cumplimiento de los tratados de prohibición de pruebas nucleares. En ese contexto, la FFT se convirtió en una herramienta esencial para distinguir entre eventos naturales, como terremotos, y detonaciones de armamento nuclear.

Tras la Segunda Guerra Mundial, se propuso el Plan Baruch para controlar materiales nucleares y evitar su reproducción, pero la desconfianza entre Estados Unidos y la URSS desencadenó una carrera nuclear. Al prohibirse las pruebas de bombas nucleares en la atmósfera y bajo el agua, se intensificaron las pruebas subterráneas, que eran difíciles de detectar.
Para monitorear estas pruebas, se utilizaron sismómetros que captaban vibraciones en la corteza terrestre, pero distinguir entre una explosión nuclear y un terremoto era extremadamente difícil. Aunque toda la información necesaria estaba contenida en las señales sísmicas, el problema era cómo analizarla eficientemente.

Para analizar estas señales, era esencial conocer las frecuencias presentes y en qué proporciones aparecían. Esto se logra con la Transformada de Fourier, que descompone una señal en sus componentes de frecuencia. Sin embargo, el método tradicional requería realizar n^2 cálculos siendo n el número de datos. Para señales complejas, esto representaba un costo computacional muy elevado, que necesitaba una solución más eficiente para la época.
Es así como aparece la Transformada Rápida de Fourier (FFT).

El algoritmo de la Transformada Rápida de Fourier (FFT) reduce la complejidad del cálculo a $O(n \log n)$ utilizando la técnica de divide y vencerás.
La idea principal es descomponer la transformada en problemas más pequeños y fáciles de resolver, combinando las soluciones parciales de manera eficiente. En lugar de calcular la transformada para todos los puntos a la vez, el algoritmo divide la señal en segmentos, aplica la transformada a cada uno y luego combina los resultados aprovechando simetrías en las operaciones y minimizando el número total de cálculos.

La FFT es de suma importancia en áreas que requieren el análisis de señales complejas en tiempo real, como el radar, la comunicación por Wi-Fi, la transmisión de datos 5G y el procesamiento de imágenes. Se ha convertido en el algoritmo numérico más importante para nuestra civilización (según el matemático Gilbert Strang), posibilitando avances en la detección temprana de sismos y en la mejora de la calidad de las telecomunicaciones.
Aunque el matemático Carl Friedrich Gauss ya conocía los principios detrás del FFT en el siglo XIX, su descubrimiento no fue publicado hasta 1965, cuando se popularizó como una herramienta computacional esencial hoy en día.

Problema 2

Considera un arreglo de enteros $A[1, \ldots, n]$ tal que en sus entradas aparece cada entero $1 \leq i \leq n$ una y sólo una vez (básicamente A codifica una permutación de los enteros $1, \ldots, n$). Una *inversión* de A es un par $1 \leq i < j \leq n$ tal que $A[i] > A[j]$. Lo que buscamos en este ejercicio es calcular el número de inversiones en A. Por ejemplo, si $A = [1, 4, 2, 3]$, el número de inversiones es 2, pues $A[2] > A[3]$ y $A[2] > A[4]$, y para todos los demás casos se cumple que si $i < j$, entonces $A[i] < A[j]$.

```
1.  Algo CountInversion(A, p, r)
2.     if p == r then
3.        return 0
4.     else
5.        q = ceil((p + r) / 2)
6.        left = CountInversion(A, p, q)
7.        right = CountInversion(A, q + 1, r)
8.        mix = MergeAndCount(A, p, q, r)
9.        return left + right + mix
10.    endif
11. End CountInversion
```

Usando dos ciclos for anidados en el que se comparan cada par $A[i]$ y $A[j]$, con $i < j$, es posible calcular el número de inversiones en tiempo $O(n^2)$. Sin embargo, esta complejidad se puede reducir a $O(n \log n)$ usando un estrategia divide y vencerás. La estrategia se puede ver en el algoritmo que aparece arriba; la primera llamada al algoritmo es $CountInversion(A, 1, n)$. La clave está en la función $MergeAndCount(A, p, q, r)$ que presupone que los segmentos $A[p, \ldots, q]$ y $A[q+1, \ldots, r]$ están ordenados ascendentemente, y hace dos cosas: devuelve el número de inversiones i, j tales que $p \leq i \leq q$ y $q + 1 \leq j \leq r$, y deja el segmento $A[p, \ldots, r]$ ordenado.

Haz lo siguiente:

1. **Demuestra que el algoritmo** $CountInversion$ **es correcto, suponiendo que** $MergeAndCount$ **hace lo que se dijo antes.**

   Demostración:

   Se demuestra por inducción sobre el tamaño del arreglo; n.

   - **Caso Base** ($n = 1$).
     En esta situación se tiene un arreglo con solo un elemento, este es de la forma: $A = [a_0]$, por lo que se ejecuta la línea 2 del algoritmo, así, se regresa 0, confirmando que no hay inversiones hasta el momento. Por lo que se cumple el caso base.
   - **Hipótesis de inducción:**
     Suponga que el algoritmo `CountInversion(A,p,r)` es correcto para cualquier arreglo de longitud menor o igual a n.
   - **Paso inductivo**
     Tómese un arreglo de longitud $n + 1$, llámese A'.
     Al ejecutar el algoritmo `CountInversion(A',p,r)` por las líneas 6 y 7 se hacen dos llamadas recursivas con subinstancias (arreglos) de longitud menor o igual a n, esto porque con q se

busca partir el arreglo A' en mitades. Dichas llamadas son `left` y `right`.

Así, se aplica el algoritmo a dos subinstancias cuya longitud es menor o igual a n, entonces, por hipótesis de inducción, el algoritmo es correcto para `left` y `right` y retorna dos subarreglos $A'[p, ..., q]$ y $A'[q+1, ..., r]$ ordenados de manera ascendente.

Finalmente, se hace uso de la función `MergeAndCount` para juntar las soluciones `left` y `right`, y por hipótesis esta función devuelve el número de inversiones y deja el segmento $A'[p, ..., r]$ ordenado.

Por lo que el algoritmo `CountInversion` es correcto.

2. **Demuestra que si la complejidad de tiempo de $MergeAndCount$ es $O(r-q)$, entonces la complejidad de tiempo de $CountInversion$ es $O(n \log n)$.**

   Demostración

   Se demuestra por inducción sobre n, la longitud del arreglo de entrada.

   - **Caso Base:**
     Cuando $n = 1$, el arreglo tiene un solo elemento, por lo que no hay inversiones y la función retorna 0 en tiempo constante, que es $O(1)$.
   - **Hipótesis Inductiva:**
     Supongamos que para cualquier subarreglo de tamaño menor o igual a n, la función `CountInversion` tiene una complejidad de tiempo $O(n \log n)$, esto es, $T(n) \in O(n \log n)$, con $T(n)$ la función de complejidad temporal.
   - **Caso Inductivo:**
     Consideremos un arreglo de tamaño $n+1$. Al llamar a `CountInversion` en este arreglo, se realizan dos llamadas recursivas en subarreglos de tamaño aproximadamente $n/2$.
     Además, se realiza una llamada a `MergeAndCount` en un subarreglo de tamaño menor a n. Según la hipótesis, esta llamada toma $O(r-q)$ tiempo.
     Es así que el tiempo total $T(n+1)$ se puede expresar como:

     $$T(n+1) = 2T\Big((n+1)/2\Big) + O(r-q)$$

     Ahora, como $r - q \leq n$ es cierto que $O(r-q) \subseteq O(n)$.

     Entonces, por hipótesis; $T\Big((n+1)/2\Big) \in O(n \ \log n)$ y además $O(r-q) \subseteq O(n) \subseteq O(n \ \log n)$.

     Por lo que $T(n+1) \in O(n \ \log n)$, que era lo que se quería probar.

3. Da un algoritmo para $MergeAndCount$ con complejidad de tiempo $O(r-q)$. Demuestra la corrección de tu algoritmo y la complejidad de tiempo del mismo.

Algoritmo MergeAndCount

```
1  function MergeAndCount(A, p, q, r):
2      nLeft = q-p+1
3      nRight = r-q
4      Left = A[p:q]            # Subarreglo desde A[p] hasta A[q]
5      Right = A[q+1:r]         # Subarreglo desde A[q+1] hasta A[r]
6
7      i=0
8      j=0
9      k=p
10     inversiones=0
11     while(i<nLeft and j<nRight)
12         if(Left[i] > Right[j])
13             A[k] = R[j]
14             inversiones = inversiones + nLeft - i
15             j = j + 1
16         else
17             A[k] = L[i]
18             i = i + 1
19         end if-else
20
21         k=k+1
22     end while
23
24     while(i<nLeft)
25         A[k]=Left[i]
26         i=i+1
27         k=k+1
28     end while
29
30     while(j<nRight)
31         A[k]=Right[j]
32         j=j+1
33         k=k+1
34     end while
35
36     return inversiones
```

Corrección del algoritmo

Se hace uso de dos invariantes de ciclo, los cuales son los siguientes:

Al inicio de cada iteración del bucle `while (i <nLeft and j <nRight)` se cumple:

1. **Invariante 1:**

   El subarreglo $A[p, \ldots, k-1]$ (es decir, desde p hasta $k-1$) contiene todos los elementos más pequeños que los elementos restantes en `Left` (`Left[i]`) y `Right` (`Right[j]`), y está ordenado en orden ascendente.

2. **Invariante 2:**

   La variable `inversiones` contiene el número exacto de pares (i', j') tales que $p \leq i' \leq q$ y $q+1 \leq j' \leq r$ con $A[i'] > A[j']$, considerando solo los elementos que ya se han añadido a $A[p, \ldots, k-1]$ y los que quedan por procesar en `Left` y `Right`.

- **Inicialización**

  En este caso se tiene: $i = 0, j = 0, k = p$

  - $A[p, \ldots, k-1]$ es el subarreglo vacío (no contiene elementos), y por lo tanto está ordenado.
  - `inversiones` $= 0$ porque aún no se ha procesado ningún elemento.

  Así, las dos partes del invariante se cumplen al inicio.

- **Mantenimiento**

  Suponga que el invariante es verdadero al inicio de alguna iteración del bucle. Se tienen los siguientes casos:

  - **Caso 1:** `Left[i] >Right[j]`.
    - Se copia `Right[j]` en $A[k]$.
    - Esto significa que todos los elementos restantes en `Left` forman inversiones con `Right[j]`.
    - Incrementamos `inversiones` en $nLeft - i$ (el número de elementos restantes en `Left`).
    - Se incrementan j y k.

    Es así que $A[p, \ldots, k]$ está ordenado e `inversiones` se ha actualizado correctamente con los nuevos pares invertidos.

  - **Caso 2:** `Left[i] <= Right[j]`.
    - Se copia `Left[i]` en $A[k]$.
    - Esto preserva el orden porque `Left[i]` es menor o igual que cualquier elemento restante en `Right`.
    - Se incrementan i y k, y no se modifica `inversiones` (no hay nuevas inversiones al tomar un elemento de `Left`).

    Es así que $A[p, \ldots, k]$ está ordenado e `inversiones` sigue contando correctamente los pares (i, j).

En ambos casos, el invariante se sigue manteniendo.

- **Finalización**

  El bucle `while` termina cuando uno de los subarreglos (`Left` o `Right`) se agota, es decir, cuando $i \geq nLeft$ o $j \geq nRight$. En ese momento:

  - Todo el subarreglo $A[p, \ldots, r]$ está completamente fusionado y ordenado.
  - `inversiones` contiene el número exacto de inversiones entre `Left` y `Right` según la definición dada.

  Luego, los bucles `while` adicionales (`while i <nLeft` y `while j <nRight`) solo se encargan de copiar los elementos restantes de `Left` o `Right`, lo cual no afecta la corrección del conteo ni el orden del subarreglo resultante.

  Por lo que el algoritmo es completo.

Complejidad Temporal

Se sabe que:

- p es el índice inicial del subarreglo izquierdo.
- q es el punto de partición entre los dos subarreglos.
- r es el índice final del subarreglo derecho.

Por lo tanto, se tienen dos subarreglos:

- Subarreglo izquierdo: $A[p, \ldots, q]$
  - Tamaño del subarreglo izquierdo ($nLeft$): $nLeft = q - p$.
- Subarreglo derecho: $A[q + 1, \ldots, r]$
  - Tamaño del subarreglo derecho ($nRight$): $nRight = r - q$.

El número total de elementos en el rango $A[p, \ldots, r]$ es: $n = r - p$.

Sin embargo, el rango que el algoritmo maneja en cada llamada es de tamaño:

$$n = r - q + (q - p) = (r - p)$$

El pseudocódigo consta de tres secciones principales:

1. **Subarreglos:**
   `Left` $= A[p : q + 1]$
   `Right` $= A[q + 1 : r + 1]$
   Dado que `Left` y `Right` suman un total de $nLeft + nRight = (q - p) + (r - q) = r - p$ elementos, la creación de estos subarreglos tiene complejidad:

$$O(r - p)$$

2. **Bucle**
   El bucle compara los elementos de `Left` y `Right`, insertándolos en $A[k]$ y moviendo los índices i, j, y k a la siguiente posición.
   Cada comparación y movimiento de índices cuesta $O(1)$ y se ejecuta un número máximo de veces igual a la suma de los elementos de `Left` y `Right` (es decir, $nLeft + nRight = r - p$).
   Por lo tanto, el bucle while tiene una complejidad de:

$$O(r - p)$$

3. **Elementos restantes:**
   Estos dos bucles adicionales solo se ejecutan si quedan elementos sin procesar en `Left` o `Right` después de la fusión principal.
   El número total de iteraciones de estos dos bucles es menor a $O(r - p)$.

Sumando las complejidades de cada sección:

- Creación de subarreglos: $O(r - p)$.
- Fusión y conteo de inversiones: $O(r - p)$.
- Copiado de elementos restantes: $O(r - p)$.

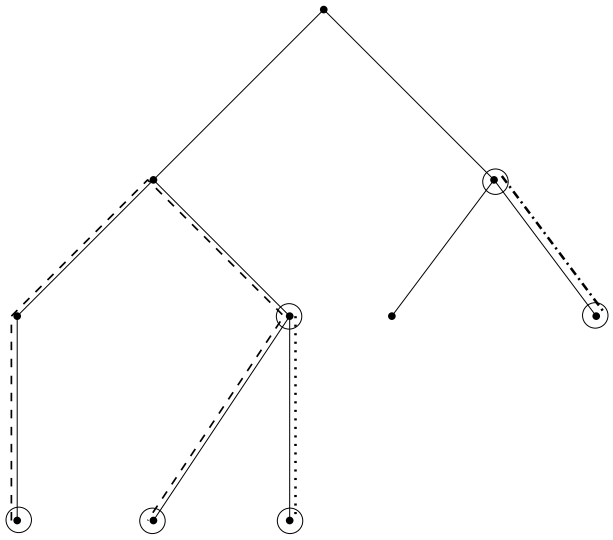
El tamaño del rango específico que el algoritmo fusiona es siempre $r - p$ por lo que, la complejidad total es:

$$O(r - p)$$

Problema 3

Sea T un árbol binario con raíz r. Todos los vértices de T tienen un atributo binario $flag$, y solo para un número par de ellos, a los que llamaremos *estaciones*, su atributo es $true$. Este ejercicio busca *emparejar* las estaciones de forma tal que en T existen caminos disjuntos en aristas (es decir, los caminos no pueden compartir aristas, pero sí vértices) que conectan las parejas. Por ejemplo, la figura siguiente muestra un ejemplo en el que hay seis estaciones (encerrados en círculos) y tres caminos (denotados con tres tipos distintos de líneas punteadas) que conectan las seis estaciones; la raíz es el vértice de más arriba.

Muestra un algoritmo divide y vencerás de tiempo $O(n)$ que imprima las parejas de las estaciones, donde n denota el número de vértices de T. Para imprimir puedes suponer la existencia de una instrucción $Print(s)$ que imprime la cadena s, y toma tiempo $O(1)$ en ejecutarse. La raíz r puede ser parte de la entrada a tu algoritmo, y considera que T está representado como una lista de adyacencia en la que para cada vértice v, $left(v)$ y $right(v)$ denotan sus hijos izquierdo y derecho, respectivamente, y si v es hoja, entonces tanto $left(v)$ como $right(v)$ son NIL.
Tip: En el análisis de complejidad haz un análisis similar al de DFS visto en clase.



Solución:

La ideal del algoritmo propuesto es la siguiente.

- **DFS:** Se realiza un recorrido en profundidad (DFS) en el árbol binario, explorando primero los subárboles izquierdo y derecho y recolectando las estaciones no emparejadas en cada uno.
- **Combinar listas:** Se combinan las listas de estaciones no emparejadas obtenidas de los subárboles izquierdo y derecho en una sola lista, que representa todas las estaciones no emparejadas en el subárbol con raíz en el nodo actual.
- **Emparejar:** Se emparejan las estaciones de la lista combinada de dos en dos, asegurando que los caminos utilizados para conectar estas estaciones sean disjuntos en aristas. Si queda una estación sin emparejar, se devuelve para ser considerada en futuros emparejamientos.

ALGORITMO

```
1: function EmparejarEstaciones(T, r)
       // T es la representación del árbol binario como lista de adyacencia
       // r es la raíz del árbol
2:     function DFS-Emparejar(v)
3:         if v = NIL then
4:             return lista vacía                    ▷ No hay estaciones en un nodo vacío
5:         end if
       // Realizar recorrido DFS en los subárboles izquierdo y derecho
6:         left_unpaired = DFS-Emparejar(left(v))
7:         right_unpaired = DFS-Emparejar(right(v))
       // Combinar las listas de estaciones sin emparejar de ambos subárboles
8:         combined_unpaired = left_unpaired ∪ right_unpaired
       // Si el nodo actual es una estación (flag es True)
9:         if flag(v) = True then
10:            Agregar v a combined_unpaired
11:        end if
       // Emparejar las estaciones de dos en dos
12:        i = 0
13:        while i < len(combined_unpaired) − 1 do
14:            u = combined_unpaired[i]
15:            w = combined_unpaired[i + 1]
16:            Emparejar (u, w)
17:            i = i + 2
18:        end while
       // Si el número de estaciones no emparejadas es impar, devolver la última
19:        if len(combined_unpaired) mód 2 = 1 then
20:            return último elemento de combined_unpaired
21:        else
22:            return lista vacía                    ▷ Todas las estaciones están emparejadas
23:        end if
24:    end function
25:    DFS-Emparejar(r)                              ▷ Llamar DFS desde la raíz
26: end function
```

Corrección del algoritmo:

Demostración:

Se demuestra por inducción sobre la altura del árbol T, $h(T)$.

- **Caso Base ($h(T) = 0$).** En este caso se tiene una hoja (la raíz).
  - Si el nodo no es una estación ($flag$ = False), entonces la lista devuelta es vacía ([]).
  - Si el nodo es una estación ($flag$ = True), la lista devuelta contiene solo esa estación ($[v]$).

Para esta situación se satisface lo requerido, pues solo se tiene un nodo qué procesar además de que no existen caminos que se sobrepongan.

- **Hipótesis de Inducción**:
  Para cualquier subárbol con raíz en un nodo v y altura menor o igual a h, el algoritmo DFS-Emparejar(v) empareja correctamente todas las estaciones dentro de ese subárbol, utilizando caminos disjuntos en aristas.

- **Paso Inductivo:**

  - Consideremos un subárbol con altura $h+1$ y raíz en un nodo v:
    El algoritmo llama recursivamente a `DFS-Emparejar` en los hijos izquierdo y derecho de v (líneas 6 y 7):
    - `left_unpaired = DFS-Emparejar(left(v))`
    - `right_unpaired = DFS-Emparejar(right(v))`

    De donde los hijos izquiero y derecho de v son árboles que tiene longitud menor a $h+1$
  - Por la hipótesis inductiva, las listas `left_unpaired` y `right_unpaired` contienen las estaciones no emparejadas de los subárboles izquierdo y derecho, respectivamente.
  - El algoritmo combina las listas de estaciones no emparejadas de ambos subárboles (línea 8):
    - `combined_unpaired = left_unpaired` $\cup$ `right_unpaired`
  - Ahora, `combined_unpaired` contiene todas las estaciones no emparejadas de los subárboles izquierdo y derecho.
  - Si el nodo actual v es una estación (`flag(v) = True`), se agrega a `combined_unpaired`:
    - `combined_unpaired = combined_unpaired` $\cup\{v\}$
  - El algoritmo empareja las estaciones en `combined_unpaired` de dos en dos, ciclo `while 13-18`. Esto asegura que todas las estaciones posibles en el subárbol con raíz en v están correctamente emparejadas utilizando caminos disjuntos.
  - Al final, si `combined_unpaired` tiene un número impar de estaciones, retorna la última como la única estación sin emparejar. De lo contrario, devuelve una lista vacía.

  Se concluye que el algoritmo es correcto.

Complejidad Temporal

- **Recorrido DFS:**
  El algoritmo realiza un recorrido DFS desde la raíz y visita cada nodo una sola vez. Para cada nodo v en el árbol:

  - Se llama a `DFS-Emparejar` en los hijos izquierdo y derecho.
  - Se combinan las listas `left_unpaired` y `right_unpaired` (operación de unión).
  - Como `DFS-Emparejar` se llama exactamente una vez por cada nodo, el tiempo de recorrido es $O(n)$, donde n es el número de nodos en el árbol.

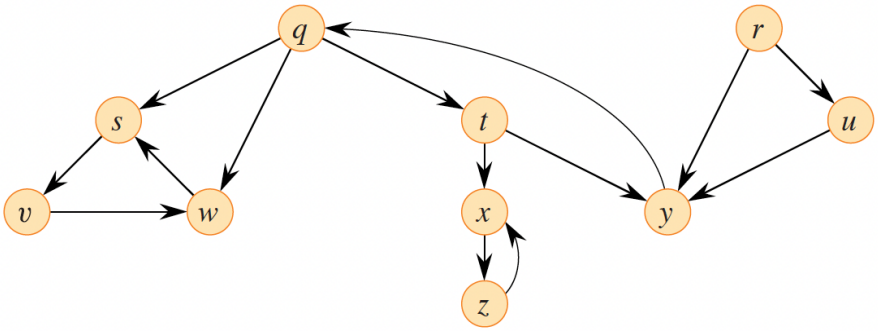
Otra forma de calcular la complejidad para este fragmento de código es haciendo lo siguiente.

Se sabe que la complejidad temporal de DFS es $O(|V| + |E|)$, siendo $|V|$ el número de nodos de la gráfica y $|E|$ el número de aristas.
Como se tiene un árbol, entonces $|E| = n - 1$, siendo $|V| = n$, por lo que la complejidad es de la forma: $O(|V| + |E|) = O(n + n - 1) = O(2n - 1) = O(n)$

- **Operaciones de Emparejamiento y Combinación:**
  - **Combinar listas** (`combined_unpaired = left_unpaired` ∪ `right_unpaired`):
    - Combinar dos listas (`left_unpaired` y `right_unpaired`) tiene un costo lineal en el número de elementos de las listas.
    - Como cada estación se puede emparejar como máximo una vez en todo el recorrido, la suma total de operaciones de combinación y emparejamiento es $O(m)$, donde m es el número total de estaciones.
    - Sin embargo, en el peor de los casos, el número de estaciones es proporcional a n, por lo que esta operación tiene costo $O(n)$.
  - **Emparejamiento de Estaciones:**
    - Cada emparejamiento se realiza en tiempo constante, ya que el número de estaciones no emparejadas se recorre de dos en dos.
    - Nuevamente, el número total de estaciones es como máximo $O(n)$, por lo que el costo total de emparejamiento es $O(n)$.
  - **Verificar si el número de estaciones es impar:**
    - Esta verificación se realiza en tiempo constante $O(1)$.
- **Complejidad Total:**
  - La complejidad total del algoritmo es la suma de las complejidades de las operaciones:
    - **Recorrido DFS:** $O(n)$
    - **Combinación de listas:** $O(n)$
    - **Emparejamiento de estaciones:** $O(n)$
  - Por lo tanto, la complejidad total del algoritmo es $O(n)$.

Problema 4

Ejecuta el algoritmo de componentes fuertemente conexas sobre la gráfica G que aparece en la siguiente figura. Muestra el orden topológico que se induce sobre G_{CC} después de la primera ejecución de DFS, y también indica el orden en el que quedan los vértices en la lista L. Finalmente, muestra las componentes que calcula tu ejecución.



Solución:

A continuación se muestra el algoritmo `DFS`.

DFS

```
1  DFS(G)
2      for each vertex u in G.V:
3          u.color = WHITE
4          u.pi = NIL
5      time = 0
6      for each vertex u in G.V:
7          if u.color == WHITE:
8              DFS-VISIT(G, u)
9
10 DFS-VISIT(G, u)
11     time = time + 1
12     u.d = time
13     u.color = GRAY
14     for each vertex v in G.Adj[u]:
15         if v.color == WHITE:
16             v.pi = u
17             DFS-VISIT(G, v)
18     time = time + 1
19     u.f = time
20     u.color = BLACK
```

Mientras que el algoritmo de componentes fuertemente conexas es de la siguiente forma:

Algoritmo de componentes fuertemente conexas

- `Entrada:` Una gráfica dirigida G.

```
0.- Crear una lista vacía L.

1.- LLamar DFS(G) y al momento de fijar 'u.f' agregar 'u' al frente de L.

2.- Crear el grafo transpuesto `G^T`.

3.- Llamar a `DFS(G^T)`, seleccionando los vértices usando L, en orden descendente
con respecto a su tiempo de terminación, i.e., del último que se agregó al primero

4.- Los vértices descubiertos en el mismo árbol son una CFC.
```

Para una mejor lectura, se hace una tabla en donde se registran los tiempos de iniciación de los vértices y los tiempos de finalización al ejecutar DFS desde s.

Nodo	**Descubrimiento** (d)	**Finalización** (f)
s	1	6
v	2	5
w	3	4
q	7	16
t	8	13
x	9	12
z	10	11
y	14	15
r	17	20
u	18	19

Con ayuda de la tabla anterior, se crea la lista `L`, la cual es de la siguiente forma:

$$L = [r, u, q, y, t, x, z, s, v, w]$$

En donde se han ordenado los nodos con base a su tiempo de finalización (f).
El siguiente paso es crear G^T, la cual es mostrada en la figura 1.

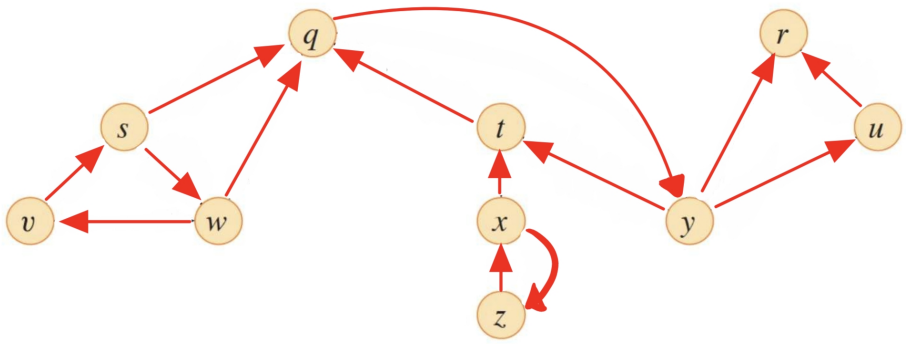


Figura 1: G^T

Finalmente se ejecuta `DFS` sobre la gráfica G^T obteniendo las componentes fuertemente conexas, tal como advierte la imagen 2.

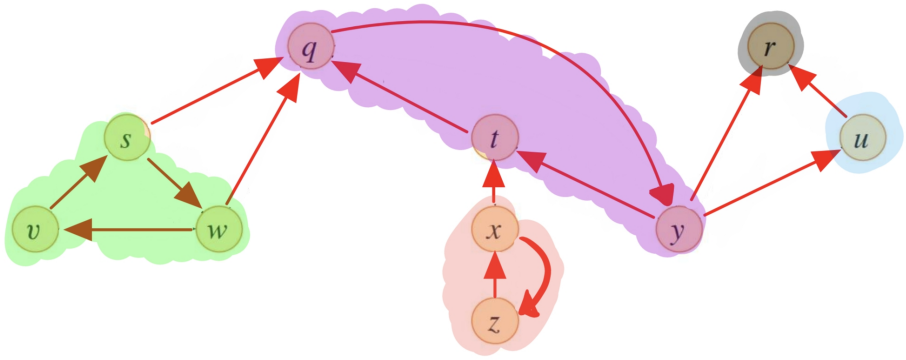


Figura 2: Componentes Fuertemente Conexas

Así, se obtiene la gráfica de componentes conexas; GCC.

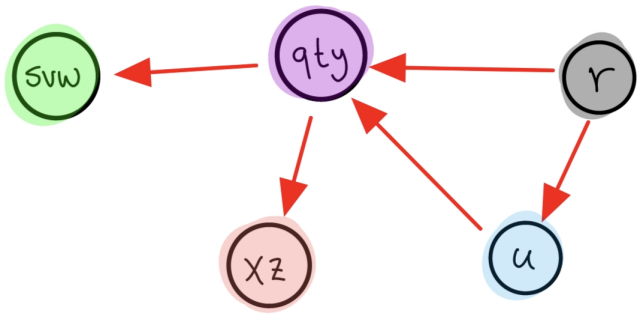


Figura 3: GCC

Por lo que el orden topológico inducido sobre GCC es el siguiente:

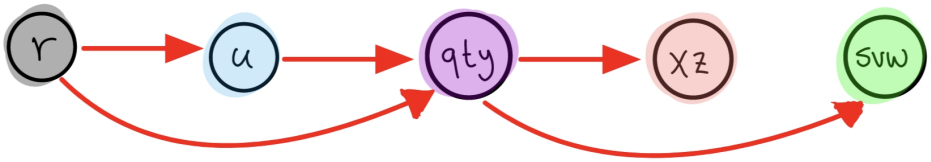


Figura 4: Orden topológico inducido sobre GCC.