Análisis del Algoritmo Quicksort Aleatorizado

Miguel Oviedo

1. Resumen

En este trabajo explicamos la implementación del algoritmo de ordenamiento Quicksort, conocido por ser uno de los más eficientes y ampliamente usado para ordenar arrays de grandes dimensiones. Primero hacemos un análisis del algoritmo y luego analizamos la esperanza matemática del tiempo de implementación cuando se ingresan elementos de forma aleatoria y de igual probabilidad. Por último mostramos una implementación de dicho algoritmo (en jupyter) y medimos el tiempo en diversos casos de distribución de los elementos.

2. Introducción

2.1. Algoritmo Quicksort

El algoritmo de Quicksort se basa en el método de *conquista y vencerás* para ordenar los elementos de un array. El array es divido en dos, luego cada subarray es ordenado independientemente del otro. El punto de referencia para particionar el array es un elemento de éste, llamado **pivote**. Los elementos restantes del array se comparan con el pivote desplazándolos a los extremos según sean mayor o menor que éste, entonces, los elementos menores que el pivote se encuentran a la izquierda y los mayores a la derecha. Habiendo particionado el array se repite el proceso llamando recursivamente al quicksort para cada subarray.

2.2. Partición del vector

La partición es un procedimiento que toma un array y sus elementos extremos. Se fija a el último elemento como el pivote y se utilizan dos índices auxiliares: i y j, j recorre todo el array excepto en la posición del pivote, A[j] se compara con el pivote: si A[j] es menor o igual que x, i de desplaza una posición a la derecha y los elementos en las posiciones de los índices $i + 1$ y j se intercambian. Cuando

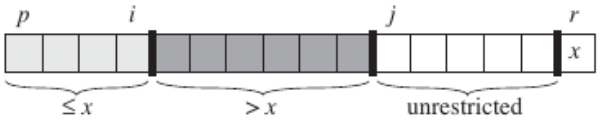

Figura 1: La zona ligeramente sombreada corresponde a los elementos menores o iguales que el pivote y la zona más sombreada a los elementos mayores que el pivote. La zona blanca corresponde a los elementos que todavía no han sido comparados con el pivote. Al final, el pivote se intercambia con *i + 1* quedando en medio de los dos subarrays.

j haya recorrido el vector hasta la posición *r - 1* el pivote se intercambia con el elemento en la posición *i + 1* que es el primer elemento mayor que el pivote.

PARTICION(A, p, r)

```
1  for j = p to r − 1
2  if A[j] ≤ x
3        i = i + 1
4        intercambiar A[i] con A[j]
5  intercambiar A[i + 1] con A[r]
6  return i + 1
```

PARTITION presenta un buble for el cual recorrerá el array $A[p..r]$, por lo tanto su tiempo de implementación será $\Theta(n)$, donde $N = r - p + 1$.

2.3. Algoritmo Quicksort

El algoritmo de quicksort es simplemente un procedimiento recursivo cuyos parámetros son el array y los extremos de cada subarray:

QUICKSORT(A, p, r)
1 **if** $p < r$
2 $\quad q =$ PARTICION(A, p, r)
3 $\quad$ QUICKSORT$(A, p, q-1)$
4 $\quad$ QUICKSORT$(A, p, q+1)$

La condicional evita que se analice más allá de los límites del vector, el procedimiento PARTICION nos devuelve el pivote, y las otras llamadas recursivas operan para subvectores de elementos menores y mayores que el pivote respectivamente.

2.4. Rendimiento del Quicksort

El rendimiento del quicksort depende de cómo están dispuestos inicialmente los elementos del array. Si la distribución es balanceada, entonces se asemeja al algoritmo de ordemaniento **mergesort** ($\Theta(nlg(n))$, pero si está desbalanceado su rendimiento será similar al **insertion sort** ($\Theta(n^2)$).

Se considera que el algoritmo de quicksort es muy ineficiente cuando los elementos están ordenados en sentido opuesto al que el algoritmo lo hace, es decir, se da cuando quicksort ordena elementos de forma creciente un array ordenado de forma decreciento (o viceversa), esto provoca una partición de un subarray de longuitud cero y otro de longuitud $n-1$. Este gran desbalance se repetirá en cada llamada recursiva pasando subvectores de longuitudes $n, n-1, n-2, ..., 2, 1$. Formalmente podemos expresar el análisis anterior con el **método de sustitución** para encontrar su complejidad.
Sea $T(n)$ la función que representa el peor caso del algoritmo quicksort con un array de tamaño n, entonces $T(n) = \max\limits_{0 \leq q \leq n-1}(T(q) + T(n-(q+1))) + \Theta(n)$ donde $T(q)$ y $T(n-(q+1))$ representan cada subarray y $\Theta(n)$ al rendimiento del procedimiento PARTICION. Supongamos que $T(n) \leq cn^2$, entonces

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1}(cq^2 + c(n-(q+1)^2) + \Theta(n) \\ &= c \cdot (\max_{0 \leq q \leq n-1}(q^2 + (n-(q+1)^2)) + \Theta(n) \end{aligned}$$

La expresión $q^2 + (n-(q+1)^2$ tiene segunda derivada con respecto a q igual a 4 ($0 \leq q \leq n-1$), lo que indica que alcanza un máximo. $\max\limits_{0 \leq q \leq n-1}(q^2 + (n-(q+1)^2) \leq (n-1)^2 = n^2 + 2n + 1$ Con esto tenemos una nueva cota para T:

$$\begin{aligned} T(n) &\leq cn^2 - c(2n-1) + \Theta(n) \\ &\leq cn^2 \end{aligned} \tag{1}$$

Por lo tanto $T(n) = O(n)$.
Por otro lado el mejor caso se da cuando el array de elementos está distribuido de tal manera que un subarray es de tamaño $\lfloor \frac{n}{2} \rfloor$ y el otro de $\lceil \frac{n}{2} \rceil - 1$. En este caso el tiempo computacional del quicksort es $T(n) = 2T(n/2) + \Theta$, el cual da una complejidad de $\Theta(nlg(n))$.

3. Quicksort Aleatorizado

Como ya se ha comentado, el rendimiento del quicksort depende de cómo están distribuidas los elementos en el vector. Para hacer un análisis del tiempo de implementación cuando se distribuyen aleatoriamente los elementos, solo se considera que el pivote puede ser cualquier elemento del vector tomado al azar, lo demás se mantiene inalterado. En resumen, lo que deseamos analizar es la esperanza del tiempo de implementación cuando el pivote se elige aleatoriamente y no a priori. En torno al algortimo, agregamos un procedimiento que tomará un pivote aleatorio y devuelve un nuevo array a PARTICION:

PARTICION_ALEATORIZADA(A, p, r)
1 i = RANDOM(p, r)
2 intercambiar $A[r]$ con $A[i]$
3 **return** PARTICION(A, p, r)

RANDOM escoge un elemento del array al azar y lo intercambia con el "pivote predeterminado", luego se pasa el nuevo array a PARTICION para que forme dos subarrays. El algoritmo de quicksort no cambia porque el proceso recursivo sigue siendo el mismo.

QUICKSORT_ALEATORIZADO(A, p, r)
1 **if** $p < r$
2 q = PARTICION_ALEATORIZADA(A, p, r)
3 QUICKSORT_ALEATORIZADO$(A, p, q - 1)$
4 QUICKSORTA_LEATORIZADO$(A, p, q + 1)$

Solo cambia el procedimiento PARTICION por PARTICION_ALEATORIZADA y el nombre de QUICKSORT a QUICKSORT_ALEATORIZADO con fines de claridad.

3.1. Tiempo de implementación esperado

El tiempo de implementación depende fundamentalmente de PARTICION. Cada partición del vector se escoge un pivote que no se repite en ningún otro subarray, entonces en el peor de los casos se escogen n pivotes y por lo tanto n llamadas a PARTICION. Otro factor que se debe analizar es el número de comparaciones $A[j] \leq x$ que se hace entre cada elemento de un vector y el pivote,

este punto es fundamental ya que esto determinará los futuros subarrays.
Sea $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ el conjunto de elementos de A comprendidos entre z_i y z_j, inclusive, entonces definimos X_{ij} como la variable aleatoria indicadora del número total de comparaciones que se hacen cuando se llama a PARTICION:

$$X_{ij} = I\{z_i \text{ es comparado con } z_j\}$$
$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

tomando esperanza en ambos lados y aplicando la propiedad lineal, tenemos

$$\begin{aligned} E[X] &= E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{P}\{z_i \text{ es comparado con } z_j\} \end{aligned}$$

Como comentamos líneas arriba, el pivote divide un array en dos subarrays que son totalmente independientes uno del otro, por lo tanto ningún elemento de un subarray se comparará con uno de otro subarray. Por ejemplo, si tenemos el array {9,3,7,8,5,4,6,2} y elegimos como pivote al 5, entonces cuando se llama a PARTICION obtenemos los subarrays {3,1,4,2} y {7,6,8,9}, vemos entonces que cuando $z_i < x < z_j$, z_i y z_j nunca se compararán. Por lo tanto, las futuras comparaciones se harán únicamente ente aquellos elementos del array que tengan posiciones menores al pivote y entre aquellos que tengas posiciones mayores al pivote. En nuestro ejemplo cuando se hizo la primera llamada a PARTICION se dió z_{25} y z_{58}, pero z_{28} no.
Por lo tanto, dos elementos son comparables si y solo si el primer elemento a elegir como pivote en Z_{ij} es z_i o z_j.

Debido a que cualquier elemento de Z_{ij} es igualemente probable de ser el pivote y la elección es independientemente de otra, la probabilidad de que un elemento sea el pivote es $\frac{1}{j-i+1}$, por lo tanto

$$\begin{aligned}\mathbb{P}\{z_i \text{ es comparado con } z_j\} &= \mathbb{P}\{z_i \text{ o } z_j \text{ es el pivote elegido de } Z_{ij}\} \\ &= \mathbb{P}\{z_i \text{ es el pivote elegido de } Z_{ij}\} \\ &+ \mathbb{P}\{z_j \text{ es el pivote elegido de } Z_{ij}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1}\end{aligned}$$

Haciendo un cambio de variables ($k = j - i$)

$$\begin{aligned}E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k+1}\end{aligned}$$

La última expresión se puede acotar mediante la serie armónica

$$\begin{aligned}&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\ &< \sum_{i=1}^{n-1} \int_{1}^{n} \frac{2}{k} dk \\ &= \sum_{i=1}^{n-1} (2lgn) \\ &= \sum_{i=1}^{n-1} O(lgn) \\ &= O(nlgn)\end{aligned}$$

de donde se concluye que el tiempo esperado es *O(nlgn)*.

4. Conclusión

- Según los cálculos obtenidos de diferentes pruebas no hay un punto de comparación computaciones para un rango de hasta cien mil elementos debido que la potencia de cómputo de una computadora éstandar es mayor, pero matemáticamente se puede extrapolar que el algoritmo de quicksort aleatorizado tiene tiempo de cómputo similar al del mejor caso.

Referencias

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2009.

[2] PKS Prakash, Achyutuni Sri Krishna Rao *R Data Structures and Algorithms*. Packt Publishing, 2016

[3] James Aspnes *Notes on Randomized Algorithms CPSC 469/569: Fall 2016*