MO417 – Complexidade de Algoritmos

Nathana Facion - RA 191079 - Lista 4

10/04/2017

Questão 1.

R: A principal característica do k-way, ser usado para ordenar arquivos muito grandes, ou seja, é um algoritmo de ordenação externa. A ordenação externa ocorre devido os dados não couberem na RAM, em vez disso eles devem residir na memória externa, o disco rígido. Uma grande vantagem desse algoritmo em relação a versão 2-way é que a mesclagem pode ocorrer em um número menor de passos. Isso faz com que o acesso a disco seja realizado menos vezes, e como a memória em disco é mais lenta, quanto mais evitarmos o acesso a ela, melhor a solução. Neste algoritmo ocorre a geração das corridas (cada arquivo ordenado é uma subcorrida) envolve apenas acesso sequencial aos arquivos. Por meio de algoritmos como heapsort que podem ser utilizados para trabalhar em memória ou disco ou fita com um custo de um pequeno overhead, é possível criar um subconjunto ordenado do arquivo através do seguinte processo:
1) Leia os registros da memória até lotar.
2) Ordenar esses registros em memória (RAM), por meio de ordenação interna.
3) Escreva os registros ordenados em sub-arquivo (ordenado)
4) Repete-se os passos acima até encerrar o arquivo original. Se a RAM disponível comporta (1/k * no. de regs. do arq. original), então ter-se-ão k arquivos ordenados
5) Aplica-se então multi-way merging nos arquivos ordenados.

Para realização do exemplo abaixo foi usado a referência [1]. Como no nosso enunciado, suponha que você tenha uma caixa-preta. Considere que esses arquivos sejam de 80MB e a memória de 1MB. Então temos um 80-way e que o tempo final de execução é 9 minutos. Desse tempo de execução a maior parte do tempo é passado realizando acesso ao disco, ou seja, a intercalação usa a menor parte do tempo. Suponha que os arquivos sejam de 800 MB. O problema encontrado nesse caso é que o arquivo aumenta mas a memória não. Então teremos de aumentar o número de corridas, e continuaremos com 1MB de memória. Teremos problema de tempo para esse caso, pois demorará 36 vezes a mais que o caso anterior. A diferença ocorre devido o tempo de acesso as corridas. Como estamos com K corridas, gastando K seek, cada uma das operações requer K^2 seeks, o que implicaria em $O(n^2)$. Para reduzir esses problema é necessário: 1) Usar mais hardware; 2) Aumentar o tamanho das corridas iniciais ; 3) Realizar a intercalação em mais de uma etapa, o que reduz a ordem de cada intercalação, aumentando o tamanho do buffer para cada corrida, portanto, transfere mais registros com um único seek.

Outro algoritmo de ordenação externa que citaremos aqui é o keysort[2][3]. A ordenação de um arquivo em disco torna-se complicada justamente porque os arquivos estão em disco e não em RAM. Entretanto suponha que você tenha identificadores únicos de 1 a n desses arquivos. E você quer apenas classifica-los (ordenar) então precisamos das chaves (identificadores). Então se todas as chaves se encaixam na memória principal:
1) Trazer as chaves para a memória principal mais identificador único correspondente.
2) Fazer a ordenação interna das chaves.
3) Reescrever o arquivo de forma ordenada. Então em termos de esforço precisamos ler o arquivo sequencialmente uma vez, percorrer cada arquivo em ordem aleatória e escrever cada registro uma vez (sequencialmente)

O indice é ordenado em memória RAM e os registros aparecem no arquivo na ordem em que foram inseridos. O índice permite localizar registros rapidamente, sem necessidade de reorganizar o arquivo de dados quando novas informações são inseridas. Basta reorganizar o índice . Para buscar esses arquivos depois podemos usar pesquisa binária no indice para encontrar o id que nos leva ao campo referenciado. Então basta procurar o registro na posição referenciada. Como a ordenação ocorre em RAM ela é muito mais rápida, o acesso em disco também é reduzido.

Referência:
[1] http://wiki.icmc.usp.br/images/1/1e/SCC0203-1o-2012-15.OrdenacaoExterna.pdf
[2] http://web.itu.edu.tr/bkurt/Courses/blg341/lecture8.pdf
[3] http://www.ic.unicamp.br/ thelma/gradu/MC326/2010/Slides/Aula04b-ordenacao.pdf

Questão 2.

a) Seja D o vetor da árvore d-nária. Seja j um índice desta árvore. Como cada pai tem d filhos, então seja j um filho, o seu pai será: $Pai(j) = \lfloor \frac{j-2}{d} + 1 \rfloor$.

Os filhos serão representados por : $Filho(j,i) = (j-1)d + i + 1$. Cada vez que mudamos o i temos outro filho de j. Onde i é um número que está entre 1 e d $(1 \leq i \leq d)$.

Demonstração:

É fácil de ver que heap binária é um tipo especial de um heap d-ário em que d=2, se você substituir d com 2, então você verá que a fórmula é similar a do livro.
Para demonstrar que as fórmulas são válida basta observarmos as seguintes provas.

Vamos mostrar que o pai do filho é o pai: $Pai(Filho(j,i)) = j$

$Pai(Filho(j,1)) = \frac{(j-1)d+i+1-2}{d} + 1 = \frac{(j-1)d+i+1-2+d}{d}$

$= \frac{dj-d+i+1-2+d}{d} = \frac{dj+i+1-2}{d} = \frac{dj+1+1-2}{d}$

$= \frac{dj}{d} = j$

Para i diferente de 1 , teremos j + $\lfloor (\frac{i-1}{d}) \rfloor$. Como i é no máximo d, este valor nunca chegará ao valor 1, assim $\lfloor (\frac{i-1}{d}) \rfloor$ = 0 e continuamos com $Pai(Filho(j,i)) = j$.

Vamos mostrar que o filho do pai é o filho: $Filho(Pai(j),i)) = j$

$Filho(Pai(j),1)) = (\frac{j-2}{d} + 1 - 1)d + 1 + 1 = (\frac{j-2}{d})d + 2$

$= (j-2) + 2 = j$

Conseguimos encontrar os filhos variando o i.

b)

Algoritmo:

Algorithm 1 Heap máximo d-nário

```
1:  procedure MAXHEAP(A, i, n, d)
2:      maximo ← i
3:      for k = Filho(i, 1) to Filho(i, d) do
4:          if k ≤ n and A[k] > A[maximo] then
5:              maximo ← k
6:      if maximo! = i then
7:          aux ← A[i]
8:          A[i] ← A[maximo]
9:          A[maximo] ← aux
10:         maxHeap(A, maximo, n, d)
```

Complexidade:

O MaxHeap do d-ário é um algoritmo diferente do heap-binário. Como visto acima, algumas modificações foram realizadas. Em relação a complexidade, cada nível executa o loops d vezes, e esses, se necessário, são executados até a base da árvore.

A necessidade de ir até a base para comparar o nó M com todos os nós, com cada um dos seus filhos, isso é preciso para que seja respeitado a regra de que o pai tem que ser maior que o filho para ser considerado um heap. Ou seja, essa verificação é novamente necessária quando realizada uma troca entre nós para garantir que todos os filhos do nó M são menores do que ele, garantindo a ordem que os nós devem aparecer na árvore.

Então é verificado o nó de maior valor entre os filhos, se o mesmo for maior que o o nó M então é realizado a troca entre os dois nós e é feito uma recursão no filho. Como mostrado por meio da descrição acima a complexidade é $O(d * \log_d n)$.

Corretude:

Base : considera h = 0, não há caminho para as folhas, então está provado.

Hipótese de indução: Considere que para Max heaps de altura < h o algoritmo funciona.

Passo de indução: O máximo guarda o maior elemento entre os filhos. Realizamos trocas entre os elementos nas linhas de 7-9. Teremos uma subárvore com raiz maior(variável máximo) em um max-heap, podemos afirmar isso pois a hipótese é aplicada. Nas outras subárvore a propriedade do heap se mantem. Com isso podemos afirmar que a subárvore com raíz i é um max-heaps. E assim está provado.

Algoritmo:

Algorithm 2 Build Heap

```
1: procedure BUILDHEAP(A, n, d)
2:     for i = ⌊((n − 2)/d) + 1⌋ descresce até 1 do
3:         maxHeap(A, i, n, d)
```

Complexidade:

Na árvore d-aria com n items, para realizar a verificação se um nó precisa mudar de lugar com um nó filho é necessário fazer d operações, $d-1$ comparações para encontrar o maior filho do nó e mais uma operação para verificar a necessidade de troca (filho maior que o pai, propriedade definida pelo heap)

Portanto, o tempo para deletar o item raíz, e ordenar os valores dos filhos é $O(dlogn/logd)$. Quando criar um heap d-ario de um conjunto de n items, existem itens que estão em posição de folhas de uma árvore d-aria, e nenhuma troca para baixo é realizada nesses items.

Consideramos então que no máximo $n/(d+1)$ items são não-folhas, e podem ser trocadas para baixo pelo menos uma vez, a um custo de O(d) para encontrar o filho e trocá-los. Então, em no máximo $n/(d^2+1)$ nós podem ser trocados para baixo duas vezes, adicionando um custo de $O(d)$ para a segunda troca, além do custo já contado no primeiro termo. Portanto, o tempo total para se criar um heap dessa forma é $O(n)$.

$\sum_{i=1}^{\log_d n} \left(\frac{n}{d^i} + 1\right) = O(n)$

Corretude:

Inicialização: Antes da primeira iteração temos $i = \lfloor((n-2)/d)+1\rfloor$. Cada nó $\lfloor((n-2)/d)+1\rfloor$, $\lfloor((n-2)/d)+1\rfloor$ + 1, $\lfloor((n-2)/d)+1\rfloor$ + 2... n é uma folha e portanto a raíz de um heap máximo.

Manutenção: Para ver que cada iteração mantém loop invariante, vemos que os filhos do nó são numerados com valores mais altos que i. Assim, pelo loop invariante, ambos são raízes de heaps máximos. Isso é uma condição necessária para ser um heap, o max heap torna o nó i a raiz de um heap máximo. O max heap, portanto mantém a propriedade que os nós i+1, i + 2.. n são todos raízes de heaps máximos.

Término: Quando i = 0 , de acordo com o loop invariante o vetor estará com os nós 1,2..n com raízes de heap máximo. Assim temos um build max heap com o heap que respeita as propriedades necessária para ser um heap.

Questão 3.

Intuição:

Se k=1, retornamos uma lista vazia
Podemos dividir em subproblemas, dependemos do k:

- Considere k par, com isso encontramos a mediana, particionamos em volta dela, resolvemos 2 subproblemas similares de tamanho $\lfloor n/2 \rfloor$ e retornamos suas soluções mais a mediana.
- Considere que k ímpar, então diminuimos para 2 subproblemas, cada um com tamanho menor que n/2.

Algoritmo

Algorithm 3 Quantiles

```
1:  procedure QUANTILES(A, k, Q, ix)
2:      if k==1 then
3:          return nil, ix
4:      else
5:          n ← A.Comprimento
6:          i ← ⌊k/2⌋
7:          quantile ← Select(A, ⌊i∗n/k⌋)
8:          Partition(A, quantile)
9:          q1,ix ← Quantiles(A[1...⌊i∗(n/k)⌋], i, Q, ix)
10:         if q1! = nil then
11:             Q[ix] ← q1
12:             ix ← ix+1
13:         q2, ix ← Quantiles(A[⌊i∗n/k⌋+1...n], ⌈k/2⌉, Q, ix)
14:         if q2! = nil then
15:             Q[ix] ← q2
16:             ix ← ix+1
17:     return quantile, ix
```

Obs: Q está sendo acessado externamente por referência de fora. O último quantil é adicionado por fora da função.

Corretude:

Base : Considere k = 1, retornamos uma lista vazia. Está provado.

Hipótese de indução: Considere que para quantidade de quantiles < k e quantidade de elementos < n o algoritmo funciona.

Passo:Seja k a quantidade de quantiles . E seja n a quantidade de elementos. Na primeira chamada da função, encontramos primeiro o quantil que divide o vetor pela metade, e o salvamos na variável quantile. Com esse valor, realizamos a partição do vetor usando este quantile como pivô. Então, recursivamente, chamamos a função para encontrar o quantil q1 que divide a parte do vetor menor que o valor de quantile, e o quantil q2 que divide a parte maior. Pela hipótese de indução, conseguiremos encontrar esse valores. Ao final, adicionamos q1 e q2 (se não forem vazios) ao vetor Q, e retornamos o valor encontrado de quantile. Ao sair de todas as chamadas recursivas e terminar a função, devemos inserir o último quantile retornado para o vetor Q.

Complexidade:

Seja n o tamanho do vetor e k a quantidade de quantis. Usando o SELECT E PARTITION para encontrar o i-ésimo menor termo, temos uma complexidade $\theta(n)$. Então reduzimos o problema em dois subproblemas menores n/2 . Então temos um $T(n,k) = 2T(n/2, k/2) + cn$.

$$T(n,k) = \begin{cases} \theta(1), & k = 1. \\ 2T(n/2,k/2) + \theta(n), & k > 1. \end{cases} \quad (1)$$

Na primeira execução, queremos encontrar o primeiro quantile, que terá um custo de O(n) pelo uso dos algoritmos SELECT e PARTITION. Depois, a função é chamada recursivamente 2 vezes, para encontrar os $\lfloor (k/2) \rfloor$ e $\lceil (k/2) \rceil$ quantis, a um custo também de O(n). Como cada nível da recursão irá chamar o método para um valor de $k/2$, então teremos que a profundidade da árvore de recursão do algorítmo será dado por $log(k-1)$.

Provando por substituição a complexidade:
T(n) $\leq$ cnlogk
$\leq$ 2T(n/2,k/2) + dn
$\leq$ (2c(n/2)log(k/2) + dn
$\leq$ cnlog(k/2) + dn
$\leq$ cnlogk - cn + dn

Precisamos de : - cn + dn $\leq$ 0

Considerando cn - dn $\geq$ 0, logo $c > d$ e n $\geq$ 1 então temos O(nlogK).

Questão 4.

R:

Algorithm 4 Convidar os estudantes

```
1:  procedure CONVIDARALUNOS(A)                          ▷ Considere que A está ordenado pela idade
2:      maisNovo, maisVelho ← A[1], A[A.comprimento]     ▷ sabemos onde temos o mais novo e mais velho
3:      encontrouMaisNovo ← false                        ▷ Detecta se mais novo foi sorteado
4:      encontrouMaisVelho ← false                       ▷ Detecta se mais velho foi sorteado
5:      convite ← 0                                      ▷ Quantidade de convite distribuido
6:      maior ← 1                                        ▷ mais velho até agora
7:      menor ← A.comprimento                            ▷ mais novo até agora
8:      while not(encontrouMaisNovo and encontrouMaisVelho) do
9:          i ← random(1...menor ∪ maior...A.comprimento)
10:         if A[i].convite! = true and (A[i].idade = maisNovo or A[i].idade = maisVelho) then
11:             A[i].convite ← true
12:             if A[i].idade = maisNovo then            ▷ verifica se o sorteado é o mais novo
13:                 encontrouMaisNovo ← true
14:             else
15:                 encontrouMaisVelho ← true
16:         else
17:             if A[i].convite = true then              ▷ sorteado já havia sido sorteado anteriormente
18:                 convite ← convite − 1                ▷ convite eh adicionado novamente abaixo
19:             else
20:                 A[i].convite ← true                  ▷ se não tinha convite, agora tem
21:         if convite == 0 then                         ▷ guarda valor da primeira iteração p/ menor e maior
22:             maior ← i
23:             menor ← i
24:         else
25:             maior ← max(maior, i)                    ▷ maior valor p/ diminuir o intervalo do problema no random
26:             menor ← min(menor, i)                    ▷ menor valor p/ diminuir o intervalo do problema no random
27:         convite ← convite + 1                        ▷ aumenta o número de convites distribuidos
28:     return convite
29:
```

Melhor e pior caso:

O melhor caso ocorre quando os alunos mais novo de todos e o mais velhos de todos são sorteados nas 2 primeiras vezes, pois assim todos os outros alunos comprarão convites. O pior caso é quando os alunos citados anteriormente são os últimos a serem sorteados, com isso todos terão sido sorteados e nínguem comprará convite.

Corretude:

A cada iteração do algoritmo conhecemos um novo aluno que vai ganhar convite, e os alunos que tiverem entre o mais novo e o mais velho atuais comprarão convites.

Inicialização: Antes de iniciar o while nenhum aluno decidiu comprar convites e nenhum ganhou convites.

Manutenção: Ao começar o laço temos uma aletoridade de 1...menor ∪ maior...A.comprimento, que é usado para sortear os convites . A cada sorteio temos um i o qual verificamos se é o mais velho de todos ou mais novo de todos. Caso não seja nenhum dos dois, verificamos se o mesmo já tem convite, se não tiver, damos convite a ele. Na primeira rodada o menor e maior é atualizado com o mesmo número para os dois, e adicionamos um ao contador de convite.

O mesmos passos acima ocorre na segunda iteração. Entretanto como mais de um aluno foi sorteado, quem estiver entre o mais novo e mais velho atual já decidiu ir, portanto não faz mais parte do sorteio, assim diminuimos as

possibilidades(conjunto) de sorteio para a próxima interação. A cada sorteio verificamos se é a variável menor ou maior que é atualizada: $maior \leftarrow max(maior, i)$ e $menor \leftarrow min(menor, i)$, assim sabemos qual dos intervalos entre 1...menor $\cup$ maior...A.comprimento deve se diminuir . Esses valores (maior e menor) são usados na próxima iteração para excluir o intervalo de quem já comprou o convite(entre o mais novo atual e o mais velho atual). O i que foi sorteado, recebe o convite. E o número de convites é atualizado.

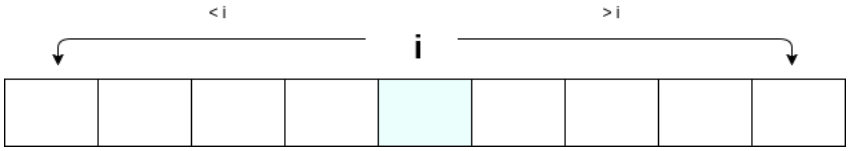
Término: Após finalizar o laço sabemos que o mais velho de todos e o mais novo de todos foram sorteados e que todos vão ter convites (alguns por comprarem outros por ganharem). Sabemos que o algoritmo para pois o intervalo de sorteio é reduzido a cada iteração. Assim, em algum momento, os alunos sorteados serão o mais novo e mais velho de todos.

Suponha por contradição que o algoritmo parou, e o aluno mais novo de todos ainda não foi sorteado. Pela condição de parada, um aluno mais novo foi sorteado, e que todos mais velhos que ele são convidados. Porém, o maisNovo ainda não foi sorteado, Logo, temos uma contradição pois significa que pelo menos o maisNovo não foi convidado para o jantar, e portanto, faltariam sorteios a ser feitos

Complexidade:

A forma de entender a complexidade deste algoritmo é similar ao do quicksort aleatório.

Seja A um vetor ordenado. Vamos resolver primeiramente para o caso do aluno mais velho, ou seja, um subproblema do original. Então considere que o i foi sorteado, com isso vamos verificar apenas o lado do vetor que contem os elementos maiores que i.



Seja X uma variável aleatória binária, pois podemos ter os números do vetor A comparados com i ou não comparados. De forma mais clara apenas os números maiores que i serão comparados, pois os mais novos não importam. Assim temos a somatória abaixo:

$\sum_{i=1}^{n-1} \frac{1}{(n-i+1)}$

$< \sum_{i=1}^{n} \frac{1}{i}$

$= O(lgn)$

O subproblema do mais novo é similar, assim o resultado é o mesmo. No problema original temos que considerar o mais novo e o mais velho e ignorar o que está entre essas idades. Então seria $Pr\{X = i\} = Pr\{A\} \cup Pr\{B\} = Pr\{A\} + Pr\{B\} - Pr\{A \cap B\}$. Os convites que estão na intersecção entre as probabilidades devem ser retirados pois senão excederemos o número de convites.

Temos então que esse resultado é menor que lgn + lgn + 1 , já que temos os dois subproblemas de mais velho, mais novo e o elemento i. Podemos concluir que será menor que $2lgn + O(1)$. Logo, assintoticamente temos: $O(lgn)$