Análise de desempenho do particionamento do algoritmo *Quicksort* para os métodos de *Hoare* e *Lomuto*

Eugenio Souza Carvalho[1], **Hugo Santos Piauilino Neto**[1]

[1]Departamento de Computação
Universidade Federal do Piauí (UFPI)
Teresina – PI – Brazil

{hugos94,eugeniucarvalho}@gmail.com

Abstract.

Resumo. *Este trabalho apresenta uma análise de desempenho do particionamento do algoritmo de ordenação Quicksort para os métodos propostos por Hoare e Lomuto [Cormen et al. 2009], além de apresentar um resumo geral sobre a história e funcionamento do algoritmo de ordenação.*

1. Introdução

2. *Quicksort*

2.1. Dimensão de Desempenho

Para uso prático, facilidade de implementação pode ser sacrificado em prol de eficiência. Em uma base teórica, podemos determinar o número de comparações de elementos e trocas para comparar o desempenho. Além disso, o tempo de funcionamento real será influenciado por outros fatores, como desempenho de *caches* e escalonamento de *threads*.

Como será mostrado abaixo, os algoritmos possuem comportamento semelhante em permutações aleatórias, exceto pelo número de trocas. Aqui, o método de *Lomuto* necessita de três vezes a mais que o de *Hoare.*

2.2. Número de Comparações

Ambos os métodos podem ser implementados utilizando $n - 1$ comparações para particionar um *array* de comprimento n. Isto é essencialmente ideal, uma vez que precisamos comparar cada elemento com o pivô para decidir onde colocá-lo.

2.3. Número de Trocas

O número de trocas é aleatório para ambos os algoritmos, dependendo dos elementos no *array*. Se assumirmos permutações aleatórias, ou seja, todos os elementos são distintos e cada permutação dos elementos é igualmente provável, podemos analisar o número esperado de trocas.

Como apenas a ordem relativa conta, assumimos que os elementos são os números $1, ..., n$. Isso faz com que a discussão abaixo se torne mais fácil pois a posição de um elemento e seu valor coincidem.

2.4. Método de *Lomuto*

A variável índice j escaneia o *array* completo e sempre que encontra um elemento $A[j]$ menor que o pivô x, a troca é realizada. Entre os elementos $1, ..., n$, exatamente $x - 1$ são menores que x, então nós teremos $x - 1$ trocas se o pivô for x.

A expectativa geral então resulta do cálculo da média de todos os pivôs. Cada valor em $\{1, ..., n\}$ tem a mesma probabilidade de se tornar pivô (especificamente probabilidade de $\frac{1}{n}$), então temos

$$\frac{1}{n}\sum_{x=1}^{n}(x-1) = \frac{n}{2} - \frac{1}{2} \tag{1}$$

trocas, em média, para particionar um *array* de comprimento n com o método de *Lomuto*.

2.5. Método de *Hoare*

Para este método, a análise é um pouco mais complexa. Mesmo fixando o pivô x, o número de trocas permanece aleatório.

Mais precisamente: os índices i e j correm um em direção ao outro até que eles se cruzem, que sempre acontece em x (por correção do algoritmo de particionamento de *Hoare*). Isto divide eficazmente o *array* em duas partes: a parte esquerda que é verificada pela variável índice i e uma parte direita que é verificada pela variável índice j.

Agora, uma troca é feita para cada par de elementos "fora do lugar", isto é, um elemento grande (maior do que x, que pertence a partição direita) que atualmente está localizado na partição esquerda e um elemento pequeno localizado na partição direita. Note-se que este par formado trabalha sempre para fora, ou seja, o número de pequenos elementos inicialmente na partição direita é igual ao número de grandes elementos na partição esquerda.

Pode-se mostrar que o número destes pares é hiper geometricamente distribuído $Hyp(n-1, n-x, x-1)$: para os $n - x$ maiores elementos nós aleatoriamente traçamos suas posições no *array* e temos $x - 1$ posições na partição esquerda . Por conseguinte, o número esperado de pares é $(n - x)(x - 1)/(n - 1)$ dado que o pivô é x.

Finalmente, nós tiramos a média de todos os valores dos pivôs para obter o número total esperado de trocas para o método de particionamento de *Hoare*:

$$\frac{1}{n}\sum_{x=1}^{n}\frac{(n-x)(x-1)}{n-1} = \frac{n}{6} - \frac{1}{3}. \tag{2}$$

Mais informações podem ser encontradas em [Wild 2013, Pág. 29].

2.6. Padrão de Acesso a Memória

Ambos os algoritmos usam dois ponteiros que escaneiam o *array* sequencialmente. Portanto, ambos possuem comportamento quase ideal.

2.7. Elementos iguais e Listas Ordenadas

A performance dos algoritmos diferem mais drasticamente para listas que não estão aleatoriamente permutadas.

Em um *array* já ordenado, o método de *Hoare* não realiza nenhuma troca, já que não existem pares mal posicionados, ao passo que o método de *Lomuto* realiza cerca de $\frac{n}{2}$ trocas.

A presença de elementos iguais requere cuidados especiais na utilização do algoritmo *Quicksort*. Considere um exemplo extremo onde um *array* é preenchido apenas com elementos 0. Para este *array*, o método de *Hoare* realiza um troca para cada par de elementos - configurando o pior caso para o particionamento de *Hoare* - mas i e j sempre encontram-se no meio do *array*. Assim, temos um particionamento ideal e o tempo total de execução permanece em $\mathcal{O}(n \log n)$.

O método de *Lomuto* possui comportamento pior para o *array* apenas com elementos 0: a comparação $A[j] <= x$ sempre irá retornar verdadeira, então serão realizadas trocas para todos os elementos. Entretanto piora: após o *loop*, sempre teremos $i = n$, então observamos o pior caso de particionamento, fazendo com que a performance seja degradada para $\Theta(n^2)$.

3. Resultados

O algoritmo *Quicksort* e os métodos de *Hoare* e *Lomuto* foram implementados utilizando a linguagem de programação C. O ambiente de desenvolvimento integrado (IDE - Integrated Development)

Windows 10 pro AMD Phenom(tm) II X4 B97 Processor 3.20 GHz Sistema operacional de 64 bits

DDR3

Linguagem C

Code::Blocks 13.12!

gcc (TDM-2 mingw32) 4.4.1 2009

4. Conclusão

O método de *Lomuto* é simples e de fácil implementação, porém, não deve ser utilizado quando alto desempenho é exigido.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Wild, S. (2013). Java 7's dual pivot quicksort. Master's thesis, Technische Universitat Kaiserslautern.