Algunas soluciones, discusiones y otros

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro
Auxiliares: Joshimar Cordova, Jorge Bahamonde
Ayudante: Sebastián Ferrada

15 de Octubre de 2016

2 Cotas inferiores

3. El problema de *búsqueda aproximada en texto* consiste en, dado un string $T[1,n]$, llamado *texto*, otro string más corto $P[1,m]$, llamado patrón, y un entero $0 \leq k < m$, determinar si P ocurre en T permitiendo a lo más k *errores* (inserciones, borrados o reemplazos de caracteres en P (o T)). Ambos T y P son secuencias de caracteres de un alfabeto $[1,\sigma]$.

   Yao demostró en 1976 que no es posible resolver el problema de búsqueda exacta ($k = 0$) en menos de $\Omega(n\log_\sigma(m)/m)$ inspecciones de caracteres de T *en promedio* (el promedio supone que los caracteres de T se eligen al azar, uniformemente en $[1,\sigma]$).

   (a) Demuestre que un adversario impide resolver el problema general inspeccionando menos de $k+1$ caracteres en cualquier ventana posible de T de largo m.

   (b) Deduzca de lo anterior una cota inferior de la forma $\Omega(kn/m)$ para el problema, incluso en el caso promedio.

   (c) Deduzca la cota para el problema $\Omega(n(k+\log_\sigma m)/m)$ en promedio, demostrada por Chang y Marr en 1994. Ellos también diseñaron un algoritmo con esa complejidad promedio, $O(n(k+\log_\sigma m)/m)$. ¿Qué puede decir entonces de la complejidad promedio del problema?

Solución:

(a) El siguiente adversario hace fallar a cualquier algoritmo que inspecciona menos de $k+1$ caracteres en una ventana de largo m. Dada una ventana V y un patrón P, ambos de tamaño m:

- El adversario comienza con la ventana vacía (en su memoria).
- Cada vez que el algoritmo pregunta por V_i (es decir, intenta inspeccionar el caracter), el adversario contesta con algún elemento del alfabeto distinto de P_i, escogido aleatoriamente.

Supongamos que el algoritmo realiza menos de $k+1$ consultas y responde (que existe o no un calce aproximado).

Si el algoritmo responde que encontró el calce, el adversario setea todos los caracteres no inspeccionados con valores distintos a los correspondientes en el patrón, y muestra que el algoritmo se equivocó (ya que P no ocurre en la ventana si sólo permitimos a lo más k errores).

En el caso en que el algoritmo diga que no existe un calce en la ventana, el adversario setea todos los caracteres no inspeccionados de V como los correspondientes de P. En ese caso, existe un calce aproximado en la ventana, pues a lo más k caracteres quedaron con errores.

De esta forma, el adversario muestra que cualquier algoritmo que inspeccione menos de $k+1$ caracteres no capaz de determinar si existe o no un calce aproximado en la ventana V (es decir, declarar si los caracteres $V[1]...V[m]$ forman *un* calce).

(b) Dado un texto T de largo n, podemos dividirlo en n/m ventanas de largo m. El adversario anterior fuerza a cualquier algoritmo a realizar $k+1$ inspecciones en cada ventana (vamos repitiendo el procedimiento anterior), para declarar si la ventana corresponde a un calce aproximado o no. Luego el algoritmo debe realizar al menos $\Omega(kn/m)$ inspecciones.

Noten que pueden existir muchas otras ventanas candidatas además de estas n/m, pero este argumento muestra que cualquier algoritmo debe tomar al menos el tiempo especificado para decidir acerca de las n/m ventanas (pudiera o no tomar más tiempo para las otras posibles ventanas, pero no afecta al argumento). Aquí está la duda que escribió su compañero sobre si el problema busca encontrar una o todas las ocurrencias.

(c) Sea g la complejidad del problema, $f_1 = kn/m$, $f_2 = log_\sigma(m)/m$. Del enunciado tenemos las cotas $\Omega(f_1)$ y $\Omega(f_2)$, y nos piden $\Omega(f_1 + f_2)$ para g. Ahora, $g \in \Omega(f)$ significa que $g > C \cdot f$ para algún C positivo. Más formalmente, tenemos:

$$g(m,n) \in \Omega(f_1(m,n)) \iff \exists C_1 > 0,\ \exists m_1, n_1\ \forall m > m_1, n > n_1,\ g(m,n) \geq C_1 \cdot f_1(m,n)$$
$$g(m,n) \in \Omega(f_2(m,n)) \iff \exists C_2 > 0,\ \exists m_2, n_2\ \forall m > m_2, n > n_2,\ g(m,n) \geq C_2 \cdot f_2(m,n)$$

y queremos llegar a

$$\exists C^* > 0, \exists m^*, n^*\ \forall m > m^*, n > n^*,\ g(m,n) \geq C^* \cdot f^*(m,n)$$

con $f^* = f_1 + f_2$. ¿Qué valores k^*, m^*, n^* usamos para que se cumpla la desigualdad? Primero, queremos usar las desigualdades para f_1 y f_2, así que usando $m^* = \max(m_1, m_2)$ y $n^* = \max(n_1, n_2)$ estamos bien. Ahora que sabemos que se cumplen las dos desigualdades, podemos sumarlas, despejando antes f_1 y f_2:

$$\begin{aligned}
\frac{g(m,n)}{C_1} &\geq f_1(m,n) \\
\frac{g(m,n)}{C_2} &\geq f_2(m,n) \\
\implies (\frac{1}{C_1} + \frac{1}{C_2})g(m,n) &\geq f_1(m,n) + f_2(m,n) \\
\implies g(m,n) &\geq \frac{1}{\frac{1}{C_1} + \frac{1}{C_2}}(f_1(m,n) + f_2(m,n))
\end{aligned}$$

Con lo que se vuelve evidente que elegir $C^* = (\frac{1}{C_1} + \frac{1}{C_2})^{-1}$ es un valor que hace válida la expresión que queríamos demostrar. Luego $g \in \Omega(f_1 + f_2)$.

Finalmente, como existe tanto una cota inferior $\Omega(f^*)$ como una cota superior $O(f^*)$ para g, se tiene que $g \in \Theta(f^*)$. En otras palabras, la complejidad del problema de búsqueda aproximada en texto está en $\Theta(n(k + \log_\sigma m)/m)$.

3 Memoria Secundaria

2. Considere una lista de N elementos, y una memoria principal de tamaño $M \in \Theta(1)$. Diseñe algoritmos eficientes para:

   (a) Encontrar un elemento que tenga la mayoría absoluta (es decir, que aparezca más de $\frac{N}{2}$ veces), y reportar su frecuencia. Si no existe tal elemento, debe reportarse `no`.

   (b) Encontrar k elementos que aparezcan más de $\frac{N}{k+1}$ veces, con $k < M$, suponiendo que existen.

Solución:

(a) Haremos un algoritmo que hace dos pasadas lineales por el archivo y encuentra el elemento mayoritario, si existe. Una forma de explicar la idea detrás del algoritmo que plantearemos es la siguiente:

Supongamos que en una votación participan N personas. Cada una puede votar por sí misma o votar en contra de otro de los participantes (restándole un voto, permitiendo una cantidad negativa de votos). Si estoy participando, lo peor que me puede pasar es que todos los demás participantes se pongan de acuerdo para votar en mi contra. Si tengo al más $N/2$ aliados (incluyéndome, claro), puedo anular todos esos votos: los que confabularon en mi contra quedarán con 0 votos y yo tendré al menos 1 (y ganaré :)).

El algoritmo, entonces, fuerza este tipo de votación entre los elementos, y registra el ganador.

```
Input: Un arreglo L con N elementos.
Output: El elemento m mayoritario, o no si no existe.
m ← NaN;
i ← 0;
for x ∈ L do
    if i = 0 then
        m ← x;
        i ← 1;
    else if x = m then
        i ← i + 1
    else
        i ← i − 1
    end
end
i ← 0;
for x ∈ L do
    if x = m then
        i ← i + 1
    end
end
if i > N/2 then
    return m
else
    return no
end
```

Supongamos que existe un elemento mayoritario m^*. Sea K el número definido (¡para el análisis, no para el algoritmo!) de la siguiente forma: igual a c si $m = m^*$ e igual a $-c$ en caso contrario. Notemos que K es positivo sólo si $m = m^*$.

Cada vez que se encuentra m^*, K crece en 1 (tanto si $m = m^*$ como si no); si aparece otro número, K puede crecer o disminuir en 1. Al final del algoritmo, dado que m^* es mayoritario,

el primer caso debe haberse dado más veces que el segundo, por lo que K debe quedar en un valor positivo. Pero esto sólo se da si $m = m^*$, con lo que el algoritmo debe haber encontrado el mayoritario.

En el caso en que no exista un mayoritario, el segundo ciclo del algoritmo se da cuenta de ello, con lo que responde `no` y preserva su correctitud.

Este algoritmo hace 2 pasadas por la entrada, con lo que toma tiempo $O(N/B)$.

(b) Había un **error de enunciado**. La idea es extender el algoritmo anterior; sin embargo, no podemos usar $O(1)$ de memoria y determinar los candidatos en una pasada por el archivo (como hacía el algoritmo anterior). De hecho, la extensión del algoritmo provee una solución *aproximada* al problema, lo que corresponde a materia que veremos después [1]. Así que posiblemente lo veamos después.

Lo que sí podemos hacer (para que esta parte no quede vacía) es mostrar por qué es imposible en ciertos casos. En particular, demostraremos que no existe forma de resolver el problema haciendo sólo una pasada por la entrada.

Sea $k = n/2$, con lo que es necesario reportar los valores que ocurren al menos dos veces. Supongamos que los primeros $N - 1$ elementos de L son distintos, y que ya hemos pasado por ellos. El único candidato a ser reportado es el último. Lo que estamos respondiendo esto, en este caso, es el problema de preprocesar un conjunto S y luego responder preguntas de la forma "¿pertenece y a S?". Es claro que para responder estas preguntas, es necesario almacenar S en alguna forma (si hay alguna parte de S que no se almacene, pueden luego preguntarnos por esa parte).

4. Se nos entrega un conjunto P de N puntos en $\mathbb{R}^d$ en *posición general*; es decir, ningún par de puntos comparte el mismo valor en alguna dimensión. Considere que $N \gg M$. Dado un punto $p \in \mathbb{R}^d$, llamaremos $p[i]$ a su i-ésima coordenada. Un punto p_1 *domina* a otro punto p_2 (denotado como $p_1 \prec p_2$) si se cumple $p_1[i] < p_2[i] \forall i = 1, ..., d$.

   Se nos pide calcular el *skyline* de P, denotado como $SKY(P)$, que incluye todos los puntos de P que no son dominados por ningún otro:

$$SKY(P) = \{p \in P | \nexists p' \in P, p' \prec p\}$$

   (a) Resuelva el problema para $d = 2$ usando $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}))$ operaciones de disco.

   (b) Resuelva el problema, ahora para $d = 3$. Llegue a la misma cota para la cantidad de operaciones de disco que en el caso anterior.

Solución:

(a) Como $d = 2$, tenemos puntos en dos dimensiones. Ordenamos los puntos por coordenada x, y luego por coordenada y (i.e. lexicográficamente). Consideremos un punto p, y sea P' el conjunto de los puntos que quedaron antes de éste. Es claro que p no puede ser dominado por ningún punto que venga después de éste (pues tendrán una coordenada x mayor que p).

Por otro lado, p será dominado por algún punto en P' si y sólo si la coordenada y de p es más grande que la del mínimo y en P'.

Luego el siguiente algoritmo resuelve el problema:

[1] Si les interesa ver el algoritmo y otras variantes que usan materia futura: `https://www.cs.utah.edu/~jeffp/teaching/cs5955/L12-Heavy-Hitters.pdf`

Input: Un arreglo P con N puntos en 2D.
Output: $SKY(P)$.
$y_{min} \leftarrow \infty$;
$out \leftarrow \emptyset$;
$\texttt{Sort}_{\text{lex}}(P)$;
for $i \in 1, ..., N$ **do**
  $(x, y) \leftarrow P_i$;
  if $y < y_{min}$ **then**
    $y_{min} \leftarrow y$;
    $out \leftarrow out \cup \{P_i\}$
  end
end
return out

Este algoritmo consiste en ordenar más una pasada (lineal) por el archivo, con lo que toma $O(\frac{N}{M} \log_{\frac{M}{B}} \frac{N}{B})$ operaciones de disco.

(b) En el caso de puntos en 3D, el algoritmo será recursivo. Primero, ordenamos de forma lexicográfica, como antes (este paso no es repetido en el caso de una llamada recursiva).

Si $N < M$, encontramos $SKY(P)$ en memoria. El costo en operaciones de disco es simplemente cargar los puntos en memoria: $O(\frac{N}{B})$

Si $N > M$, dividimos P en $s = \Theta(\frac{M}{B})$ particiones $P_1, ..., P_s$ de tamaño $\lceil \frac{N}{s} \rceil$ tal que cada punto en P_i tiene una coordenada x menor que las de los puntos en P_j, si $i < j$. Como el arreglo de puntos ya está ordenado, esto toma tiempo lineal. Luego invocamos el algoritmo de forma recursiva en cada P_i.

Forzaremos el siguiente invariante: dado que no se pide retornar $SKY(P)$ en un orden particular, los retornaremos ordenados según la coordenada y.

Cuando tenemos $SKY(P_i)$ para cada P_i, debemos mezclarlos para producir $SKY(P)$. Comenzamos con $SKY(P) = \emptyset$. Escaneamos todos los $SKY(P_i)$ de forma síncrona, avanzando en orden de la coordenada y (es decir, como si hiciéramos un merge). Como son $s = \Theta(\frac{M}{B})$ trozos, este merge se puede hacer manteniendo un bloque de cada trozo en memoria.

Mantenemos en memoria, además, un arreglo Z_{min}. Este arreglo contendrá, en su posición i, el mínimo valor de la coordenada z observada para P_i. Tenemos el siguiente lema:

$$p_i \in SKY(P) \iff \forall j < i,\ p_i[3] < Z_{min}[j]$$

donde p_i representa un elemento que fue obtenido desde P_i. La demostración del lema es como sigue.

- Claramente, p no puede ser dominado por ningún punto en $P_{i+1}, ...P_s$, ya que tiene una coordenada x menor que los puntos en estos conjuntos.
- Sea S el conjunto de puntos en P_j escaneados antes que p, para un $j < i$. Ningún punto en $P_j \backslash S$ (es decir, los que no han sido escaneados) puede dominar a p, pues p tiene una coordenada y menor que cualquier punto en este conjunto.
- Por otro lado, todos los puntos en S dominan a p en el plano xy: como $P_j \subset S$ con $j < i$, los elementos de S poseen coordenadas x menores a la de p; como todos los P_k están ordenados por coordenada y (y el merge respeta ese orden) los elementos ya escaneados tienen menor coordenada y que p.
- Luego, un punto en S domina, a p en $\mathbb{R}^3$ si y sólo si se mantiene la desigualdad (es decir, la coordenada z es la que decide).

Luego, al hacer el merge de los P_i, observamos si cada p cumple la desigualdad antes especificada, en cuyo caso agregamos p a $SKY(P)$. Este merge toma $O(\frac{N}{B})$ operaciones de disco.

Si $F(N)$ es el costo (en operaciones de disco) del algoritmo al aplicarse sobre N puntos, tenemos del análisis que hemos hecho que:

$$F(N) = \begin{cases} O(N/B) & \text{si } N \leq M \\ \sum_{i=1}^{s} F(|P_i|) + O(N/B) & \text{en otro caso} \end{cases}$$

donde $\sum_{i=1}^{s} |P_i| = N$, y $|P_i| \approx \lceil N/s \rceil$ (estrictamente, $\leq$). Esta recurrencia resulta (aplicando el Teorema Maestro) en $F(N) = O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$.

4 Propuestos anteriores

1. En este problema daremos una cota inferior de $\Omega(nlogn)$ para el problema de determinar si existe un par de elementos idénticos en un conjunto de n, en el modelo de comparaciones.

   (a) Suponga que los elementos son $a_0 \leq a_1 ... \leq a_{n-1}$. Definimos las comparaciones *adyacentes* como las comparaciones entre a_i y a_{i+1}, para cada $i \in [0, n]$. Muestre, utilizando un argumento de adversario, que cualquier algoritmo que determine si existe un par de elementos idénticos debe realizar todas las comparaciones adyacentes.

   (b) Muestre que para problema de ordenar n elementos $a_0 \leq a_1 ... \leq a_{n-1}$ es suficiente conocer el resultado de las comparaciones adyacentes. Es decir, muestre que existe un algoritmo que recibe n elementos, un conjunto de resultados (representados, por ejemplo, de la forma $(a, b, v), v \in \{-1, 0, 1\}$) de comparaciones que incluya todas las adyacentes, y que ordena los elementos.

   (c) Concluya, notando que el paso anterior es una forma de *reducción*.

Solución

(a) Supongamos que un algoritmo para el problema de determinar si n elementos son diferentes. Sea el siguiente adversario que obliga al algoritmo a comparar todos los argumentos adyacentes:

- El adversario elige n elementos tales que $a_0 < a_1 ... < a_{n-1}$ y responde todas las comparaciones usando estos elementos.
- Si el algoritmo termina y existe un par a_i, a_{i+1} tales que el algoritmo no los compara, el adversario observa la respuesta del algoritmo.
- Si el algoritmo responde que hay duplicados, el adversario muestra los elementos $a_0, ..., a_{n-1}$ y declara que el algoritmo se equivocó.
- Si el algoritmo responde que no hay duplicados, el adversario reemplaza a_{i+1} por a_i y muestra los elementos, declarando que el algoritmo se equivocó.

Este adversario funciona porque, dada la forma de la entrada que escoge, cualquier elemento comparado con a_i (excepto a_{i+1}) producirá el mismo resultado que si fuera comparado con a_{i+1}. Luego el reemplazo hecho en el último punto es perfectamente válido. Notemos que este adversario obliga al algoritmo a realizar estas comparaciones si los elementos de entrada son distintos, pero esto es suficiente para el siguiente paso.

(b) Tenemos un arreglo de n elementos diferentes $a_0, ..., a_{n-1}$ y deseamos ordenarlos. Utilizamos el siguiente algoritmo:

- Ejecutamos el algoritmo que determina si son todos diferentes o no, de una forma supervisada: cada vez que el algoritmo compara un par de elementos, anotamos cuáles comparó y qué resultado se obtuvo.
- Construimos un grafo dirigido en el que cada elemento a ordenar es un vértice. Luego revisamos el registro de las comparaciones: si se comparó a y b y se determinó que $a > b$, agregamos un arco de a hacia b en el grafo.
- Realizamos una búsqueda en profundidad para determinar el orden topológico del grafo y lo usamos para ordenar los elementos.

¿Por qué funciona el último paso? El grafo que se construye es conexo (pues todos los elementos adyacentes fueron comparados) y acíclico (pues no puede darse, por ejemplo, $a < b < ... < a$). Esto significa que un algoritmo de búsqueda en profundidad en el grafo puede encontrar un orden topológico de éste en tiempo lineal en el número de arcos y vértices. Este orden topológico corresponde al orden de los elementos del arreglo, por la misma forma de construcción del grafo.

(c) El primer paso del algoritmo anterior toma tiempo $Dif(n)$ (el tiempo de determinar si todos los elementos son distintos). El segundo paso también (es escanear la lista de comparaciones hechas). El tercer paso toma tiempo $O(Dif(n)+n)$ (pues es $O(|V|+|E|)$). Luego el algoritmo en su totalidad toma tiempo $O(Dif(n)+n)$. Si $Dif(n)$ tomara un tiempo (asintóticamente) menor a $n\log n$ tendríamos una contradicción, pues habríamos construido un algoritmo que viola la cota inferior para el problema de ordenar de $\Omega(n\log\, n)$, que conocemos.

Otras formas de ver este problema: `http://www.sciencedirect.com/science/article/pii/0022000079900540` ; `http://dimacs.rutgers.edu/~gkindler/ds02c/ex5_ans.ps`

2. Se tienen dos arreglos A y B de largo n almacenados en memoria secundaria. Si bien son diferentes, ambos contienen los enteros entre 1 y n. Se desea construir el arreglo C, dado por $C[i] = A[B[i]]$. Diseñe un algoritmo eficiente para esto, y analícelo.

3. Dadas dos listas crecientes de largos m y n, y suponiendo que $m < n$, diseñe y analice un algoritmo para unirlas en una única lista ordenada, que tome tiempo $O(m + m\log\, \frac{n}{m})$.

Solución

Sean X e Y las listas de tamaño n y m, respectivamente. El siguiente algoritmo logra la cota pedida:

- Dividimos X en m trozos de tamaño n/m.
- Definimos $s_1, s_2, ...s_m$ como los primeros elementos de cada trozo.
- Definimos i y j como 0. La idea es que i será un puntero en X y j un puntero en Y.
- En cada paso (mientras queden elementos de ambas listas) hacemos lo siguiente:
  - Si $s_i < y_j$, incrementamos i en 1 (¡avanzamos un bloque entero!)
  - Si no, hacemos una búsqueda binaria entre s_{i-1} y s_i para encontrar el lugar donde iría y_j en X. Esto nos dice que todos los elementos anteriores a esta posición ya pueden ser escritos en la salida, para luego escribir y_j. Luego incrementamos j en 1.

Recordando que estamos en el modelo de comparaciones, notemos que este algoritmo termina cuando i y j llegan m. Para incrementar i hasta m, es necesario caer en el primer caso del ciclo m veces, lo que requiere m comparaciones. Para que j llegue a m, es necesario que sucedan m búsquedas binarias del segundo caso, lo que toma $\log\, \frac{n}{m}$ comparaciones. Luego, en conjunto, el algoritmo toma $2m + m\log\, \frac{n}{m}$ en el peor caso, que corresponde a $O(m + m\log\, \frac{n}{m})$