MO417 Complexidade de Algoritmos - Ata

Redatora: Sheila Maricela Pinto Cáceres

abril de 2009

Problema 8-2: Ordenação local em tempo linear

Vamos supor que temos um arranjo de n registros de dados para ordenar e que a chave de cada registro tem o valor 0 ou 1. Um algoritmo para ordenar tal conjunto de registros poderia ter algum subconjunto das três características desejáveis a seguir:

1. O algoritmo é executado no tempo $O(n)$.
2. O algoritmo é estável.
3. O algoritmo ordena localmente, sem utilizar mais que uma quantidade constante de espaço de armazenamento além do arranjo original.

a. Dê um algoritmo que satisfaça aos critérios 1 e 2 anteriores.
Solução: *Counting Sort*.

b. Dê um algoritmo que satisfaça aos critérios 1 e 3 anteriores.
Solução:
Poderia ser usado o algoritmo *PARTITION* do *QuickSort*.
Se considerarmos o pivô com valor 0, por exemplo, então o algoritmo vai pôr todos os elementos menores ou iguais a 0 (todos os 0) na esquerda e todos os valores maiores que 0 (todos os 1) na direita. Assim, o algoritmo ordena o vetor, é local e é executado em tempo $O(n)$. Cabe ressaltar que não é preciso se preocupar pela correta posição do pivô, dado que são somente dois valores possíveis: 0 e 1.

c. Dê um algoritmo que satisfaça aos critérios 2 e 3 anteriores.
Solução: *Insertion Sort*.

d. Algum dos seus algoritmos de ordenação das partes (a)-(c) pode ser usado para ordenar n registros com chaves de b bits usando *Radix Sort* no tempo $O(bn)$? Explique como ou por que não.
Solução: O algoritmo *Radix Sort* utiliza um algoritmo de ordenação estável como passo intermediário, e portanto, esse algoritmo determina seu tempo final.
Por restrição do problema, é sabido que o tempo do algoritmo intermediário a ser usado deve ser linear. O ***Counting Sort*** é o único dos algoritmos das partes (a)-(c) que cumpre estas condições. Como cada número do conjunto a ser ordenado é 0 ou 1, tem-se números de 1 digito que ocupam b bits onde $b = 1$. Portanto, segundo o Lema 8.4, o *Radix Sort* ordena n números de b bits em tempo $\theta((b/r)(n + 2^r))$, para $r \leq b$. Considerando que $r = 1$, então o tempo usado é $\theta((b/1)(n + 2^1))$ ou seja $O(bn)$.

e. Suponha que os n registros tenham chaves no intervalo de 1 a k. Mostre como modificar a ordenação por contagem de tal forma que os registros possam ser ordenados localmente no tempo $O(n+k)$. Você pode usar o espaço de armazenamento $O(k)$ fora do arranjo de entrada. Seu algoritmo é estável? (Sugestão: Como você faria isso para $k = 3$?)

Solução: O Algoritmo por contagem poderia ser modificado como é mostrado a seguir:

Algoritmo 1: "Modificação do Algoritmo por Contagem"

```
1  for i <- 1 to k do
2          C[i] <- 0
3          pos[i] <- 0
4  for j <- 1 to n do
5          C[A[j].key]<-C[A[j].key]+1
6  for i <- 1 to k do
7          C[i]<-C[i]+C[i-1]
8  index<-n
9  while index >= 0 do
10         valor <- A[index].key
11         if(index <= C[valor] and index > C[valor]-pos[valor]) then
12                 index -> index-1
13         else
14                 destino <- C[valor]-pos[valor]
15                 exchange A[index] <-> A[destino]
16                 pos[valor] <- pos[valor]+1
```

Além do vetor de entrada, somente é permitido ocupar espaço de armazenamento $O(k)$. Por tanto para que o algoritmo seja local não pode ser usado o vetor B do *Counting Sort* dado que tem tamanho n. Neste algoritmo o vetor C assim como o vetor *pos* têm tamanho k.
As três primeiras linhas inicializam os vetores C e *pos*. Nas linhas 4-5, é contado o número de vezes que cada elemento do vetor de entrada A se repete, sendo armazenada a resposta no vetor C assim como acontece no *Counting Sort*. Do mesmo jeito, as linhas 6-7 cumprem a função de inserir em C[i] o número de elementos menores ou iguais a i. A linha 8 inicializa a variável *index* com o número de elementos. O *while* das linhas 9-16 definem o foco da solução do problema. A variável *index* vai percorrendo o vetor A e cada elemento $A[index]$ é colocado na posição final correta ordenada do mesmo vetor A. Na linha 10 a variável *valor* é inicializada tendo entre os seus valores possíveis de 1 até k. O if da linha 11-12 decrementa o *index* e mantém o conteúdo de $A[index]$ se ele está localizado na posição certa (entre $C[valor]$ e $C[valor]-pos[valor]$), caso contrario, nas linhas 13-16 o conteúdo de $A[index]$ é trocado com $A[destino]$ onde $destino = C[A[index].key]-pos[A[index].key]$ até que o elemento certo encontra a sua posição no vetor.
O vetor *pos* vai armazenando em cada uma das suas posições, o número de vezes que cada elemento $pos[valor]$ foi posto na posição correta no vetor A. Ao finalizar o algoritmo, o vetor *pos* terá em cada posição o numero total de vezes que o elemento *valor* apareceu no vetor A.
O algoritmo é linear e não é estável.

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill, 1990.