Grafos dirigidos, DAG y orden topológico

Colectivo Estructuras de Datos y Algoritmos

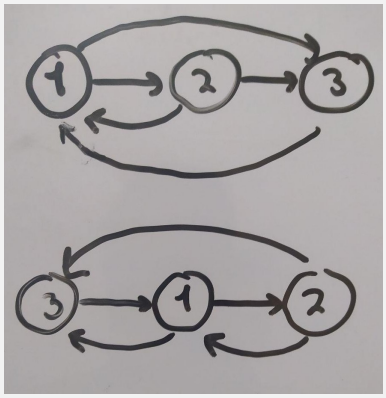
Octubre 2020

1. Responda V o F. Justifique en cada caso:

   (a) Si un grafo dirigido G contiene ciclos, el algoritmo $TOPOLOGICAL\_SORT(G)$ visto en conferencia produce una ordenación de los vértices de G que minimiza de el número de arcos "hacia atrás".

   Respuesta

   (F). Contrajemplo. En la imagen se muestran dos órdenes topológicos generados por distintos recorridos DFS sobre el mismo grafo. Ambos tienen distinta cantidad de arcos "hacia atrás".

   Como el algoritmo no dice en qué orden se recorren los arcos en el DFS, entonces puede pasar que para el grafo mostrado, el algoritmo genere la respuesta que aparece debajo.

   (b) En un DAG existe al menos un vértice con *indegree* igual a 0 y uno con *outdegree* igual a 0.

   Respuesta

   (V) Sea G un DAG, y sea $C = \{u, ..., v\}$ el camino de mayor longitud en G (no aclaro camino simple, porque todos los caminos en G son simples). Vamos a demostrar que $indegree(u) = outdegree(v) = 0$.

   Si $indegree(u) \neq 0$ entonces existe w tal que también existe el arco $\langle w, u\rangle$. Noten que $w \notin C$, ya que si perteneciera, existiría un ciclo en G. Como $w \notin C$, entonces C no es el camino más largo en G. Esto genera una contradicción, luego $indegree(u) = 0$.

   Análogamente se demuestra que $outdegree(v) = 0$. ■

   (c) La mayor cantidad de arcos que puede tener un DAG con n vértices es $\theta(n^2)$.

Respuesta

Para enfrentar este ejercicio pensemos en una manera de construir un DAG con la mayor cantidad de arcos posible. Como vamos a demostrar en el ejercicio 2, un grafo dirigido es un DAG si y solo si tiene un orden topológico. Entonces, pensemos como disponer arcos en un DAG de n vértices sin violar la existencia de un orden topológico. Si ordenamos arbitrariamente los n vértices entonces una buena idea sería $\forall i, j$ con $i < j$ colocar un arco $\langle v_i, v_j \rangle$, donde i es la posición (comenzando por 0) de v_i en el ordenamiento que definimos.

De esta forma el ordenamiento $v_1, v_2, ..., v_n$ es un orden topológico de los vértices ya que no tiene arcos "hacia atrás". Definido de esta forma, el número de arcos de dicho grafo es:

$$|E| = \sum_{i=1}^{n} i - 1 = \frac{n(n-1)}{2}$$

Demostremos que este número es máximo. Hagamóslo por inducción en n. Formalmente queremos demostrar que el número máximo de arcos en un grafo dirigido y acíclico $G = \langle V, E \rangle$ donde $|V| = n$, es $\frac{n(n-1)}{2}$.

Caso base: $n = 1$ Solo existe un DAG de un solo vértice y es aquel que no tiene arcos. Luego 0 es la máxima cantidad de arcos que puede tener. En este caso:

$$|E| = 0 = \frac{1(1-1)}{2}$$

Paso inductivo $P(n-1) \Rightarrow P(n)$. En este caso $P(n-1)$ es: *El número máximo de arcos en un DAG de $n-1$ vértices es* $\frac{(n-1)(n-2)}{2}$.

Sea $G = \langle V, E \rangle$ un DAG de n vértices con la mayor cantidad de arcos posible. Como se demostró en el ejercicio anterior, en un DAG siempre existe al menos un vértice con *outdegree* 0. Luego, sea v un vértice de G tal que $outdegree(v) = 0$.

Noten que $indegree(v) = n - 1$ porque como la cantidad de arcos en G es máxima, si $indegree(v)$ no fuera $n-1$, entonces se pudiera añadir un arco desde algún vértice distinto de v hacia v, lo cual no generaría un ciclo porque $outdegree(v) = 0$. Luego la cantidad de arcos en G no sería máxima.

Si quitamos este vértice de G y todos los arcos que inciden en él $(n-1)$, se crea un grafo G' con $n-1$ vértices, que es un DAG (porque si no existían ciclos en G, quitarle vértices y arcos no puede crear un ciclo) y además la cantidad de arcos en G' es máxima (porque si no lo fuera entonces no lo sería tampoco en G).

Entonces, por hipótesis de inducción:

$$|E'| = \frac{(n-1)(n-2)}{2}$$

y

$$|E| = |E'| + n - 1 = \frac{(n-1)(n-2)}{2} + n - 1 = \frac{n(n-1)}{2} \quad \blacksquare$$

Demostremos que $f(n) = \frac{n(n-1)}{2}$ es $\theta(n^2)$. Noten que $\forall n \geq 2$:

$$\frac{n^2}{4} \leq f(n) \leq n^2$$

Luego, sea $n_0 = 2, c_1 = \frac{1}{4}$ y $c_2 = 1$, $\forall n \geq n_0$ se cumple que $c_1 n^2 \leq f(n) \leq c_2 n^2$. Por lo que $f(n) \in \theta(n^2)$.

2. Demuestre que un grafo dirigido es un DAG si y solo si existe un orden topológico de sus vértices.

Respuesta

($\Rightarrow$) Esta implicación se desprende directo del hecho de que existe un algoritmo (visto en conferencia) que computa correctamente un orden topológico dado un DAG.

($\Leftarrow$) Asumamos que G tiene un orden topológico y no es acíclico. Sea C un ciclo en G y $T = [v_1, v_2, ...v_n]$ un orden topológico de sus vértices. Demostremos que en C existe un arco "hacia atrás" en T.

Sea $e = \langle v_i, v_j \rangle$ un arco de C, donde i, j son los índices de dichos vértices en T. Si $i > j$ (v_i está a la derecha de v_j en T) entonces e es "hacia atrás" y termina la demostración. Asumamos entonces que $i < j$ (v_j está a la derecha de v_i en T). Sabemos además que existe un camino C' formado por arcos de C que va desde v_j hasta v_i, porque C es un ciclo. Entonces, en C' tiene que existir un arco "hacia atrás" porque si todos fueran "hacia adelante", se cumpliría que $i > j$, lo cual contradice lo asumido[a]. ■

[a] Esta explicación no es formal. Una manera de formalizar este razonamiento es utilizando inducción sobre la longitud de C'.

3. Diseñe un algoritmo que dado un grafo dirigido $G = \langle V, E \rangle$ y una lista T que constituye una permutación de sus vértices, permita determinar, en $O(|V| + |E|)$ si T es un orden topológico de G.

Respuesta

En la conferencia hay propuesto un algoritmo para la solución de este ejercicio. El algoritmo es simple, y fíjense que lo que haces es verificar que no existan arcos "hacia atrás" en T.

Demostremos que es correcto. Noten que esto es un problema de decisión, luego para probar que es correcto necesitamos demostrar que:

(a) Si T es un orden topológico, el algoritmo devuelve $True$

(b) Si T no es un orden topológico, el algoritmo devuelve $False$

Si T es un orden topológico, entonces no tiene arcos "hacia atrás", luego en un vértice cualquiera de T, cuando ya se han eliminado todos lo vértices a su izquierda, tiene que tener *indegree* igual a 0. Luego, el algoritmo devuelve $True$.

En el otro sentido, si T no es un orden topológico entonces existe un arco hacia atrás $e = \langle u, v \rangle$ en T. Cuando se llega al vértice v en el algoritmo, todavía no se va a haber eliminado el vértice u de G, por lo que el $indegree(v)$ va a ser al menos 1. Luego el algoritmo devuelve $False$ en este caso. ■

4. Demuestre que siendo G un DAG, el siguiente algoritmo devuelve un orden topológico de los vértices de G.

```
1: T ← lista vacía
2: while G tenga vértices do
3:     v ← vértice con indegree igual a 0 en G
4:     quitar v y todos sus arcos de G
5:     T.append(v)
6: end while
7: return T
```

Respuesta

El algoritmo esta constituido por un ciclo (línea 2) que de completarse itera $|V|$ veces, y en cada iteración busca un vértice con *indegree* igual a 0, lo elimina del grafo junto a todos sus arcos, y lo agrega como próximo vértice en la lista que finalmente devuelve. Para demostrar que la lista que el algoritmo computa define un orden topológico de G debemos probar dos cosas:

(a) Que el algoritmo termina (no es trivial que siempre existe en el grafo restante un vértice con *indegree* igual a 0).

(b) Que cuando termina, la lista T constituye un orden topológico de los vértices del grafo de entrada.

Para la primera parte solo necesitamos demostrar que en cada iteración del ciclo del algoritmo, existe un vértice con *indegree* igual a 0 en el grafo que va resultando. Inicialmente el grafo G dado es un DAG, por lo que, según demostramos en el ejercicio 1b, tiene un vértice con *indegree* igual a 0. Además, eliminar vértices y arcos de un grafo no puede generar ciclos, de lo que en cualquier iteración, como el grafo restante es el resultado de haber eliminado vértices y sus arcos de G, siempre será un DAG. Luego, siempre tiene al menos un vértice con *indegree* 0.

Para la segunda parte, denotemos como v_i el i-ésimo elemento de la lista T, el cual es además el vértice que se seleccionó en la iteración i del ciclo del algoritmo. Demostremos que no existe en G ningún arco $\langle v_i, v_j \rangle$ tal que $i > j$. Asumamos que existe. Noten que para todo arco $\langle v_i, v_j \rangle$, para que *indegree* de v_j llegue a 0, se necesita eliminar primero a v_i del grafo. Luego, el algoritmo necesariamente elimina a v_i antes que a v_j de G; de lo que $i < j$. Y esto contradice lo asumido, por lo que no existen arcos "hacia atrás" en G.

5. Diseñe un algoritmo que dado un grafo dirigido $G = \langle V, E \rangle$ permita determinar, en $O(|V| + |E|)$ si G es o no un árbol. Se dice que un grafo dirigido es un árbol si su grafo subyacente es un árbol libre y existe en el grafo un nodo (raíz) desde el cual se alcanza al resto de los vértices.

Respuesta

La definición de árbol dada en el ejercicio para grafos dirigidos tiene dos elementos que hay que garantizar:

(a) Que el grafo subyacente sea un árbol libre.

(b) Que existe en el grafo un nodo desde el cual se alcanzan el resto de los vértices.

Para garantizar la primera parte basta construir el grafo subyacente y determinar si es un árbol libre con algunos de los algoritmos vistos para ello anteriormente en el curso.

Ahora, ¿cómo garantizamos la segunda parte?

Noten que para ser un árbol, el grafo dirigido no puede tener ciclos(porque si los tuviera entonces el grafo subyacente no sería un árbol libre). Entonces, de acuerdo al ejercicio 1b, un grafo dirigido que no contiene ciclos tiene que tener al menos un vértice con *indegree* igual a 0. Si hacemos un DFS_VISIT a partir de uno de esos vértices, digamos v, y se alcanza el resto de los vértices del grafo, se cumple la condición. A su vez, si esto no ocurre y quedan vértices por visitar, entonces v no alcanza a ningún vértice de los que restan, además ningún vértice alcanza a v porque $indegree(v) = 0$, luego no existiría ningún vértice desde el cual se alcance al resto.

El razonamiento anterior implica que en un grafo dirigido G cuyo grafo subyacente sea un árbol, hacer DFS_VISIT desde un vértice v de $indegree(v) = 0$ y que este visite todos los vértices de G, es condición necesaria y suficiente para que G sea un árbol dirigido.

Entonces, el pseudocódigo de esta algoritmo queda de la siguiente forma:

```
1: G' ← grafo subyacente de G
2: if G' es árbol then
3:     v ← vértice con indegree 0 en G
4:     visited ← DFS_VISIT(G, v)
5:     return len(visited) == |V|
6: else
7:     return false
8: end if
```

Garantizar la primera parte es $O(|V|+|E|)$, encontrar un vértice v con $indegree(v) = 0$ es $O(|V|+|E|)$ y hacer un DFS sobre el grafo dirigido es $O(|V|+|E|)$. Luego, la complejidad temporal del algoritmo es $O(|V|+|E|)$.

Como ejercicio adicional, piensen cómo la definición de árbol dirigido es equivalente a que G tenga uno y solo un vértice con *indegree* igual a 0. Lo cual deriva en un algoritmo mucho más sencillo (igual de eficiente), para determinar si G es un árbol dirigido.

6. Se dice que un grafo dirigido $G = \langle V, E\rangle$ es **semiconexo** si para todo par de vértices $u, v \in V$ se cumple que en G existe un camino de u a v y no de v a u o existe un camino de v a u y no de u a v. Implemente un algoritmo que permita determinar en $O(|V|+|E|)$ si un grafo $G = \langle V, E\rangle$ dado es semiconexo.

Respuesta

Vamos a comenzar con una línea de razonamiento que pudieran haber seguido para dar con la solución del ejercicio.

Lo primero que se deben dar cuenta es que un grafo semiconexo tiene que ser un DAG. Porque si el grafo tuviera un ciclo, entonces existirían u, v tales que existe camino de u a v y viceversa, lo cual contradice la definición de semiconexo.

Como G es un DAG, existe un vértice u tal que $indegree(u) = 0$. Pero además, noten que si se quita dicho vértice de G creando el grafo G', entonces G' también es semiconexo. ¿Por qué? Noten que $\forall x, y$ vértices de G', tal que en G existía un camino de x a y, ese camino sigue existiendo en al quitar u, ya que u no podía ser vértice interior de ningún camino en G porque $indegree(u) = 0$. Además, como G es semiconexo, no existe en G camino de y a x, y quitar u no crea nuevos caminos. Luego en G' tampoco existe camino de y a x.

Como G' es semiconexo y por tanto DAG, entonces existe v con $indegree(v) = 0$ en G'. Pero además, en G tenía que haber un camino de u a v ya que no podía ser de v a u porque $indegree(u) = 0$ en G. Sabemos además que cuando se quita u de G, el *indegree* de v pasa a 0. Esto significa que la única forma de llegar a v era a través de u. Tenemos entonces que:

(a) En G existe un camino de u a v

(b) A v solo se puede llegar a través de u

Esto implica que en G existe el arco $\langle u, v\rangle$.

Si repetimos este proceso iterativamente, y se siguen quitando los vértices en este orden, de acuerdo al algoritmo visto en el ejercicio 4, se genera un orden topológico $v_1, v_2, ..., v_n$, ya que siempre estamos quitando vértices con *indegree* igual a 0. Donde además, sabemos que $\forall i < n$ existe el arco $\langle v_i, v_{i+1}\rangle$.

Fíjense adicionalmente, que la selección del próximo vértice con *indegree* 0 siempre es arbitraria, o sea, nuestro razonamiento no depende de qué vértice se escoja, y eso "puede significar" que se tiene que

cumplir para cualquier orden topológico[a].

En este punto del razonamiento podemos presumir que la condición a la que llegamos no es solo necesaria, sino también suficiente para que G sea semiconexo.

Recuerden que hasta aquí estamos tratando que razonar el problema, nada de lo que hemos dicho hasta ahora demuestra nada y esto es un proceso de debe ocurrir en sus cabezas, no en el papel. Lo que se supone va en el papel es lo que viene adelante.

Formalmente queremos demostrar que:

Un grafo dirigido G es semiconexo si y solo si G es un DAG y para cualquier orden topológico $v_1, v_2, ..., v_n$ de G, $\forall i < n$ se cumple que existe en G el arco $\langle v_i, v_{i+1} \rangle$.

Empecemos por la implicación más fácil.

($\Leftarrow$) Partimos de que G es un DAG. Sea $v_1, v_2, ..., v_n$ un orden topológico cualquiera de G. Sabemos que $\forall i < n$ se cumple que existe en G el arco $\langle v_i, v_{i+1} \rangle$.

Para todo par de vértices v_i, v_j de G con $i < j$ se cumple que existe el camino $v_i, v_{i+1}, ..., v_{j-1}, v_j$, y además no existe camino de v_j a v_i porque de existir, el grafo no sería acíclico. Luego G es semiconexo.

Ahora la buena, que si se fijan el razonamiento seguía la dirección de esta implicación.

($\Rightarrow$) Primeramente, como ya dijimos, si G es semiconexo no tiene ciclos, por lo que es un DAG.

Ahora, sea $v_1, v_2, ..., v_n$ un orden topológico cualquiera de G. Noten que $\forall i < n$, como G es semiconexo se cumple que existe un camino particularmente de v_i a v_{i+1}. Pero dicho camino no puede pasar por ningún vértice v_j con $j > i+1$ o $j < i$, porque en cualquiera de los dos casos el orden topológico tendría un arco "hacia atrás", lo que generaría una contradicción. Entonces ese camino solo puede ser el arco $\langle v_i, v_{i+1} \rangle$. ■

[a] Digo que "puede significar" porque en ningún momento se demostró en el ejercicio 4 que con distintas selecciones de los vértices de *indegree* igual a 0 se obtienen todos los órdenes topológicos posibles.

7. Sea $G = \langle V, E \rangle$ un grafo dirigido y acíclico y sea k la longitud del camino más largo en G. Diseñe un algoritmo que particione el conjunto de vértices en a lo sumo $k + 1$ grupos de modo que para todo par de vértices u, v con $v \neq u$ pertenecientes a un mismo grupo, no exista camino entre u y v y tampoco exista camino entre v y u. La complejidad temporal de su algoritmo debe ser $O(|V| + |E|)$.

Respuesta

En este ejercicio hay varias líneas de pensamiento que se pueden seguir que son incorrectas, pero, para no hacer esto excesivamente largo, vamos directo a un razonamiento que lleva a una solución.

Como ustedes saben desde primer año, una forma de abordar los problemas es iterativamente. En EDA, y de manera general, esto significa que una idea para enfrentar un problema, es pensar en alguna propiedad que tenga dicho problema inicialmente, extraer a partir de esa propiedad parte de la solución del problema y repetir este proceso garantizando que se mantenga invariante dicha propiedad o propiedades. Si se fijan esto es lo que hace el algoritmo visto en el ejercicio 4 para encontrar un orden topológico.

Si abordan este ejercicio pensando de esta forma, inicialmente en el grafo G ¿qué vértices podemos garantizar que pueden estar en el mismo grupo? Hay varias respuestas a esta pregunta, pero una de ellas es definitivamente: "aquellos que tienen *indegree* 0" (que ya sabemos que existen en G porque es un DAG). Si dos vértices u, v tienen *indegree* igual a 0, no puede existir un camino de u a v ni viceversa.

Vamos a enumerar los grupos de 1 a $k+1$. Podemos asignar entonces a todos los vértices con *indegree* 0 en G el grupo 1. Ahora, pensemos que sucede si quitamos de G todos estos vértices y sus arcos. Definitivamente el grafo que resulta sigue siendo un DAG.

Ahora, ¿podemos volver a repetir lo que hicimos? En este nuevo grafo también existen vértices con *indegree* igual a 0, y no existen caminos entre ellos. Entonces podemos poner a todos en el mismo grupo. Pero, ¿puede ser en el grupo 1? No, porque para todo vértice v que tiene *indegree* 0 en este punto, existe un vértice u en el grupo 1 tal que en G existe el arco $\langle u, v\rangle$.

Repetir este proceso, asignando en cada iteración un grupo nuevo al conjunto de vértices con *indegree* 0, hasta que se hayan agrupado todos los vértices del grafo, genera una asignación de grupos que cumple con los requisitos del problema.

El pseudocódigo de este algoritmo luce como sigue.

```
1: grupos ← lista de tamaño |V|
2: grupo_actual ← 1
3: while G tenga vértices do
4:    lista_indegree_cero ← todo v con indegree(v) = 0 en G
5:    for v in lista_indegree_cero do
6:       grupos[v] ← grupo_actual
7:       quitar v y todos sus arcos de G
8:    end for
9:    grupo_actual + +
10: end while
11: return grupos
```

Demostremos que este algoritmo es correcto. Dividamos la demostración en dos partes:

(a) La asignación en grupos que se crea es factible[a].

(b) Se crean a lo sumo $k+1$ grupos.

En el caso de la primera parte, demostremos que para cualquier par de vértices u, v tal que exista un camino de u a v, el algoritmo asigna a u y a v grupos diferentes.

Noten que el algoritmo le asigna a un vértice x un grupo una vez que su *indegree* alcanza el valor 0. Si existe un camino de u a v, no puede pasar que ambos vértices alcancen *indegree* 0 en la misma iteración, puesto que para que v tenga *indegree* 0, al menos es necesario eliminar a u del grafo. Luego, como los dos alcanzan *indegree* 0 en iteraciones diferentes, el algoritmo le asigna grupos diferentes.

Para explicar la segunda parte vamos a utilizar la propiedad de G que nos dice que el camino más largo tiene longitud k. Noten que un vértice v que se procesa en la iteración i alcanzó *indegree* 0 en la iteración actual y no en la anterior. Esto implica que tiene que existir un vértice u procesado en la iteración $i-1$ tal que en G exista el arco $\langle u, v\rangle$ (de tal forma que cuando se quitó u de G es que v llegó a *indegree* 0). Formalmente, si las iteraciones van de 1 a n, para toda iteración $i > 0$,siendo v_i un vértice analizado en dicha iteración, existe un vértice v_{i-1} analizado en la iteración $i-1$, tal que existe en G el arco $\langle v_{i-1}, v_i\rangle$.

Luego, si la última iteración de G es $k+1$, entonces existe en G un camino $v_1, v_2, ..., v_{k+1}$ de longitud k. Como el camino más largo en G tiene longitud k, el ciclo exterior no se puede ejecutar más de $k+1$ veces. De lo que el algoritmo solo asigna vértices en a lo sumo $k+1$ grupos diferentes. ■

Adicionalmente, piensen como resolver este ejercicio usando la técnica de programación dinámica en DAGs, que se va a explicar a continuación.

[a] Una asignación factible en este caso es aquella donde los grupos que se crean cumplen con las restricciones del problema

Programación Dinámica en DAG

Muchos son los problemas sobre grafos en los que computando una función por cada vértice del mismo se obtiene la solución. Un subcojunto de estas funciones, que atrapan gran parte de los problemas a resolver sobre grafos, se calculan exclusivamente en función de los vértices que inciden en un vértice en cuestión. Esto es, dado un grafo G y un vértice v de G:

$$F(v) = \begin{cases} Op(X) & \text{si } indegree(v) > 0 \\ VALOR\_BASE & \text{en otro caso} \end{cases}$$

donde X es la secuencia $\{F(w) \mid \langle w, v\rangle \in E(G)$.

Si el grafo G es un DAG, existe un algoritmo genérico que nos permite computar una función F con estas propiedades. Fíjense que la única condición necesaria para computar iterativamente la función F sobre los vértices de G, es que cuando se visite un vértice v, los valores de $F(w_i)$ ya se hayan calculado correctamente $\forall w_i \mid \langle w_i, v\rangle \in E(G)$. Esto lo podemos lograr exactamente recorriendo los vértices en un orden topológico de G.

Un pseudocódigo genérico quedaría de la siguiente forma:

```
1: ts ← TopologicalSort(G)
2: f[v] ← VALOR_BASE  ∀v ∈ V(G)
3: for i = 0 to n − 1 do
4:     v ← ts[i]
5:     f[v] ← F(v)
6: end for
7: return  f
```

Demostremos que este algoritmo computa correctamente el valor de la función F para todos los vértices de G. Para ello solo necesitamos probar que $\forall v \in V(G)$ se cumple que $f[v]$ contiene el valor de $F(v)$. Hagámoslo por inducción fuerte en el índice i de los vértices en el orden topológico (que es a su vez el índice del ciclo de la línea 4). Sea v_i el vértice que está en la posición i-ésima del orden topológico.

Caso base: $i = 0$

Sabemos, por propiedad del orden topológico que $indegree(v_0) = 0$. Luego, por definición de la función F, $f[v_0] = VALOR\_BASE$.

Paso inductivo: $P(j)_{j<i} \Rightarrow P(i)$

La hipótesis de inducción en este caso establece que $f[v_j] = F(v_j)$ para todo $j < i$. Por definición, F solo depende del valor de $F(w)$ para aquellos $w \mid \langle w, v\rangle \in E(G)$. Y, por propiedad del orden topológico, para cualquiera de esos vértices w, se cumple que su índice j en el orden topológico es menor que i. Luego, en el momento de calcular $f[v_i]$, por hipótesis de inducción $f[w] = F(w)$ para todo $w \mid \langle w, v\rangle \in E(G)$. Por lo tanto, por definición de $F(v)$, esta se calcula correctamente y se almacena en $f[v]$. ■

Cuando se enfrenten un problema en específico, fíjense que solo necesitan sustituir la línea 5 por un código que compute correctamente la función F en el vértice v y lo guarde en $f[v]$. Y a la hora de explicar un ejercicio de este tipo, solo se hace necesario definir $F(v)$ correctamente en función de los vértices que inciden en v.

Esta forma de calcular funciones sobre los nodos de un DAG utilizando el orden topológico para validar que el momento de calcular el valor de $F(v)$ toda la información necesaria haya sido calculada previamente, se conoce como **Dinámica en DAGs**. Disímiles son las formas en las que se pueden llevar a cabo, pero esta es una de las más completas y el resto de los ejercicios propuestos en esta clase

práctica van a usar este modelo de solución.

Asumiendo que el costo de evaluar $F(v)$ sea $O(indegree(v))$ y aplicando la definición de O, el costo de computar los valores de la función para cada vértice es $\sum_{v_i \in V(G)} O(indegree(v_i))$ y cumple lo siguiente:

$$\sum_{v_i \in V(G)} O(indegree(v_i)) \leq \sum_{v_i \in V(G)} c_i * indegree(v_i)$$

Luego, para $C_{max} = max(c_i)$:

$$\sum_{v_i \in V(G)} O(indegree(v_i)) \leq C_{max} \sum_{v_i \in V(G)} indegree(v_i) = C_{max} * |E|$$

$$\Rightarrow \sum_{v_i \in V(G)} O(indegree(v_i)) \in O(|E|)$$

Por lo que revisar todos los vértices del grafo y guardar el valor de la función F más calcular todos los valores de la función F es $O(|V| + |E|)$ por el principio de la suma.

8. Dado un grafo dirigido y acíclico $G = \langle V, E \rangle$ y un vértice $s \in V$, diseñe un algoritmo que determine $\forall u \in V$ la longitud del camino de longitud máxima que va de s a u. La complejidad temporal de su algoritmo debe ser $O(|V| + |E|)$.

   (a) ¿Cómo se puede modificar su algoritmo para encontrar la longitud del camino de longitud máxima que va de u a s, para todo vértice u?

Respuesta

Para resolver este problema primero se define computacionalmente la función que para cada vértice computa la longitud del camino de longitud máxima que va de s a todos los vértices del grafo.

$$F(v) = \begin{cases} 0 & \text{si } v = s \\ -\infty & \text{si } indegree(v) = 0 \\ 1 + \max_{w_i \mid \langle w_i, v \rangle \in E} F(w_i) & \text{en otro caso} \end{cases}$$

Esta función se extrae directamente de la definición del problema:

- El camino de longitud máxima que va de s a s es $\langle s \rangle$, debido a que el grafo es acíclico.
- Si no existe un camino entre s y v la longitud de dicho camino es $-\infty$.
- El camino de longitud máxima entre s y v tiene que pasar obligatoriamente por un adyacente a v por tanto:

  $$\max(1 + longitud(s, w_i)) = 1 + \max(longitud(s, w_i)) = 1 + \max(F(w_i))$$

  . Donde $\langle w_i, v \rangle \in E$

Luego se resuelve como una **Dinámica en DAG** dado que $F(u)$ solo depende de sus adyacentes para computar su valor. El siguiente pseudocódigo brinda solución al problema, que con el mismo razonamiento realizado en la respuesta 7 tiene complejidad temporal $O(V + E)$

```
1: ts ← TopologicalSort(G)
2: f[v] ← −∞  ∀v ∈ V(G)
3: f[s] = 0
4: for i = 0 to n − 1 do
5:    v ← ts[i]
6:    for w_i tal que ⟨w_i, v⟩ ∈ E do
7:       if f[w_i] ≥ 0 then
8:          f[v] ← max(f[v], 1 + f[w_i])
9:       end if
10:   end for
11: end for
12: return  f
```

a-) Este problema se resuelve reduciéndolo al problema anterior, usando como entrada G^T en vez de G; ya que existe una biyección entre todos los caminos que empiezan en v en G y los que terminan en v en G^T y ambos tienen la misma longitud, por tanto se puede buscar el máximo en el conjunto de los caminos que empiezan en s en G^T.