Auxiliar #3 - Memoria Secundaria

28 de septiembre de 2020 - Bernardo Subercaseaux

Problema 1. (⋆⋆⋆) Asuma que conoce un algoritmo de ordenamiento en memoria secundaria con complejidad $O(n \log_m n)$. Dada una relación binaria $\mathcal{R} \subseteq \{1, \ldots, k\} \times \{1, \ldots, k\}$, representada como una secuencia de N pares (i, j), que se almacena en un archivo de memoria externa, diseñe algoritmos eficientes para determinar si $\mathcal{R}$ es:

1. Refleja
2. Simétrica

Solución 1 . La reflexividad requiere que para todo $i \in \{1, \ldots, k\}$ se encuentren presentes los pares (i, i). Para esto se puede mantener un contador de pares de la forma (i, i), e iterar linealmente sobre el archivo aumentando el contador al encontrar un par de esa forma. Finalmente se compara si el contador vale k o no, lo que determina la respuesta. Al requerir una lectura lineal del archivo, se realizan $O(N/B) = O(n)$ operaciones de I/O.

Para estudiar la simetría comparemos tres estrategias.

- Se itera linealmente por $\mathcal{R}$, y por cada par (i, j) se itera linealmente revisando si existe el par (j, i). Dado que por cada uno de los N pares se hace una lectura lineal, esto requiere $O(nN) = O(N^2/B)$ operaciones de I/O.
- Se construye $\mathcal{R}'$, un archivo que contiene el inverso de cada par en $\mathcal{R}$. Esto toma $O(n)$ operaciones de I/O. La pregunta es luego si $\mathcal{R}$ y $\mathcal{R}'$ tienen los mismos elementos. Para esto podemos ordenar ambos archivos, en tiempo $O(n \log_m n)$, y luego revisar si son idénticos (a través de una lectura lineal de ambos en tiempo $O(n)$). La complejidad total está dominada por el ordenamiento, resultando en $O(n \log_m n)$.
- En el caso en que k no sea muy grande, podemos construir un archivo $\mathcal{A}$ como matriz de adyacencia de $\mathcal{R}$. Es decir, por cada uno de los k^2 pares (i, j) posibles, $\mathcal{A}$ tendrá un 1 si ese par está en $\mathcal{R}$ y un 0 si no. Inicialmente se puede construir $\mathcal{A}$ solo con 0's y al iterar por $\mathcal{R}$ se introducen los 1's correspondientes. Esto tarda tiempo $O(k^2)$ (notando que necesariamente $N \leq k^2$). Luego, se realiza otra iteración por $\mathcal{R}$ en que por cada par (i, j) se va a buscar a $\mathcal{A}$ si está su par contrario. Dado que la posición del 0 o 1 que indica la presencia de tal par está fija, solo se realiza un acceso a $\mathcal{A}$, lo que resulta en $O(N)$ accesos extra. En total la complejidad es $O(k^2)$ operaciones de I/O. Si $N \sim k^2$ y $\log_m n >> B$ esto es mejor que la estrategia anterior.

Problema 2. (⋆⋆⋆) La idea de esta pregunta es permutar aleatoriamente un gran archivo que se posee en memoria externa.

1. Diseñe un algoritmo que dado un arreglo de n elementos en memoria principal, genera con probabilidad uniforme una de sus $n!$ permutaciones, en tiempo $O(n)$.
2. Diseñe un algoritmo que permute aleatoriamente (nuevamente con distribución uniforme sobre las permutaciones) un archivo en memoria externa de N elementos, con la hipótesis de que $N \leq M^2/B$

3. Resuelva el mismo problema pero sin la hipótesis anterior

Solución 2 . Es fácil ver que el siguiente algoritmo funciona:

1. Se tiene una lista de elementos. Inicialmente ningún elemento se considera *descartado*.
2. Se inicializa una lista A donde quedará la permutación. A comienza vacía.
3. Mientras queden elementos no descartados descartados:
   a) Se elige un elemento al azar entre los no descartados.
   b) Se añade el elemento elegido en el paso anterior al final de A, y se descarta de la lista original.

Este algoritmo se puede implementar en tiempo lineal si en lugar de usar una lista auxiliar A, se implementa sobre la lista original. Los elementos *descartados* se irán dejando al final de la lista. Por lo que cada vez que se elige un elemento para descartar, se permuta su posición para que quede al final de la lista. Más concretamente, se define un índice i inicializado en n (asumiendo indexación desde 1). En cada paso se elige un elemento al azar en $[1, i]$ y se permuta con el i-ésimo, luego se decrementa i y se repite el proceso hasta que $i = 1$.

Teniendo en cuenta la restricción $N \leq M^2/B$, podemos separar input en $m = M/B$ archivos, cada uno de tamaño a lo más M. Por cada archivo tenemos un buffer de tamaño B en memoria principal. Leemos los N elementos del archivo inicial, por bloques, y cada bloque lo distribuímos aleatoriamente entre los buffers. Cuando un buffer se llena se escribe en su archivo correspondiente y se vacía. Esto distribuye uniformemente los N elementos entre los archivos. Luego leemos cada uno de los archivos, permutamos el interior de cada archivo (usando el algoritmo de permutación en memoria principal), e imprimimos. Dado que los elementos del input ya están distribuidos uniformemente entre los archivos, al concatenar la permutación de cada uno se obtiene una distribución uniforme. Notando que el algoritmo consiste en unas pocas pasadas lineales por los datos, el costo total es $O(N/B) = O(n)$.

Al no tener la restricción, si dividimos el input en m archivos, estos podrán tener tamaño $> M$. Esto implica que no podemos permutar el interior de cada archivo como antes (cargando en RAM y permutando gratis ahí). Sin embargo, podemos proceder recursivamente, usando el algoritmo anterior para permutar cada uno de los m archivos, obteniendo así m^2 archivos. Siguiendo con esta idea sucesivamente, generamos m^i archivos, tomando i hasta que los sub-archivos creados si tengan tamaño $< M$, y por lo tanto puedan ser permutados en memoria principal. La profundidad de la recursión será de $O(\log_m N) = O(\log_m n)$. Dado que por cada nivel de la recursión solo se pasa linealmente por los datos, el costo total es $O(n \log_m n)$, que es el mismo de ordenar en memoria secundaria.