PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

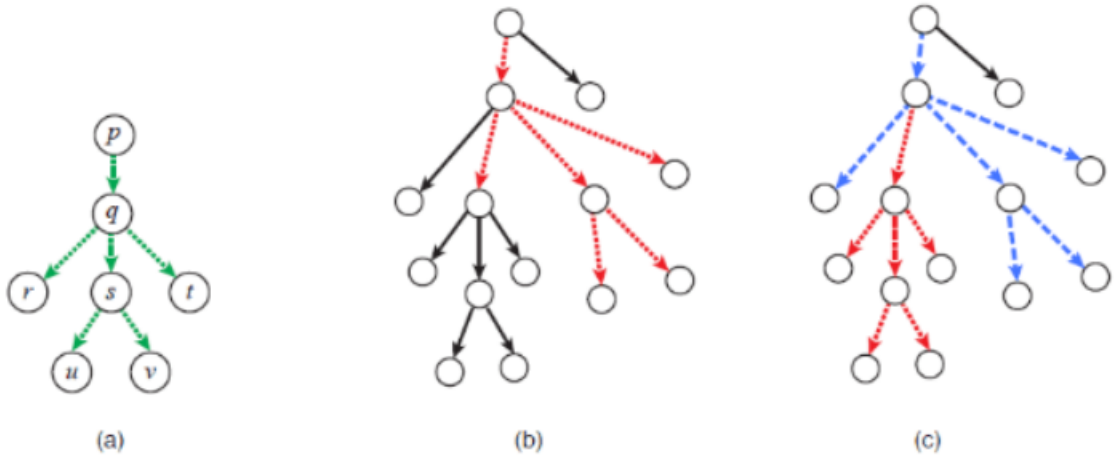
IIC2283 Diseño y análisis de algoritmos
2° semestre 2019

Ayudantía 4

Dudas a Antonio López (alopez7@uc.cl)

Problema 1

Un *stickman* es un grupo de 6 aristas $(p,q),(q,r),(q,t),(q,s),(s,u),(s,v)$ con la forma que se muestra en la figura (a).



1. Has un algoritmo que dado un árbol de grado arbitrario indica cual es el máximo de *stickmans* que se pueden construir de manera de que no compartan aristas entre ellos.
2. Demuestra que el algoritmo anterior es correcto.
3. Calcula la complejidad del algoritmo con respecto al número de nodos del árbol.

Solución: Un stickman es construido con 3 tipos de aristas: cuello, torso y extremidades. Cualquier arista puede ser una extremidad pero no cualquier arista puede ser un torso o un cuello. Para ser un torso, una arista tiene que llegar a un nodo del cual salen otras 2 aristas. Para ser cuello, la arista debe llegar a un nodo que tenga un torso y otras 2 aristas saliendo de el (extremidades). Sabiendo esto, el algoritmo para encontrar el máximo de strickmans es el siguiente:

- Separar aristas por niveles de profundidad en el árbol
- Iterar por los niveles desde el más profundo hasta la raíz:
  - Determinar cuales aristas del nivel pueden ser torso de stickman y cuales pueden ser cuello
  - Sumar 1 por cada arista que puede ser cuello y eliminar esas aritas del árbol

Ahora demostremos que el algoritmo es correcto:

- Notemos primero que el algoritmo selecciona las aristas que pueden ser cuello de un nivel, suma 1 por cada cuello y elimina esos cuellos del árbol. Esto es correcto porque no es posible construir 2 stickmans con cuellos distintos en el mismo nivel de profundidad del árbol y que intersecten entre ellos. Por lo tanto todos los cuellos de un mismo nivel definen stickmans distintos y correctos que no se intersectan.
- Al sacar los cuellos ya utilizados del árbol hace que se desconecte un subárbol hacia abajo eliminando del árbol todas las aristas que ya se usaron para un stickman. Esto hace que no sea posible que el algoritmo cree stickmans con aristas en común.
- Ya que se usan todos los posibles cuellos por nivel, no se puede agregar más stickmans a la solución creada por el algoritmo (esto no significa que sea óptima).
- Ahora definiremos la solución dada por el algoritmo como un conjunto $S = \{s_1, s_2, \ldots, s_n\}$ donde cada s_i es un stickman.
- Dado S demostraremos que para toda solución óptima $O = \{o_1, o_2, \ldots, o_m\}$ se cumple que $m = n$. Para esto partiremos desde la solución O y la transformaremos en S hasta que sean la misma. Esto será suficiente para demostrar que $m = n$ ya que en S no es posible agregar más stickmans.
- Dado S y O podemos ordenar los stickmans de ambas soluciones por nivel de abajo hacia arriba. Denominamos S_i y O_i como los conjuntos de stickmans de las soluciones S y O del nivel i respectivamente. Luego comparamos el nivel más profundo de ambas soluciones. Si ambos niveles tienen los mismos stickmans pasamos al siguiente hasta que sean distintos.
- Sea i el nivel tal que $|S_i| \neq |O_i|$. Dada la forma en la que se construye S sabemos que $|S_i| > |O_i|$, por lo tanto, existe un s_j en S_i que no pertenece a O_i. Además sabemos que O es la solución óptima para el árbol por lo que no podemos incluir a s_j en O para hacer una mejor solución, por lo tanto s_j intersecta con otro stickman o_k en O. Esto a su vez nos dice que o_k no pertenece a S.
- Es necesario notar acá que no es posible que s_j intersecte a 2 stickmans distintos en O_l donde $l < i$ ya que o sino estos 2 stickmans en O_l intersectan entre ellos. Para demostrar esto hay que probar que dado 3 stickmans a, b, c donde el cuello de a está más abajo que el cuello de b y de c, si a colisiona con b y con c implica que b colisiona con c. Esto es simple ya que a debe colisioan al menos en el cuello con b y con c por lo tanto b y c colisionan entre ellos en el cuello de a.
- Por lo tanto, se puede remplazar en O el stickman o_k por s_j sin que este colisione con ningún otro stickman en O. Este proceso se puede repetir hasta hacer que O_i sea igual a S_i. Si hacemos esto hasta la raíz, tendremos que O tiene los mismos elementos que tenía inicialmente y además sus elementos son todos iguales a los en S. Por lo tanto S también es una solución óptima.

La complejidad del algoritmo depende solo de ordenar las aristas por nivel e iterar por cada nivel revisando si las aristas pueden ser cuellos o torsos. Por lo tanto el algoritmo toma tiempo $O(n \log n)$

Problema 2

Se tienen dos arreglos A y B de números positivos, de largos n y m respectivamente. Se puede realizar la siguiente operación un número arbitrario de veces: se toma un subarreglo continuo de un arreglo y se reemplaza por la suma de los elementos en el subarreglo. Por ejemplo, del arreglo [1, 10, 100, 1000, 10000] se puede obtener el arreglo [1, 1110, 10000], [11, 100, 1000, 10000], [1, 10, 11100], etc.
Dos arreglos A y B son iguales si tienen el mismo largo y para todo i se cumple que A[i] = B[i]. Dado dos arreglos A y B, se quiere realizar la operación descrita anteriormente hasta que ambos arreglos sean iguales. Más aún, se busca que al terminar el largo de A y B sea máximo.
Entregue un algoritmo que dado dos arreglos A y B retorne el largo máximo de los arreglos tras realizar la transformación descrita anteriormente, o NO si no es posible hacer esto. Analice la complejidad del algoritmo.

Solución: Notemos que si la suma de todos los elementos de A suman distinto que todos los elementos de B entonces no es posible agrupar ambos arreglos hasta que sean iguales. Dicho esto el algoritmo es el siguiente:

- $max\_length(A, B)$:
  - $n = A.length$
  - $m = b.length$
  - $if(sum(A) \neq sum(B))$, $return$ NO
  - $length = 0$
  - $i = 0$
  - $j = 0$
  - $while(i < n)$:
    - $s_1 = A[i]$
    - $s_2 = B[j]$
    - $i++$
    - $j++$
    - $while(s_1 \neq s_2)$:
      - $if(s_1 \leq s_2)$, $s_1 += A[i]$, $i++$
      - $else$, $s_2 += B[j]$, $j++$
    - $length++$
  - $return\ length$

Lo que se hace es sucesivamente tomar los prefijos de A y B más pequeños que sumen lo mismo. Luego esos prefijos se colapsan a un solo número, por lo que se le suma 1 al largo de la solución. Falta demostrar que el algoritmo es correcto. Primero, notemos que si un cierto subarreglo fue colapsado, entonces podemos asumir que el elemento resultante no es colapsado nuevamente (sería equivalente a colapsar un subarreglo más grande inicialmente). Con esto, los distintos índices donde se colapsa el arreglo generan particiones. Sean $i_1, i_2, \ldots, i_l$ las particiones de A por nuestro algoritmo MaxLength, y sean $j_1, j_2, \ldots, j_k$ las particiones óptimas, con $k > l$. Sea y el mínimo número tal que $i_y \neq j_y$. Notar que $i_y < j_y$ por el funcionamiento del algoritmo. ¡Tenemos que $j_1, j_2, \ldots, j_{y-1}, i_y, j_y, ..., j_k$ define una partición mejor que la óptima! Se concluye que la solución óptima no puede ser mejor que la solución encontrada por el algoritmo MaxLength. Sobre la complejidad del algoritmo, se tiene que i y j aumentan monotónicamente desde 0 hasta n y m, respectivamente. Además, se hace una cantidad constante de operaciones por cada aumento de i y j. Por lo tanto, el algoritmo es O(n + m).

Problema 3

El algoritmo de Kruskal encuentra el MST de un grafo (minimun spanning tree) de la siguiente forma:

- Ordenar las aristas por costo de menor a mayor
- Mientras quedan aristas
  - Sea a la primera arista de la lista
  - Si a crea un ciclo en el MST, se descarta
  - Sino, se agrega al MST
  - Se elimina la arista a de la lista

Demuestra que el algoritmo de Kruskal encuentra un MST.

Solución: En cada iteración del algoritmo de kruskal, el algoritmo toma una arista y revisa si esta crea un ciclo en el MST. Si esta arista no crea un ciclo en el MST entonces se agrega a este. Demostraremos que la arista agregada por el algoritmo en cada iteración pertenece a un MST. Notar que pueden existir múltiples MST para un grafo. Para demostrar esto asumiremos que existe un árbol de cobertura con menor costo que el producido por nuestro algoritmo de Kruskal:

- Sea $K = \{e_1, \ldots, e_{|V|-1}\} \subseteq E$ el árbol generado por el algoritmo de Kruskal en el grafo $G(V, E)$.
- Dada una arista $e_i = (a, b) \in K$ cualquiera se separan los nodos de V en 2 grupos A y B donde $a \in A$ y $b \in B$ y el resto de los nodos quedan en el grupo A y B dependiendo del árbol en el que quedarían si se saca e_i del árbol K. Esta división es un corte de del grafo.
- La arista e_i es la arista del grafo de menor costo que pasa de un lado del corte al otro. Esto es así dado el funcionamiento del algoritmo.
- Demostraremos que en un grafo si definimos un corte cualquiera, la arista de menor costo que pasa por el corte debe estar en el MST. Esto es suficiente para demostrar que Kruskal encuentra un árbol de cobertura mínimo.
- Si dado un corte, la arista más barata que cruza el corte no está en el MST, entonces podemos agregar la arista más barata que pasa por el corte al MST. Esto generaría un ciclo en el árbol que incluye al menos una arista que pasa por el corte. Esta arista que pasa por el corte y es parte del ciclo tiene costo mayor a la agregada al MST por lo tanto si la sacamos tendríamos como resultado otro árbol de cobertura pero de menor costo. Por lo tanto el MST original no era en realidad mínimo.