Universidad de Chile
Departamento de Ciencias de la Computación
CC4102 - Diseño y Análisis de Algoritmos

Auxiliar 3 - "Costo Promedio"

Profesor: Pablo Barceló
Auxiliar: ~~Manuel~~ Ariel Cáceres Reyes

07 de Abril del 2017

P1. Shuffle

El análisis probabilístico de los algoritmos nos permite estudiar su comportamiento promedio asumiendo cierta distribución en la entrada. Por otro lado, los algoritmos aleatorizados son algoritmos a los que se les introduce una componente aleatoria.
Un procedimiento estándar en los algoritmos aleatorizados consiste en generar una permutación al azar (escogida uniformemente entre las permutaciones de n elementos) de los elementos de una lista o "shuffle".
Considere el problema de "shuffle" sobre una lista de enteros $A[0, \ldots, n-1]$

a) Muestre un algoritmo de costo $\mathcal{O}(n^2)$.

b) Muestre un algoritmo de costo $\mathcal{O}(n \log n)$.

c) Muestre un algoritmo que sea *in-place* y de costo $\mathcal{O}(n)$.
Demuestre su correctitud.

P2. InsertionSort

Una inversión en un arreglo corresponde a un par de elementos $A[i] > A[j]$ con $i < j$.
Supongamos que los elementos de $A[1, n]$ vienen dados por una permutación aleatoria Π de $\{1, \ldots, n\}$.

a) Calcule la cantidad esperada de inversiones en $A[1, n]$.

b) Utilice la parte anterior para demostrar que el número promedio de intercambios en InsertionSort es $\mathcal{O}(n^2)$

P3. Codificación de Huffman

Supongamos que tenemos las probabilidades $\{p_1, \ldots, p_n\}$, tales que $\sum_{i=1}^{n} p_i = 1$. El algoritmo de Huffman construye un árbol binario que tiene estas probabilidades en sus hojas y minimiza $\sum_{i=1}^{n} p_i n_i$, donde n_i es la profundidad de p_i en el árbol.

a) Escriba en pseudocódigo el algoritmo

b) ¿Cómo implementaría el algoritmo para que tuviera costo $\mathcal{O}(n \log n)$?

c) Suponga ahora que las probabilidades vienen ordenadas de forma creciente. ¿Cómo implementaría el algoritmo para que tuviera costo $\mathcal{O}(n)$?

d) Demuestre la correctitud del algoritmo

Soluciones

P1. Shuffle

a) Ponemos los elementos en un TDA *Conjunto* y vamos extrayendo de el los elemento al azar dejándolos en el arreglo en el orden que fueron extraídos. Lo anterior lo podemos implementar con una lista enlazada lo que deriva en un costo de $\mathcal{O}(n^2)$.

b) Si generamos números al azar entre 0 y 1 por cada elemento del arreglo y luego ordenamos el arreglo original (con MergeSort por ejemplo) según los números asociados generaremos una permutación de este en $\mathcal{O}(n \log n)$.

c) El siguiente código en "Python" es *in-place* y de costo lineal:

```
for i in range(n):
    r = randint(i,n)
    a[i], a[r] = a[r], a[i]
```

Para mostrar que genera una permutación al azar uniformemente entre todas las permutaciones, se demostrará lo siguiente:

"Después de la i-ésima iteración el arreglo $a[1,\ldots,i]$ es cada i-permutación de a con probabilidad $\frac{(n-i)!}{n!}$"

Si lo anterior es cierto entonces después de la última iteración se cumplirá que a es cada permutación de a con probabilidad $\frac{(n-n)!}{n!} = \frac{1}{n!}$, que es precisamente lo que buscamos.

La demostración es por inducción en i. El caso base $i = 1$ nos dice que después de la primera iteración $a[1]$ es cada elemento (1 permutaciones) de a con probabilidad $\frac{(n-1)!}{n!} = \frac{1}{n}$, lo que es cierto pues en esa iteración $a[1]$ se intercambia uniformemente con alguno de los elementos de a. Para el paso inductivo asumamos el enunciado cierto para valores menores a i y consideremos la probabilidad:

$$\mathbb{P}\left(a[1,\ldots,i] \; es \; x_1,\ldots,x_i\right)$$

, donde $x_1,\ldots,x_i$ es alguna i-permutación de a.
Veamos que para que lo anterior sea cierto se tiene que cumplir que en la (i-1)-ésima iteración $a[1,\ldots,i-1]$ haya sido $x_1,\ldots,x_{i-1}$ y en la i-ésima el algoritmo puso x_i en $a[i]$. Por lo que la probabilidad anterior es igual a :

$$\mathbb{P}\left(a[1,\ldots,i-1] \; es \; x_1,\ldots,x_{i-1} \wedge x_i \text{ es puesto en } a[i]\right)$$

Ahora notemos que estos eventos no son independientes (piense en que si no se supiese que $a[1,\ldots,i-1]$ es $x_1,\ldots,x_{i-1}$, entonces x_i podría estar allí y en ese caso la probabilidad de

que sea puesto en $a[i]$ es 0, dado que el algoritmo solo busca intercambios hacia adelante). Entonces usamos probabilidad condicionada de modo que lo anterior es:

$$\mathbb{P}\left(x_i \text{ es puesto en } a[i] \mid a[1,\ldots,i-1] \text{ es } x_1,\ldots,x_{i-1}\right) \cdot \mathbb{P}\left(a[1,\ldots,i-1] \text{ es } x_1,\ldots,x_{i-1}\right)$$

Pero lo primero es la probabilidad de escoger x_i uniformemente entre las $n-i+1$ posibilidades que hay hacia adelante y lo segundo es nuestra hipótesis inductiva lo que queda:

$$\begin{aligned}\frac{1}{n-i+1} \cdot \frac{(n-(i-1))!}{n!} &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!}\end{aligned}$$

P2. InsertionSort

a) Si Π es el conjunto de las permutaciones e I es una función que toma una permutación π y le asigna el número de inversiones que esta genera, entonces estamos interesados en:

$$\begin{aligned}\mathbb{E}(I(\pi)) &= \sum_{\pi\in\Pi} \frac{1}{|\Pi|} I(\pi) \\ &= \sum_{\pi\in\Pi} \frac{1}{|\Pi|} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{1}_{\pi(i)>\pi(j)} \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{|\Pi|} \sum_{\pi\in\Pi} \mathbb{1}_{\pi(i)>\pi(j)}\end{aligned}$$

Veamos ahora que la suma mas interna es $\frac{|\Pi|}{2}$ pues si fijamos i y j por cada permutación que tiene $\pi(i) > \pi(j)$ otra exactamente igual que tiene lo que está en i intercambiado con lo que está en j. Por lo tanto:

$$\begin{aligned}\mathbb{E}(I(\pi)) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{|\Pi|} \cdot \frac{|\Pi|}{2} \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{2} \\ &= \frac{n(n-1)}{4} \in \mathcal{O}(n^2)\end{aligned}$$

b) Por lo visto en la P1b de la auxiliar 2, el número de intercambios realizados por InsertionSort es exactamente el número de inversiones del arreglo, por lo que el tiempo esperado de intercambios es $\mathcal{O}(n^2)$.

P3. Codificación de Huffman

a) Sea $node(x, n_1, n_2)$ la operación que crea un nodo de árbol binario, cuyo costo es x y sus hijos izquierdo y derecho son n_1 y n_2 respectivamente, entonces el algoritmo de Huffman queda descrito como:

$S \leftarrow \{node(p_1, null, null), \ldots, node(p_n, null, null)\}$
while $|S| > 1$ **do**
  $n_i \leftarrow extractMin(S)$
  $n_j \leftarrow extractMin(S)$
  $put(S, node(n_i.peso + n_j.peso, n_i, n_d)$
end while

b) Si implementamos S como una cola de prioridad con un heap el algoritmo queda $\mathcal{O}(n \log n)$.

c) Si usamos 2 colas, una para nodos sin hijos C_1, en donde ponemos inicialmente los nodos con sus probabilidades ordenadas de forma creciente, y otra para los árboles que tienen hijos C_2 formados por el algoritmo, luego lo que hacemos es tomar los dos mínimos desde el frente de esas colas fusionarlos como antes y dejarlos al final de C_2 podemos hacer el algoritmo en tiempo $\mathcal{O}(n)$ (notar que el costo del algoritmo es simplemente el número de fusiones de Huffman que son $\leq 2n$ pues lo formado finalmente es un árbol).

d) Para demostrar la correctitud de Huffman demostraremos lo siguiente:

"Dado un conjunto de i probabilidades, el algoritmo de Huffman construye un código óptimo"
[1]

Por inducción en i. En el caso base $i = 2$ Huffman entrega el único árbol que representa un código libre de prefijos, por lo que este es óptimo. Para el paso inductivo asumiremos el enunciado cierto para valores menores a i.
Sea T_H el árbol entregado por el algoritmo de Huffman corrido sobre las probabilidades $p_1, \ldots, p_i$ y definamos el peso del árbol $W(T_H)$ como el largo esperado de la codificación, es decir $\sum_{j=1}^{i} p_j n_j$. Supongamos ahora que p_l y p_m son los primeros 2 símbolos escogidos por Huffman (es decir los dos menores) y sea T_H^* el árbol de aplicar Huffman sobre las probabilidades $\{p_1, \ldots, p_i, p_l + p_m\} \setminus \{p_l, p_m\}$ que por hipótesis inductiva ($i - 1$ probabilidades) es óptimo.

Notemos ahora que se cumple $W(T_H) = W(T_H^*) + (p_l + p_m)$, puesto que correr el algoritmo sobre las nuevas probabilidades es lo mismo que correrlo sobre las antiguas a partir de la segunda iteración y además este nuevo nodo $p_l + p_m$ estará un nivel más arriba que los dos nodos p_l y p_m.

[1] Para demostrar que el código construído es óptimo veremos que su largo esperado es $\leq$ al de algún otro código óptimo.

Ahora consideremos un árbol óptimo T para las probabilidades $p_1, \ldots, p_i$ de altura h. Notemos que tanto p_l como p_m deben estar ambos a profundidad h, pues:

- Si ambos están a una profundidad $< h$ podría cambiar cualquiera de ellos con un nodo de profundidad h generando un árbol de menor (contradicción con que T es óptimo) o igual (renombro T de modo que el nuevo T también es óptimo) W.
- Si solo uno de ellos está a profundidad h significaría que es el único a profundidad h (si no intercambio el otro con el que no está a profundidad h generando un árbol de menor (contradicción con que T es óptimo) o igual (renombro T de modo que el nuevo T también es óptimo) W), pero esto es una contradicción pues podría quitarle un bit a su codificación y obtener un árbol de peso menor.

Asumamos ahora que p_l y p_m son hermanos en T (si no lo fuesen intercambiamos al hermano de p_l por p_m renombrando T y obteniendo un nuevo T que también es óptimo). Consideremos además el árbol T^* obtenido de reemplazar el padre de p_l y p_m por el nodo $p_m + p_l$. Con un razonamiento análogo al anterior obtenemos que $W(T) = W(T^*) + (p_l + p_m)$.

Finalmente como T_H^* es óptimo (por H.I.), se cumple que:

$$\begin{aligned} W(T_H^*) &\leq W(T^*) \\ W(T_H^*) + (p_l + p_m) &\leq W(T^*) + (p_l + p_m) \\ W(T_H) &\leq W(T) \end{aligned}$$

Y como T es óptimo T_H, el árbol generado por Huffman, también lo será.