Estructuras de Datos y Algoritmos – iic2133

I2

3 octubre 2016

Contesta sólo 4 preguntas

1. Un *puente* en un grafo no direccional es una arista que, si se saca, separaría el grafo en dos subgrafos disjuntos. Describe un algoritmo eficiente que encuentre todos los puentes de un grafo no direccional; justifica la eficiencia y la corrección de tu algoritmo.

Respuesta:

Una arista (u, v) es un puente Û (u, v) no pertenece a un ciclo; dfs encuentra los ciclos. Por lo tanto, ejecutemos dfs y miremos el bosque dfs producido. Recordemos que dfs en un grafo no direccional sólo produce aristas de árbol y hacia atrás; **no** produce aristas hacia adelante ni cruzadas.

Una arista de árbol (u, v) en el bosque dfs es un puente Û no hay aristas hacia atrás que conecten un descendiente de v a un ancestro de u. ¿Cómo determinamos esto? Recordemos el tiempo de descubrimiento $v.d$ que dfs asigna a cada vértice v : una arista hacia atrás (x, y) conectará un descendiente x de v a un ancestro y de u si $y.d < x.d$.
Luego, si $v.d$ es menor que todos los tiempos d que pueden ser alcanzados mediante una arista hacia atrás desde cualquier descendiente de v, entonces (u, v) es un puente.

Es decir, basta modificar un poco `dfsVisit` visto en clase: asignar a cada vértice v un tercer número $v.min$ que sea el menor de todos los $y.d$ alcanzables desde v mediante una secuencia de cero o más aristas de árbol seguida de una arista hacia atrás; si al retornar `dfsVisit`(v), $v.d = v.min$, entonces (u, v) es un puente. Esta modificación no cambia la complejidad de dfsVisit.

El puntaje distribuye así:

- 4 ptos por el algoritmo (Se toma en cuenta cuán eficiente es).
- 1 pto por justificar correctamente la eficiencia, independiente que tan bueno sea el algoritmo y suponiendo que este es correcto.
- 1 pto por justificar la correctitud del algoritmo.

2. *Otro algoritmo de inserción en árboles rojo-negros*. En lugar de primero insertar el nuevo nodo, pintarlo de rojo y después hacer rotaciones y/o cambios de color (*hacia arriba*) para restaurar las propiedades del árbol, podemos ir haciendo los ajustes a medida que vamos *bajando* por el árbol hacia el punto de inserción, de modo que cuando insertamos el nuevo nodo simplemente lo pintamos de rojo y sabemos que su padre es negro. El procedimiento es el siguiente.

> Mientras bajamos por el árbol, cuando *en la ruta de inserción* vemos un nodo *X* que tiene dos hijos rojos, intercambiamos colores: pintamos *X* de rojo y sus hijos de negro. Esto producirá un problema sólo si el padre *P* de *X* es rojo. Pero en ese caso, simplemente aplicamos las rotaciones apropiadas, que se ilustran en las figs. A y B adjuntas.

a) Muestra cómo opera este nuevo algoritmo de inserción cuando insertamos un nodo con la clave **45** en el árbol de la fig. C adjunta; específicamente, muestra lo que ocurre a medida que llegas a cada nivel.

Respuesta:

Cuando se baja de la raíz 30 a su hijo derecho 70 no pasa nada, ya que 70 (el *X* del algoritmo) tiene sólo un hijo rojo, 60. Cuando se baja de 70 a 60 tampoco pasa nada, ya que 60 (el *X* del algoritmo) es rojo. Cuando se baja de 60 a 50 estamos en el caso descrito en el algoritmo: 50 (el *X* del algoritmo) es negro y sus dos hijos, 40 y 55, son rojos. [*Hasta aquí no hemos cambiado nada, sólo bajamos por el árbol*: **0.5 pts.**]

Entonces, cambiamos colores: pintamos a 50 de rojo y a sus hijos 40 y 55 de negro. Pero ahora 50 y su padre 60 son rojos; como el hermano 85 de 60 es negro, estamos en la situación descrita en la fig. A: 50 es *X*, 60 es *P*, 70 es *G*, 85 es *S*. [**1 pt.**]

Aplicando la rotación y cambios de colores sugeridos en la fig. A, queda 60 de negro como nuevo hijo derecho de 30 y con dos hijos rojos, 50 y 70, que en total tienen cuatro hijos negros, 40, 55, 65 y 85 (este último mantiene sus dos hijos rojos, 80 y 90, aunque ahora los tres están un nivel más abajo). [**1 pt.**]

Nosotros seguimos en 50, que ahora es rojo y está un nivel más arriba, así como sus hijos, 40 y 55, ahora negros. Bajamos a 40, que es una hoja negra, y por lo tanto insertamos 45 como hijo derecho de 40 y lo pintamos rojo. [**0.5 pts.**]

b) Los casos ilustrados en las figs. A y B se producen cuando el hermano *S* del padre *P* de *X* es negro. Explica claramente qué pasa con el caso en que *S* es rojo *en este nuevo algoritmo de inserción*.

Respuesta:

Este caso en realidad *no se produce*, debido a los ajustes que vamos haciendo a medida que bajamos por el árbol. Si es que encontramos un nodo *Y* con dos hijos rojos, sabemos que sus nietos tienen que ser todos negros; y como también pintamos de negro los hijos de *Y*, entonces incluso después de las posibles rotaciones no vamos a encontrar otro nodo rojo en los próximos dos niveles.

3. Tienes el texto completo de la novela *A Tale of Two Cities*, de Charles Dickens.

> *It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness …*

Un algoritmo de reconocimiento de texto necesita obtener la lista de palabras distintas en la novela. Escribe un algoritmo que haga esta operación en tiempo O(*n*) promedio, en que *n* es el número total de palabras del texto; y justifica esta complejidad.

Respuesta:

Una solución es hacer una tabla de hash con encadenamiento sobre los posibles strings y en base a eso detectar las palabras iguales:

$TablaHash \leftarrow$ Arreglo tamaño m de listas vacías.
$Resp \leftarrow$ Lista vacía de respuesta.
for $pal \in Palabras$ **do**
  $key \leftarrow hash(pal)\%m$
  if $TablaHash[key] == \emptyset \vee pal \notin TablaHash[key]$ **then**
    $TablaHash[key].add(pal)$
    $Resp.add(pal)$
  end if
end for
return $Resp$

Donde la función de hash es tal que no produce colisiones: se puede transformar el String a un entero fácilmente usando potencias de 128 en ASCII.

Cabe destacar que la tabla si tendrá colisiones, ya que a la función de hash luego se le aplica el módulo del tamaño del arreglo "m".

Finalmente como nuestra función de hash distribuye bien y suponiendo que satisface HUS, el riesgo de colisiones distribuye uniformemente en el caso promedio y por lo tanto la búsqueda en promedio será O(1) si tomamos un "m" lo suficientemente grande. (Se pueden hacer otras funciones o suponer que existe una, siempre y cuando se diga que satisface HUS).

El puntaje se entrega de la siguiente forma:

- 1 pto si es que se hizo un algoritmo correcto pero no de la complejidad pedida
- 3 ptos si es que se hizo una tabla de hash pero sin colisiones *
- 6 ptos por una respuesta correcta con justificación de complejidad, también existen otras soluciones como usar direccionamiento abierto.

* Es imposible hacer una tabla de hash sin colisiones ya que las palabras posibles son muchas (en teoría infinitas), notamos que además la cantidad de strings posibles crece exponencialmente según el largo, por lo que por ejemplo si hiciéramos una tabla sin colisiones para strings de largo 100 se necesitaría un arreglo de tamaño $128^{100} = 5.26 * 10^{210}$ lo cual es gigantesco y no es posible para ningún computador (Un computador normal soporta del orden de 10^{9} de memoria).

4. **a**) Escribe un algoritmo que ordene una lista doblemente ligada *L* usando *quickSort*. Puedes usar de todas las operaciones pertinentes en listas, incluyendo *crear* y *destruir*.

b) Digamos que existe un algoritmo que permite mezclar dos listas ordenadas, obteniendo una lista ordenada, en tiempo O(1). Analiza la eficiencia de *mergeSort* usando este algoritmo para mezclar las listas

c) Explica qué propiedad de *radixSort* permite garantizar la ordenación de los elementos.

Pauta:

a) Si el algoritmo no es quicksort. 0 pts.

Si el algoritmo es quicksort pero no está hecho para listas, por ejemplo usa lista[i], lo que en una lista ligada es O(n). 0 pts.

Si el algoritmo es quicksort y tiene muchos errores. 1 pt.

Si el algoritmo es quicksort y tiene pocos o no tiene errores: 2pts.

b) 1 punto por plantear la ecuación de recurrencia o el problema maestro o plantear una sumatoria que resuelva el problema.

1 punto por resolver el problema con lo anterior.

c) 2 puntos si dice estabilidad o si explica la propiedad de estabilidad con palabras.

0 puntos en caso contrario.

5. Para cada una de las siguientes situaciones, indica qué algoritmo visto en clase es el que más se beneficia, y cuál, el que se ve más perjudicado, justificando el porqué. Además, propón un algoritmo específico para el caso, que sea especialmente eficiente, indicando su complejidad. Si no hay algoritmos que se beneficien o perjudiquen, indícalo.

0.5 por cual se beneficia
0.5 por cual se perjudica
0.5 por el algoritmo propuesto eficiente

i. Datos ordenados en sentido contrario.

Beneficiado: No hay algoritmo beneficiado

Perjudicado: Insertion Sort, Shell Sort, QuickSort

Propuesto: Dar vuelta los elementos del arreglo. O(n)

ii. Datos ordenados de a pares, en que cada par de elementos $L[2*i]$ y $L[2*i+1]$ en el arreglo L cumplen que $L[2*i] < L[2*i+1]$.

Beneficiado: Insertion Sort, Shell Sort.

MergeSort no se beneficia porque a pesar de que venga ordenado de a pares, MergeSort no lo sabe y hará las comparaciones igual.

Perjudicado: No hay algoritmo perjudicado

Propuesto: MergeSort pero solo haciendo merge, saltándose las comparaciones iniciales O(n*log(n))

Ordenar de a pares e impares y luego hacer merge de ambos resultados O(n*log(n))

iii. Datos en que cada uno es un número natural entre 1 y 100.

Beneficiado: No hay algoritmo beneficiado.

Perjudicado: No hay algoritmo perjudicado.

Propuesto: Counting Sort. O(n)

iv. Datos ordenados excepto por un dato insertado de manera aleatoria.

Beneficiado: Insertion Sort, Shell Sort

Perjudicado: QuickSort

Propuesto: Ej. (Listas) Encontrar el elemento, sacarlo y luego ingresarlo haciendo búsqueda binaria.

(Arreglo) Encontrar el elemento y hacer swap hasta que quede en su posición correcta.

Insertion Sort hasta que ordene un elemento.

Todos estos son O(n).