Trabalho prático de paradigmas de programação

Raphael Rodrigeus Campos *

DCC - Universidade Federal de Minas Gerais

Maio 2015

1 O problema

Temos n cérebros rotulados de $1, ..., n$ com cada cérebro i tendo um peso p_i e um QI qi_i associados, onde p_i e qi_i são números inteiros entre 1 e 10000. Pede-se a maior subsequência de cérebros na qual tenhamos os pesos crescentes e os QIs decrescente, formalmente, pede-se a maior subsequência de cérebros $C \subseteq \{1, ..., n\}$ que satisfaça

$$p_{c_1} < p_{c_2} < ... < p_{c_m} \wedge qi_{c_1} > qi_{c_2} > ... > qi_{c_m} \tag{1}$$

2 Algoritmos

Suponhamos que quiséssemos, somente, a maior subsequência $C \subseteq \{1, ..., n\}$ na qual os pesos são crescentes. Se assumirmos que todos os pesos são inteiros distintos temos uma solução trivial para o problema, que é a ordenação dos cérebros em função dos pesos. De maneira análoga, se quiséssemos a maior subsequência de cérebros com QIs decresecentes seria necessário somente ordenar os cérebros em função de seus QIs decrescentemente. A fim de simplicidade, assumimos que os pesos e QIs são inteiros distintos, todavia, não há tal restrição no problema. A restrição imposta pelo problema em (1) é mutalmente exclusiva, ou seja, se houver QIs ou pesos iguais, um ou nenhum deles fará parte da solução do problema.

Como queremos encontrar a maior subsequência que satisfaça (1), podemos ordenar a entrada pelos pesos ou QIs que não afetará a restrição do problema. Quando ordenamos pelos pesos ou QIs temos a maior subsequência $C \subseteq \{1, ..., n\}$ na qual os pesos são crescentes ou os QIs são decrescentes, como elucidado previamente. Portanto, após a ordenção, podemos modelar esse problema como uma extensão do LIS (Longest Increasing Subsequence) visto em sala, onde tínhamos como restrições $j < i$ e $A[j] < A[i]$.

Vamos assumir que a sequência de cérebros está ordenada decrescentemente por QI: $qi_1 \geq qi_2 \geq ... \geq qi_n$, nós dizemos que um cérebro i vem antes de j se $i < j$. Agora podemos modelar o problema como LIS, ou seja, queremos encontrar a maior subsequência de cérebros tal que $i < j$, $p_i < p_j$ e $qi_i > qi_j$.

*Matrícula: `2015660911`

2.1 Força bruta

Seja $P[1,..i]$ e $QI[1,..i]$ os i primeiros elementos dos arrays P e QI, respectivamente. Seja $OPT(i)$ o tamanho da subsequência de cérebros que termina em i. Nós assumimos que OPT(i) satisfaz:

$$OPT(i) = \begin{cases} 0 & \text{se } i = 0 \\ \max(1, \max_{j < i,\ p_j < p_i \text{ e } qi_j > qi_i}(1 + OPT(j))) & \text{caso contrário} \end{cases} \quad (2)$$

Teorema 1. *OPT(i) satisfaz recorrência (2)*

Demonstração. Se i = 0, há apenas uma subsequência de 0 elementos, ou seja, vazia. Se $i \geq 1$, suponha que O é uma solução ótima, onde O é a maior subsequência de cérebros que os pesos são crescentes e os QIs decrescentes. Seja S todos elementos de O exceto o último. Se S for vazio, então OPT(i) = 1. Caso S não seja vazio, S tem de terminar em uma posição j, onde $j < i$. Como O é ótimo então $p_j < p_i$ e $qi_j > qi_i$. Além do mais, S é a maior subsequência de cérebros que os pesos são crescentes e os QIs decrescentes de $P[1,..k]$ e $QI[1,..k]$ para $k < i$, onde $p_k < p_i$ e $qi_k > qi_i$. Se fosse falso, poderíamos trocar S por uma outra subsequência de tal array, que aumentaria o tamanho de O, que é uma contradição, já que O é ótimo. Portanto, o tamanho de O é dado por $\max(1, \max_{j < i,\ p_j < p_i \text{ e } qi_j > qi_i}(1 + OPT(j)))$. □

Algorithm 1 *Brute_Force(n)*

```
1:  ordenar os cérebros em ordem decrescente de QI
2:  S = ∅
3:  prev[1..n] ← −1
4:  max_ref ← 1
5:  Brute_Force_LIS(n)
6:  { Recupera o LIS do array prev }
7:  i ← mi
8:  while i ≠ −1 do
9:      S ← S ∪ {i}
10:     i ← prev[i]
11: end while
12: return reverse(S)
```

O algoritmo força bruta é representado pelo algoritmo 1, na linha 1 ordenamos a entrada em função do QI de forma decrescente e nas linhas $2-4$ as variáveis são inicializadas. Como mostrado no inicio da seção 2, após a ordenação, pode-se estender o problema do LIS e utilizar a equação de recorrência 2 para implementar o algoritmo força bruta para encontrar a maior subsequência de cérebros com pesos crescentes e QIs decrescentes, vide algoritmo 2 : **Brute_Force_LIS**. Após execução da linha 5 do algoritmo 1, o arranjo $prev[1..n]$, que armazena em cada posição a referência para seu predecessor na subsequência, é percorrido para encontrar a maior subsequência (linhas $7-11$).

2.1.1 Análise de complexidade

A complexidade do algoritmo 2 é dado pela função de recorrência:

$$T(n) = \begin{cases} 0 & \text{se i} = 1 \\ \sum_{i=1}^{n-1} T(i) + n - 1 & \text{caso contrário} \end{cases} \quad (3)$$

Portanto, o algoritmo força bruta é $O(2^n)$ em complexidade de tempo. E ele é O(n) em complexidade de espaço , pois utilizamos o arranjo $prev[1...n]$ para manter o rastro da maior subsequência.

Algorithm 2 $Brute\_Force\_LIS(i)$

```
1: if i = 1 then
2:    return 1
3: end if
4: max ← 1
5: for j=1 to i-1 do
6:    max_length ← Brute_Force_LIS(j)
7:    if max < max_length + 1 and p[j] < p[i] and qi[j] > qi[i] then
8:       max ← max_length + 1
9:       prev[i] ← j
10:   end if
11:   if max > max_ref then
12:      max_ref ← max
13:      mi = i
14:   end if
15:   return max
16: end for
```

2.2 Programação dinâmica

O algoritmo recursivo força bruta está de fato resolvendo, somente, n subproblemas distintos (Fig. 1). O fato que o faz rodar em tempo exponencial é o grande número de cálculos redundantes. Nós computamos o algoritmo 2 de forma mais inteligente memorizando o resultados das chamadas recursivas no arranjo $M[1..n]$, que foi, primeiramente, iniciado com zeros. Assim, se $M[j]$ diferente de zero, não é preciso recalcular $Dynamic\_Programming\_LIS(j)$, mas simplesmente retornar $M[j]$, como pode ser observado no algoritmo 4.

Até o momento, nós computamos o valor da solução ótima, porém, queremos o conjunto ótimo completo de cérebros. Nós sabemos que o cérebro i faz parte da solução ótima para o conjunto de cérebros $\{1, ..., i, j\}$ se e somente se $opt(i)+1 = opt(j) \wedge (p[i] < p[j] \wedge qi[i] > qi[j])$, onde j é o indice do maior valor em M. Usando essa observação, podemos rastrear o arranjo M e encontrar a subsequência de cérebros ótima, como pode ser visto no algoritmo 5 : **DP_Find_Solution**.

Algorithm 3 $Dynamic\_Programming(n)$

```
1: ordenar os cérebros em ordem decrescente de QI
2: M[1..n] ← 0
3: max_ref ← 1
4: Dynamic_Programming_LIS(n)
5: return DP_Find_Solution()
```

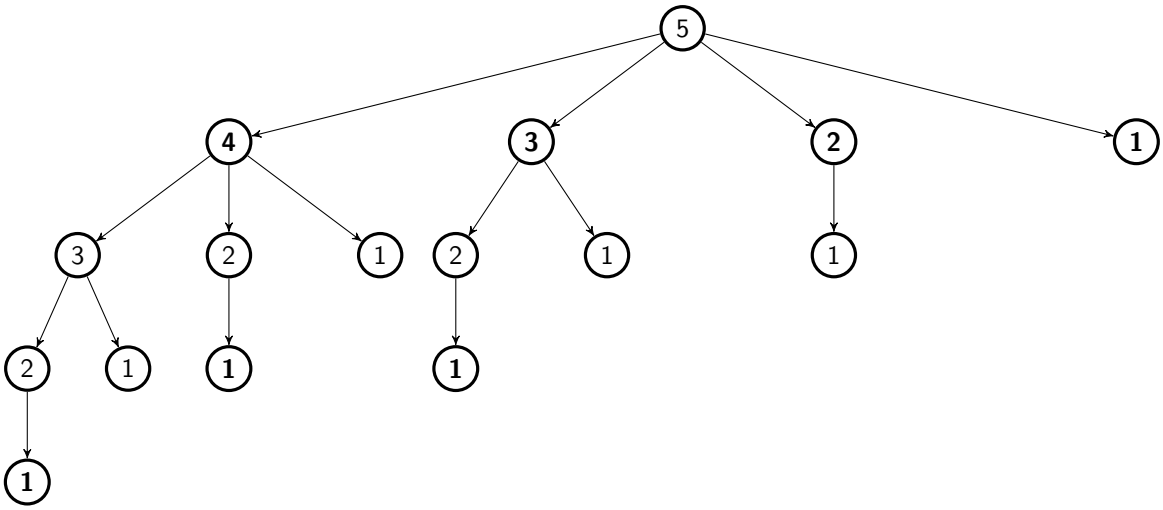

Figura 1: Árvore de recursão

Algorithm 4 *Dynamic_Programming_LIS*(i)

```
1: if i = 1 then
2:    return 1
3: else
4:    if M[i]! = 0 then
5:       return M[i]
6:    end if
7: end if
8: max ← 1
9: for j=1 to i-1 do
10:   max_length ← Dynamic_Programming_LIS(j)
11:   if max < max_length + 1 and p[j] < p[i] and qi[j] > qi[i] then
12:      max ← max_length + 1
13:   end if
14:   if max > max_ref then
15:      max_ref ← max
16:      mi = i
17:   end if
18:   return M[j] ← max
19: end for
```

Algorithm 5 *DP_Find_Solution*()

```
1: S ← {mi}
2: j ← mi
3: max ← M[mi]
4: i ← j − 1
5: while i > 0 and j > 0 and max > 1 do
6:    if M[i] + 1 = max and p[i] < p[j] and qi[i] > qi[j] then
7:       S ← S ∪ {i}
8:       j ← i
9:       max ← M[i]
10:   end if
11: end while
12: return S
```

2.2.1 Análise de complexidade

O algoritmo 3 leva $O(n \log n)$ para ordenar os elementos em função dos QIs. E o algoritmo 4 leva $O(n^2)$ para ser executado para uma entrada de tamanho n, além do mais, encontrar o conjunto de cérebros ótimo a partir do arranjo de memorização M leva $O(n)$, como observado no algoritmo 5. Portanto, a complexidade de tempo do algoritmo de programação dinâmica é $O(n^2)$. Como guardamos os resultados em $M[1..n]$, a complexidade de espaço é igual a $O(n)$.

2.3 Algoritmo guloso

A implementação do algoritmo guloso é baseado no algoritmo ***patience sort***[2][1]. A estratégia gulosa utilizada é: **sempre colocar o cérebro na pilha mais a esquerda possível (linha 6 a 11 do algoritmo 6)**.

Teorema 2. *O número de pilhas gerados pelo algoritmo guloso é igual ao tamanho l_n da maior subsequência de cérebros com pesos crescentes e QIs decrescentes;*

Demonstração. Se cérebros aparecerem em ordem crescente de peso e em ordem decrescente de QIs, ou seja, $p_1 < p_2 < ... < p_n$ e $qi_1 > qi_2 > ... > qi_n$. Então, sob qualquer estratégia legal, c_i tem de estar em alguma pilha a direita da pilha contendo c_{i-1}, pois o peso do cérebro no topo daquela pilha só pode decrescer , ou então o QI crescer. Dessa maneira, o número de pilhas é no mínimo l_n. Reciprocamente, utilizando a estratégia gulosa quando o cérebro c é colocado em uma pilha diferente da primeira pilha, coloque um ponteiro desse cérebro para o cérebro c' atualmente no topo da pilha a esquerda, onde $p_{c'} < p_c$ e $qi_{c'} > qi_c$ (linha 14-16). Ao final de todas iterações, seja c_u o cérebro no topo da última pilha u (pilha mais a direita). A sequência

$$c_1 \leftarrow c_2 \leftarrow ... \leftarrow c_{u-1} \leftarrow c_u \tag{4}$$

obtida seguindo os ponteiros é a subsequência de cérebros com pesos crescentes e QIs decrescentes cujo o tamanho é o número de pilhas.[2] □

Algorithm 6 $Greedy(n)$

1: ordenar os cérebros em ordem decrescente de QI e quando houver empate, ordenar em ordem decresecente de peso
2: $S \leftarrow \emptyset$
3: $pilhas \leftarrow \emptyset$
4: $prev[1..n] \leftarrow nil$
5: **for** $i \leftarrow 1$ **to** n **do**
6: $pilha \leftarrow primeira\_pilha\_que\_elemento\_topo\_nao\_satisfaca\{p[top\_elem] < p[i]$ **and** $qi[top\_elem] > qi[i]\}$
7: **if** $pilha = \emptyset$ **then**
8: $pilhas \leftarrow pilhas \cup cria\_pilha\_coloca\_no\_topo(i)$
9: **else**
10: $pilha.push(i)$
11: **end if**
12: $ant\_pilha \leftarrow pega\_pilha\_anterior(pilha)$
13: $ref\_topo \leftarrow ant\_pilha.top()$
14: **if** $p[ref\_topo] < p[i]$ **and** $qi[ref\_topo] > qi[i]$ **then**
15: $prev[i] \leftarrow ref\_topo$
16: **end if**
17: **end for**
18: $topo \leftarrow$ primeiro elemento da última pilha que tem referência para a pilha anterior
19: **while** $topo \neq nil$ **do**
20: $S \leftarrow S \cup \{topo\}$
21: $topo \leftarrow prev[topo]$
22: **end while**
23: **return** $reverse(S)$

n	FB	DP	AG
100000	-	$104.706 \pm (3.4x10^{0})$	$0.960196 \pm (6.3x10^{-3})$
50000	-	$25.889 \pm (1.9x10^{-1})$	$0.46389 \pm (4.9x10^{-2})$
10000	-	$1.039 \pm (6.4x10^{-3})$	$0.0044 \pm (1.2x10^{-3})$
1000	-	$0.011 \pm (2.6x10^{-4})$	$0.0018 \pm (5.9x10^{-5})$
100	-	$0.00013 \pm (6.0x10^{-6})$	$0.00010 \pm (4.3x10^{-5})$
50	-	$0.00004 \pm (4.3x10^{-6})$	$0.00005 \pm (4.4x10^{-6})$
20	$0.014951 \pm (4.0x10^{-3})$	$0.00001 \pm (2.4x10^{-6})$	$0.00002 \pm (3.2x10^{-6})$
9	$0.00004 \pm (1.3x10^{-5})$	$0.000004 \pm (1.9x10^{-6})$	$0.00001 \pm (3.2x10^{-6})$

Tabela 1: Tempo médio de execução em segundos

2.3.1 Análise de complexidade

O algoritmo guloso leva $O(n \log n)$ para executar a linha 1. No corpo do laço ***for*** a operação na linha 6 leva $O(n)$ no pior caso, quando todos os cérebros da entrada fazem parte da maior subsequência, $O(1)$ no melhor caso, quando a maior subsequência tem tamanho 1, e no caso médio $O(\sqrt{n})$ [1]. Já que a operação é executada para os n cérebros da entrada, o algoritmo guloso tem complexidade de tempo no pior caso de $O(n^2)$, no melhor caso $O(n \log n)$ e no caso médio $O(n\sqrt{n})$. Portanto, o algoritmo guloso no pior caso é $O(n^2)$. Para armazenar as pilhas é gasto de espaço $O(n)$ e para armazanar os ponteiros no vetor $prev[1..n]$ é, também, $O(n)$. Logo, a complexidade de espaço é $O(n)$.

3 Resultados experimentais e conclusão

Os experimentos foram realizados em um PC - Toshiba Satellite i7 2.1GHz quad-core com 8GB de memória ram. As implementações dos algoritmos - FB(Força bruta), DP(Programação dinâmica) e AG(algortimo guloso) - são sequênciais.

Para a coleta dos dados, cada algoritmo foi executado 5 vezes para cada instância de entrada, e foi calculada a média de seus respectivos tempos de execução. Foram utilizadas entradas, de tamanhos {*9, 20, ..., 100000*} cada linha da entrada é um par ordenado (*p, qi*) que representa dados de um cérebro c_i e são números inteiros entre 1 e 10000 criados aleatoriamente.

Podemos analisar, claramente, através da tabela 1 que os tempos de execução seguem as complexidades dos algoritmos apresentadas anteriormente.

Vemos uma grande superioridade do algoritmo guloso em relação às outras abordagens, como era esperado pela análise de complexidade do algoritmo guloso no caso médio, já que o guloso depende do número de pilhas que no pior caso é n e, assim, tornando-o quadrático como o algoritmo DP, todavia, em seu caso médio o número de pilhas tende a ser bem menor que n, tornando algoritmo mais eficiente.

Referências

[1] Longest increasing subsequence. Disponível em `http://www.cs.princeton.edu/courses/archive/spr05/cos423/lectures/patience.pdf`.

[2] D. ALDOUS and P. DIACONIS. *LONGEST INCREASING SUBSEQUENCES: FROM PATIENCE SORTING TO THE BAIK-DEIFT-JOHANSSON THEOREM.*, volume 36. 1999.