
ÁRVORES DE BUSCA ÓTIMAS (Cormen, §15.5)

Dada uma árvore binária de busca A de "n" nós, seja "f_A [i]", ou simplesmente f[i], o nível do i-ésimo nó de A, sendo:

a) "Nível" sinônimo de "profundidade", mas contado de 1 em diante, ou seja, o nível de um nó "x" é o número de nós no caminho da raiz até "x".

b) A "ordenação" dos nós é feita da esquerda para a direita, ou seja, o "i-ésimo nó de A" é o nó de i-ésima menor chave de A (no problema em questão, as chaves dos nós são todas distintas).

Além disso, dado também um vetor p[1..n] de números reais quaisquer, seja C_p (A), ou simplesmente C(A) -- lido como "o custo de A" --, dado por

$$C(A) = \text{SOMATÓRIO}_{i=1}^{n} f[i] * p[i] .$$

O objetivo do problema é então, dado apenas um vetor p[1..n], encontrar uma árvore binária de busca A de custo mínimo com relação a "p". NA VERSÃO SIMPLIFICADA DO PROBLEMA, o objetivo é apenas retornar o custo c(A) de uma árvore A de custo mínimo.

O problema acima modela a situação prática em que se sabe que cada nó de uma árvore binária de busca é consultado na árvore com probabilidade p[i], e onde se deseja minimizar o custo esperado de uma consulta na árvore.

1. EXERCÍCIO: prove que, se A é uma árvore de "n" nós, raiz "r", subárvore esquerda E e subárvore direita D, então

$$C(A) = C(E) + C(D) + \text{SOMATÓRIO}_{i=1}^{n} p[i] .$$

O resultado acima leva ao algoritmo de programação dinâmica a seguir:

```
Algoritmo:    ABB_ótima
Entrada:      vetor p[1..n] de números reais.
Pré-condição: nenhuma.
Saída:        um número real "x".
Pós-condição: x = mín{ C_p (A) : A é uma árvore binária de busca de "n" nós }.
```

```
01. PARA i DE 1 A n
02. | M[i][i] := p[i]
03. PARA t DE 2 A n
04. | PARA i DE 1 A n -t +1
05. | | f := i +t -1, soma_prob := 0
06. | | PARA k DE i A f
07. | | | soma_prob := soma_prob + p[k]
08. | | melhor_custo := mín{ soma_prob + M[i+1][f] ,
09. | |                      soma_prob + M[i][f-1]   }
10. | | PARA r DE i+1 A f-1
11. | | | custo_r := soma_prob + M[i][r-1] + M[r+1][f]
12. | | | melhor_custo := mín{ melhor_custo, custo_r }
13. | | M[i][f] := melhor_custo
14. RETORNE M[1][n].
```

Observe que, por brevidade, o algoritmo acima calcula "soma_prob" em cada iteração, mas que seria mais eficiente calcular esse valor para cada par (i,f) antes do laço da linha 3, armazenando os resultados numa matriz.

2. EXERCÍCIOS:

a) Observando que cada iteração do laço da linha 4 executa em tempo θ(t), e que esse laço executa "n -t +1" iterações, segue que cada iteração do laço da linha 3 executa em tempo θ((n -t +1)*t), e portanto que o algoritmo completo executa em tempo θ de

SOMATÓRIO_{t=1}^n (n -t +1)*t .

É fácil verificar que o termo acima é O(n³): basta observar que n -t +1 <= n, que t <= n e que SOMATÓRIO_{t=1}^n n² = n*n² = n³.

Assim sendo, prove OU QUE o algoritmo acima executa em tempo Ω(n³) OU QUE ele executa em tempo o(n³).

b) Uma generalização do problema abordado acima é considerar que uma busca também pode acontecer por uma chave que não esteja na árvore. Assim, as chaves "i" e "i+1" são buscadas com probabilidade p[i] e p[i+1], e chaves maiores que a chave "i" e menores que a chave "i+1" são buscadas com probabilidade q[i]. Nesse caso, além do vetor p[1..n], a entrada possui também um vetor q[0..n]. O custo de buscar uma chave inexistente é uma unidade a mais do que o nível do último nó percorrido na busca -- o que é equivalente a considerar que cada chave "i" que é uma folha na árvore possui, na verdade, dois filhos d_{i-1} e d_i, d_i representando as chaves maiores que a chave "i" e menores que a chave "i+1"; nesse caso, o custo de consultar uma chave maior que a chave "i" e menor que a chave "i+1" é o nível do "pseudo-nó" d_i. Assim sendo, estenda o algoritmo acima para resolver a generalização em questão.

c) Estenda o algoritmo "ABB_ótima" de forma a retornar não apenas o custo de uma árvore ótima, mas também alguma representação explícita da árvore em si. Escolha uma representação eficiente (em particular, a complexidade do algoritmo não deve mudar).

LEITURA ADICIONAL

O algoritmo de programação dinâmica apresentado acima pode ser melhorado de forma a executar em tempo O(n²), inclusive no caso mais geral do exercício 2.b. Isso é provado no seguinte artigo de Knuth (de 1971, mas veja também o exercício 15.5-4 do Cormen, 2ª edição):

http://dx.doi.org/10.1007/BF00264289 .

Um algoritmo de Hu e Tucker / Knuth (1971) resolve um caso particular do problema do exercício 2.b, a saber quando p[i] = 0 ∀i, em tempo O(n*lg n):

http://dx.doi.org/10.1137/0125012 .

A seguinte é uma extensa revisão científica do assunto (1997):

http://dx.doi.org/10.1016/S0304-3975(96)00320-9 .