Construção e Análise de Algoritmos

aula 24: Dois problemas sobre árvores

1 Introdução

(. . .)

2 Cadeias de multiplicação de matrizes

O problema dessa seção nasce da observação de uma pequena curiosidade.

Suponha que A, B, C são matrizes.

Todo mundo sabe que existem duas maneiras de multiplicar essas 3 matrizes (nessa ordem):*

$$A \cdot \big(B \cdot C\big) \quad = \quad \big(A \cdot B\big) \cdot C$$

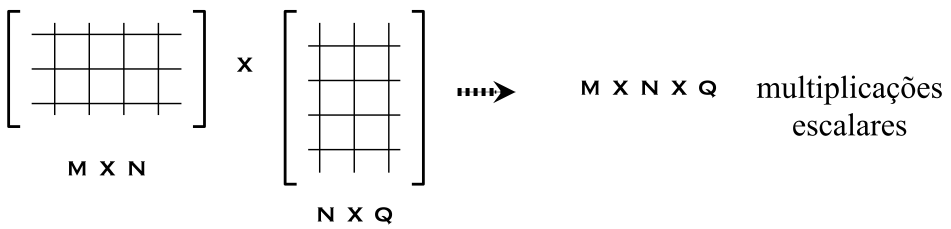
Quer dizer, não importa a ordem em que as multiplicações individuais são realizadas, o resultado é sempre o mesmo.

O que nem todo mundo sabe, ou se dá conta, é que realizar as multiplicações em ordens diferentes nem sempre dá no mesmo.

Veja só.

Suponha que A tem dimensão 3×4, B tem dimensão 4×2, e C tem dimensão 2×6.

Agora, note que a multiplicação de uma matriz $m \times n$ por uma matriz $n \times q$ requer $m \cdot n \cdot q$ multiplicações escalares



Calculando o número total de multiplicações escalares associadas às duas maneiras de multiplicar A, B, C, nós descobrimos que

- $A \cdot \big(B \cdot C\big) \; = \; [3 \times 4] \cdot \Big([4 \times 2] \cdot [2 \times 6]\Big) \;\Rightarrow$ *120 mult. escalares*
- $\big(A \cdot B\big) \cdot C \; = \; \Big([3 \times 4] \cdot [4 \times 2]\Big) \cdot [2 \times 6] \;\Rightarrow$ *60 mult. escalares*

---

*Todo mundo sabe também que, em geral, $A \cdot (B \cdot C) \neq A \cdot (C \cdot B)$, por exemplo.

É verdade, não dá no mesmo!

Algumas pessoas, quando vêem esse tipo de coisa, logo pensam

- *E se eu tivesse n matrizes* $M_1, M_2, \ldots, M_n$

  Como é que eu posso descobrir a maneira ótima de multiplicá-las?

(e já começam a perder a tranquilidade ...)

Certo.

O problema já está relativamente bem definido.

Mas, nós podemos acrescentar que as dimensões das matrizes $M_1, \ldots, M_n$ são definidas por uma sequência de números

$$d_1,\ \ d_2,\ \ ,d_3,\ \ \ldots,\ \ d_n,\ \ d_{n+1}$$

Isto é,

- a matriz A_1 tem dimensão $d_1 \times d_2$
- a matriz A_2 tem dimensão $d_2 \times d_3$
- e assim por diante ...

O nosso objetivo é encontrar uma ordem para a realização das multiplicações

$$M_1 \ \times\ M_2 \ \times\ M_3 \ \times\ \ldots \ \times\ M_{n-1} \ \times\ M_n$$

que esteja associada ao menor número de multiplicações escalares possível.

Ok, mas onde está a árvore aqui?

Quer dizer, na Introdução nós dissemos que hoje a gente estaria manipulando árvores, mas tudo o que se vê acima é uma sequência ...

Sim, é verdade.

Para ver a árvore é preciso olhar para a solução do problema, e não para o problema.

Por exemplo,

$$\Big(\big(M_1 \ \cdot\ (M_2 \ \cdot\ M_3)\big) \ \cdot\ M_4 \Big) \ \cdot\ \big(M_5 \ \cdot\ M_6\big)$$

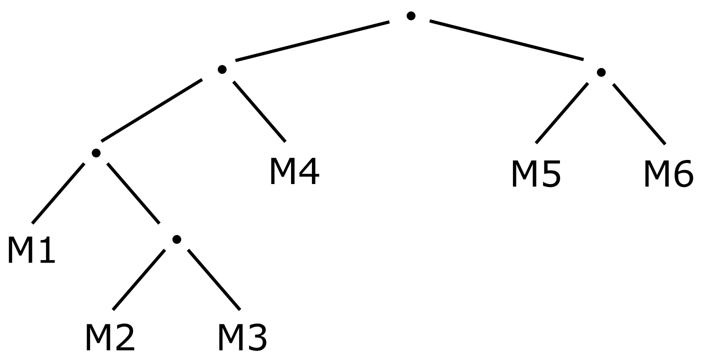
Pronto, você vê a árvore aqui?

Uma sequência de operações matemáticas parentizada não é mais uma sequência ...

Ela é mais parecida com uma coleção de caixas colocadas umas dentro das outras.

E um árvore é uma boa maneira de visualizar que está dentro de quem.

Veja só



Não é legal?

Agora, nós podemos reformular o problema da seguinte maneira

- Dada uma cadeia de multiplicação de matrizes

$$M_1 \;\times\; M_2 \;\times\; M_3 \;\times\; \ldots \;\times\; M_{n-1} \;\times\; M_n$$

  encontrar uma árvore binária que corresponda à maneira mais eficiente possível de realizar essas multiplicações.

A ideia, é claro, é utilizar a técnica de programação dinâmica.

E, para isso, nós precisamos descobrir como decompor o problema.

Mas, antes disso, é útil chamar atenção para um detalhe importante.

Digressão:

Há muitas aulas que nós temos trabalhado com problemas de otimização de *seleção*.

Quer dizer, problemas onde a solução é um subconjunto que precisa ser selecionado a partir de um conjunto base que define o problema†.

Nesses casos, a seleção de um elemento qualquer do problema, em geral, já nos dá uma maneira de decompor o problema.

Mas, desta vez, nós temos um problema de otimização de *organização*, digamos assim.

Quer dizer, um problema onde a solução é dada por uma forma particular de organização para *todos* os elementos do conjunto base.

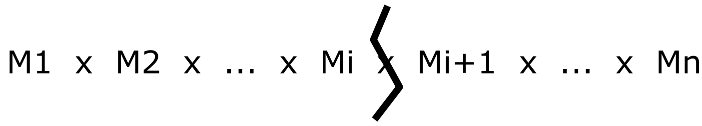
Isso significa que a nossa estratégia padrão de decomposição não vai funcionar aqui.

E que nós precisamos de novas ideias ...

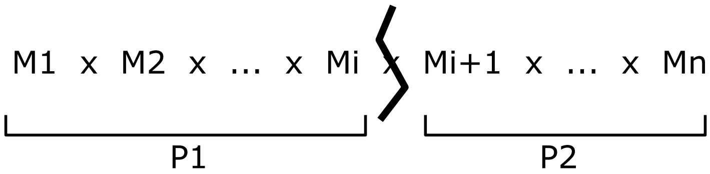
Mas, não é tão difícil encontrar uma nova ideia: basta olhar para as operações de multiplicação (e não para as matrizes).

Veja só

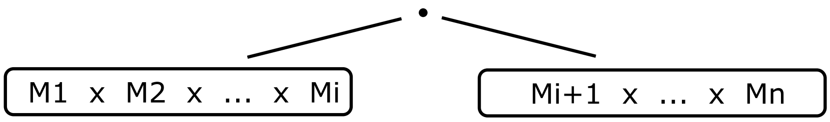
†O problema dos códigos de prefixo é uma exceção.



Como se pode ver no esquema acima, qualquer operação de multiplicação decompõe o problema em dois subproblemas independentes, que podem ser resolvidos em separado



O mais interessante é que a escolha dessa operação de multiplicação determina a raiz da árvore solução



Nesse sentido, a primeira escolha está definindo a multiplicação que será realizada por último.

Não é engraçado? (e contra-intuitivo)

Seja como for, agora nós já temos uma estratégia de decomposição.

O próximo passo consiste em verificar como se obtém a solução do problema (de otimização) a partir dessa decomposição.

Você consegue ver que a cadeia de multiplicações $M_1 \cdot \ldots \cdot M_i$ produz uma matriz resultado com dimensão $[d_1 \times d_{i+1}]$?

(Verifique isso.)

E que a cadeia de multiplicações $M_{i+1} \cdot \ldots \cdot M_n$ produz uma matriz resultado com dimensão $[d_{i+1} \times d_n]$?

Bom, isso significa que a última multiplicação da solução acima realiza

$$\big[d_1 \times d_{i+1}\big] \;\cdot\; \big[d_{i+1} \times d_n\big] \quad \Rightarrow \quad d_1 \cdot d_{i+1} \cdot d_n \quad \textsf{multiplicações escalares}$$

Agora, é fácil ver que o custo da solução S acima é dado por

$$\textsf{Custo}(S) \quad = \quad \textsf{Custo}(S_1) \;+\; \textsf{Custo}(S_2) \;+\; d_1 \cdot d_{i+1} \cdot d_n$$

onde S_1, S_2 são soluções para os subproblemas P_1, P_2.

Legal.

Mas, como usual, alguém pode perguntar: "*Quem disse que a multiplicação que você escolheu no início é a melhor escolha para a raiz da árvore?*"

É verdade, mas ninguém disse isso ...

Quer dizer, aqui, nós vamos ter que utilizar a estratégia padrão de testar todas as possibilidades.

Abaixo nós temos o pseudo-código do algoritmo que implementa essa ideia (sem memoização, para simplificar as coisas).

```
Procedimento  Multi-Mat-PD  ( Mi,...,Mj : sequência de matrizes )
 {
    Se ( i = j )
     {
        S  <--  árvore com único nó Mi;   Retorna (S,0)
     }

    Smin  <--  Null;    Cmin  <--  infinito

    Para  k <-- i  Até  j-1
     {
         (S1,C1)  <--  Multi-Mat-PD ( Mi,...,Mk )

         (S2,C2)  <--  Multi-Mat-PD ( Mk+1,...,Mj )

               S  <--  árvore com subárvores S1 e S2
               C  <--  C1  +  C2  +  (di * d_k+1 * d_j+1)

         Se ( C < Cmin )
          {
             Smin  <--  S;     Cmin  <--  C;
     }    }

    Retorna (Smin,Cmin)
 }
```

Análise de complexidade

Não é difícil ver que cada chamada recursiva é caracterizada pelos índices i, j da sequência de matrizes que ela recebe.

Cada um desses índices pode variar de 1 a n.

Logo, são realizadas no máximo $O(n^2)$ chamadas recursivas distintas.

A seguir, observe que o laço do algoritmo pode realizar no máximo n voltas, e que todas as instruções executam em tempo $O(1)$.

Portanto, o algoritmo como um todo executa em tempo

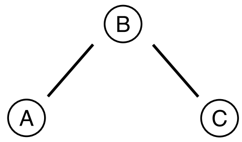
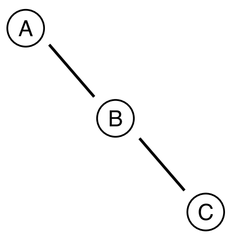
$$O(n^2) \;\times\; O(n) \quad = \quad O(n^3)$$

3 Árvores de busca ótimas

O problema dessa seção tem uma solução muito semelhante ao problema anterior.

Por isso, faz sentido pegar uma carona e apresentá-lo nessa aula também.

Esse problema também é baseado em uma observação simples.

> *Suponha que queremos construir uma árvore binária de busca para armazenar registros com as chaves A, B, C.*
>
> *Intuitivamente, nós deveríamos construir uma árvore com a menor altura possível.*
>
> 
>
>
> *Mas, imagine que a chave A tem uma frequência de acesso, digamos, 3 vezes maior que B e C.*
>
> *Por exemplo, a cada 10 consultas, 6 buscam pela chave A, 2 pela chave B, e 2 pela chave C.*
>
> *Como A se encontra no segundo nível da árvore, uma busca por essa chave precisa realizar 2 comparações — o mesmo vale para C, enquanto que B é encontrada com apenas 1 comparação.*
>
> *Isso significa que, a cada 10 consultas, são realizadas*
>
> $$6 \cdot 2 \ + \ 2 \cdot 1 \ + \ 2 \cdot 2 \quad = \quad 18 \ \textit{comparações}$$
>
> *Por outro lado, se construímos essa outra árvore*
>
> 
>
>
> *que tem altura maior, as 10 consultas realizam*
>
> $$6 \cdot 1 \ + \ 2 \cdot 2 \ + \ 2 \cdot 3 \quad = \quad 14 \ \textit{comparações}$$
>
> *Algumas pessoas, quando vêem esse tipo de coisa, pensam*
>
> - *E se eu tivesse n registros com chaves $c_1, c_2, \ldots, c_n$*

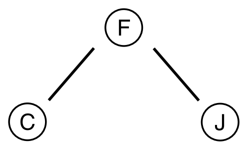
Como é que eu posso descobrir a maneira ótima de armazená-las em uma árvore binária de busca?

(e lá se foi a sua paz de espírito embora ...)

O problema que nós vamos resolver aqui é ligeiramente mais sofisticado que esse.

Note que a descrição acima não leva em conta as consultas que buscam por uma chave que não está na árvore.

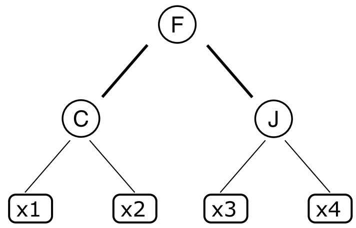
Por exemplo, considere a seguinte árvore binária de busca



Uma busca pela chave A nessa árvore nos leva até o nó C, quando então descobrimos que A não está na árvore.

E uma busca pela chave G nos leva até o nó J, quando então descobrimos que G não está na árvore.

Uma maneira simples de lidar com esse tipo de situação consiste em adicionar chaves artificiais X_1, X_2, X_3, X_4 na árvore[‡]:



A ideia aqui é que X_1 corresponde à coleção de chaves menores que C, X_2 corresponde à coleção de chaves entre C e F, e assim por diante.

Em outras palavras, quando ocorre uma consulta, digamos, pela chave B, nós imaginamos que isso é uma consulta por X_1. E quando ocorre uma consulta pela chave E, nós imaginamos que isso é uma consulta por X_2.

De acordo com essa ideia, nós associamos a X_1 a soma das frequências de todas as chaves menores que C. Associamos a X_2 a soma das frequências de todas as chaves entre C e F. E assim por diante.

Agora, nós podemos formalizar o problema da seguinte maneira

[‡]Esse truque está sendo usado apenas para nos ajudar a pensar sobre o problema de otimização. A estrutura de dados real não irá armazenar chaves artificiais.

- Dada uma coleção de chaves

$$c_1,\ c_2,\ c_3,\ \ldots,\ c_{n-1},\ c_n$$

onde cada chave c_i está associada a uma frequência f_i, e uma coleção de chaves artificiais

$$x_1,\ x_2,\ x_3,\ \ldots,\ x_n,\ x_{n+1}$$

onde cada chave artificial x_i está associada a uma frequência g_i, o problema consiste em encontrar uma árvore binária de busca A tal que

1. as chaves artificiais x_j aparecem todas como folhas na árvore
2. o custo médio de A

$$\mathsf{Custo}(A) \quad = \quad \sum_i \mathsf{alt}_A(c_i) \cdot f_i \ + \ \sum_j \mathsf{alt}_A(x_j) \cdot g_j$$

é o menor possível.

A seguir, nós vamos resolver esse problema por programação dinâmica.

A dica é que nós não precisamos nos preocupar demais com as chaves artificiais x_j, e podemos raciocinar sobre as chaves c_i de maneira semelhante ao que foi feito na seção anterior.

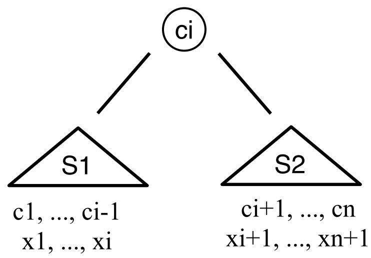
Vejamos.

Nós começamos escolhendo uma chave c_i qualquer para ser a raiz da árvore

$$c_1,\ c_1,\ \ldots,\ c_{i-1}\ c_i\ c_{i+1},\ \ldots,\ c_{n-1}\ c_n$$

É fácil ver que essa escolha decompõe o problema em dois subproblemas independentes P_1 e P_2, que podem ser resolvidos em separado para produzir as árvores solução A_1 e A_2.

E também é fácil ver que a árvore solução A do problema original é construída da seguinte maneira



O próximo passo consiste em determinar o custo dessa solução.

E a chave é observar que ao colocar S_1 e S_2 como subárvores de c_i, a altura de todos os seus nós aumenta de 1.

(Você consegue ver isso?)

Isso significa que nós podemos escrever o custo da parte direita da árvore S como

$$\sum_{k=1}^{i-1} \Big(\mathsf{alt}_{A_1}(c_k)+1\Big)\cdot f_k \quad + \quad \sum_{l=1}^{i} \Big(\mathsf{alt}_{A_1}(x_l)+1\Big)\cdot g_l$$

Uma pequena manipulação algébrica nos permite reescrever essa expressão como

$$\underbrace{\sum_{k=1}^{i-1} \mathsf{alt}_{A_1}(c_k)\cdot f_k \quad + \quad \sum_{l=1}^{i} \mathsf{alt}_{A_1}(x_l)\cdot g_l}_{\mathsf{Custo}(S_1)} \quad + \quad \sum_{k=1}^{i-1} f_k \;+\; \sum_{l=1}^{i} g_l$$

onde nós reconhecemos a expressão que dá o custo de S_1.

A mesma coisa também pode ser feita com a parte direita da árvore S.

Finalmente, colocando tudo junto, nós obtemos

$$\begin{aligned}\mathsf{Custo}(S) \quad = \quad & \mathsf{Custo}(S_1) \;+\; \sum_{k=1}^{i-1} f_k \;+\; \sum_{l=1}^{i} g_l \\ & +\; \mathsf{Custo}(S_2) \;+\; \sum_{k=i+1}^{n} f_k \;+\; \sum_{l=i+1}^{n+1} g_l \\ & +\; f_i\end{aligned}$$

que pode ser simplificada para

$$\mathsf{Custo}(S) \quad = \quad \mathsf{Custo}(S_1) \;+\; \mathsf{Custo}(S_2) \;+\; \sum_{k=1}^{n} f_k \;+\; \sum_{l=1}^{n+1} g_l$$

Legal.

Mas, é sempre bom lembrar que essa é a solução que se obtém quando escolhemos c_i com a raiz da árvore.

Não há nenhuma garantia que essa seja a solução ótima do problema.

Para obter essa garantia, é preciso examinar todas as possíveis escolhas de raiz.

Essa última observação nos dá a oportunidade de mencionar um detalhe relacionado com as chaves artificiais.

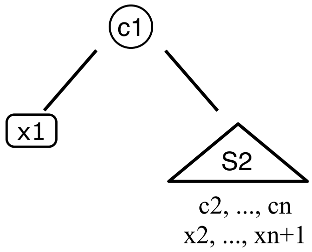
É o seguinte.

Suponha que c_1 foi escolhida como a raiz da árvore.

Isso significa que todas as outras chaves $c_2, \ldots, c_n$ vão ficar do lado direito da árvore.

Mas, isso não significa que o lado esquerdo vai ficar vazio.

Quer dizer, do lado esquerdo nós vamos encontrar a chave artificial x_1.



A mesma coisa acontece quando nós escolhemos c_n para a raiz.

É preciso atenção com esses casos na hora de escrever o pseudo-código.

```
Procedimento  ABB--PD  ( C = {ci,...,cj}, X = {xi,...,x_j+1} )
 {
    Se ( C está vazio )
     {
        S  <--  árvore de um único nó com a chave de X;
        Retorna (S,g(X))
     }

    (S1,C1)  <--  ABB-PD ( vazio, {xi} )
    (S2,C2)  <--  ABB-PD ( {c_i+1,...,cj} , {x_i+1,...,x_j+1} )

    Smin  <--  Construir-Árvore ( S1, ci, S2 )
    Cmin  <--  C1 + C2 + (fi+...+fj) + (gi+...+g_j+1)

    Para  k <-- i+1  Até  j
     {
         (S1,C1)  <--  ABB-PD ( {c_i,...,c_k-1} , {x_i,...,x_k} )
         (S2,C2)  <--  ABB-PD ( {c_k+1,...,cj} , {x_k+1,...,x_j+1} )

               S  <--  Construir-Árvore ( S1, ck, S2 )
               C  <--  C1 + C2 + (fi+...+fj) + (gi+...+g_j+1)

         Se ( C < Cmin )
          {
             Smin  <--  S;     Cmin  <--  C;
          }
     }
    Retorna (Smin,Cmin)
 }
```

Análise de complexidade

Não é difícil ver que cada chamada recursiva é caracterizada pelos índices i, j do subconjunto de chaves `C` que ela recebe.

Cada um desses índices pode variar de 1 a n.

Logo, são realizadas no máximo $O(n^2)$ chamadas recursivas distintas.

A seguir, observe que o laço do algoritmo pode realizar no máximo n voltas, e que todas as instruções executam em tempo $O(1)$.

Portanto, o algoritmo como um todo executa em tempo

$$O(n^2) \;\times\; O(n) \quad = \quad O(n^3)$$