MO417 – Complexidade de Algoritmos

Nathana Facion - RA 191079 - Lista 5

20/04/2017

Questão 1.

R:

Definição

A entrada é uma sequência de palavras com tamanhos l_1,l_2,...,l_n medidos em caracteres. Queremos imprimir esse parágrafo de forma elegante em um número de linhas que cabem no máximo M caracteres. Para isso, assumiremos que nenhuma palavra mais longa caberá em uma linha, isto é, $\forall$ i temos $l_i \leq$ M .

Abaixo temos algumas considerações que foram feitas para solução do problema.

Número	Variável	Definição
1	$extraSpace[i,j]$	$M - j + i - \sum_{k=i}^{j} l_k$
2	$rowCost[i,j]$	$\begin{cases} \infty, & if extraSpace[i,j] < 0. \\ 0, & if\, j = n \text{ and extraSpace[i,j]} \geq 0 \text{ (última linha custo zero)} \\ \text{(extraSpace[i,j])}^3, & \text{outros} \end{cases}$ (1)

1) Número de espaços extras no final da linha que contem as palavras de i até j.

2) Custo de incluir uma linha contendo palavras de i até j. Este é o valor que queremos minimizar. Quando temos extraSpace[i,j] < 0, adicionamos o valor ∞ para que este conjunto não faça parte da soma minimal. O zero é atribuido quando temos que $j = n$ e quando $extraSpace[i,j] \geq 0$, pois nesse caso as palavras são menores que M, este caso só é aplicado a última linha.

Fazendo o custo da linhas ser infinito quando as palavras não cabem nela, nós prevenimos tal conjunto de fazer parte da soma minimal, e fazendo o custo 0 para a última linha (se as palavras couberem), prevenimos que o conjunto da última linha de infuenciar a soma sendo minimizada.

Subestrutura ótima:

Considere uma solução ótima de exibir as palavras 1 até n. Seja i o índice da primeira palavra da última linha dessa solução. Então o conjunto das palavras 1,...,i-1 deve ser ótima. Se não fosse, poderíamos conseguir um conjunto ótimo para essas palavras, melhorando o custo total da solução. Com isso teriamos uma contradição.

Seja k o valor de uma linha qualquer. A mesma argumentação é válida se considerarmos i como sendo o índice da primeira palavra na k-ésima linha, onde 2$\leq$k$\leq$n. Portanto, esse problema apresenta subestrutura ótima.

Custos das palavras:

Considere um conjunto ótimo para as palavras 1,..,j. Suponha que sabemos que a última linha, que termina com a palavra j, começa com a palavra i. As linhas anteriores, portanto, contém as palavras 1,...,i-1. De fato, elas devem conter um conjunto ótimo das palavras 1,...,i-1.

Seja wordCost[j] o custo de um arranjo ótimo das palavras 1,...,j.

Número	Variável	Definição
1	wordCost[j]	$\begin{cases} 0, & if\, j = 0. \\ \min_{(1 \leq i \leq j)}(wordCost[i-1] + rowCost[i,j]) & if\, j > 0 \text{ (*)} \end{cases}$ (2)

(*) Considere que a última linha contém as palavras i...j.

1) Seja $wordCost[j]$ um vetor responsável de descobrir as palavras que iniciam a última linha para o subproblema das palavras 1,...,j. Para isso tentamos todas as possibilidades para a palavra i. A de menor custo é a escolhida.

Seja *wordLocation* uma tabela que mostra a localização de cada palavra em *wordCost*. Por exemplo: quando $wordCost[j]$ for calculado, se $wordCost[j]$ é baseado no valor de $wordCost[k-1]$, marque $wordLocation[j] = k$. Então, depois que $wordCost[n]$ for computado, podemos seguir os ponteiros para ver onde ocorrem as quebras de linhas. A última linha começa na palavra wordLocation[n] e vai até a palavra n. A linha anterior começa em $wordLocation[wordLocation[n]]$ e vai até $wordLocation[n]-1$.

Define uma ótima solução:

Seja wordCost(j) o custo ótima de exibir as palavras 1 até j. Pela prova da subestrutura ótima, fica claro que dado um i ótimo (isto é, o índice da primeira palavra exibida na última linha da solução ótima), nós temos:

$wordCost(j) = wordCost(i-1) + rowCost(i, j)$

No entanto, como nós não sabemos qual i é ótimo, precisamos considerar todo possível i, então nossa definição para o custo ótimo é:

$wordCost(j) = min\{wordCost(i-1) + rowCost(i, j)\}$

Então para o caso base definimos $wordCost(0) = 0$

Algoritmos

Considere o seguinte pseudocódigo:

Algorithm 1 Cálculo de custo de linhas

procedure ROWCOST($extraSpace, j, n$)
    if extraSpace < 0 **then**
        return ∞
    else
        if $j = n$ and extraSpace > 0 **then** ▷ Usado apenas para última linha
            return 0
        else
            return $(\text{extraSpace})^3$

Algorithm 2 Imprimir Elegante

procedure PRINT($n, words, M$) ▷ words é um vetor de palavras
    aloca um vetor wordCost ▷ Responsavel pelo custo das palavras
    aloca um vetor wordLocation ▷ Responsavel pela localização para reconstrução
    inicialziar wordCost tudo com zero
    for j = 1 até n **do**
        $extraSpace \leftarrow M$
        $maximum \leftarrow max(1, j+1-\lceil(M/2)\rceil)$ ▷ Precisamos verificar apenas até M/2
        for $i = j$ até maximum decrementando -1 **do**
            $extraSpace \leftarrow extraSpace - words[i].Comprimento - 1$
            $currentCost \leftarrow wordCost[i-1] + rowCost(extraSpace, j, n)$
            if $currentCost < wordCost[j]$ **then**
                $wordLocation[j] \leftarrow i$ ▷ Atualiza localização
                $wordCost[j] \leftarrow currentCost$ ▷ Atualiza menor custo
    return $wordCost, wordLocation$ ▷ Retorna o menor custo e Localização

Complexidade:

Linha	Comentário	Complexidade
1	constante	c1
2 a 4	for que vai até n	$O(n)$
5 a 10	for e while	$O(n*M)$
11	constante	c2

As complexidades de tempo e espaço são ambas $\theta(n*M)$.

No máximo $\theta(M/2)$ palavras cabem em uma linha. Isso pode ser considerado pois cada palavra tem pelo menos 1 caracter, e ainda temos espaços entra as palavras. Com isso podemos concluir que cada linha com palavras i,...,j contem j-i+1 palavras. Então se j-i+1 > $\lceil(M/2)\rceil$, logo sabemos que rowCost[i,j]= ∞. Precisamos calcular e guardar $extraSpace[i,j]$ e $rowCost[i,j]$ para $j-i+1 \leq \lceil(M/2)\rceil$. E o loop interno pode rodar até max(1, j-$\lceil(M/2)\rceil$+1). Então temos θ(n*M).

Questão 2.

R:
Seja w um vetor com o peso dos itens e seja c um vetor com valor dos itens. Considere que os itens sejam indexados de modo que $w_1 \leq w_2 \leq w_3 ... \leq w_n$ e $c_1 \geq c_2 \geq c_3 ... \geq c_n$. Desta forma podemos concluir que o maior valor estará no menor peso e o menor valor no maior peso. Para um algoritmo ser eficiente basta que percorrer o vetor ordenado w, removendo o item de W que é a capacidade total da mochila.

Algoritmo:

Algorithm 3 Algoritmo da Mochila modificado

```
1:  procedure MOCHILA(c, w, W, n)
2:      itensMochila = ∅                                   ▷ Itens da mochila
3:      valorTotal = 0                                     ▷ Valor total de item que cabe na mochila
4:      w ← mergesort(w, 0, w.comprimento)                 ▷ Ordena de menor peso para maior peso
5:      c ← mergesortInverse(c, 0, c.comprimento)          ▷ Ordena de maior valor para menor valor
6:      for i = 1 to n do do
7:          if w[i] ≤ W then
8:              W ← W − w[i]
9:              itensMochila ← i ∪ itensMochila
10:             valorTotal = valorTotal + c[i]
11:     return itensMochila, valorTotal
```

Corretude:

De acordo com o CLRS, existem dois ingredientes que são exibidos pela maioria dos problemas que se prestam a uma estratégia gulosa : a (1) propriedade de escolha gulosa e a (2) subestrutura ótima. Segue abaixo a prova das propriedades e argumentos adicionais, devido a solução ser iterativa.

Seja W o peso total da mochila, o quanto a mochila suporta. Seja c um vetor com os custos de cada item, ordenados do mais alto para o mais baixo. Seja w o vetor com o peso de cada item, ordenado do menor para o maior.

(1) Propriedade de escolha gulosa Seja S uma solução ótima para a mochila, uma solução não vazia. Podemos assumir que i_1 pertence a S. Entretanto assuma que i_1 não pertença a S. Seja m o menor indice de um item em S. Considere a solução $S_1 = (S \backslash i_m) \cup i_1$. Como $w_1 \leq w_m$, então $w(S_1) \leq w(S) \leq W$, entao S_1 é um solução viável. Temos que $c_1 \geq c_m$, o que implica em $c(S_1) \geq c(S)$. Concluímos então S_1 tambem é otimo.

(2) Subestrutura ótima Considere agora que i_1 pertence a S . Seja S_2 um subconjunto de S, $S_2 = S \backslash i_1$ é ótimo para os items $i_2, ..., i_n$ e W_2 = W-w_1. Se S_2 não for ótimo, seria possível melhorar a solução S. Se isso ocorrer S não seria uma solução ótima, já que a melhoria de S_2 implicaria em sua melhoria também.

(3) Argumentos adicionais:
Seja p o indice do último elemento de uma solução ótima S_3. Como o algoritmo seleciona os primeiros elementos que cabem na bolsa de forma sequencial, então qualquer elemento está nesse intervalo (1...p) estará na solução ótima. Isso ocorre pois o vetor está ordenado de forma crescente de peso e decrescente de valor. Então se estamos verificando se o item i vai para a mochila isso significa que todos os itens anteriores a ele já estão na mochila.

Com isso, podemos concluir que se algum item de $1 \leq i \leq p$ não estiver na solução ótima, significa que a solução pode ser melhorada e que está solução sem este item não pode ser ótima.

Complexidade:

Linha	Comentário	Complexidade
1 a 3	O tempo é constante	$O(1)$
4 a 5	a função mergesort	$O(nlogn)$
6 a 10	o for e seu conteúdo	$O(n)$
11	tempo é constante	$O(1)$

A complexidade então do algoritmo da mochila modificado para resolver o problema é $O(nlogn)$.

Questão 3.

R: Em sala vimos o algoritmo de Huffman, abaixo segue o algoritmo modificado para palavras de código ternário, (Usamos 0,1,2 em sua representação).

Algoritmo:

Algorithm 4 Algoritmo de Huffman modificado

procedure TERNARY-HUFFMAN(C) ▷ Conjunto de caracteres
  $n \leftarrow |C|$
  if n mod 2 == 0 **then**
    adiciona um novo nó com frequência zero em C
    $n \leftarrow n+1$
  $Q \leftarrow C$ ▷ Q é a fila de prioridade dada pela frequência dos vértices ainda não intercalados
  for 1 até $\lfloor n/2 \rfloor$ **do**
    alocar novo registro z;
    $z.esq \leftarrow x \leftarrow EXTRAI-MIN(Q)$ ▷ 0
    $z.mid \leftarrow w \leftarrow EXTRAI-MIN(Q)$ ▷ 1
    $z.dir \leftarrow y \leftarrow EXTRAI-MIN(Q)$ ▷ 2
    $z.f \leftarrow x.f + y.f + w.f$ ▷ f : Frequência dos caracteres
    $INSERE(Q,z)$
  $return\ EXTRAI-MIN(Q)$

Detalhes:

Novo nó: Adicionamos o nó auxiliar com frequência 0, pois,como o algoritmo funciona retirando 3 nós e adicionando um nó com a frequência desses 3, se o conjunto de frequências fosse par, acabaríamos na situação do último passo sobrar apenas 2 nós para realizar o cálculo, o que quebraria a regra de todos os nós terem 3 filhos.

For até n/2: Na primeira iteração retiramos 3 nós e os substituímos por 1 nó. Então agora temos n-2 nós no total. A cada passo iremos pegar outros 3 nós, e se não for possível, pegaremos 1(acontece no último passo quando só tem a raíz), e adicionar 1, Então diminuimos o total em 2 nós acada iteração. Portanto, só precisamos executar o for n/2 vezes.

Corretude:

A corretude também é uma versão modificada do que vimos em sala:

Lema 1. *Seja C um alfabeto onde cada caracter $c \in C$ tem frequência f[c]. Sejam x ,y e w três caracteres em C com as menores frequências. Então, existe um código otimo livre de prefixo para C no qual os códigos para x,y,w tem o mesmo comprimento e diferem apenas no último bit.*

Demonstração. Seja T uma árvore ótima. Seja a uma folha de profundidade maximal. Se a não tem irmãos, então deletando a da árvore (e usando o pai de a para representar o caracter antes representado por a) produz uma codificação melhor, contradizendo a árvore ser ótima. Então a terá dois irmão b e c, e já que a tem profundidade maximal, b e c são folhas. Então a,b e c são nós de profundidade maximal, e sem perda de generalidade, podemos dizer que $a.freq \leq b.freq \leq c.freq$.

Também sem perda de generalidade, podemos dizer que $x.freq \leq w.freq \leq y.freq$. Troque as folhas a e x; chame a árvore resultante de T_1. Então B(T_1) $\leq$ B(T), já que x.freq $\leq$ a.freq. Então este deve ser o caso que x.freq = a.freq e T_1 é ótima também. Similarmente, troque as folhas b e w em T_1; chame a árvore resultante de T_2. Este deve ser o caso que w.freq = b.freq e T_3 é ótima. Em T_2, as folhas x e w são irmãs. Similarmente, troque as folhas c e y em T_2; chame a árvore resultante de T_3. Este deve ser o caso que $y.freq = c.freq$ e T_3 é ótima. Em T_3, as folhas w e y são irmãs. Com isso podemos concluir que x,w e y são irmãs.

□

Lema 2. *Seja C um alfabeto com frequência f [c] definida para cada caracter $c \in C$. Sejam x, y e w três caracteres de C com as menores frequencias. Seja C_1 o alfabeto obtido pela remoção de x,y e w e pela inclusao de um novo caracter z, ou seja, $C_1 = C \cup$ {z} - { x, y, w} . As frequencias dos caracteres em $C_1 \cap C$ sao as mesmas que em C e f [z] e definida como sendo f [z] = f [x] + f [y] + f [w]. Seja T_1 uma arvore binaria representado um código otimo livre de prefixo para C_1. Entao a arvore binaria T obtida de T_1 substituindo-se o vertice (folha) z pela por um vertice interno tendo x , y e w como fihos, representa uma código otimo livre de prefixo para C.*

Demonstração. Suponha que T não seja ótima. Pelo lema 1, Seja F uma árvore ótima para C em que x,y e w são irmãs. Apague x,y,w de F, e marque seu pai (agora uma folha) com o z. Chame a árvore resultante de F_1. Note que F_1 é uma árvore para C_1, além de que:

B(F_1) = B(F) - x.freq - y.freq - w.freq

< B(T) - x.freq - y.freq - w.freq

= B(T_1).

Isso contradiz o fato que T_1 é ótima. Concluímos que T deve ter sido ótima desde o começo. □

Teorema 3. *O algoritmo de Huffman constroi um codigo otimo (livre de prefixo).*

Demonstração. Segue imediatamente dos Lemas 1 e 2. □

Complexidade:

Linha	Comentário	Complexidade
1	O tempo é constante	$O(1)$
2	usando Build-Min-Heap	$O(logn)$
4 a 10	a for n - 1 e operação no heap O(logn)	$O(nlogn)$
11	operação no heap O(logn)	$O(logn)$

A complexidade então do do algoritmo modificado do huffman para resolver o problema é $O(nlogn)$.