Examen #4

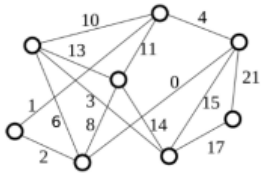
Matemáticas Discretas

07/Diciembre/2021

Acosta Imandt Daniel
316002228

Problema 1

(2.5 pts) Ejecuta manualmente el algoritmo de Prim y el algoritmo de Kruskal en la siguiente gráfica con pesos:



Solución

Lo primero que vamos a hacer será ponerle nombres a los vertices para que la explicación sea más sencilla

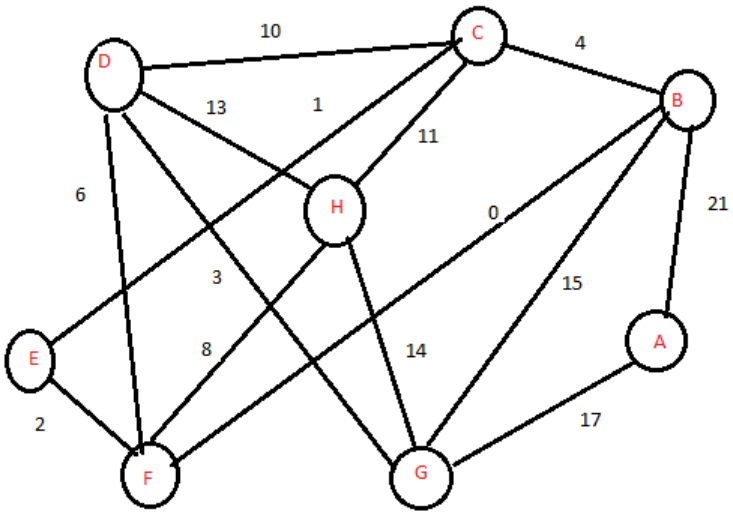


Figura 1: Gráfica con nombres

Prim

Lo primero que vamos a hacer será tomar la arista que tiene peso 0 (B,F) ,ya que es la que tiene menor peso, de allí vemos que la siguiente que nos conviene es la que pesa menos de las aristas con vértices en nuestro árbol, por lo que tomamos la que pesa 2(F,G), de allí pasamos a la que pesa 1(E,C), notamos que ya no tenemos que tomar la que pesa 4(C,B), ya que se genera un ciclo , así que tomamos la de peso 6 (D,F) y pasamos a la de 3(D,G),seguida de la de 8(F,H),por ultimo tomamos la que pesa 17(G,A) y notamos que ya terminamos ya que hemos conectado a todos los vértices de la gráfica con

el peso mínimo el cual es $0+2+1+6+3+8+17=37$.

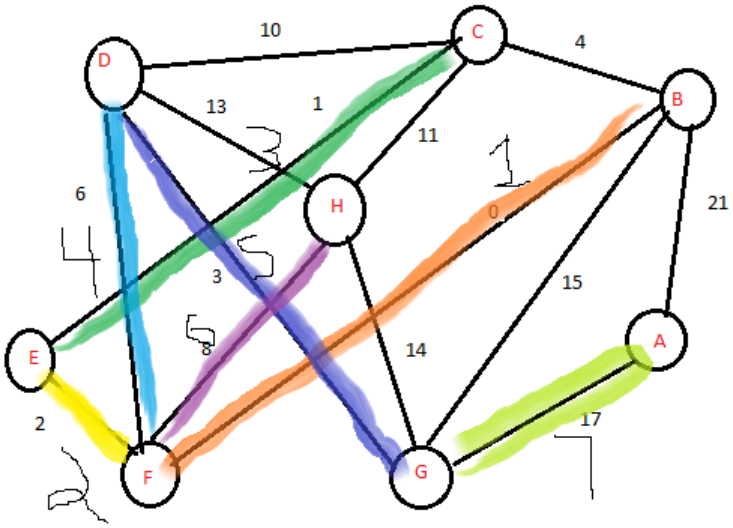


Figura 2: Orden de como tomamos las aristas con Prim

Kruskal
Recordando que lo que se debe de hacer es ordenar las aristas de menor a mayor de acuerdo a su peso.Luego vamos agregando aristas en orden procurando que no se hagan ciclos y por ultimo terminamos cuando se hayan agregado $n-1$ aristas.
Por lo que ordenando las aristas tenemos lo siguiente

$$[0,1,2,3,4,6,8,10,11,13,14,15,17,21]$$

Por lo que agarramos la arista (B,F) con peso 0, después (E,C) con peso 1, proseguimos con (F,E) con peso de 2, seguido de (D,G) de 3, y seguimos con (D,F) de peso de 6,vemos que no debemos de tomar la arista (C,B) ya que se crea un ciclo,proseguimos con el (F,H) de 8,ahora se crean ciclos para las aristas de (C,H),(H,G),(D,H),(C,D).Por ultimo tomamos la arista(G,A) de peso 17.Aquí notamos que ya tenemos 7 aristas que es una menos de los vértices por lo que hemos acabado y el peso total es de $1+2+3+6+8+17=37$

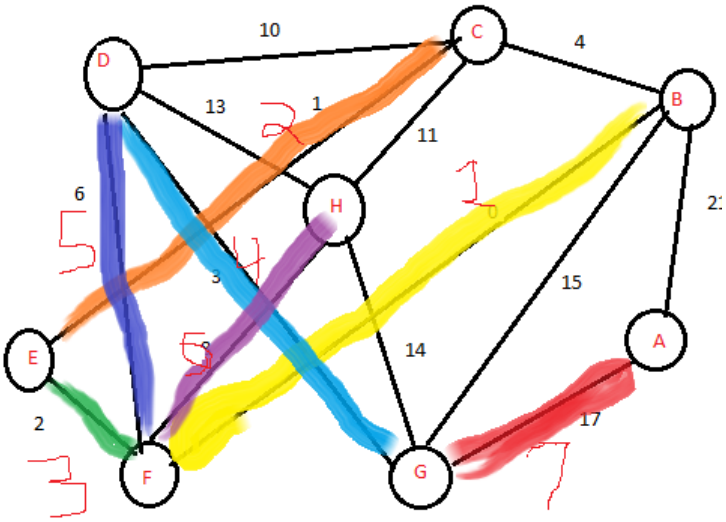


Figura 3: Orden de como tomamos las aristas

2

Un fragmento de ADN es una palabra que usa únicamente las letras A,C,G y T. Se tienen n fragmentos de ADN de longitud 4.Si se tienen dos fragmentos de ADN , podemos pegarlos si pasa alguna de las siguientes dos condiciones:

- Las dos ultimas letras del primero son iguales a las dos primeras letras del segundo.
- Las tres ultimas letras del primero son iguales a las tres primeras letras del segundo.

Por ejemplo los fragmentos ACGG y CGGT se pueden pegar para obtener ACGGT.Los fragmentos ACTG y TGAA se pueden pegar para obtener ACTGAA.
Dados dos de estos n fragmentos F_1 y F_2 , propón un pseudocódigo un algoritmo que determine si se pueden pegar algunos de los n fragmentos para obtener un fragmento (de la longitud que sea) que comience con F_1 y termine con F_2.
Explica las suposiciones de tu modelo y evalúa asintoticamente la complejidad de tu propuesta.

Solución 2

Lo primero que requerimos son todos los fragmentos posibles y los que utilizaremos como F_1 y F_2. A partir de esto creamos una gráfica dirigida en donde cada vértice es un fragmento.
Ahora veamos que si las ultimas dos letras de un fragmento son iguales a las dos primeras letras de otro fragmento o en otro caso si las ultimas tres letras de un fragmento son iguales a las primeras tres letras de otro fragmento se genera un enlace.
Entonces ahora tenemos nuestra gráfica con todos los enlaces que existen.Por lo que ahora solo veremos si existe algún camino entre el fragmento F_1 y el fragmento F_2 para eso vamos a utilizar búsqueda por anchura, en donde empezamos en el vértice F_1 y si la búsqueda por anchura llega en algún momento a F_2 decimos que si existe un fragmento total que las una.En caso de que no exista, sabemos que la gráfica no es conexa y regresamos que no existe tal fragmento total.
Ahora lo escribiremos en pseudocódigo

Algorithm 1 Encontrar si existe un camino entre dos fragmentos

Requerimientos Una lista de todos los fragmentos L y F_1, F_2
Regresa Si existe un fragmento total que una F_1 y F_2

Creamos una gráfica G en donde cada nodo es un fragmento de la lista de fragmentos

n=longitud(lista fragmentos)

while i in n **do**

  while j in n **do**

    if L[i][-3:]=L[j][:3] **or** L[i][-2:]=L[j][:2] **then** Agregamos un enlace dirigido que va de i a j

    end if

  end while

end while Aplicamos búsqueda por anchura que empieza en F_1

if Si BFS encontró a F_2 **then** Regresamos que si existe un fragmento total que une a F_1 Y F_2

end if
Else Regresamos que no existe un fragmento total que une a F_1 y F_2

Notamos que solo hay un enlace entre los nodos si se cumplen las instrucciones y luego usamos BFS ,el cual es un algoritmo greedy y que ya probamos su correctitud en la clase, por lo cual nuestro algoritmo es correcto.
También estamos suponiendo que ya tenemos un algoritmo de búsqueda por anchura preparado, ya que no lo estoy describiendo en este pseudocódigo.
Por ultimo evaluaremos este algoritmo para encontrar su complejidad asintótica.
En la primera parte vemos que hay dos ciclos `while` uno dentro del otro, por lo que esa parte tiene complejidad $O(n^2)$, por otro lado sabemos que búsqueda por anchura tiene complejidad de $O(n^2)$.
Por lo que llegamos a que la complejidad total de esta propuesta es de $\underline{O(n^2)}$

3

(2.5 pts) Da un algoritmo en pseudocódigo que en tiempo $O(n)$ encuentre un subconjunto de tamaño máximo de aristas disjuntas en un árbol con n vértices. Recuerda que para que dos aristas sean disjuntas debe suceder que no comparten vértices.

Solución

Lo primero que hacemos es crear una copia de solo los vértices del árbol que nos dan. Luego encontramos el nodo hoja más alejado de la raíz y vemos su arista.
Aquí usamos un ciclo `while` mientras el árbol tenga aristas.
Proseguimos a agarrar la arista encontrada anteriormente y ponemos esta en la copia de los vértices del árbol y borramos las aristas de este vértice.
Con lo que obtenemos un nuevo árbol y volvemos a llamar a ciclo `while` sobre este nuevo árbol hasta que al árbol ya no le quede ninguna arista.
A continuación viene el pseudocódigo

Algorithm 2 Encontrar el subconjunto de tamaño máximo de aristas disjuntas

Requerimientos Un árbol.

Regresa Gráfica G con el máximo de aristas disjuntas del árbol

Creamos una gráfica G donde los nodos son los vértices del árbol, pero no ponemos sus aristas.

while aristas del árbol $\neq 0$ **do** Elegir el nodo hoja más alejado de la raíz y tomar su arista .
Ponemos en la gráfica G esta arista y borramos del árbol esta arista más sus aristas adyacentes y los vértices que ya no tengan aristas.

end while

Notamos que el algoritmo pasado solo tiene un ciclo `while` por lo que su complejidad es de $O(n)$

4

(2.5 pts) Se tienen n intervalos finitos en R, dados como $[a_i, b_i]$ para cada i en $1, 2, ..., n$. A partir de ellos, se crea una gráfica G de la siguiente manera: hay un vértice por cada intervalo y hay una arista si los dos intervalos correspondientes se intersectan. Es sencillo verificar si esta gráfica es la gráfica completa en tiempo $O(n^2)$. Veremos cómo hacerlo en menos tiempo.

- Escribe qué desigualdades deben cumplir a_i, b_i, a_j, b_j para que los intervalos $[a_i, b_i]$ y $[a_j, b_j]$ se intersecten.

- Propón en pseudocódigo un algoritmo que use el inciso anterior para determinar si la gráfica G es completa en tiempo $O(n^2)$.
- Sea b el mínimo de los b_j .Muestra que todas las parejas de intervalos se intersectan si y sólo si b está en todos los intervalos.
- Propón en pseudocódigo un algoritmo que use el inciso anterior para determinar si la gráfica G es completa en tiempo $O(n)$

Solución

1)
Para que los intervalos $[a_i, b_i]$ y $[a_j, b_j]$ se intersecten deben de cumplir lo siguiente que a_j sea mayor o igual a a_i y al mismo tiempo que a_j sea menor o igual a b_i, de otra forma que b_j sea mayor o igual a a_i pero y menor o igual a b_i, visto de otra forma requerimos que:

$$a_i \leq a_j \leq b_i$$

$$o$$

$$a_i \leq b_j \leq b_i$$

2)
Ahora veamos un pseudocódigo para ver si la gráfica es completa en tiempo $O(n^2)$.Para eso notamos que todos los intervalos deben de intersectarse entre si.
Por lo que cada intervalo sera un vértice y existe un enlace entre dos intervalos si estos se itersectan.
Una vez obtenida esta gráfica obtenemos su matrix de adyacencia , en donde checamos cada entrada de esta, donde si la entrada $ij \neq 0 \quad \forall \quad 0 < i, j \leq n$ y $i \neq j$ decimos que la gráfica es completamente conectada, ya que se pude llegar de cada nodo a cualquier otro en un solo paso.
Nota: notamos que si $i = j$ su entrada de la matriz de adyacencia si puede ser cero ya que para que sea

completa no es necesario que tenga auto-enlaces.
Por lo que ahora lo escribimos en pseudocodigo.

Algorithm 3 Decir si la gráfica es completa o no

Requerimientos n intervalos .
Regresa Si si todos los intervalos se intersectan, no en caso contrario

Creamos una gráfica G donde los nodos son los intervalos y hay enlace si se intersectan estos intervalos.
.
sacamos su matriz de adyacencia

while i in n **do**
    while j in n **do**
        if i !=j **then**
            if ij != 0 **then** status=verdadero
            end if
            if ij ==0 **then** status=falso
Terminamos el ciclo
            end if
        end if
    end while
end while
if status=verdadero **then** Decimos que G es completa
end if
if status=falso **then** Decimos que G no es completa
end if

Ahora vemos en que tiempo asintótico corre este algoritmo. Notamos que tenemos un ciclo `While` anidado dentro de otro y es lo único que le agrega complejidad a nuestro algoritmo por lo que nos corre en tiempo $O(n^2)$
3)
Por hipotesis tenemos que $b = min\{b_1, ..., b_i, ...b_n\}$
Primero hacemos el regreso de la demostración.
$\Leftarrow$ Suponemos que b esta en todos los intervalos .**PD** todos los intervalos se intersectan.
Denotaremos como I_j el intervalo de $[a_j, b_j] \quad \forall j = 1, ..., n$.
Tenemos que b se encuentra en $I_j \; \forall j = 1, ..., n$ entonces tenemos que $b \in \bigcap_{i=1}^{n} I_i$.
De aquí vemos que hay una intersección no vacía de todos los intervalos, por lo que todas los intervalos se interactúan si b pertenece a todos los intervalos.

Ahora hacemos la ida.
$\Rightarrow$ Suponemos que todos los intervalos se intersectan.
Lo haremos por contradicción.
Ahora vamos a suponer que hay un intervalo el cual b no pertenece, por lo que tenemos dos casos posibles $b < a_i$ o $b_i < b$.
En el caso donde $b_i < b$ llegamos a que $b \neq min\{b_1, ..., b_i, ...b_n\}$ lo que es una contradicción con nuestra hipótesis de que b era el mínimo.

En el caso donde $b < a_i$ llegamos a una contradicción ya que también suponemos que todos los intervalos se intersectan y allí vemos que el mínimo de los intervalos superiores es menor a un limite inferior de un intervalo, por lo que no todos los intervalos se intersectan.
Con esto llegamos a una Contradicción !!!!.
Por lo tanto si todas los intervalos se intersectan entonces b esta en todos los intervalos.
Por lo tanto llegamos a que: todas las parejas de intervalos se intersectan si y sólo si b esta en todos los intervalos.

$$QED$$

4)
Ahora utilizando lo anterior vamos a hacer un algoritmo que corra en $O(n)$ para ver si la gráfica es completa.
Primero vamos a encontrar el mínimo de todos sus cotas superiores y lo llamaremos b.Ahora veremos si para cada intervalo I, b esta en este.
Si encontramos un intervalo i en dude $b \notin I$ Decimos que no es completa.En caso contrario decir que si es completa.

Algorithm 4 La gráfica de intervalos es completa o no

Requerimientos Gráfica de intervalos G.
Regresa Si es completa o no.

b=min{$b_1, ...b_i, ..., b_n$}
$Bandera = True$
for $nodo$ en nodos de G. **do**
If

B no esta en nodo **then** Bandera=False

end if
end for

if Bandera=True **then** La gráfica es completa

end if

if Bandera=False **then** La gráfica no es completa

end if

Aquí notamos que solo hay un ciclo `for` por lo que el algoritmo corre en tiempo $O(n)$

Extra

(+2 pts extra) Sea A la matriz de adyacencia de una gráfica G con n vértices. Toma enteros i, j en $\{1, ..., n\}$ y un entero positivo k. Demuestra por inducción que la entrada ij de la matriz A^k es igual a la cantidad de caminos que hay del vértice i al vértice j que tengan longitud k. Usa esto para proponer un algoritmo que calcule esta cantidad de caminos y analiza su tiempo de ejecución.

Demostración

Para probar esto vamos a hacer inducción sobre k.
Para nuestra base inductiva sea $k = 1$, por lo que la matriz A^1 es igual a matriz de adyacencia, que por como esta construida es la longitud de el camino de i a j si no tiene pesos es 1 si hay un vetice que las uno o 0 si no hay vertice que las une.Por lo tanto es el total de camino de longitud 1 que van de i a j.

Ahora supongamos que para cierto $d \geq 0$, que la entrada ij de A^d dice la cantidad de caminos posibles de i a j de longitud d.
Ahora tenemos que demostrar que es válido para $d + 1$.

$$(A^{d+1})_{ij} = (A^d \times A)_{ij}$$
$$= \sum_{k=1}^{n} (A^d)_{ik} \times A_{kj}$$

Por lo que vemos que un camino de $d + 1$ de i a j esta conformado de dos caminos mas pequeños un que va de i a k y otro de k a j.
Sabemos por paso inductivo que A_{kj} es distinto de cero si existe un vértice entre k y j.
Por lo que $\sum_{k=1}^{n} (A^d)_{ik} \times A_{kj}$ es el total de camino que van de i a j

$$QED$$

Ahora visto esto queremos hacer el algoritmo para encontrar la cantidad de camino y analizar su tiempo de ejecución.
Para eso solo basta saber la longitud del camino que quieren que encontremos k y de que vértice a que vértice i,j.Luego multiplicamos la matriz de adyacencia por si misma k veces y nos fijamos en la entrada i, j la cual nos dice el total de camino posibles entre estos dos vértices.

Algorithm 5 Decir el total de caminos de i a j con longitud de k

---

Requerimientos matriz de adyacencia A, el nodo saliente i y el nodo incidente j y la longitud del camino k .
Regresa Total de caminos de longitud k de i a j
multiplicamos A k-veces
Imprimimos ij

Una vez que ya tengamos la matriz de adyacencia solo se tiene que multiplicar k veces para tener su camino.Así que su complejidad en tiempo viene dada por la de multiplicación de matrices.
Sabemos que el algoritmo estándar de multiplicación de matrices tiene complejidad de $O(n^3)$ de tiempo, por lo que al hacerlo k veces tenemos $kO(n^3)$, por lo que llegamos que la complejidad de tiempo de este algoritmo es de $O(n^3)$