Programação Dinâmica

Cid C. de Souza – IC-UNICAMP

12 de julho de 2005

Programação Dinâmica: conceitos básicos

▷ Quando se aplica:

- Problemas de Otimização.
- Solução = Seqüência de Decisões.
- **Existência de subestrutura ótima** (*Princípio da otimalidade de Bellman*): as soluções ótimas do problema contêm soluções ótimas de subproblemas.
- **Fórmula de Recorrência**: descreve relação entre as soluções ótimas dos subproblemas.

Programação Dinâmica: conceitos básicos (cont.)

- Sobreposição de Subproblemas (*Overlaping subproblems*): a solução ótima de um mesmo subproblema é usada várias vezes mas **calculada uma única vez** !
- Soluções de subproblemas são armazenadas em **tabelas**, logo é preciso que o número total de subproblemas que precisam ser resolvidos é pequeno (polinomial no tamanho da entrada).
- *Programação Dinâmica é uma estratégia "bottom-up" !*

Multiplicação de cadeias de matrizes

- ▷ *O problema*: calcular o número mínimo de operações de multiplicação (escalar) para encontrar a matriz M dada por:

$$M = M_1 \times M_2 \times \ldots M_i \ldots \times M_n$$

onde M_i é uma matriz de b_{i-1} linhas e b_i colunas, para todo $i \in \{1, \ldots, n\}$.

- ▷ **Observação 1:** as matrizes são multiplicadas aos pares.
- ▷ **Observação 2:** para calcular a matriz M' dada por $M_i \times M_{i+1}$ são necessárias $b_{i-1} * b_i * b_{i+1}$ multiplicações entre os elementos de M_i e M_{i+1}.

Multiplicação de cadeias de matrizes (cont.)

- *Exemplo:* $M = M_1 \times M_2 \times M_3 \times M_4$ com $b = \{200, 2, 30, 20, 5\}$.
- A ordem das multiplicações faz **muita** diferença !

$$
\begin{array}{lcl}
M = (((M_1 \times M_2) \times M_3) \times M_4) & \rightarrow & 152.000 \text{ operações} \\
M = (M_1 \times ((M_2 \times M_3) \times M_4)) & \rightarrow & 3400 \text{ operações}
\end{array}
$$

- Algoritmo "força bruta" é impraticável: existem $\approx 4^n/n^{\frac{3}{2}}$ possíveis parentizações !

Multiplicação de cadeias de matrizes (cont.)

▷ **Observação 3:** solução = parentização (associação) de produtos entre matrizes. Seqüência de decisões: onde colocar os parênteses.

▷ **Observação 4:** dada uma solução ótima, existem dois pares de parênteses que identificam o último par de matrizes que serão multiplicadas. Ou seja, existe k tal que $M = A \times B$ onde $A = M_1 \times \ldots M_k$ e $B = M_{k+1} \times \ldots M_n$.

▷ Subestrutra ótima: A e B precisam ser computados de maneira ótima !

Multiplicação de cadeias de matrizes (cont.)

▷ **Observação 5:** se $m[i,j]$ é a solução ótima para realizar o produto $M_i \times M_{i+1} \times \ldots \times M_j$ então $m[i,j]$ é dado por:

$$m[i,j] := \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + b_{i-1} * b_k * b_j\}.$$

▷ **Observação 6:** O cálculo do mesmo $m[i,j]$ pode ser requerido em vários subproblemas mas o número de total de $m[i,j]$'s é $O(n^2) \Longrightarrow$ tabelar $m[i,j]$!

Multiplicação de Matrizes (cont.)

▷ O algoritmo:

`Multiplica_matrizes`(b);

Para $i = 1$ **até** n **faça** $m[i, i] \leftarrow 0$;

(* calcula o valor ótimo de todas sub-cadeias de tamanho $u + 1$ *)

Para $u = 1$ **até** $n - 1$ **faça**

Para $i = 1$ **até** $n - u$ **faça**

$j \leftarrow i + u$;

$m[i, j] \leftarrow \min_{i \leq k < j}\{m[i, k] + m[k + 1, j] + b_{i-1} * b_k * b_j\}$.

$s[i, j] \leftarrow \arg\min_{i \leq k < j}\{m[i, k] + m[k + 1, j] + b_{i-1} * b_k * b_j\}$.

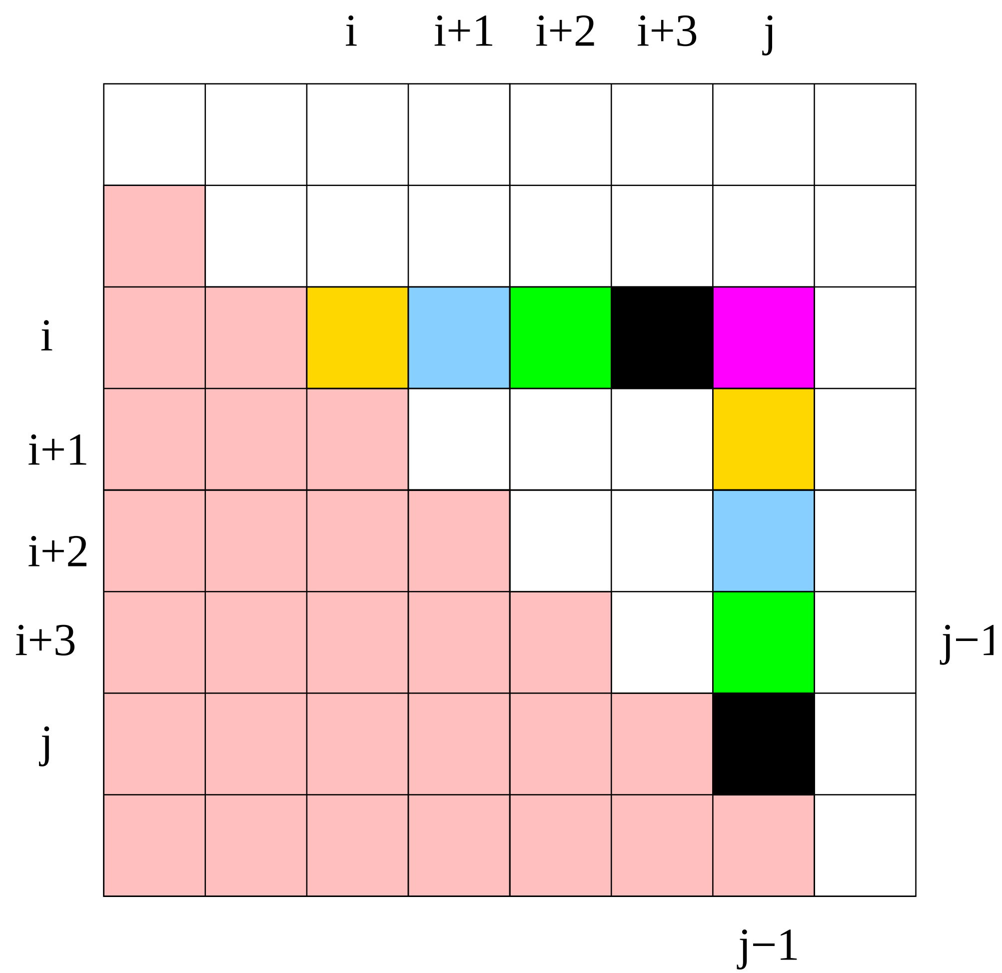
fim-para

fim-para.

Retorne(m,s).

▷ **Complexidade:** $O(n^3)$.

Multiplicação de cadeias de matrizes (cont.)



Multiplicação de cadeias de matrizes (cont.)

	1	2	3	4
1	0			
2		0		
3			0	
4				0

m

	1	2	3	4
1	_			
2		_		
3			_	
4				_

s

Multiplicação de cadeias de matrizes (cont.)

m

	1	2	3	4
1	0	12000		
2		0	1200	
3			0	3000
4				0

s

	1	2	3	4
1	_	1		
2		_	2	
3			_	3
4				_

{ 200, 2, 30, 20, 5 }

Multiplicação de cadeias de matrizes (cont.)

m

	1	2	3	4
1	0	12000	9200	
2		0	1200	
3			0	3000
4				0

s

	1	2	3	4
1	–	1	1	
2		–	2	
3			–	3
4				–

{ 200, 2, 30, 20, 5 }

b0*b1*b3=200*2*20=8000

b0*b2*b3=200*30*20=120000

Multiplicação de cadeias de matrizes (cont.)

m

	1	2	3	4
1	0	12000	9200	
2		0	1200	1400
3			0	3000
4				0

s

	1	2	3	4
1	_	1	1	
2		_	2	3
3			_	3
4				_

{ 200, 2, 30, 20, 5 }

b1*b2*b4=2*30*5=300

b1*b3*b4=2*20*5=200

Multiplicação de cadeias de matrizes (cont.)

m

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3400
4				0

s

	1	2	3	4
1	_	1	1	1
2		_	2	3
3			_	3
4				_

{ 200, 2, 30, 20, 5 }

b0*b1*b4=200*2*5=2000

b0*b2*b4=200*30*5=30000

b0*b3*b4=200*20*5=20000

Multiplicação de cadeias de matrizes (cont.)

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3000
4				0

m

	1	2	3	4
1	_	1	1	1
2		_	2	3
3			_	3
4				_

s

M1 ((M2 . M3) . M4)

Multiplicação de cadeias de matrizes (cont.)

▷ *Por quê não usar Divisão-e-conquista (recursão) ?*

▷ O algoritmo recursivo: (complexidade $\Omega(2^n)$)

`Matrizes_Rec`(b, i, j);
    Se $i = j$ **então Retornar** 0;
    $m[i, j] \leftarrow \infty$;
    Para $k \leftarrow i$ **até** $j - 1$ **faça**
        $q \leftarrow$ `Matrizes_Rec`$(b, i, k) +$ `Matrizes_Rec`$(b, k + 1, j)$;
        $q \leftarrow q + b[i] \times b[k] \times b[j]$;
        Se $m[i, j] > q$ **então** $m[i, j] \leftarrow q$; $s[i, j] \leftarrow k$;
Retornar $m[i, j]$.

▷ Se $n = 4$, ocorrerão 5 chamadas para computar $m[3, 3]$!

Multiplicação de cadeias de matrizes (cont.)

▷ *Fazendo o produto (recuperando a solução):*

`Faz_prod`(M, s, i, j);
    Se $i < j$ **então**
        $X \leftarrow$ `Faz_prod`$(M, s, i, s[i, j])$;
        $Y \leftarrow$ `Faz_prod`$(M, s, s[i, j] + 1, j)$;
        Retornar `Multiplica_duas`$(X, Y, b[i-1], b[s[i, j]], b[j])$;
    se não Retornar M_i;
fim.

Multiplicação de cadeias de matrizes (cont.)

- *Programação dinâmica "top-down":* a técnica de **memorização**.
- Manter estrutura do algoritmo recursivo tabelando os valores pré-computados dos subproblemas (evita recálculos).
- Interromper a recursão sempre que ela já tiver sido computada para o conjunto de parâmetros de entrada. Esta situação será identificada quando o valor correspondente à saída daquela chamada recursiva já estiver preenchido na tabela.

Multiplicação de cadeias de matrizes (cont.)

`Matrizes_Memo`(b,n);
    Para $i \leftarrow 1$ **até** n **faça**
        Para $j \leftarrow 1$ **até** n **faça**
            $m[i,j] \leftarrow \infty$;
    Retornar `Aux`$(b,1,n)$;
`Aux`(b,i,j);
    Se $m[i,j] < \infty$ **então Retornar** $m[i,j]$;
    Se $i = j$ **então** $m[i,j] \leftarrow 0$;
    se não
        Para $k \leftarrow i$ **até** $j-1$ **faça**
            $q \leftarrow$ `Aux`$(b,i,k) +$ `Aux`$(b,k+1,j) + b[i] \times b[k] \times b[j]$;
            Se $m[i,j] > q$ **então** $m[i,j] \leftarrow q$; $s[i,j] \leftarrow k$;
    fim-se
Retornar $m[i,j]$.

O Problema Binário da Mochila

▷ *Dados:*

- um conjunto de n itens;
- w_i: o peso do item i, para todo $i = 1, \ldots, n$.
- c_i: o valor do item i, para todo $i = 1, \ldots, n$.
- W: o limite de peso que a mochila comporta.

▷ *Hipóteses:*

- $\sum_{i=1}^{n} w_i > W$
- $w_i \leq W$ para todo $i = 1, \ldots, n$.

▷ *Pergunta-se:* quais os itens que eu devo colocar na mochila de modo a **maximizar** o valor total transportado ?

O Problema Binário da Mochila

▷ *Formulação do problema:*

- *Variáveis:* $x_i = 1$ se o item i estiver na solução ótima e $x_i = 0$ caso contrário.
- *Modelo:*

$$\max \sum_{i=1}^{n} c_i x_i \tag{1}$$

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

▷ *Programação dinâmica ?*

- problema de otimização;
- solução $\equiv$ seqüência de decisões;
- tem subestrutura ótima ?

O Problema Binário da Mochila

▷ *Encontrando a subestrutura ótima:*

- Se o item n estiver na solução ótima, o valor desta solução será c_n mais o valor da melhor solução do problema da mochila com capacidade $W - w_n$ considerando-se só os $n-1$ primeiros itens.
- Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade W considerando-se só os $n-1$ primeiros itens.

O Problema Binário da Mochila

▷ *Generalizando:*

Definição: seja $z[k, d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila e os k primeiros itens da instância original.

▷ *Fórmula de recorrência:*

$$z[0, d] = 0$$

$$z[k, d] = -\infty, \quad \text{se } d < 0$$

$$z[k, d] = \begin{cases} z[k-1, d], & \text{se } w_k > d \\ \max\{z[k-1, d], z[k-1, d-w_k] + c_k\}, & \text{se } w_k \leq d \end{cases}$$

O Problema Binário da Mochila

▷ *O algoritmo*

`Mochila`(c,w,W,n);
Para $d \leftarrow 0$ **até** W **faça** $z[0,d] \leftarrow 0$;
Para $k \leftarrow 1$ **até** n **faça**
  Para $d \leftarrow 1$ **até** W **faça**
    Se $w_k > d$ **então** $z[k,d] \leftarrow z[k-1,d]$;
    se não
      $z[k,d] \leftarrow z[k-1,d]$;
      Se $c_k + z[k-1, d-w_k] > z[k,d]$ **então**
        $z[k,d] \leftarrow c_k + z[k-1, d-w_k]$;
Retornar $z[n,W]$.

Complexidade: $O(nW)$ (**pseudo-polinomial !**)

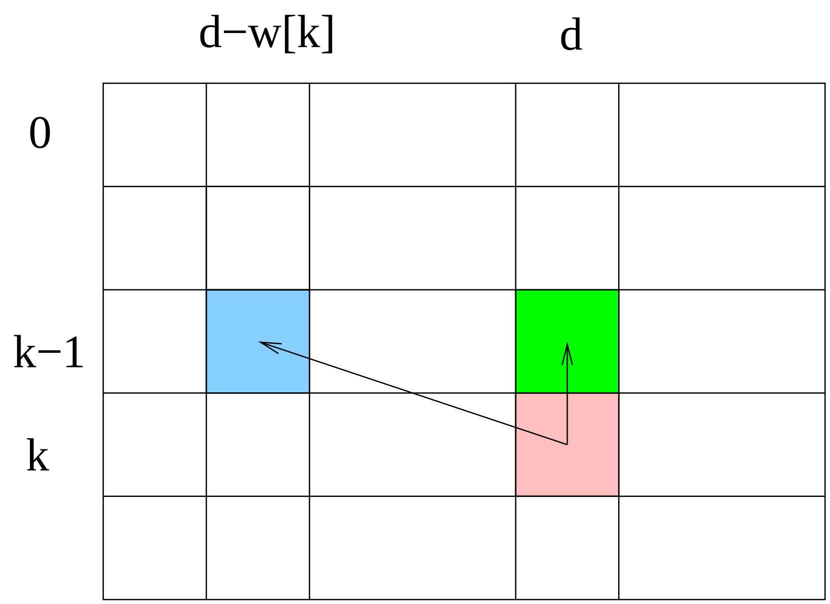
O Problema Binário da Mochila

▷ *Como recuperar a solução x ?*

`Mochila_sol`(x,z,n,W);
  Para $i \leftarrow 1$ **até** n **faça** $x[i] \leftarrow 0$;
  `Mochila_sol_aux`(x,z,n,W);
  Retornar x;
fim.

`Mochila_sol_aux`(x,z,k,d);
Se $k \neq 0$ **então**
  Se $z[k, d] = z[k-1, d]$ **então**
    $x[k] \leftarrow 0$; `Mochila_sol_aux`(x,z,$k-1$,d);
  se não
    $x[k] \leftarrow 1$; `Mochila_sol_aux`(x,z,$k-1$,$d-w_k$);
fim-procedimento

▷ *Complexidade:* $O(n)$.



$$z[k,d] = \min \{ z[k-1,d] \;,\; z[k-1,d-w[k]] + c[k]$$

O Problema Binário da Mochila

▷ *Exemplo:*

$$\begin{aligned} \max \quad & 10x_1 + 7x_2 + 25x_3 + 24x_4 \\ & 2x_1 + x_2 + 6x_3 + 5x_4 \leq 7 \\ & x_i \in \{0,1\}, i = 1,2,3,4 \end{aligned}$$

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0							
3	0							
4	0							

+ c[1] = 10

w[1]=2

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0							
4	0							

+ c[2] = 7

w[2]=1

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0							

+ c[3] = 25

w[3]=6

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

+ c[4] = 24

w[4]=5

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7	c[k], w[k]
0	0	0	0	0	0	0	0	0	
1	0	0	10	10	10	10	10	10	10, 2
2	0	7	10	17	17	17	17	17	7, 1
3	0	7	10	17	17	17	25	32	25, 6
4	0	7	10	17	17	24	31	34	24, 5

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7	c[k], w[k]
0	0	0	0	0	0	0	0	0	
1	0	0	10	10	10	10	10	10	10, 2
2	0	7	10	17	17	17	17	17	7, 1
3	0	7	10	17	17	17	25	32	25, 6
4	0	7	10	17	17	24	31	34	24, 5

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7	c[k], w[k]
0	0	0	0	0	0	0	0	0	
1	0	0	10	10	10	10	10	10	10, 2
2	0	7	10	17	17	17	17	17	7, 1
3	0	7	10	17	17	17	25	32	25, 6
4	0	7	10	17	17	24	31	34	24, 5

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7	c[k], w[k]
0	0	0	0	0	0	0	0	0	
1	0	0	10	10	10	10	10	10	10, 2
2	0	7	10	17	17	17	17	17	7, 1
3	0	7	10	17	17	17	25	32	25, 6
4	0	7	10	17	17	24	31	34	24, 5

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7	c[k], w[k]
0	0	0	0	0	0	0	0	0	
1	0	0	10	10	10	10	10	10	10, 2
2	0	7	10	17	17	17	17	17	7, 1
3	0	7	10	17	17	17	25	32	25, 6
4	0	7	10	17	17	24	31	34	24, 5

O Problema Binário da Mochila

k \ d	0	1	2	3	4	5	6	7	c[k], w[k]
0	0	0	0	0	0	0	0	0	
1	0	0	10	10	10	10	10	10	10, 2
2	0	7	10	17	17	17	17	17	7, 1
3	0	7	10	17	17	17	25	32	25, 6
4	0	7	10	17	17	24	31	34	24, 5

x[1] = x[4] = 1 , x[2] = x[3] = 0

Caminhos mínimos: algoritmo de Floyd Warshall

▷ *Dados:*

- um grafo orientado $G = (V, E)$;
- c_{ij}: o custo do arco (i, j) de E.

▷ *Hipótese:* os custos podem ser negativos mas não existem *ciclos negativos* embora possam existir arcos com custos negativos (**não aceitos pelo Dijkstra !**)

▷ *Pergunta-se:* qual o comprimento do caminho mais curto entre todos pares de vértices ?

Caminhos mínimos: algoritmo de Floyd Warshall

▷ *Encontrando a subestrutura ótima:*

- supor vértices rotulados de 1 a n;
- definir um k-caminho entre dois vértices i e j como sendo um caminho de i para j que só passa por vértices de rótulo $\leq k$;
- supor que se sabe calcular o k-caminho mais curto;

▷ *Fórmula de recorrência*:

- $d[i,j,0] = \infty$ se $(i,j) \notin E$ e $d[i,j,0] = c_{ij}$ caso contrário;
- $d[i,j,k] = \min\{d[i,j,k-1], d[i,k,k-1] + d[k,j,k-1]\}$.

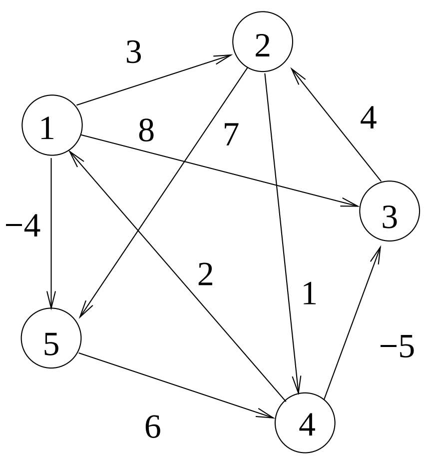
Caminhos mínimos: algoritmo de Floyd Warshall

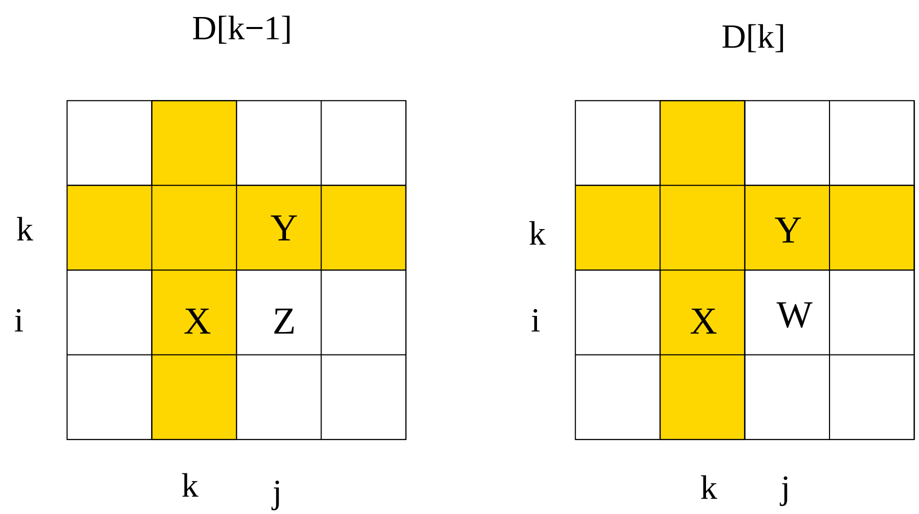
▷ *Algoritmo:*

Para $i \leftarrow 1$ **até** n **faça**
  Para $j \leftarrow 1$ **até** n **faça**
    $d[i,j] \leftarrow c[i,j]$;

Para $k \leftarrow 1$ **até** n **faça**
  Para $i \leftarrow 1$ **até** n **faça**
    Para $j \leftarrow 1$ **até** n **faça**
      Se $d[i,k] + d[k,j] < d[i,j]$ **então**
        $d[i,j] \leftarrow d[i,k] + d[k,j]$;
Retorne (d);

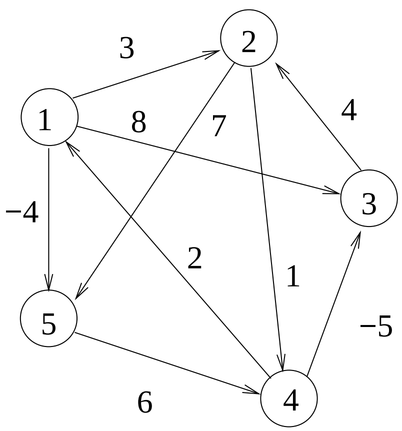
Caminhos mínimos: algoritmo de Floyd Warshall





$$W = \min \{ Z , X + Y \}$$

Caminhos mínimos: algoritmo de Floyd Warshall



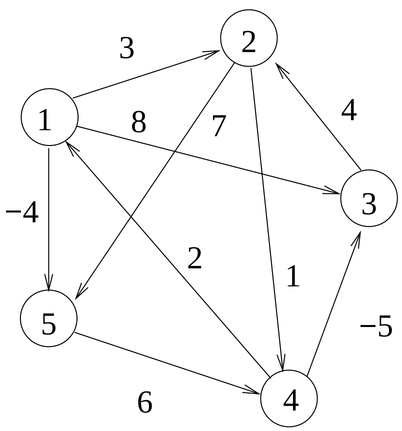
D[0]

	1	2	3	4	5
1	0	3	8	M	−4
2	M	0	M	1	7
3	M	4	0	M	M
4	2	M	−5	0	M
5	M	M	M	6	0

D[1]

	1	2	3	4	5
1	0	3	8	M	−4
2	M	0	M	1	7
3	M	4	0	M	M
4	2	5	−5	0	−2
5	M	M	M	6	0

Caminhos mínimos: algoritmo de Floyd Warshall



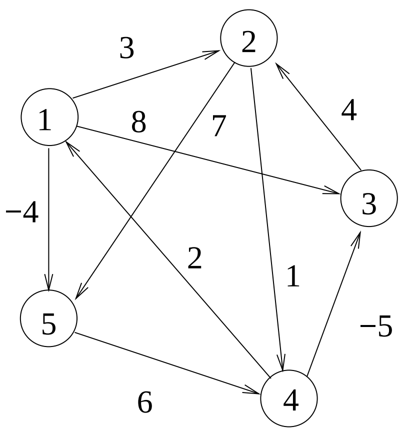
D[1]

	1	2	3	4	5
1	0	3	8	M	−4
2	M	0	M	1	7
3	M	4	0	M	M
4	2	5	−5	0	−2
5	M	M	M	6	0

D[2]

	1	2	3	4	5
1	0	3	8	4	−4
2	M	0	M	1	7
3	M	4	0	5	11
4	2	5	−5	0	−2
5	M	M	M	6	0

Caminhos mínimos: algoritmo de Floyd Warshall



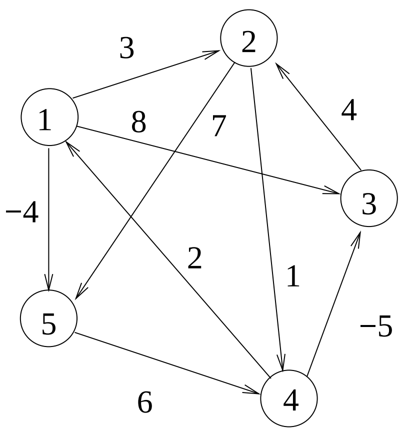
D[2]

	1	2	3	4	5
1	0	3	8	4	−4
2	M	0	M	1	7
3	M	4	0	5	11
4	2	5	−5	0	−2
5	M	M	M	6	0

D[3]

	1	2	3	4	5
1	0	3	8	4	−4
2	M	0	M	1	7
3	M	4	0	5	11
4	2	−1	−5	0	−2
5	M	M	M	6	0

Caminhos mínimos: algoritmo de Floyd Warshall



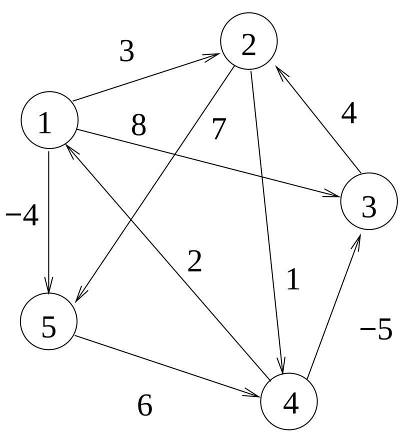
D[3]

	1	2	3	4	5
1	0	3	8	4	−4
2	M	0	M	1	7
3	M	4	0	5	11
4	2	−1	−5	0	−2
5	M	M	M	6	0

D[4]

	1	2	3	4	5
1	0	3	−1	4	−4
2	3	0	−4	1	−1
3	7	4	0	5	3
4	2	−1	−5	0	−2
5	8	5	1	6	0

Caminhos mínimos: algoritmo de Floyd Warshall



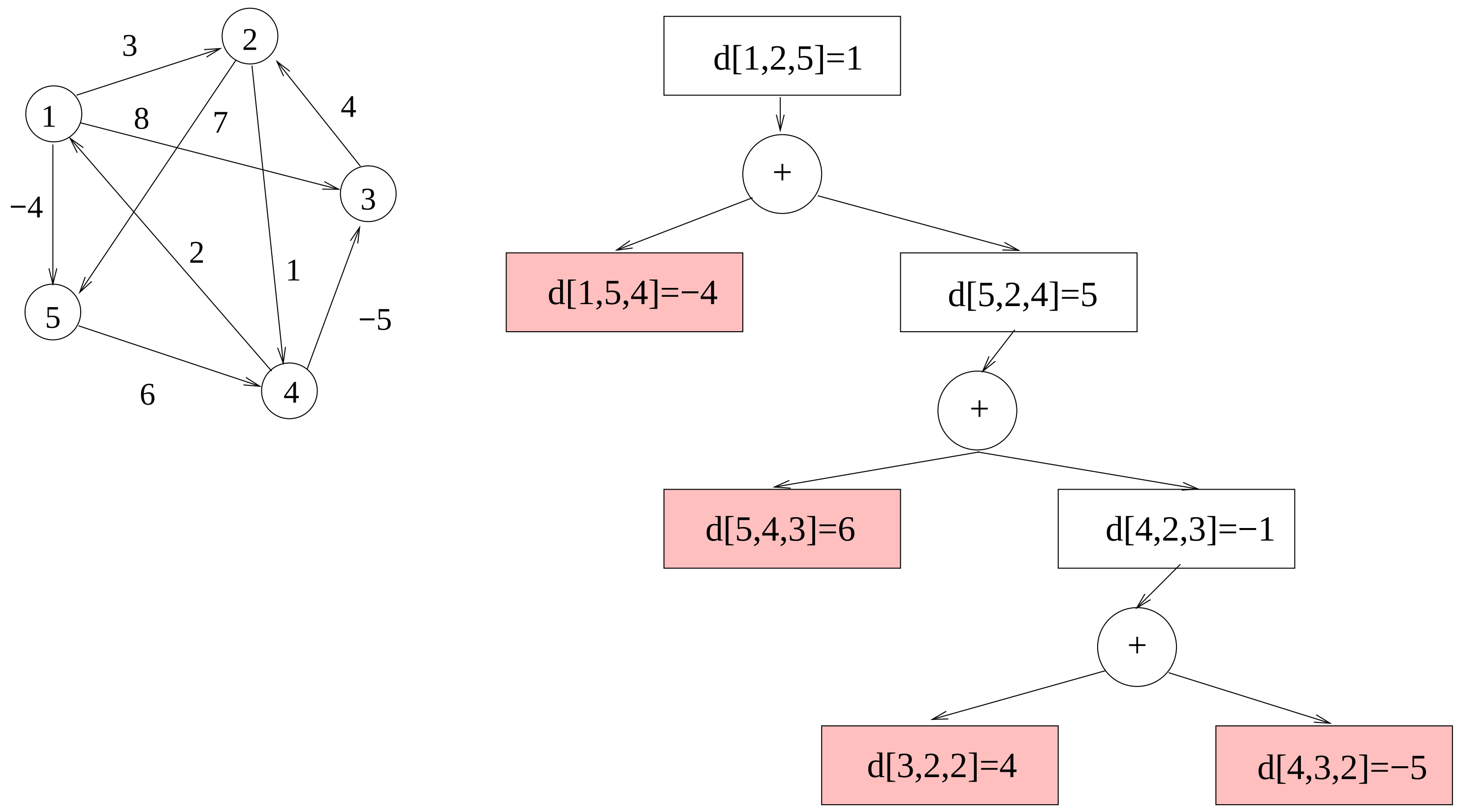
D[4]

	1	2	3	4	5
1	0	3	−1	4	−4
2	3	0	−4	1	−1
3	7	4	0	5	3
4	2	−1	−5	0	−2
5	8	5	1	6	0

D[5]

	1	2	3	4	5
1	0	1	−3	2	−4
2	3	0	−4	1	−1
3	7	4	0	5	3
4	2	−1	−5	0	−2
5	8	5	1	6	0

Caminhos mínimos: algoritmo de Floyd Warshall



Máxima subcadeia comum

▷ *Definição:* dada uma cadeia $S = \{a_1, \ldots, a_n\}$, $S' = \{b_1, \ldots, b_p\}$ é uma *subcadeia* de S se existem p índices $i(j)$ satisfazendo:

(a) $i(j) \in \{1, \ldots, n\}$ para todo $j \in \{1, \ldots, n\}$;

(b) $i(j) < i(j+1)$ para todo $j \in \{1, \ldots, n-1\}$;

(c) $b_j = a_{i(j)}$ para todo $j \in \{1, \ldots, n\}$.

▷ *Exemplo:* $S = \{ABCDEFG\}$ e $S' = \{ADFG\}$.

▷ *Dados:* duas cadeias de caracteres X e Y de um alfabeto Σ.

▷ *Pergunta-se:* qual a maior subcadeia comum de X e Y ?

Máxima subcadeia comum (cont.)

▷ *Programação dinâmica ?*

- problema de otimização;
- solução $\equiv$ seqüência de decisões ?
- tem subestrutura ótima ?

▷ *Notação:* seja S uma cadeia de tamanho n. Para todo $i = 1, \ldots, n$, o prefixo de tamanho i de S será denotado por S_i. Exemplo: para $S = \{ABCDEFG\}$, $S_2 = \{AB\}$ e $S_4 = \{ABCD\}$.

▷ *Definição:* $c[i, j]$ é o tamanho da maior subcadeia comum entre os prefixos X_i e Y_j. Logo, se $|X| = m$ e $|Y| = n$, $c[m, n]$ é o valor ótimo.

Máxima subcadeia comum (cont.)

- *Teorema (subestrutura ótima):* seja $Z = \{z_1, \ldots, z_k\}$ a maior subcadeia comum de $X = \{x_1, \ldots, x_m\}$ e $Y = \{y_1, \ldots, y_n\}$, denotado por $Z = \texttt{MSC}(X, Y)$.
  1. Se $x_m = y_n$ então $z_k = x_m = y_n$ e $Z_{k-1} = \texttt{MSC}(X_{m-1}, Y_{n-1})$.
  2. Se $x_m \neq y_n$ então $z_k \neq x_m$ implica que $Z = \texttt{MSC}(X_{m-1}, Y)$.
  3. Se $x_m \neq y_n$ então $z_k \neq y_n$ implica que $Z = \texttt{MSC}(X, Y_{n-1})$.
- *Fórmula de Recorrência*:

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Máxima subcadeia comum (cont.)

$\texttt{MSC}(X, m, Y, n, c, b)$;

Para $i = 1$ até m **faça** $c[i, 0] \leftarrow 0$;   (* Inicializações *)

Para $j = 1$ até n **faça** $c[0, j] \leftarrow 0$;

Para $i = 1$ até m **faça**   (* Cálculo da matriz c *)

  Para $j = 1$ até n **faça**

    Se $x_i = y_j$ **então**

      $c[i, j] \leftarrow c[i-1, j-1] + 1$; $b[i, j] \leftarrow$ "$\nwarrow$";

    se não

      Se $c[i, j-1] > c[i-1, j]$ **então**

        $c[i, j] \leftarrow c[i, j-1]$; $b[i, j] \leftarrow$ "$\leftarrow$";

      se não

        $c[i, j] \leftarrow c[i-1, j]$; $b[i, j] \leftarrow$ "$\uparrow$";

Retorne$(c[m, n], b)$.

$\triangleright$ *Complexidade:* $O(mn)$.

Máxima subcadeia comum (cont.)

▷ *Recuperando a solução:*

`Recupera_MSC`(b, X, m, n);
    `Recupera_MSC_aux`(b, X, m, n)
fim.

`Recupera_MSC_aux`(b, X, i, j);
    Se $i = 0$ e $j = 0$ **então retornar**
    Se $b[i, j] =$ "$\nwarrow$" **então**
        `Recupera_MSC_aux`$(b, X, i-1, j-1)$;   **imprima** x_i;
    se não se $b[i, j] =$ "$\uparrow$" **então**
        `Recupera_MSC_aux`$(b, X, i-1, j)$;
        `Recupera_MSC_aux`$(b, X, i, j-1)$;
fim.

▷ *Complexidade:* $O(m + n)$.

Máxima subcadeia comum (cont.)

▷ *Exemplo :*

	Y		b	d	c	a	b
X		0	1	2	3	4	5
	0	0	0	0	0	0	0
a	1	0	0	0	0	1	1
b	2	0	1	1	1	1	2
c	3	0	1	1	2	2	2
b	4	0	1	1	2	2	3

	Y		b	d	c	a	b
X		0	1	2	3	4	5
	0						
a	1		↑	↑	↑	↖	←
b	2		↖	←	←	↑	↖
c	3		↑	↑	↖	←	↑
b	4		↖	↑	↑	↑	↖

▷ *Economizando memória ...*

▷ *Solução "top-down" ...*

Planejamento da Produção em lotes (Lot Sizing)

▷ *Dados:*

- n: horizonte de planejamento.
- d_i: demanda no período i.
- c_i: custo unitário de produção no período i.
- h_i: custo unitário de estocagem entre os períodos $i-1$ e i.

▷ *Pergunta-se:* quanto e quando deve ser produzido em cada período para atender a demanda de modo a **minimizar** o custo total de produção e estocagem ?

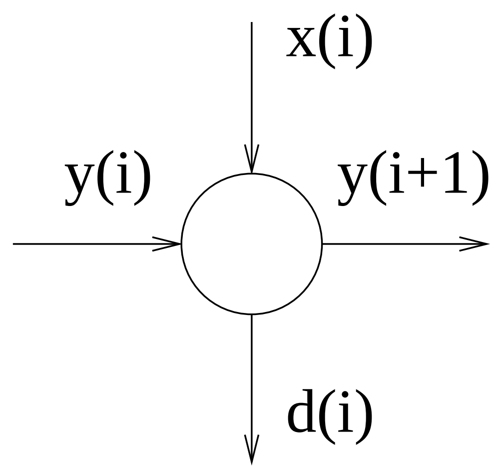
▷ *Hipóteses:* os estoques no início e no fim do horizonte são nulos.

Lot Sizing: Formulação do problema

▷ *Variáveis:*

- x_i: quantidade de itens produzidos no período i.
- y_i: quantidade de itens em estoque no início do período i.

▷ *Modelo gráfico:*



Lot Sizing: Formulação do problema (cont.)

▷ *Modelo algébrico;*

$$\min \sum_{i=1}^{n} \{c_i x_i + h_i y_i\} \tag{3}$$

$$y_i + x_i = d_i + y_{i+1}, \quad i = 1, \ldots, n \tag{4}$$

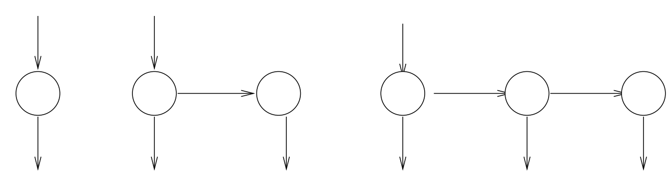
$$y_1 = y_{n+1} = 0, \tag{5}$$

$$x_i \geq 0, \; y_i \geq 0, \quad i = 1, \ldots, n \tag{6}$$

Lot Sizing (cont.)

▷ *Teorema 1*: Existe uma solução ótima tal que, para todo período $i = 1, \ldots, n$, a demanda d_i é totalmente satisfeita pela produção em i (x_i) **ou** apenas pelo estoque proveniente do período $i - 1$ (y_i).

▷ *Representação gráfica de uma solução ótima: ...*



▷ *Programação dinâmica ?*

- problema de otimização;
- solução $\equiv$ seqüência de decisões;
- tem subestrutura ótima ?

Lot Sizing (cont.)

- *Definição 1*: $E[j]$ é o custo mínimo de produção satisfazendo as demandas dos períodos 1 até $j-1$ sem que haja estoque entre os períodos $j-1$ e j (i.e., $y_j = 0$), para todo $j = 1, \ldots, n+1$.

  Observação: $E[1] = 0$ e, por hipótese, $E[n+1]$ é o valor ótimo !

- *Definição 2*: a demanda acumulada entre os períodos i e $j-1$ é dada por:

$$da[i,j] = \sum_{\ell=i}^{j-1} d_\ell.$$

Lot Sizing (cont.)

▷ *Definição 3*: o *custo acumulado de estocagem* entre os períodos i e j é dado por:

$$ha[i,j] = \sum_{\ell=i+1}^{j-1} h_\ell da[\ell, j].$$

▷ *Fórmula de recorrência (subestrutura ótima):*

$$E[j] = \min_{1 \leq i \leq j-1} \{E[i] + c_i da[i,j] + ha[i,j]\}.$$

Lot Sizing (cont.)

`Lotsizing`(c,d,h,n);

`Pre_processamento`$(d,h,n,da[1,.],ha[1,.])$;
$E[1] \leftarrow 0$;
Para $i = 2$ **até** n **faça** $E[i] \leftarrow \infty$;
Para $j = 2$ **até** $n+1$ **faça**
    Para $i = 1$ **até** $j-1$ **faça**
        $w \leftarrow E[i] + c_i * \texttt{Calc\_da}(i,j) + \texttt{Calc\_ha}(i,j)$;
        Se $E[j] > w$ **então**
            $E[j] \leftarrow w$; $b[j] \leftarrow i$;;
    Retornar $(E[n+1],b)$;
fim

Lot Sizing (cont.)

- *Exemplo:* ...
- *Complexidade:* Se o procedimento `Pre_processamento` tiver complexidade $O(n)$ e as funções `Calc_da` e `Calc_fa` forem computadas em $O(1)$, então o algoritmo `Lotsizing` terá complexidade $O(n^2)$.
- *Pergunta:* como fazer um cálculo eficiente de $da[\,]$ e $ha[\,]$?

Lot Sizing: Pré-processamento

▷ *Idéia:* calcular $d_a[1,.]$ iterativamente já que $d_a[1,i] = d_a[1,i-1] + d[i-1]$. Em seguida, note que $h_a[1,1] = h_a[1,2] = 0$ e que:

$$\begin{aligned}
h[1,3] &= \sum_{\ell=2}^{2} h[\ell] d_a[\ell,3] \\
&= h[2]d_a[2,3] = h[2]d_a[1,3] - h[2]d_a[1,2] \\
h[1,4] &= \sum_{\ell=2}^{3} h[\ell] d_a[\ell,4] \\
&= h[2]d_a[2,4] + h[3]d_a[3,4] = \\
&= h[2](d_a[1,4] - d_a[1,2]) + h[3](d_a[1,4] - d_a[1,3]) \\
&= (h[2] + h[3])d_a[1,4] - h[2]d_a[1,2] - h[3]d_a[1,3] \\
\ldots &= \ldots \\
h[1,j] &= (\sum_{\ell=2}^{j-1} h[\ell])d_a[1,j] - \sum_{\ell=2}^{j-1} h[\ell]d_a[1,\ell].
\end{aligned}$$

Lot Sizing: Pré-processamento

▷ Logo, $h_a[1,.]$ é facilmente computável ($O(n)$) se $d_a[1,.]$ estiver caculado.

▷ Seja

$$\alpha(j) = \sum_{\ell=2}^{j-1} h[\ell] \quad \text{e} \quad \beta(j) = \sum_{\ell=2}^{j-1} h[\ell] d_a[1,\ell].$$

▷ O algoritmo a seguir tem complexidade $O(n)$ e calcula $d_a[1,.]$ e $h_a[1,.]$ corretamente. Nele, os valores das variáveis α e β na j ª iteração do segundo laço **Para** corresponde aos valores $\alpha(j)$ e $\beta(j)$ respectivamente.

Lot Sizing: Pré-processamento (cont.)

`Pre_processamento`(d,h,n,$d_a[1,.]$,$h_a[1,.]$);
$d_a[1,1] \leftarrow 0$; $h_a[1,1] \leftarrow 0$; $h_a[1,2] \leftarrow 0$;
Para $i \leftarrow 2$ **até** n **faça**
  $d_a[1,i] \leftarrow d_a[1,i-1] + d[i-1]$;
fim-para
$\alpha \leftarrow 0$; $\beta \leftarrow 0$;
Para $j \leftarrow 3$ **até** $n+1$ **faça**
  $\alpha \leftarrow \alpha + h[j-1]$; $\beta \leftarrow \beta + h[j-1]d_a[1,j-1]$;
  $h_a[1,j] \leftarrow \alpha * d_a[1,j] - \beta$;
fim-para
fim.

Lot Sizing: Recuperação da solução

`Lotsizing_sol`$(x,b,n,d_a[1,.])$;
    Para $i \leftarrow 1$ **até** n **faça** $x[i] \leftarrow 0$;
    `Lotsizing_sol_Recursivo`$(x,b,n,d_a[1,.],n+1)$;
    Retornar x;
fim.

`Lotsizing_sol_Recursivo`$(x,b,n,d_a[1,.],j)$;
    Se $j > 1$ **então**
        $x[b[j]] \leftarrow d_a[1,j] - d_a[1,b[j]]$;
        `Lotsizing_sol_Recursivo`$(x,b,n,d_a[1,.],b[j])$;
    fim-se
fim-procedimento

$\triangleright$ *Complexidade:* $O(n)$.