Repaso I2

Rojo-negro, B+, Hash, Grafos y Estrategias

Joaquín Peralta - Elías Ayaach - Alex Infanta - Steven Reynolds - Paula Grune

Hashing



Tablas de Hash 2019-2

La pizzería intergaláctica M87 sirve pizzas de todos los incontables sabores existentes en todos los multiversos, y necesita ayuda para hacer más eficiente la atención de la millonaria cantidad de pedidos que recibe por segundo. Los pedidos funcionan de la siguiente manera:

- Una persona solicita una pizza del sabor que haya escogido y da su nombre. Su pedido se agrega al sistema.
- Cuando la pizza esta lista, se llama por el altavoz a la persona que haya pedido ese sabor hace más tiempo, y, una vez entregada la pizza, se borra el pedido del sistema.
  Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente. ¿Qué esquema de resolución de colisiones debería usarse y por qué? ¿Qué es lo que se guarda en la tabla?



Solución

Para eso usamos una tabla de hash que nos permita guardar múltiples values para un mismo key. En este caso key corresponde al tipo de pizza y un value corresponde al nombre de la persona que lo pidió.

Los values de un mismo key se deben guardar en una cola FIFO, de manera de agregar un nuevo value o extraer el siguiente, sea de O(1) y se atienda en orden de llegada.
La eliminación del pedido sale automática con esta estructura, ya que obtener el siguiente elemento lo extrae de la estructura.

Como el dominio de las keys es infinito, debemos ir despejando las celdas de la tabla cuando una key se queda sin values, ya que no tenemos memoria infinita. Para esto es necesario usar encadenamiento ya que permite eliminar keys de la tabla, sin perjudicar el rendimiento de esta. De esta manera, en caso de que varios sabores de pizza sean hasheados a la misma celda, las colas FIFO de los nombres de las personas que pidieron esas pizzas deberán ser encadenadas en una lista doblemente ligada que las contenga

Grafos



Contexto

Bob colecciona libros raros y descubrió recientemente un diccionario escrito en un idioma desconocido que utiliza los mismos caracteres que el idioma castellano. Sin embargo, el orden de las palabras en el diccionario es diferente de lo que uno esperaría si los caracteres estuvieran ordenados de la misma manera que en castellano. Bob intentó determinar el orden de los caracteres del extraño alfabeto usando la lista ordenada de palabras del diccionario. Frustrado por la tediosidad de la tarea, se dio por vencido y decidió pedirle ayuda a su amiga Alice, que modeló el problema usando grafos y fue capaz de resolverlo de inmediato. En esta pregunta, usted deberá replicar el proceso con el que Alice resolvió el problema, diseñando un algoritmo tal que dada una lista de palabras ordenada alfabéticamente (usando el orden desconocido de los caracteres), permita obtener una lista ordenada con los caracteres del alfabeto.

Ejemplo: Para la siguiente lista ordenada de palabras

XWY
ZX
ZXY
ZXW
YWZ

El orden de los caracteres que la produce es X < Z < Y < W.

Asumiendo que la lista ordenada de palabras contiene suficiente información para obtener el orden buscado, responda los siguientes puntos.

a) Modele el problema usando un grafo, de tal manera que a partir del mismo pueda obtener el resultado pedido (es decir, el orden de los caracteres). Indique el tipo de grafos que va a usar (dirigido, no dirigido, cíclico, acíclico) y a qué corresponde cada una de sus componentes.

Solución: A partir de los strings de entrada (y su orden) se pueden derivar relaciones de orden entre los caracteres. Se define un grafo G=(V,E) tal que para todo carácter x del alfabeto, hay un nodo v_x en V . Luego, agregaremos aristas al grafo que muestren las dependencias entre caracteres que se pueden leer de los strings. Es decir, la arista dirigida (v_x, v_y) pertenece al conjunto de aristas E del grafo si y sólo si se puede deducir x<y desde los strings de entrada.

Dado que se asume que hay suficiente información en los strings, el grafo tendrá suficientes aristas que permitan obtener el orden alfabético pedido. El grafo es acíclico y dirigido , y por lo tanto admite un orden topológico, que es lo que nos dará el orden pedido (las aristas nos dan el orden entre pares de caracteres, un orden topológico nos da el orden completo).

b) Diseñe un algoritmo tal que a partir de la lista ordenada de palabras genere el grafo del punto anterior, y luego a partir del grafo permita obtener la lista ordenada de caracteres.

```
Algorithm 1: SortAlfabeto(P[1..n])
begin
    V ← ∅
    E ← ∅
    s ← P[1]

    for i = 2 to n:
        s′ ← P[i]
        j ← 0
        while j < s.len() and j < s′.len():
            if s[j] ≠ s′[j]:
                E ← E ∪ {(s[j], s′[j])}   // relación s[j] < s′[j]
                V ← V ∪ {s[j], s′[j]}
                break
            j ← j + 1
        s ← s′

    Construir el grafo G = (V, E)
    L ← TopSort(G)
    return L
end
```



c) Muestre el grafo correspondiente a la siguiente lista de palabras, y el resultado de aplicar su algoritmo sobre el mismo:

CEDA
CFA
CFD
CBE
AFE
AD
ABC
EAC
EE
EFF
EFD

Además de la lista ordenada de caracteres, muestre (gráficamente) el estado final de la ejecución de su algoritmo que le permite determinar el resultado obtenido.

Solución: El grafo para el ejemplo dado tiene el conjunto de nodos:

$$V = \{A, B, C, D, E, F\}$$

Mientras que el conjunto de aristas que se puede deducir de las palabras es:

$$E = \{(C, A), (A, E), (A, D), (E, F), (F, B), (F, D), (D, B)\}$$

Luego de ejecutar el sort topológico, los vértices quedan marcados con los intervalos:

- A tiene el intervalo [2, 11]
- B tiene el intervalo [4, 5]
- C tiene el intervalo [1, 12]
- D tiene el intervalo [3, 6]
- E tiene el intervalo [7, 10]
- F tiene el intervalo [8, 9]



El sort topológico por lo tanto produce el orden: C, A, E, F, D, B.

Backtracking
vs
Greedy vs DP



Las 4 técnicas vistas

Para atacar los problemas complejos que se nos han presentado, contamos con las siguientes estrategias:

- Dividir para conquistar (D&C)
- Backtracking
- Algoritmos codiciosos (Greedy)
- Programación dinámica (DP)

Un breve repaso...

Dividir para conquistar

- Dividir el problema en problemas más pequeños hasta que sea trivial resolverlos
- Los problemas son disjuntos: Cada subproblema es independiente del resto y se resuelve aparte
- La solución del problema grande se construye a partir de las soluciones de los problemas pequeños

EJEMPLO: Ordenar un array con MergeSort

Backtracking

- Definimos variables, dominios y restricciones
- Asignamos valores a las variables de forma ordenada
- Si alguna restricción se rompe, deshacemos la última asignación y probamos nuevamente
- Iteramos hasta resolver el problema
- GRAN complejidad → $O(K^n)$

EJEMPLO: Salir de un laberinto dándose golpes contra la pared

Un breve repaso...

Greedy

- Iremos haciendo decisiones de forma secuencial
- Para cada paso, tomaremos la decisión que parezca mejor en el momento
- ¿Cuál es la mejor decisión? La que minimice/maximice cierta función
- Eficientes, pero podría llevarnos a mínimos/máximos locales

EJEMPLO: Queriendo encontrar el camino más corto en un grafo, tomar siempre la arista de menor peso

Programación Dinámica

- Al igual que Dividir para Conquistar, dividir el problema en subproblemas pequeños que tienen subestructura óptima
- Estos subproblemas NO son disjuntos, si no que se repiten al descomponer otros problemas
- Resolvemos los problemas sencillos y guardamos sus soluciones para usarlas al componer las soluciones de problemas mayores

EJEMPLO: Dar vuelto con la menor cantidad de monedas

¿Cuál debemos usar?

Dependiendo del problema, todos podrían encontrar solución, pero algunas soluciones podrían ser mejores que otras

Hay que hacerse preguntas tales como:

¿Queremos maximizar minimizar algo?

¿Hay restricciones?

¿Hay problemas similares más pequeños? ¿Se repiten?

Queremos encontrar la mejor estrategia para el problema

Contexto

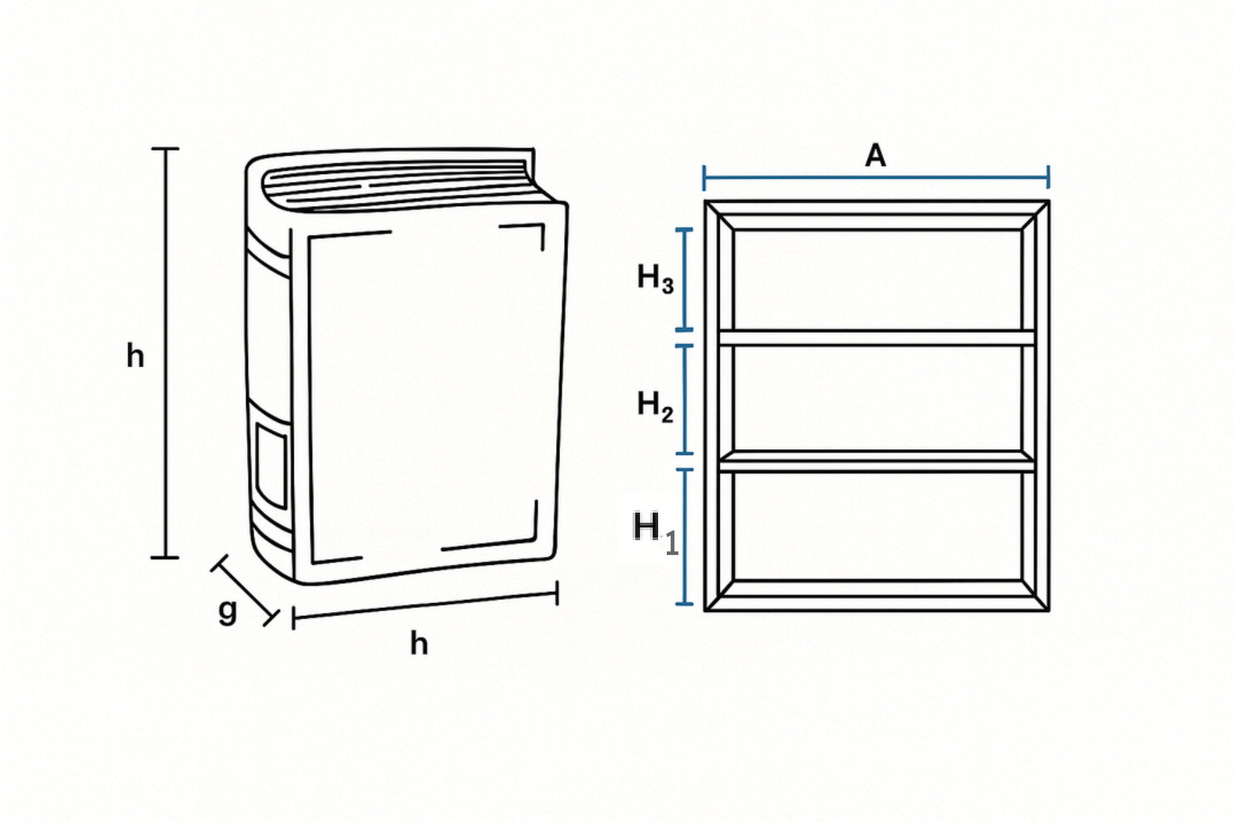
Bob acaba de comprar una repisa de libros de 3 pisos de diferentes alturas H1, H2, H3 cms y de A cms de ancho y con una capacidad máxima de W kg. Los libros se deben acomodar en forma vertical, es decir, con su grosor ocupando espacio en el ancho de la repisa.

Además, cuenta con un conjunto de libros a ser acomodados en la repisa. Cada libro Li se caracteriza por su altura hi cms, su grosor gi cms y su peso wi kg.

Suponga que todos los libros caben de alto en alguna repisa y que cada piso de la repisa puede acomodar a varios libros.

En los siguientes puntos, seleccione el uso de una estrategia algorítmica distinta (es decir, no puede usar la misma t´ecnica en dos puntos distintos) entre Backtracking, Codiciosa o Programaci´on Din´amica que mejor se ajuste para resolver cada uno de los siguientes problemas. En cada caso debe plantear las restricciones, la función objetivo, la ecuación de recurrencia o la estrategia codiciosa seg´un corresponda, describir la soluci´on y justificar la elecci´on realizada.

Imagen de Referencia



- Restriccion de largo por piso: $\sum_{i=1}^{n} g_i <= A$ para cada **piso** y $g_(i+1) > L$
- Restriccion de peso en repisa completa: $\sum_{i=1}^{n} w_i <= W$ y $\sum_{i=1}^{n} w_{i+1} > W$ para la repisa completa
- Resricción de altura por piso $h_i <= H_1, H_2, H_3$

a) Asignar libros a pisos de la repisa de modo que se cumpla con todas las condiciones.

Solución 1: Backtracking. Solo se necesita una solución cualquiera que cumpla con que la suma de los pesos de los libros acomodados no supere el límite. Se eligen los Li y se acomodan de modo que se cumplan las 3 restricciones.

Solución 2: Programación dinámica.

- Restriccion de largo por piso: $\sum_{i=1}^{n} g_i <= A$ para cada **piso** y $g_(i+1) > L$
- Restriccion de peso en repisa completa: $\sum_{i=1}^{n} w_i <= W$ y $\sum_{i=1}^{n} w_{i+1} > W$ para la repisa completa
- Resricción de altura por piso $h_i <= H_1, H_2, H_3$

b) Distribuir libros para maximizar el número total de libros ubicados, sin exceder ancho ni peso soportado por la repisa.

Solución 1: Greedy.
La estrategia codiciosa puede ser escoger por densidad (peso/grosor) o escoger los libros más livianos primero hasta alcanzar el máximo en cada piso.

Solución 2: Backtracking con heurística.

- Restriccion de largo por piso: $\sum_{i=1}^{n} g_i <= A$ para cada **piso** y $g_(i+1) > L$
- Restriccion de peso en repisa completa: $\sum_{i=1}^{n} w_i <= W$ y $\sum_{i=1}^{n} w_{i+1} > W$ para la repisa completa
- Resricción de altura por piso $h_i <= H_1, H_2, H_3$

c) Ordenar libros para minimizar el espacio sobrante en cada piso sin exceder el peso ni el ancho.

Solución: Programación dinámica.
Es similar al problema de la mochila.
El espacio disponible después del libro Li es d[i][j] por cada piso j.
La ecuación de recurrencia tiene 4 opciones: colocar el libro en Piso1, Piso2, Piso3 o no colocarlo. d[i] = A - max(A - Σ(gi), A), con j = 1, 2, 3.

Arboles Rojo-Negro



Contexto

Una empresa de retail cuenta con un sistema de gestión de clientes (CRM) que registra la información de los clientes utilizando un árbol rojo-negro C, en que la llave es el RUN del cliente (tipo int) y en cada nodo x se almacena la información del cliente de la forma x.run, x.nombre, x.edad, x.renta, x.color, x.p (puntero al padre del nodo x), etc. Solo se consideran las operaciones de búsqueda e inserción de clientes.

La empresa requiere generar de manera eficiente listas de clientes para realizar campañas de marketing. Por ejemplo, obtener desde C: “la lista de run de todos los clientes mayores de 25 años y menores de 35, ordenados por edad” (select run from C where (25 < edad < 35) order by edad;).

El arquitecto de sistemas define que se construyan índices para C utilizando tres nuevos árboles rojo-negro con llaves adecuadas para las búsquedas indicadas: E con llave edad.run, R con llave renta.run, N con llave nombre.run. Los nodos i de estos árboles índice tienen un puntero data al nodo de C correspondiente al run (i.data->x(con llave run)). Proponga para los casos siguientes un algoritmo que resuelva lo solicitado, asuma que los árboles C, E, R, N existen y están poblados correctamente.

a) Escriba en pseudo código el algoritmo buscarCliente(C,E,R,N,x,llave) que busca al cliente x por la llave indicada (run, edad, renta, nombre) en el árbol/índice adecuado retornando el nodo encontrado o null si no está en el árbol C.

```
Buscar(n, x, llave):
    if n == null :
        return n
    if llave == RUN :
        k ← x.run
    else:
        k ← x.llave + '.' + x.run
    if n.llave == k :
        return n
    if n.llave > k :
        return Buscar(n.left, x, llave)
    else:
        return Buscar(n.right, x, llave)
```

```
BuscarCliente(C, E, R, N, x, llave):
    if llave == RUN :
        n ← C
    elif llave == edad :
        n ← E
    elif llave == renta :
        n ← R
    elif llave == nombre :
        n ← N
    m ← Buscar(n, x, llave)
    return m
```

b) Escriba en pseudo código el algoritmo insertCliente(C,E,R,N,x) que inserta los datos del cliente x (run, edad, renta, nombre) en C y actualiza los índices E,R,N. Asuma que la función FixBalance(A,z) está disponible y la puede utilizar en el árbol A adecuado al insertar un nodo z.

```
InsertCliente(C, E, R, N, x):
    for llave ∈ {run, edad, renta, nombre} :
        ref ← null
        n ← BuscarCliente(C, E, R, N, x, llave)
        if n == null :
            if llave == run :
                ref ← Nuevo nodo con los datos de x
                Insert(C, ref)
                FixBalance(C, ref)
            if llave == run :
                if llave == edad :
                    b ← E
                elif llave == renta :
                    b ← R
                elif llave == nombre :
                    b ← N
                nodo ← Nuevo nodo
                nodo.llave ← ref.llave + '.' + ref.run
                nodo.data ← ref
                Insert(b, nodo)
                FixBalance(b, nodo)
```

c) Escriba en pseudo código el algoritmo rangoClientes(C,E,R,N,llave,min,max) que retorna una lista de punteros a nodos de C que cumplen que el valor de la llave se encuentra entre los parámetros min y max, y la lista está ordenada de manera ascendente en el valor de la llave

```
RangoClientes(C, E, R, N, llave, min, max):
1     if llave == RUN :
2         n ← C
3     elif llave == edad :
4         n ← E
5     elif llave == renta :
6         n ← R
7     elif llave == nombre :
8         n ← N
9     L ← lista vacía
10    ListarNodosOrdenados(n, llave, min, max, L)
11    return L
```

```
ListarNodosOrdenados(n, llave, min, max, L):
1     if n ≠ null :
2         if n.llave ≥ min AND n.llave ≤ max :
3             ListarNodosOrdenados(n.left, llave, min, max, L)
4             Agregar n al final de la lista L
5             ListarNodosOrdenados(n.right, llave, min, max, L)
6         elif n.llave < min :
7             ListarNodosOrdenados(n.right, llave, min, max, L)
8         elif n.llave > max :
9             ListarNodosOrdenados(n.left, llave, min, max, L)
```

Grafos II
Orden Topológico



Definición

- Sea G un grafo dirigido. Un orden topológico de G es una secuencia de sus nodos

$$v_0, v_1, \ldots, v_n \qquad v_i \in V(G)$$

- Tal que
  - Todo nodo del grafo aparece en la secuencia
  - En la secuencia no hay elementos repetidos
  - Si (a,b) ∈ E(G) entonces el nodo a aparece antes que el nodo b en la secuencia



Orden Topologico: Pseudocódigo

```
input  : grafo G
output: lista de nodos L

TopSort(G):
1     L ← lista vacía
2     t ← 1
3     for u ∈ V(G) :
4         u.start ← 0
5         u.end ← 0
6     for u ∈ V(G) :
7         if u.start = 0 :
8             TopDfsVisit(G, L, u, t)
9     return L
```

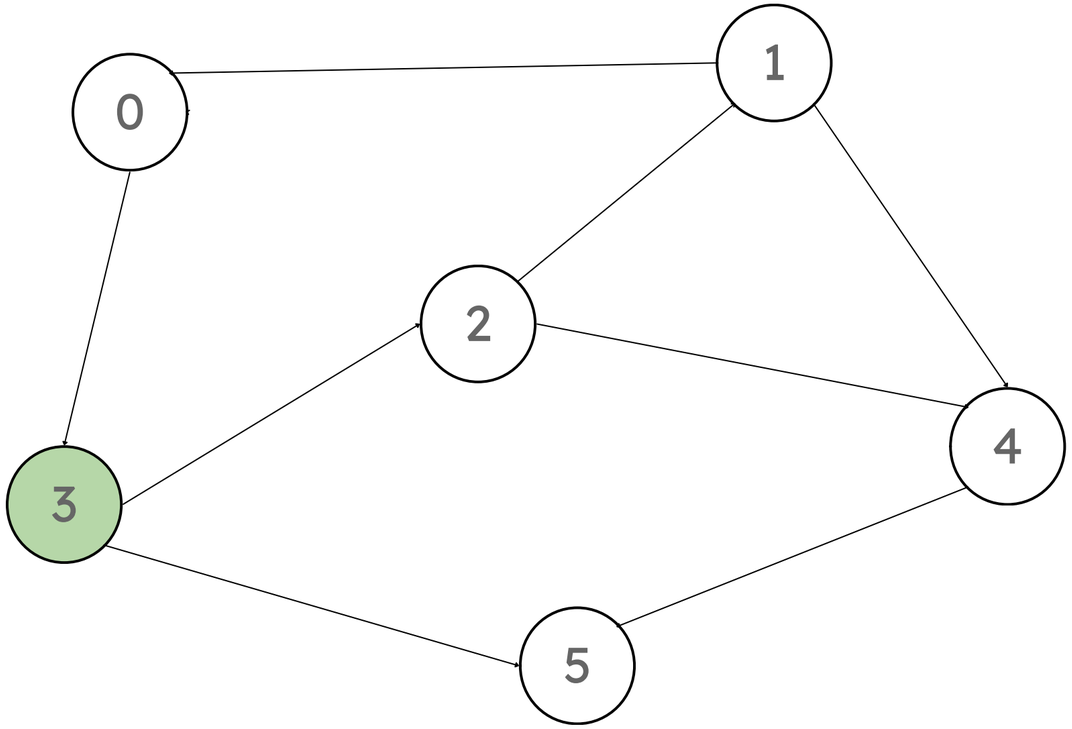
```
input  : grafo G, lista de nodos L,
         nodo u ∈ V(G), tiempo t
output: tiempo t ≥ 1

TopDfsVisit(G, L, u, t):
1     u.start ← t
2     t ← t + 1
3     for v ∈ N_G(u) :
4         if v.start = 0 :
5             TopDfsVisit(G, L, v, t)
6     u.end ← t
7     Insertar u como cabeza de L
8     t ← t + 1
9     return t
```

¿Cómo se ve?

1. Elegimos un nodo para comenzar, en este caso usaremos el nodo 3

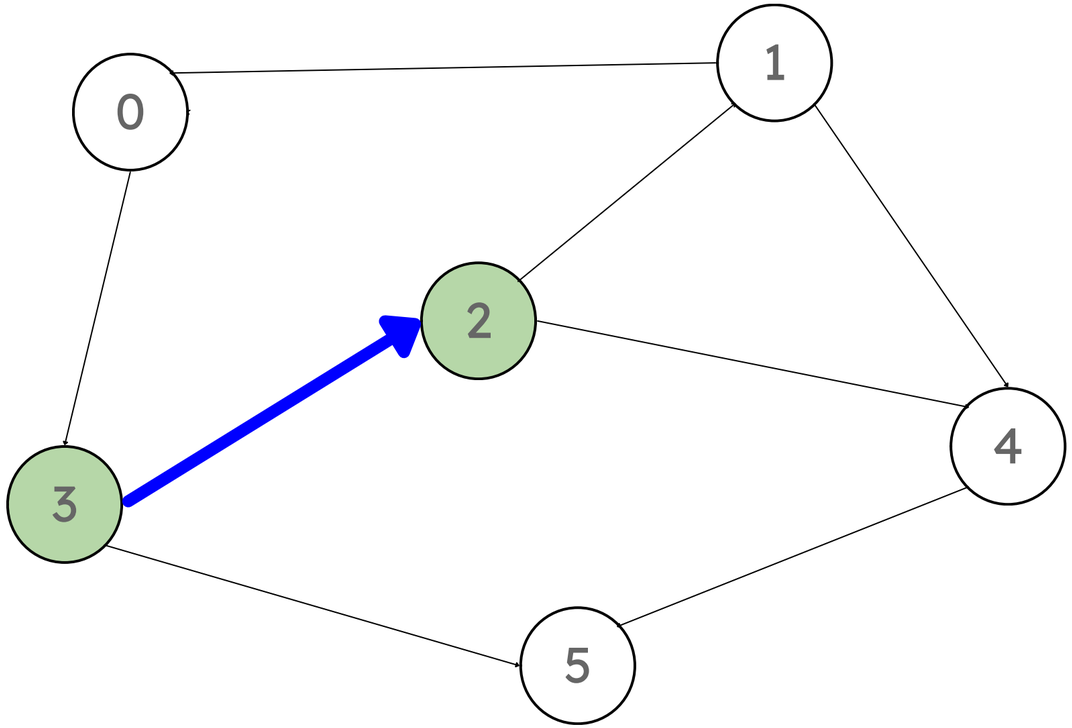
   t = 0



Nodo	Start	End	OT
0	0	0	
1	0	0	
2	0	0	
3	0	0	
4	0	0	
5	0	0	

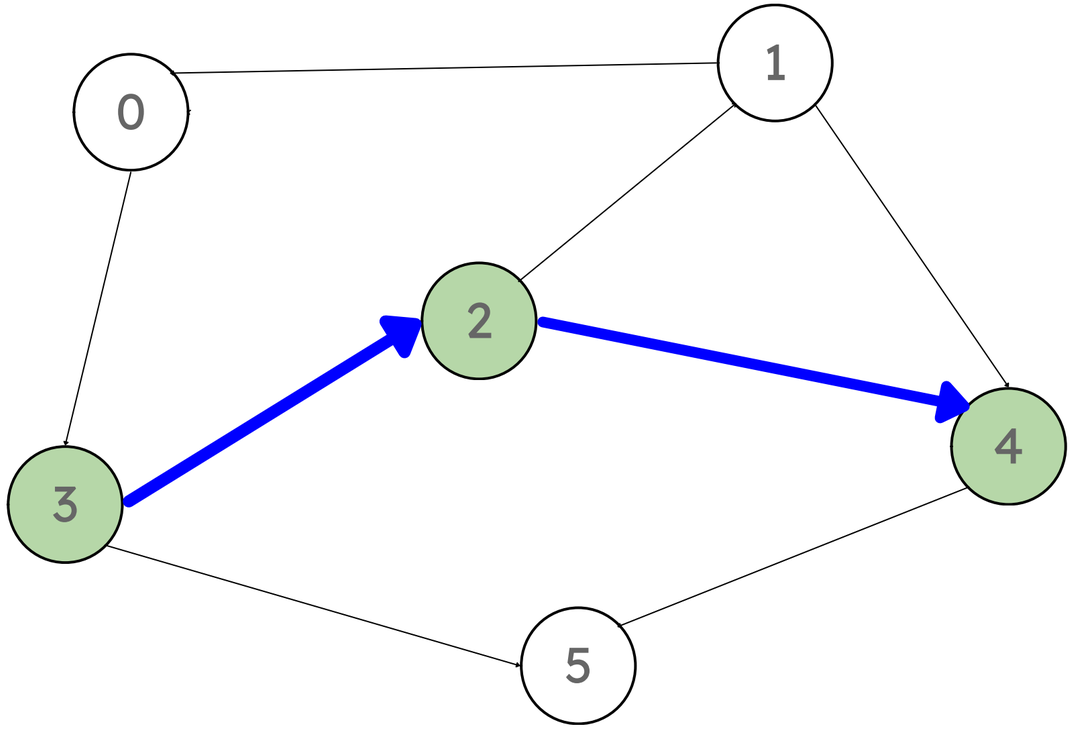
- Elegimos un nodo hijo del nodo actual: 2

  t = 1



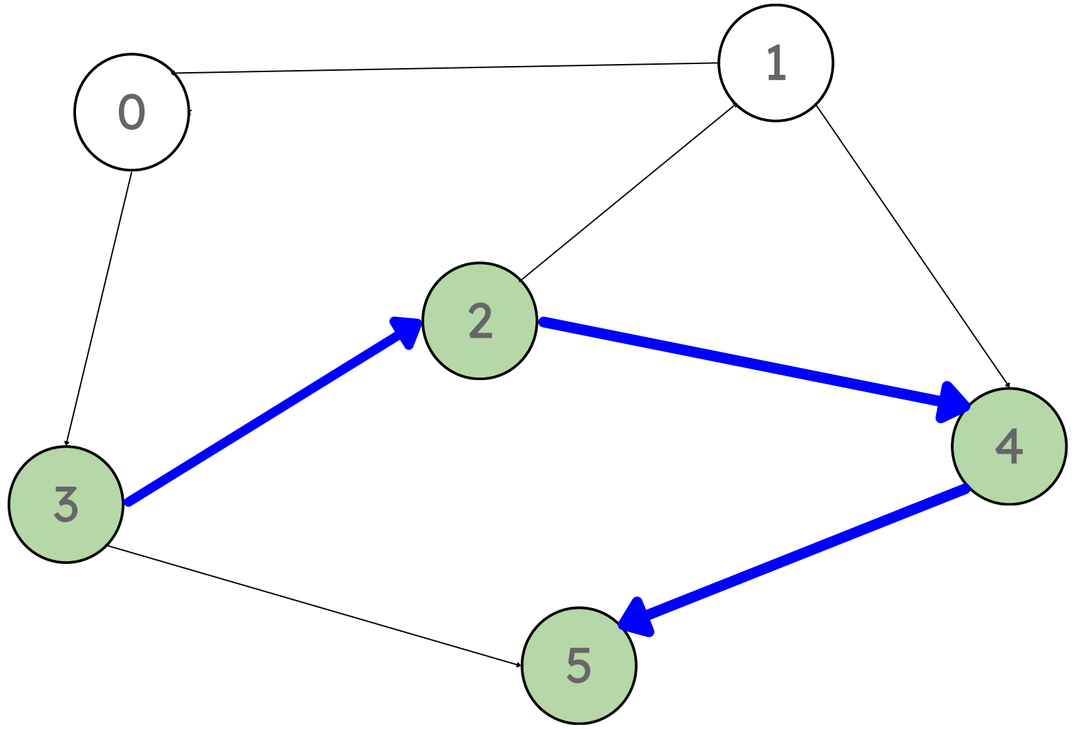
Nodo	Start	End	OT
0	0	0	
1	0	0	
2	1	0	
3	0	0	
4	0	0	
5	0	0	

- Elegimos un nodo hijo del nodo actual: 4
  t = 2



Nodo	Start	End	OT
0	0	0	
1	0	0	
2	1	0	
3	0	0	
4	2	0	
5	0	0	

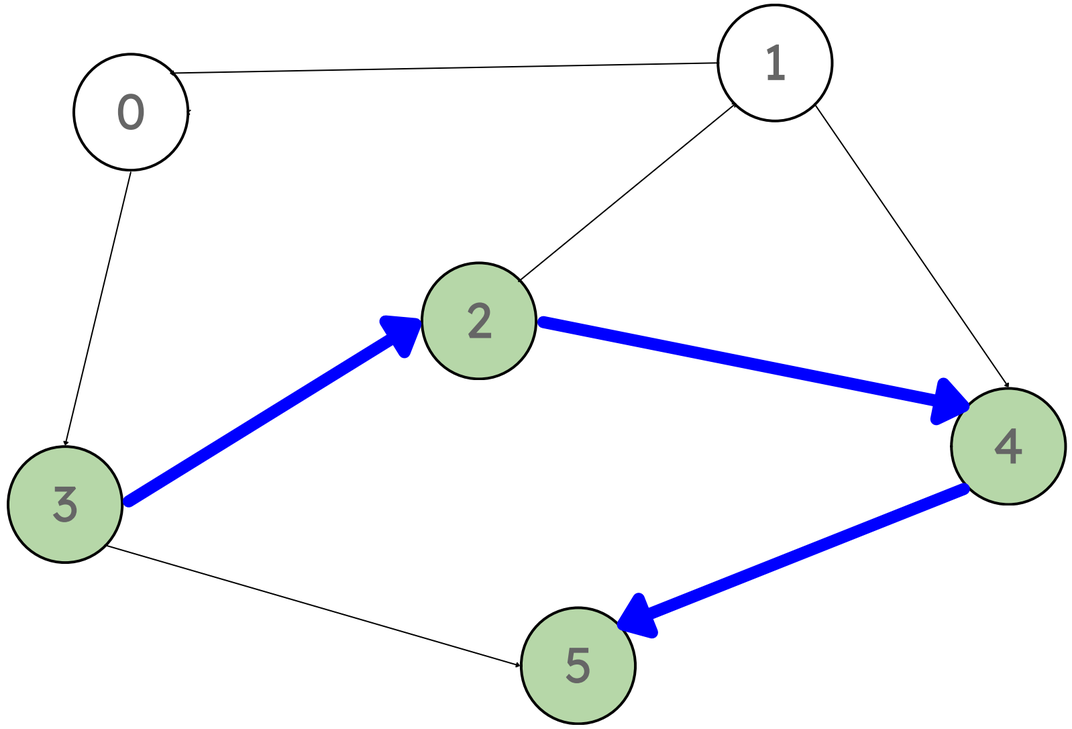
- Elegimos un nodo hijo del nodo actual: 5
  t = 3



Nodo	Start	End	OT
0	0	0	
1	0	0	
2	1	0	
3	0	0	
4	2	0	
5	3	0	

- El nodo actual no tiene nodos hijos, por lo cual lo agregamos al orden topológico
- Seteamos su End en t=4

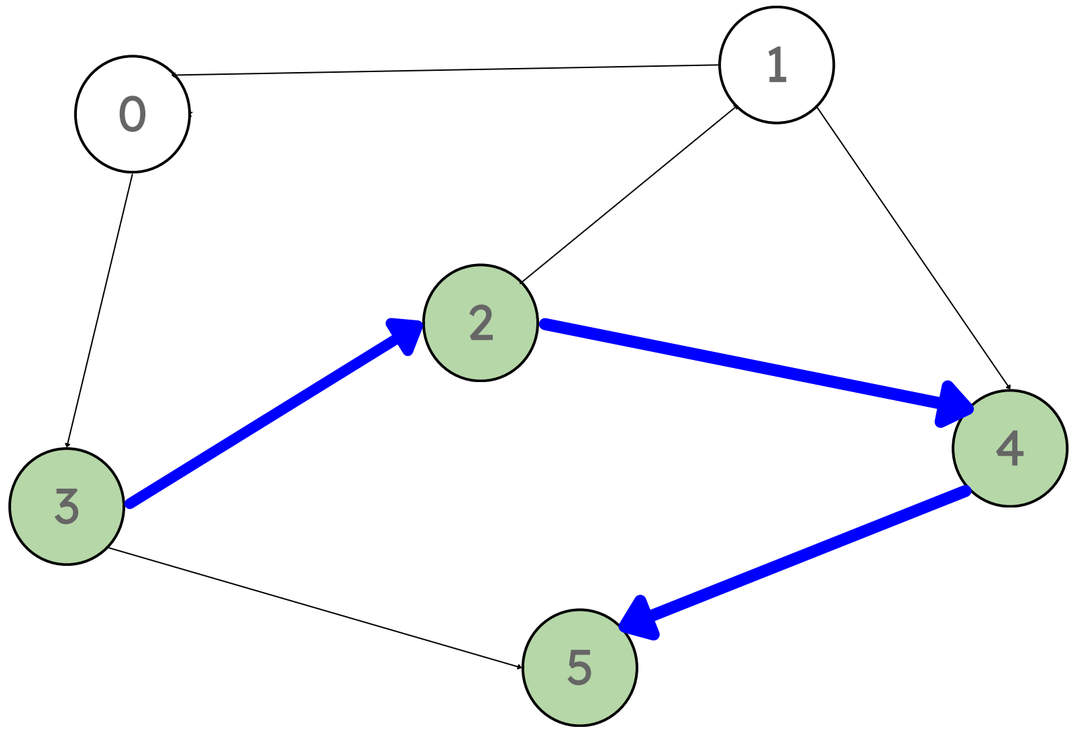
  t = 4



Nodo	Start	End	OT
0	0	0	5
1	0	0	
2	1	0	
3	0	0	
4	2	0	
5	3	4	

- Volvemos al nodo (4) y este no tiene más hijos.
  Lo agregamos al OT
- Seteamos su End en t=5

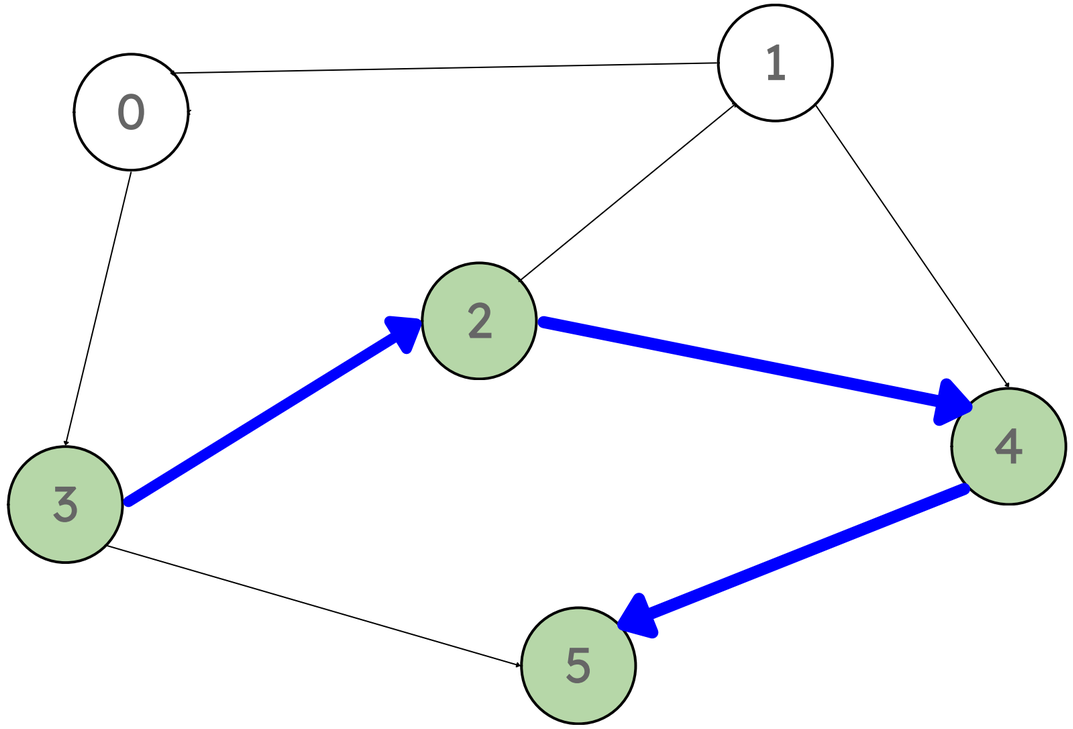
  t = 5



Nodo	Start	End	OT
0	0	0	4
1	0	0	5
2	1	0	
3	0	0	
4	2	5	
5	3	4	

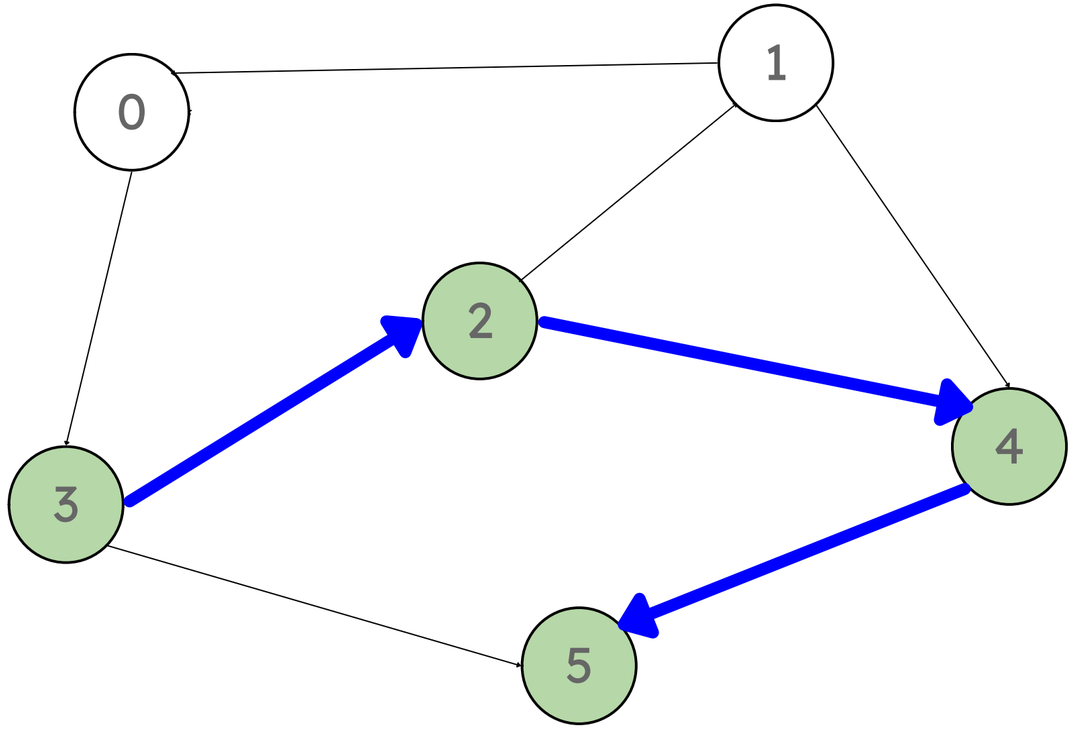
- Volvemos al nodo (2) y este no tiene más hijos.
  Lo agregamos al OT
- Seteamos su End en t=6

  t = 6



Nodo	Start	End	OT
0	0	0	2
1	0	0	4
2	1	6	5
3	0	0	
4	2	5	
5	3	4	

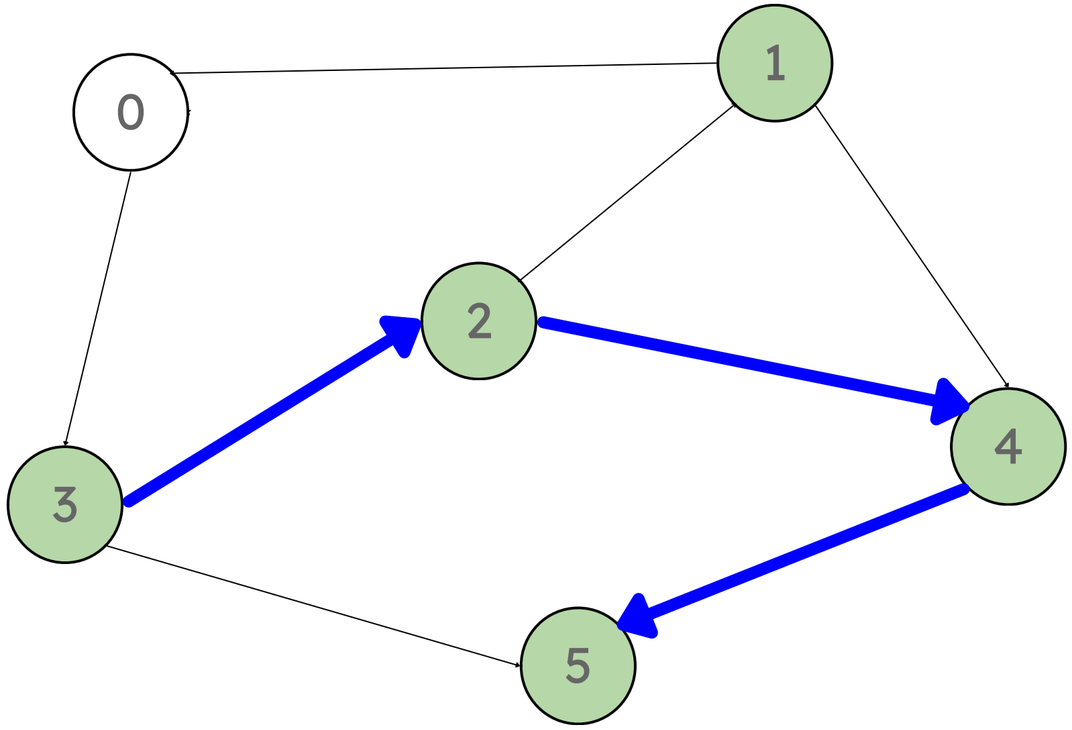
- Volvemos al nodo (3) y tiene el nodo (5) de hijo.
  El start del nodo (5) no es 0.
- Lo agregamos al OT
- Seteamos su End en t=7
  t=7



Nodo	Start	End	OT
0	0	0	3
1	0	0	2
2	1	6	4
3	0	7	5
4	2	5	
5	3	4	

- Elegimos un nodo restante, elegimos el (1)

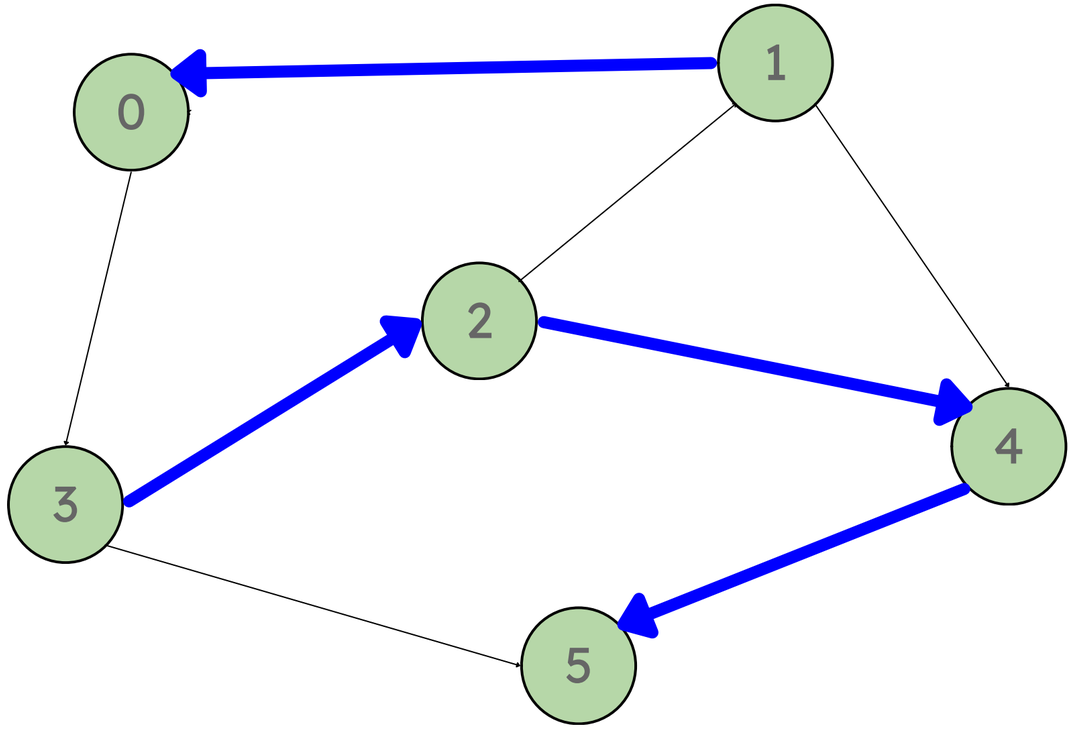
  t=8



Nodo	Start	End	OT
0	0	0	3
1	8	0	2
2	1	6	4
3	0	7	5
4	2	5	
5	3	4	

- Elegimos un nodo hijo: (0)

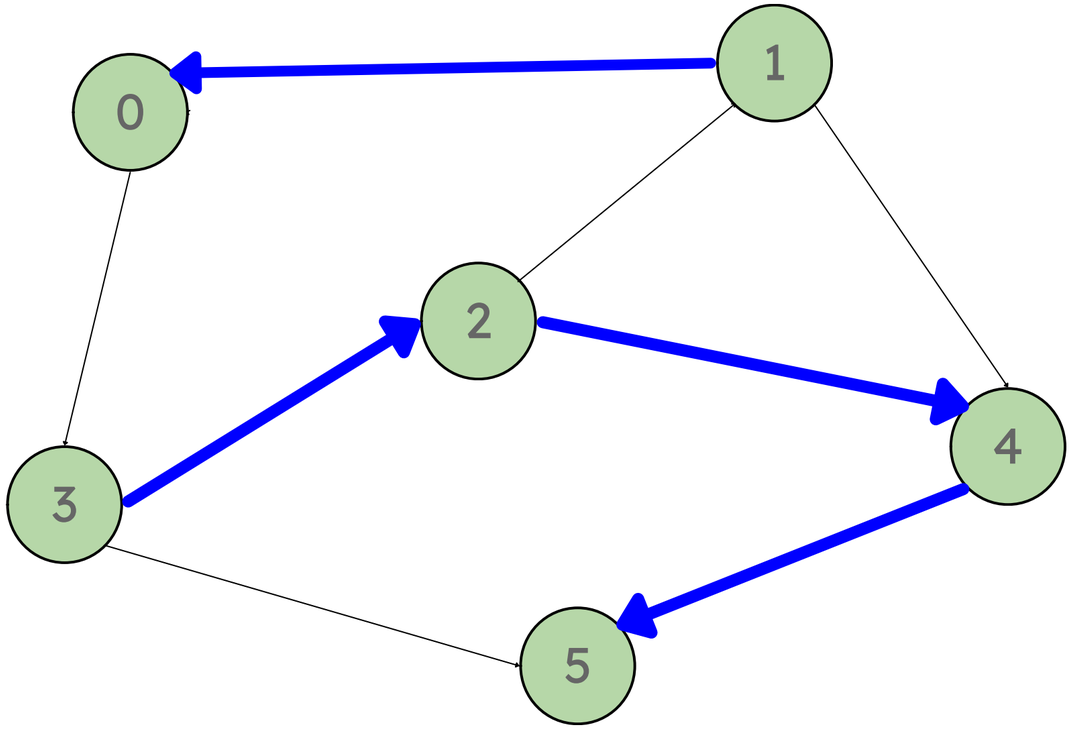
  t=9



Nodo	Start	End	OT
0	9	0	3
1	8	0	2
2	1	6	4
3	0	7	5
4	2	5	
5	3	4	

- El nodo 0 no tiene hijos con start=0.
- Lo agregamos al OT
- Seteamos su End en t=10

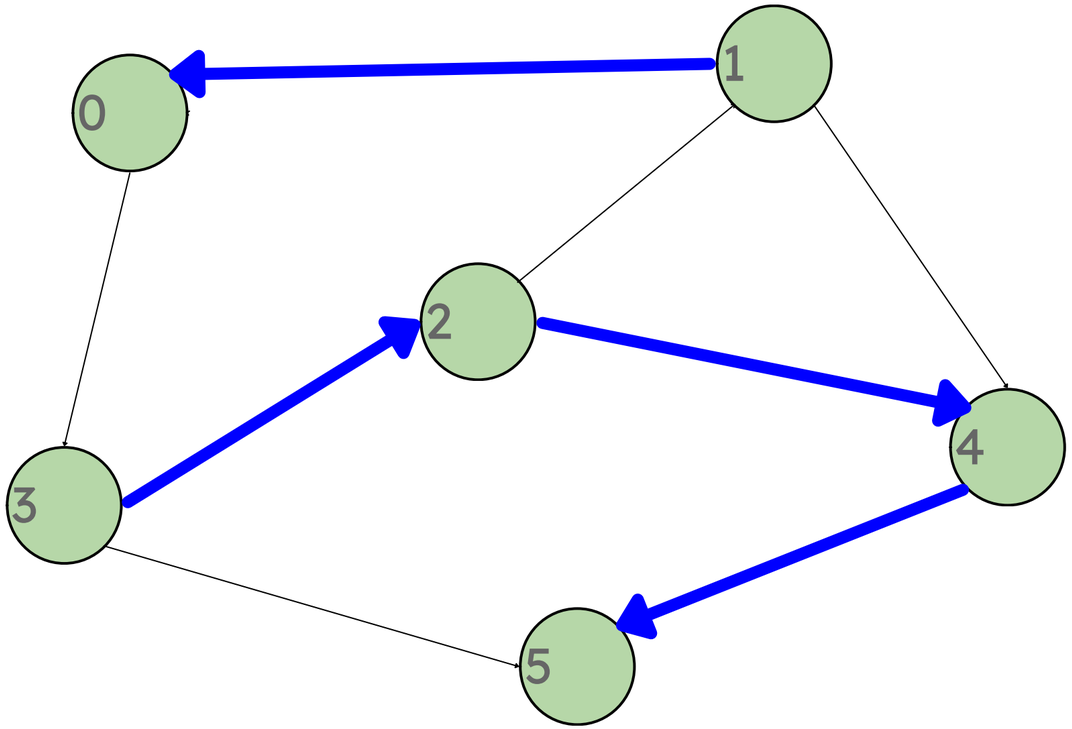
t=10



Nodo	Start	End	OT
0	9	10	0
1	8	0	3
2	1	6	2
3	0	7	4
4	2	5	5
5	3	4	

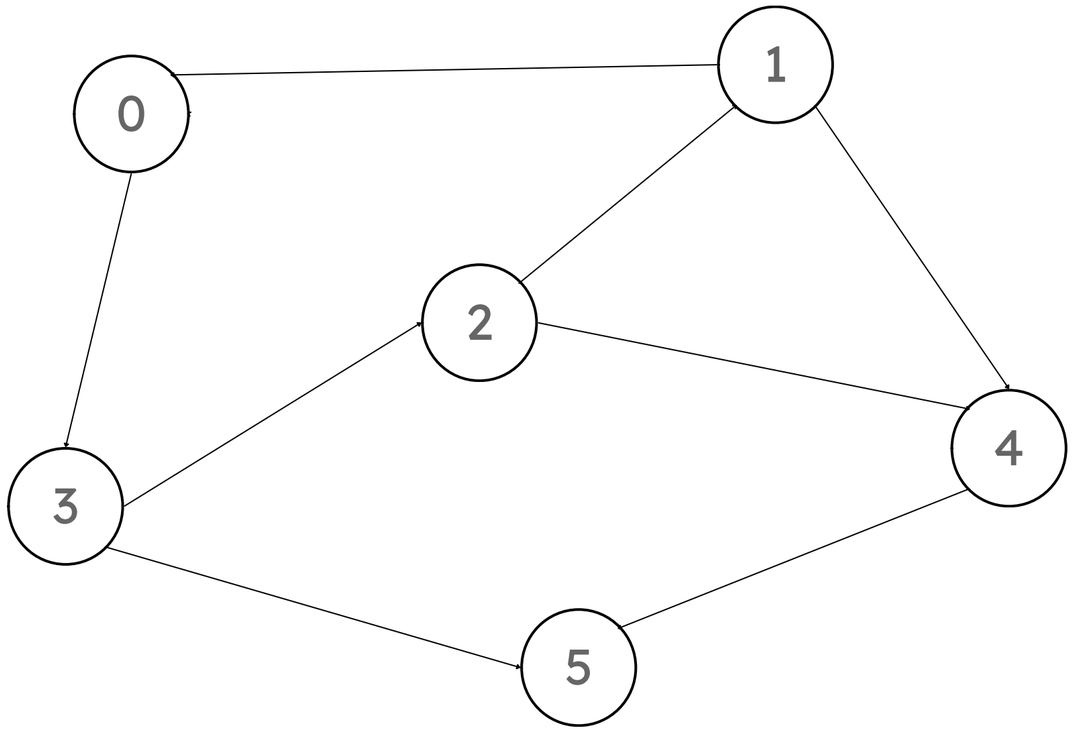
- Volvemos al nodo 1 y no tiene más hijos con start= 0.
- Lo agregamos al OT
- Seteamos su End en t=11

t=11



Nodo	Start	End	OT
0	9	10	1
1	8	11	0
2	1	6	3
3	0	7	2
4	2	5	4
5	3	4	5

Finalmente, el orden topológico del grafo es: (1)(0)(3)(2)(4)(5).



- En resumen, orden en que vamos “descubriendo los nodos”
- Los intervalos de tiempo de los nodos no se traslapan entre ellos, por construcción

Los grafos cíclicos no tienen orden topológico ¿Por qué?

Contexto

a) En clases estudiamos una versión recursiva del recorrido DFS de un grafo. Proponga el pseudocódigo de un algoritmo iterativo que implemente DFS en un grafo dirigido G = (V, E), que opere en tiempo O(V + E). Indique precisamente qué estructura(s) de datos necesita para su solución



Contexto

Se presenta una modificación del algoritmo visto en clases

```
DFS-Visit(s):
    for u ∈ V − {s} :
        u.color ← blanco;  u.δ ← ∞;  π[u] ← ∅
    s.color ← gris;  s.δ ← 0;  π[s] ← ∅
    Q ← stack LIFO vacío
    Insert(Q, s)
    while Q no está vacía :
        u ← Extract(Q)
        for v ∈ α[u] :
            if v.color = blanco :
                v.color ← gris;  v.δ ← u.δ + 1
                π[v] ← u
                Insert(Q, v)
        u.color ← negro
```

b) Un algoritmo alternativo para determinar un orden topológico en un grafo dirigido G = (V, E) es extraer un nodo sin aristas que apunten a él, eliminar todas sus aristas y repetir el proceso hasta que no queden nodos

(i) Proponga el pseudocódigo de este algoritmo de forma que en tiempo O(V + E) entregue un orden topológico para un grafo acíclico. Especifique qué estructura(s) de datos necesita.

(ii) Explique cómo este algoritmo puede detectar un ciclo en G. No requiere dar pseudocódigo

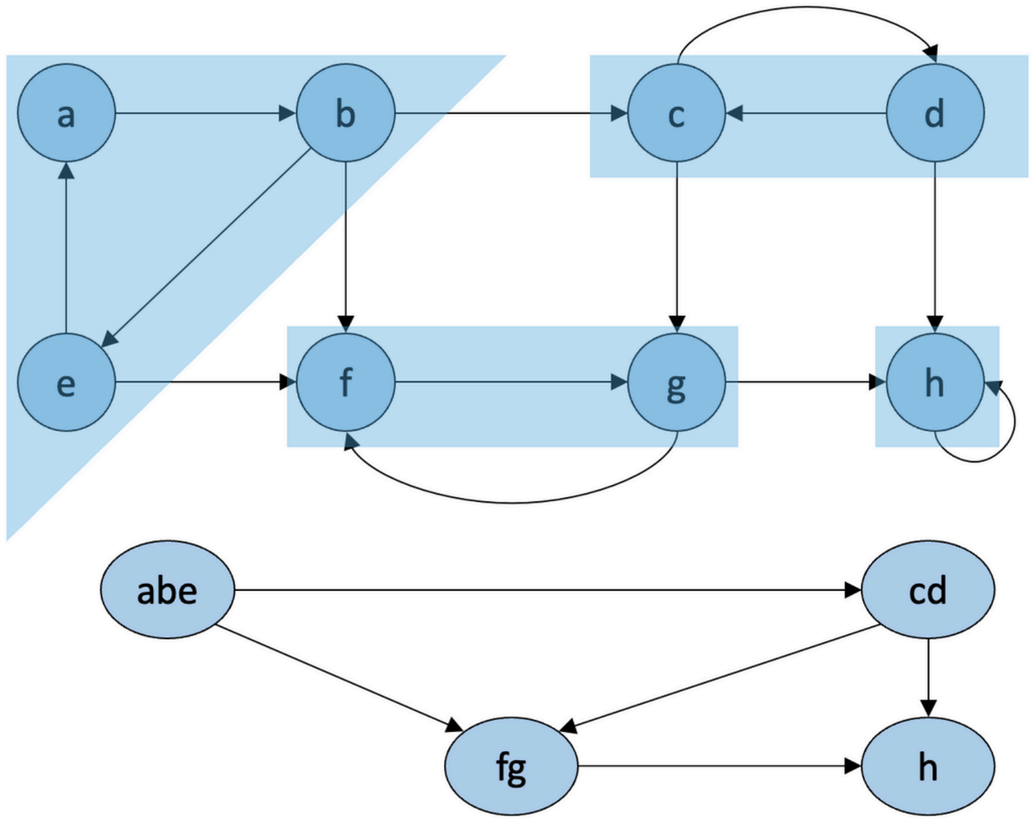
(i) Proponga el pseudocódigo de este algoritmo de forma que en tiempo O(V + E) entregue un orden topológico para un grafo acíclico. Especifique qué estructura(s) de datos necesita.

```
Algoritmo(V, E):
    L ← lista ligada vacía
    while V ≠ ∅ :
        u ← Extract(Q)
        v ← nodo de V sin aristas incidentes Extraer v de V
        for e ∈ E :
            if v es mencionado en e :
                Extraer arista e de E
        Insertar v en L
    return L
```

(ii) Explique cómo este algoritmo puede detectar un ciclo en G. No requiere dar pseudocódigo

Si L no contiene todos los nodos de V, entonces G tiene un ciclo dirigido





El nuevo grafo de componentes es acíclico, ¿Por qué?

Si tuviera ciclos, los nodos de ambas CFC estarían en realidad en una misma CFC

Éxito en la I2 :3



