

PROJETO E ANÁLISE DE ALGORITMOS

PARTE IV

TÉCNICAS PARA PROBLEMAS DE OTIMIZAÇÃO

1. Branch and Bound (Enumeração implícita)

Existem duas abordagens principais para lidar com problemas de complexidade não-polinomial ou intratáveis:

- Usar uma estratégia que garanta uma solução exata para o problema, mas que não garanta chegar a esta solução em tempo polinomial.
- Usar um algoritmo aproximado, que possa chegar a uma solução não necessariamente exata, mas bem aproximada (sub-ótima) em tempo polinomial.

A abordagem exata inclui as técnicas de força-bruta e branch-and-bound.

- A ideia de backtracking pode ser usada em problemas de otimização, onde deseja-se minimizar ou maximizar uma função-objetivo, geralmente sujeita a restrições.
- Em relação ao algoritmo de backtracking, o algoritmo de branch-and-bound requer dois itens adicionais:
 - Uma maneira de estabelecer, para cada nó da árvore um limite sobre o melhor valor da função-objetivo em qualquer solução que possa ser obtida pela adição de mais componentes à solução parcial representada pelo nó.
 - O valor da melhor solução encontrada até o momento.

- Se essas informações estiverem disponíveis podemos comparar o valor limite de um nó com o valor da melhor solução até o momento e:
 - Se o valor do limite não for melhor, o nó é podado.
 - Senão será expandido.
- Assim, um caminho de busca é encerrado quando:
 - O valor do limite do nó não é melhor do que o valor da melhor solução até o momento, ou
 - O nó representa uma solução que viola as restrições do problema, ou
 - O subconjunto das soluções possíveis representadas pelo nó consiste em um único ponto
- Ao invés de gerar um único filho do nó mais promissor como no backtracking, geramos todos os filhos e expandimos o mais promissor.
- Nós mais promissores são os que têm o melhor limite inferior ou superior.
- O desempenho de um programa *branch and bound* está fortemente relacionado à qualidade dos seus limitantes inferiores e superiores
 - Quanto mais precisos forem esses limitantes, menos soluções parciais serão consideradas e mais rápido o programa encontrará a solução ótima

- O nome *branch and bound* refere-se às duas fases do algoritmo:
 - *Branch*: testar todas as ramificações de uma solução candidata parcial
 - *Bound*: limitar a busca por soluções sempre que se detectar que o atual ramo da busca é infrutífero

Exemplo: Problema da mochila 0/1

Deseja-se carregar em uma mochila um subconjunto de objetos que some o maior valor. Cada objeto tem um peso e a mochila tem uma capacidade máxima. Os objetos não podem ser partidos.

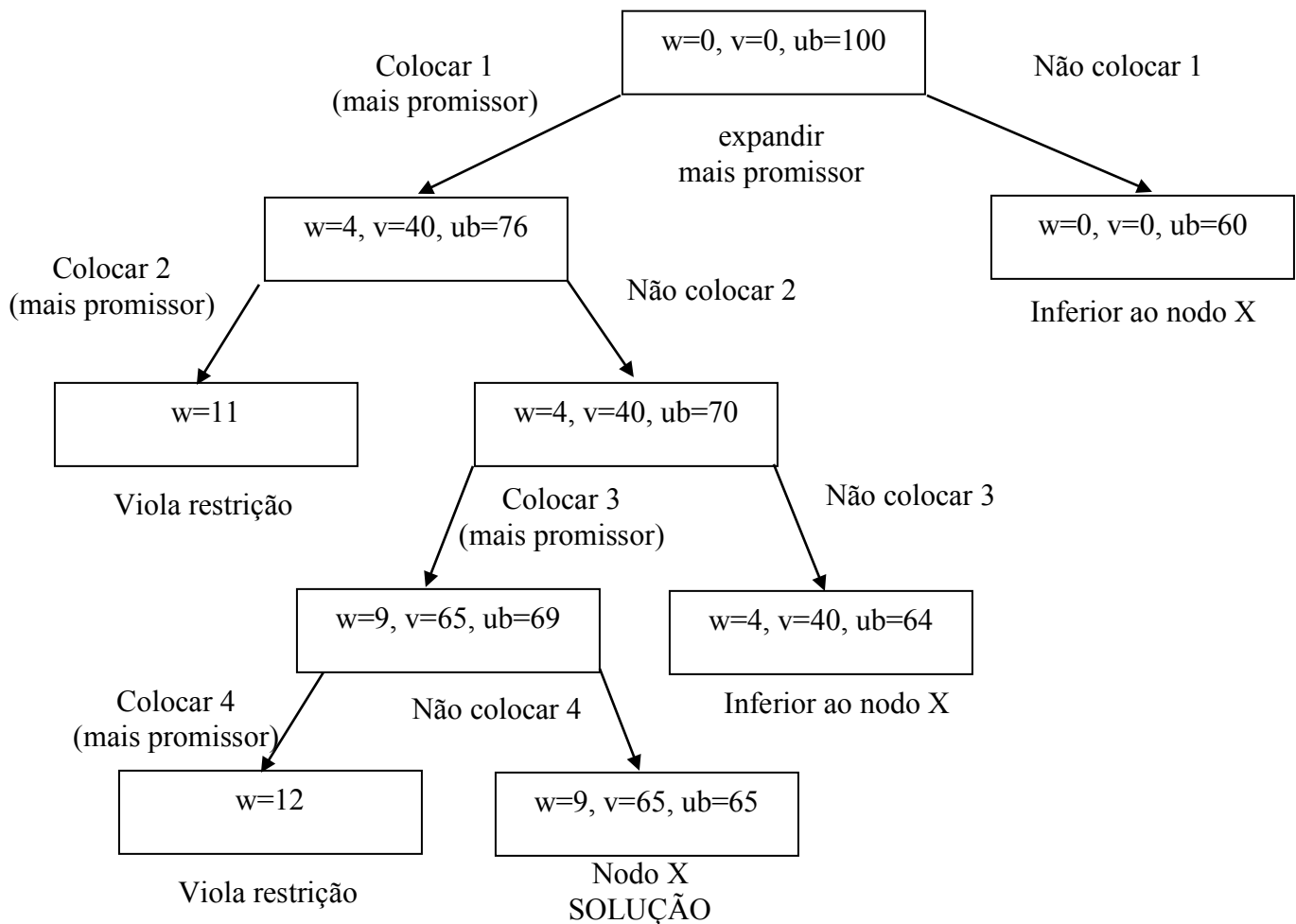
- Muitas situações de interesse comercial
- Existem muitas variantes relacionadas às restrições do problema: número de itens de cada tipo que pode ser incluído, possibilidade de partir ou não cada item, tamanhos serem números inteiros ou não. Cada variante possui complexidade muito diferente umas das outras!

Para a mochila de capacidade $W=10$ e os seguintes objetos:

Item (j)	Peso (w)	Valor (v)	Valor/peso
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

Para computarmos o limite superior, somaremos v , o valor total dos objetos já selecionados, ao produto entre a capacidade restante da mochila e a melhor relação valor/peso entre os objetos restantes:

$$ub_{t+1} = \sum_{j=1}^t v_j + (W - \sum_{j=1}^t w_j) \max_{k \notin S} (v_k / w_k)$$



2. Programação dinâmica

- Tipicamente o paradigma de programação dinâmica aplica-se a problemas de otimização (principalmente os polinomiais).
- Podemos utilizar programação dinâmica em problemas onde há:
 - Subestrutura ótima: As soluções ótimas do problema incluem soluções ótimas de subproblemas.
 - Sobreposição de Subproblemas: O cálculo da solução através de recursão implica no recálculo de subproblemas.
- A técnica de programação dinâmica visa evitar o recálculo desnecessário das soluções dos subproblemas.
- Para isso, soluções de subproblemas são armazenadas em tabelas.
- Logo, para que o algoritmo de programação dinâmica seja eficiente, é preciso que o número total de subproblemas que devem ser resolvidos seja pequeno (polinomial no tamanho da entrada).

O desenvolvimento de um algoritmo de programação dinâmica pode ser desmembrado em uma sequência de 4 etapas:

- a) Caracterizar a estrutura de uma solução ótima
- b) Definir recursivamente o valor de uma solução ótima
- c) Calcular o valor de uma solução ótima
- d) Construir uma solução ótima a partir de informações calculadas

A) Exemplo: Números de Fibonacci

- Encontrar o n -ésimo número de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

```
unsigned int FibRec (unsigned int n) {  
    if (n < 2)  
        return n;  
    else  
        return (FibRec(n-1) + FibRec(n-2));  
}
```

- Simples e elegante, mas vamos analisar o seu desempenho:

$$T(n) = T(n-1) + T(n-2) + 1 \quad \text{para } n \geq 2$$

$$T(n) = 0 \quad \text{para } n \leq 1$$

$$T(n) = \Theta(1,618^n)$$

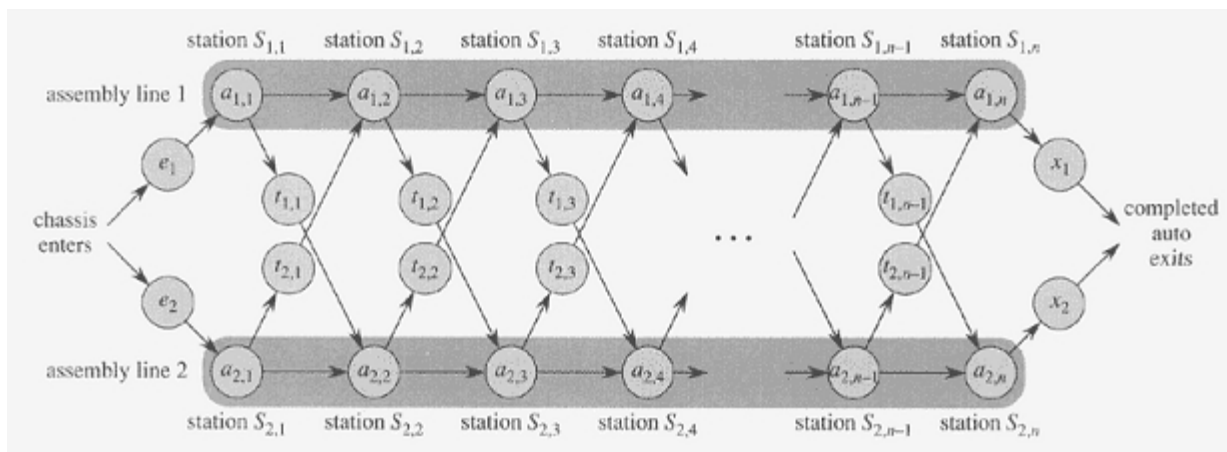
- Solução iterativa:

```
unsigned int FibIter (unsigned int n) {  
    unsigned int i = 1, k, F = 0;  
    for (k = 1; k <= n; k++) {  
        F += i;  
        i = F - i;  
    }  
    return F;  
}
```

- Resolvemos problemas menores
- Em todo estágio, guardamos as soluções para os dois problemas anteriores nas variáveis i e F
- Considerando que a medida de complexidade de tempo $T(n)$ é o número de adições, então $T(n) = \Theta(n)$

B) Exemplo: Linha de Montagem

- Uma fábrica com duas linhas de montagens.
- Um chassi de automóvel entra uma das linhas, tem peças adicionadas a ele em uma série de estações.
- Cada linha de montagem (1,2) possui n estações (1.. n)
- $S_{i,j}$ é a j -ésima estação na linha i .
- As estações com mesmo índice j executam a mesma função mas levam tempos diferentes $a_{i,j}$.
- Além disso, temos um tempo de entrada e_i , um tempo de saída x_i e um tempo para transferência de linhas $t_{i,j}$.



Problema: qual a sequência de estações leva menos tempo na construção do carro?

- Solução por força-bruta: enumerar as possíveis sequências e calcular a de menor tempo: $\Omega(2^n)$

Solução por PD:

a) Caracterizar a estrutura de uma solução ótima

- Uma solução ótima para o problema (caminho mais rápido passando pela estação $S_{i,j}$) necessariamente determina soluções ótimas para os caminhos que passam por $S_{1,j-1}$ ou $S_{2,j-1}$
- Um prefixo B de uma solução ótima A também é uma solução ótima para o subproblema correspondente, porque se ela não fosse, existiria outra melhor que poderia substituí-la, melhorando também a solução A.
- A solução ótima não é necessariamente única.

b) Definir recursivamente o valor de uma solução ótima

- O caminho mais rápido até completar a estação $S_{1,j}$ é o caminho mais rápido até completar a estação $S_{1,j-1}$, e depois passar diretamente pela a estação $S_{1,j}$, ou o caminho mais rápido até completar a estação $S_{2,j-1}$, uma transferência da linha 2 para a linha 1, e depois passar pela estação $S_{1,j}$ (toma-se o que for menor dos dois).
- Um raciocínio análogo vale para completar $S_{2,j}$.

$$f_1(j) = \begin{cases} e_1 + a_{1,1} & , j = 1 \\ \min(f_1(j-1) + a_{1,j}, f_2(j-1) + t_{2,j-1} + a_{1,j}), & j > 1 \end{cases}$$

$$f_2(j) = \begin{cases} e_2 + a_{2,1} & , j = 1 \\ \min(f_2(j-1) + a_{2,j}, f_1(j-1) + t_{1,j-1} + a_{2,j}), & j > 1 \end{cases}$$

- $f_i(j)$ é o menor tempo possível para levar um chassi desde o ponto de partida até a estação $S_{i,j}$.
- O menor tempo para levar um chassi por todo o percurso na fábrica é:

$$\min(f_1(n) + x_1, f_2(n) + x_2).$$

Mas a solução das recorrências leva $O(2^n)$ em tempo!

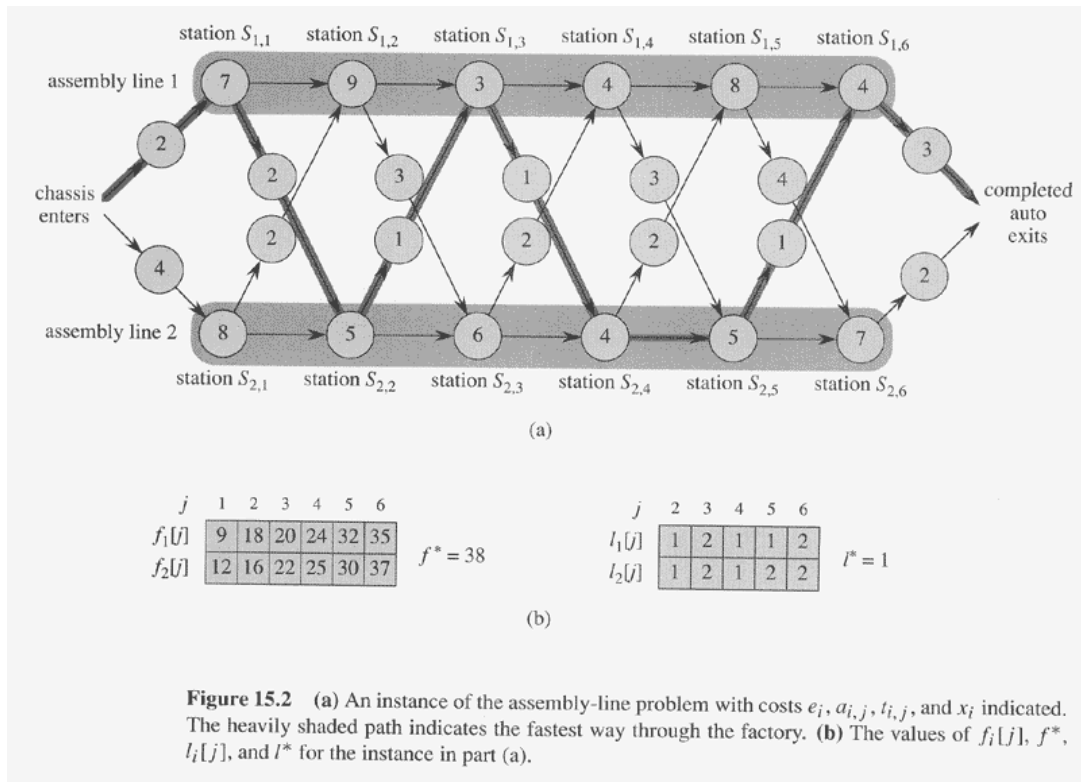
c) Calcular o valor de uma solução ótima

- Utilizando PD, a solução ótima pode ser encontrada em tempo linear, a um custo extra de memória também linear.

```

FASTEST-WAY( $a, t, e, x, n$ )
1   $f_1[1] \leftarrow e_1 + a_{1,1}$ 
2   $f_2[1] \leftarrow e_2 + a_{2,1}$ 
3  for  $j \leftarrow 2$  to  $n$ 
4      do if  $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
5          then  $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$ 
6               $l_1[j] \leftarrow 1$ 
7          else  $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
8               $l_1[j] \leftarrow 2$ 
9      if  $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
10         then  $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$ 
11              $l_2[j] \leftarrow 2$ 
12         else  $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
13              $l_2[j] \leftarrow 1$ 
14  if  $f_1[n] + x_1 \leq f_2[n] + x_2$ 
15     then  $f^* = f_1[n] + x_1$ 
16          $l^* = 1$ 
17     else  $f^* = f_2[n] + x_2$ 
18          $l^* = 2$ 

```



d) Construir uma solução ótima a partir de informações calculadas (ordem inversa)

```

PRINT-STATIONS( $l, n$ )
1   $i \leftarrow l^*$ 
2  print "line "  $i$  ", station "  $n$ 
3  for  $j \leftarrow n$  downto 2
4      do  $i \leftarrow l_i[j]$ 
5      print "line "  $i$  ", station "  $j - 1$ 

```

C) Exemplo: Multiplicação de Matrizes

Calcular o número mínimo de operações de multiplicação (escalar) necessário para computar a matriz M dada por:

$M = M_1 \times M_2 \times \dots \times M_n$ onde M_i é uma matriz de b_{i-1} linhas e b_i colunas, para todo $i \in \{1..n\}$.

- Matrizes são multiplicadas aos pares sempre. Então, é preciso encontrar uma parentização (agrupamento) ótima para a cadeia de matrizes.
- Para calcular a matriz M' dada por $M_i \times M_{i+1}$ são necessárias $b_{i-1} \times b_i \times b_{i+1}$ multiplicações entre os elementos de M_i e M_{i+1} .
- Exemplo: Qual é o mínimo de multiplicações escalares necessárias para computar $M = M_1 \times M_2 \times M_3 \times M_4$ com $b = \{200, 2, 30, 20, 5\}$?
- As possibilidades de parentização são:
 - a) $M = (M_1 \times (M_2 \times (M_3 \times M_4))) = 5300$ multiplicações
 - b) $M = (M_1 \times ((M_2 \times M_3) \times M_4)) = 3400$ multiplicações
 - c) $M = ((M_1 \times M_2) \times (M_3 \times M_4)) = 4500$ multiplicações
 - d) $M = ((M_1 \times (M_2 \times M_3)) \times M_4) = 29200$ multiplicações
 - e) $M = (((M_1 \times M_2) \times M_3) \times M_4) = 152000$ multiplicações

A ordem das multiplicações faz muita diferença!

- Poderíamos calcular o número de multiplicações para todas as possíveis parentizações, mas este procedimento é exponencial, logo a estratégia de força bruta é impraticável!

Solução por PD:

a) Caracterizar a estrutura de uma solução ótima

- Vamos adotar a notação $A_{i..j}$ para a matriz que resulta da avaliação do produto $A_i \times A_{i+1} \times \dots \times A_j$.
- Devemos obter $A_{1..n} = A_{1..k} \times A_{k+1..n}$, cujo custo ótimo é obtido pela soma do custo de $A_{1..k}$ com $A_{k+1..n}$, mais o custo do produto delas.
- A subcadeia $A_{1..k}$ deve ter parentização ótima, do contrário poderíamos substituí-la, por outra com o custo menor que o ótimo.
- Logo, uma solução ótima para uma instância do problema contém soluções ótimas para as sub-instâncias do mesmo problema, o que permite o emprego da programação dinâmica.

b) Definir recursivamente o valor de uma solução ótima

- Deve-se definir uma expressão recursiva para a solução ótima em função das sub-instâncias.
- Será utilizada uma tabela m_{ij} , $1 \leq i \leq j \leq n$, para armazenar a solução ótima para $A_{i..j}$. O valor $m[i,i] = 0$, pois $A_{i..i} = A_i$, não havendo a necessidade de qualquer cálculo.
- Para $i < j$, podemos usar a estrutura ótima delineada na etapa 1. Assim $A_{i..j}$ pode ser dividido em duas partes $A_{i..k}$ e $A_{k+1..j}$, onde $i \leq k < j$. Então $m[i,j]$ é igual ao menor custo para calcular $A_{i..k}$ e $A_{k+1..j}$, mais o custo para multiplicar essas duas matrizes.
- O custo para multiplicar $A_{i..k}$ e $A_{k+1..j}$ é de $p_{i-1} * p_k * p_j$ multiplicações escalares.
- De forma geral, se $m[i, j]$ é número mínimo de multiplicações que deve ser efetuado para computar $A_i \dots A_j$ então $m[i, j]$ é dado por:

$$m[i, j] := \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} * p_k * p_j\}$$

- Os valores de $m[i,j]$ fornecem os custos de soluções ótimas para subproblemas, mas não informações para a construção de uma solução ótima.
- Para facilitar a indicação de uma parentização ótima, basta armazenar na matriz $s[i,j]$ o valor de k usado para o valor ótimo de $m[i,j]$.

c) Calcular o valor de uma solução ótima

- Usando a programação dinâmica passamos a ter $\Theta(n^2)$ subproblemas. O tempo de execução é $O(n^3)$.

MATRIX-CHAIN-ORDER(p)

```

1   $n \leftarrow \text{length}[p] - 1$ 
2  for  $i \leftarrow 1$  to  $n$ 
3      do  $m[i, i] \leftarrow 0$ 
4  for  $l \leftarrow 2$  to  $n$            ▷  $l$  is the chain length.
5      do for  $i \leftarrow 1$  to  $n - l + 1$ 
6          do  $j \leftarrow i + l - 1$ 
7               $m[i, j] \leftarrow \infty$ 
8              for  $k \leftarrow i$  to  $j - 1$ 
9                  do  $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
10                     if  $q < m[i, j]$ 
11                         then  $m[i, j] \leftarrow q$ 
12                              $s[i, j] \leftarrow k$ 
13  return  $m$  and  $s$ 
```

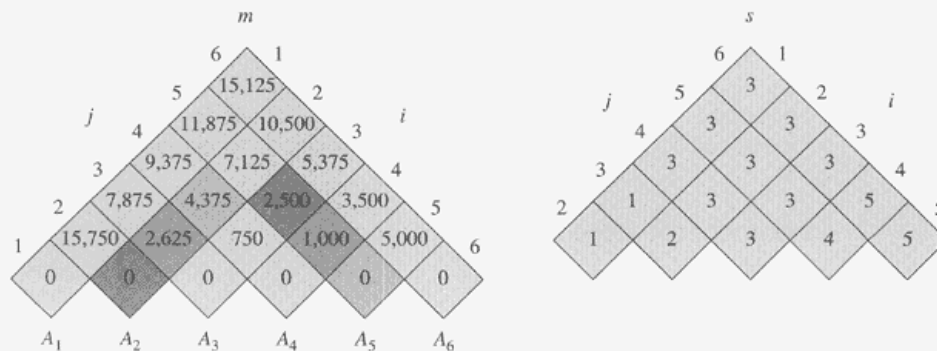


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15,125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$\begin{aligned}
 m[2, 5] &= \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000, \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases} \\
 &= 7125.
 \end{aligned}$$


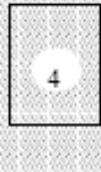

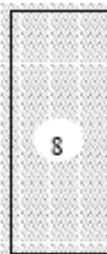

d) Construir uma solução ótima a partir de informações calculadas

- A construção de uma solução ótima se baseia na tabela s .
- Cada entrada $s[i,j]$ registra o valor de k tal que a colocação ótima de parênteses de $A_i..A_j$ divide o produto entre A_k e A_{k+1} .
- Desse modo, sabemos que a multiplicação final de matrizes no cálculo ótimo de $A_1..n$, é $A_1..s[1,n] \times A_{s[1,n]+1}..n$.
- As multiplicações de matrizes anteriores podem ser calculadas recursivamente, pois $s[1,s[1,n]]$ determina a última multiplicação de matrizes no cálculo de $A_1..s[1,n]$, e $s[s[1,n]+1,n]$ determina a última multiplicação de matrizes no cálculo de $A_{s[1,n]+1}..n$.
- O procedimento recursivo PRINT-OPTIMAL-PARENS mostra isso.

```
PRINT-OPTIMAL-PARENS( $s, i, j$ )
1  if  $i = j$ 
2    then print " $A$ " $i$ 
3    else print "("
4         PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )
5         PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )
6         print ")"
```


D) Exemplo: Problema da Mochila

Exemplo

Tamanho:					
Valor:	4	5	10	11	13
Nome:	A	B	C	D	E

- Considerando que a mochila tem capacidade 17, qual é a melhor combinação?
- Abordagem programação dinâmica:
 - Calcular a melhor combinação para todas as mochilas de tamanho até M
 - Cálculo é eficiente se feito na ordem apropriada, itens menores primeiro

```

for( j = 1; j <= N; j++ )
{
    for( i=1; i <= M; i++ )
        if ( i >= size[j] )
            if ( cost[i] < cost[i-size[j]] + val[j] )
            {
                cost[i] = cost[i-size[j]] + val[j];
                best[i] = j;
            }
}

```

- $cost[i]$: maior valor que se consegue com mochila de capacidade i
- $best[i]$: último item acrescentado para obter o máximo
- Calcula-se o melhor valor que se pode obter usando só itens tipo A, para todos os tamanhos de mochila
- Repete-se usando só A's e B's, e assim sucessivamente
- Quando um item j é escolhido para a mochila: o melhor valor que se pode obter é $val[j]$ (do item) mais $cost[i-size[j]]$ (para encher o resto)
- Se o valor assim obtido é superior ao que se consegue sem usar o item j , atualiza-se $cost[i]$ e $best[i]$; senão mantêm-se
- Conteúdo da mochila ótima: recuperado através do vetor $best[i]$
 - $best[i]$ indica o último item da mochila
 - O restante é o indicado para a mochila de tamanho $M-size[best[M]]$
- Eficiência: A solução em programação dinâmica gasta tempo $\Theta(NM)$

	k	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1	cost[k]	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
	best[k]	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2	cost[k]	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
	best[k]	A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3	cost[k]	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
	best[k]	A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4	cost[k]	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
	best[k]	A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=5	cost[k]	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
	best[k]	A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

4. Métodos gulosos ótimos

- Tipicamente algoritmos gulosos são utilizados para resolver problemas de otimização.
- Uma característica comum dos problemas onde se aplicam algoritmos gulosos é a existência de subestrutura ótima, semelhante à programação dinâmica:
 - Programação dinâmica: tipicamente os subproblemas são resolvidos quanto à otimalidade antes de se proceder a escolha de um elemento que irá compor a solução ótima
 - Algoritmo guloso: primeiramente é feita a escolha de um elemento que irá compor a solução ótima e só depois um subproblema é resolvido.
- Um algoritmo guloso sempre faz a escolha que parece ser a melhor a cada iteração, ou seja, de acordo com um critério guloso. É uma decisão localmente ótima.
- Propriedade da escolha gulosa: garante que a cada iteração é tomada uma decisão que irá levar a um ótimo global.
- Em um algoritmo guloso uma escolha que foi feita nunca é revista, ou seja, não há qualquer tipo de retrocesso.

A) Exemplo: Seleção de atividades

- $S = \{a_1 \dots a_n\}$: conjunto de n atividades que podem ser executadas em um mesmo local. Exemplo: palestras em um auditório.
- Para todo $i = 1 \dots n$, a atividade a_i começa no instante s_i e termina no instante f_i , com $0 \leq s_i < f_i$. Ou seja, supõe-se que a atividade a_i será executada no intervalo de tempo (semi-aberto) $[s_i; f_i)$.
- Definição: As atividades a_i e a_j são ditas compatíveis se os intervalos $[s_i; f_i)$ e $[s_j; f_j)$ são disjuntos.

Problema de Seleção de Atividades: Encontre em S um subconjunto de atividades mutuamente compatíveis que tenha tamanho máximo.

Exemplo:

I	1	2	3	4	5	6	7	8	9	10	11
S_i	1	3	0	5	3	5	6	8	8	2	12
F_i	4	5	6	7	8	9	10	11	12	13	14

- Pares de atividades incompatíveis: $(a_1; a_2)$, $(a_1; a_3)$
- Pares de atividades compatíveis: $(a_1; a_4)$, $(a_4; a_8)$
- Conjuntos máximos de atividades compatíveis: $(a_1; a_4; a_8; a_{11})$ e $(a_2; a_4; a_9; a_{11})$
- As atividades estão ordenadas em ordem monotonamente crescente de tempos de término. Isso será importante mais adiante ...

- Inicialmente verificaremos que o problema da seleção de atividades tem a propriedade da sub-estrutura ótima e, então, definiremos recursivamente o valor de uma solução ótima.
- Em seguida, mostraremos que há uma forma de resolver uma quantidade consideravelmente menor de subproblemas do que é feito na programação dinâmica.
- Isto será garantido por uma propriedade de escolha gulosa, a qual dará origem a um algoritmo guloso.

Definição: $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$, i.e., o conjunto de tarefas que começam depois do término de a_i e terminam antes do início de a_j .

- Tarefas artificiais: a_0 com $f_0 = 0$ e a_{n+1} com $s_{n+1} = \infty$
- Tem-se que $S = S_{0;n+1}$ e, com isso, S_{ij} está bem definido para qualquer par $(i; j)$ tal que $0 \leq i; j \leq n + 1$.
- Supondo que $f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n < f_{n+1}$, ou seja, que as tarefas estão ordenadas em ordem crescente de tempos de término, pode-se concluir que $S_{ij} = \emptyset$ para todo $i \geq j$.

Subestrutura ótima:

- considere o subproblema da seleção de atividades definido sobre S_{ij} .
- Suponha que a_k pertence a uma solução ótima de S_{ij} .
- Como $f_i \leq s_k < f_k \leq s_j$, uma solução ótima para S_{ij} que contenha a_k será composta pelas atividades de uma solução ótima de S_{ik} , pelas atividades de uma solução ótima de S_{kj} e por a_k .

Definição: para todo $0 \leq i, j \leq n + 1$, seja $c[i, j]$ o valor ótimo do problema de seleção de atividades para a instância S_{ij} . Deste modo, o valor ótimo do problema de seleção de atividades para instância $S = S_{0; n+1}$ é $c[0, n + 1]$.

Fórmula de recorrência:

$$c[i, j] = \begin{cases} 0 & S_{ij} = \phi \\ \max_{i < k < j: a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & S_{ij} \neq \phi \end{cases}$$

Teorema: (escolha gulosa)

Considere o subproblema definido para uma instância não-vazia S_{ij} , e seja a_m a atividade de S_{ij} com o menor tempo de término, i.e.:

$$f_m = \min \{f_k: a_k \in S_{ij}\}$$

Então:

- a) existe uma solução ótima para S_{ij} que contém a_m e
- b) S_{im} é vazio e o subproblema definido para esta instância é trivial, portanto, a escolha de a_m deixa apenas um dos subproblemas com solução possivelmente não-trivial, já que S_{mj} pode não ser vazio.

SelecionaAtivGulosoRec(s; f; i; j)

- Entrada: vetores s e f com instantes de início e término das atividades $i, i + 1, \dots, j$, sendo $f_i \leq \dots \leq f_j$.
- Saída: conjunto de tamanho máximo de índices de atividades mutuamente compatíveis.

1. $m = i + 1$;
// Busca atividade com menor tempo de término que pode estar em S_{ij}
2. enquanto $m < j$ e $s_m < f_i$ faça $m = m + 1$;
3. se $m \geq j$ então retorne conjunto-vazio;
4. senão
5. se $f_m > s_j$ então retorne conjunto-vazio; // $a_m \notin S_{ij}$
6. senão retorne $\{a_m\} \cup \text{SelecionaAtivGulosoRec}(s, f, m, j)$.

- A chamada inicial será $\text{SelecionaAtivGulosoRec}(s, f, 0, n+1)$.
- Complexidade: $\Theta(n)$.
- Ao longo de todas as chamadas recursivas, cada atividade é examinada exatamente uma vez no laço da linha 2. Em particular, a atividade a_k é examinada na última chamada com $i < k$.
- Como o algoritmo anterior é um caso simples de recursão caudal, é trivial escrever uma versão iterativa para ele.

$\text{SelecionaAtivGulosoIter}(s, f, n)$

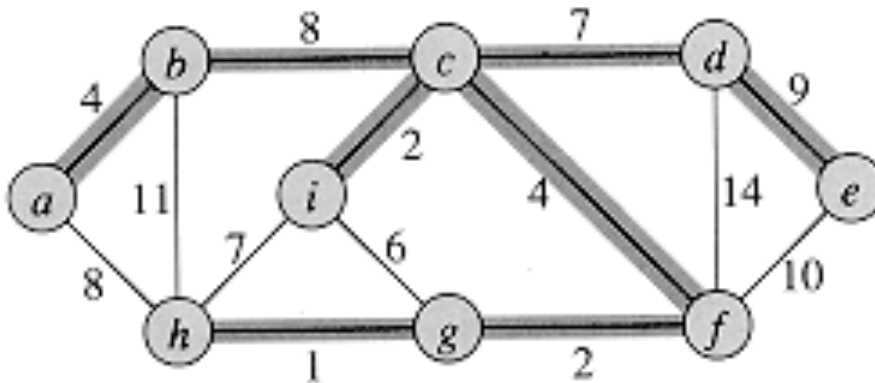
- Entrada: vetores s e f com instantes de início e término das n atividades com os instantes de término em ordem monotonicamente crescente.
- Saída: o conjunto A de tamanho máximo contendo atividades mutuamente compatíveis.

1. $A = \{a_1\}$;
2. $i = 1$;
3. para $m = 2$ até n faça
4. se $s_m \geq f_i$ então
5. $A = A \cup \{a_m\}$;
6. $i = m$;
7. retorne A .

- Complexidade: $\Theta(n)$.

B) Exemplo: Árvore Geradora Mínima


- Árvore Geradora Mínima é a árvore geradora de menor peso de G
- Dado um grafo G com pesos associados às arestas, encontrar uma árvore geradora mínima de G



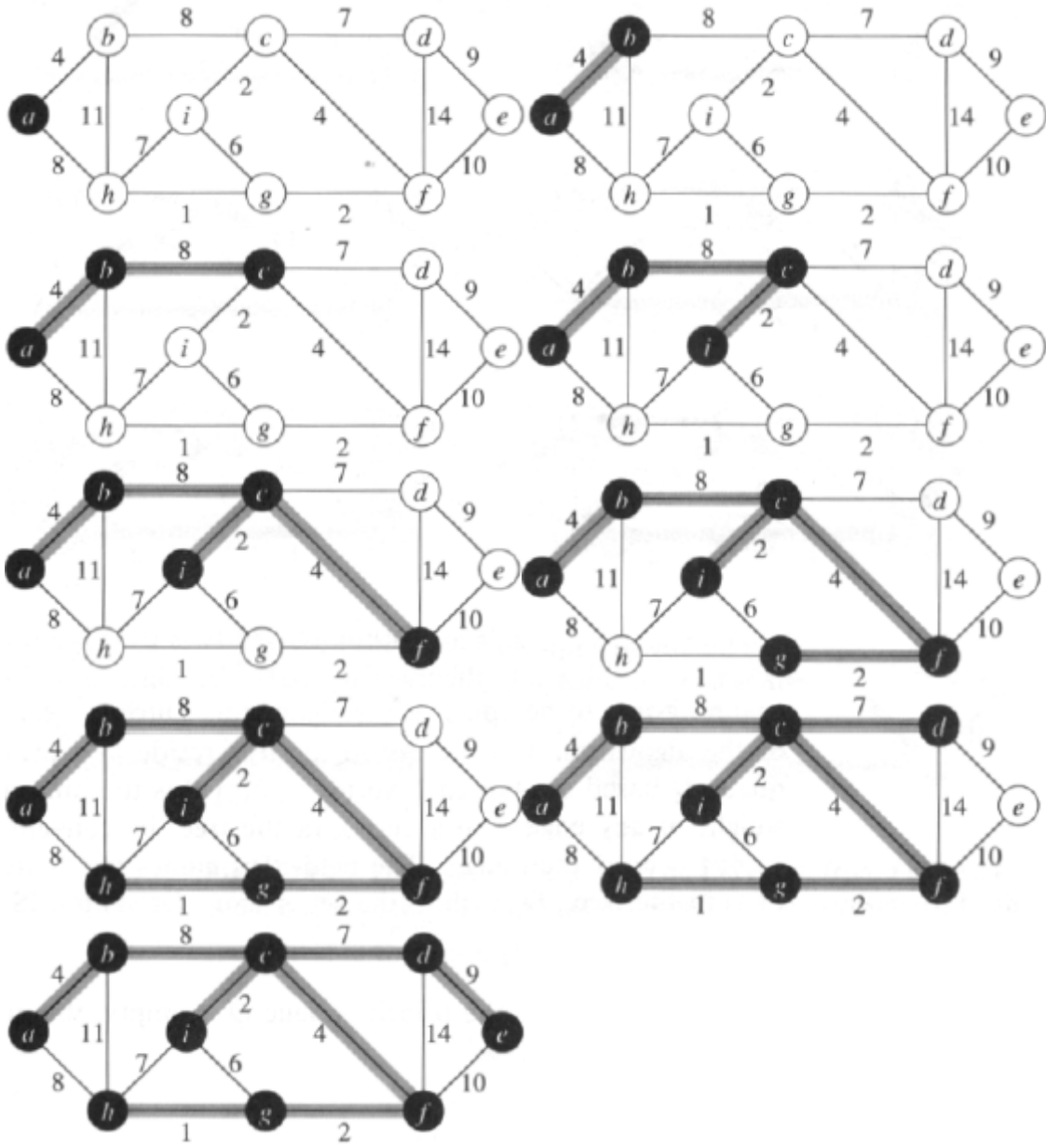
Algoritmo de Prim

- As arestas no conjunto A sempre formam uma árvore única
- A árvore começa a partir de um vértice de raiz arbitrária r e aumenta até a árvore alcançar todos os vértices em V
- Em cada etapa, uma aresta leve conectando um vértice de A a um vértice em $V-A$ é adicionada à árvore
- Quando o algoritmo termina, as arestas em A formam uma árvore geradora mínima
- Durante a execução do algoritmo, todos os vértices que não estão na árvore residem em uma fila de prioridade mínima Q baseada em um campo chave
- Para cada vértice v , $\text{chave}[v]$ é o peso mínimo de qualquer aresta que conecta v a um vértice na árvore
- $\pi[v]$ é o pai de v na árvore

```

MST-PRIM (G, w, r)
for cada  $u \in V[G]$  do
    chave[u]  $\leftarrow \infty$ 
     $\pi[u] \leftarrow \text{NIL}$ 
chave[r]  $\leftarrow 0$ 
 $Q \leftarrow V[G]$ 
while  $Q \neq \emptyset$  do
     $u \leftarrow \text{EXTRACT-MIN}(Q)$   INSERE NA AGM
    for cada  $v \in \text{Adj}[u]$  do
        if  $v \in Q$  e  $w(u, v) < \text{chave}[v]$ 
then
             $\pi[v] \leftarrow u$ 
             $\text{chave}[v] \leftarrow w(u, v)$ 

```



C) Problema da Mochila Fracionário

- Problema da Mochila 0-1 ou 0-1 *Knapsack Problem*:
 - O item i é levado integralmente ou é deixado
- Problema da Mochila Fracionário:
 - Fração do item i pode ser levada

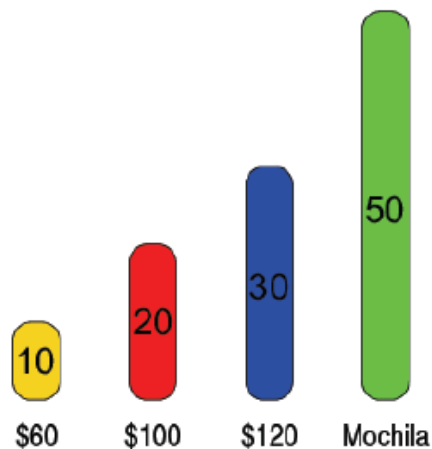
Possuem a propriedade de subestrutura ótima

- Problema inteiro:
 - Considere uma carga que pesa no máximo W com n itens
 - Remova o item j da carga
 - Carga restante deve ser a mais valiosa pesando no máximo $W - w_j$ com $n - 1$ itens
- Problema fracionário:
 - Considere uma carga que pesa no máximo W com n itens
 - Remova um peso w do item j da carga
 - Carga restante deve ser a mais valiosa pesando no máximo $W - w$ com $n - 1$ itens mais o peso $w_j - w$ do item j
- Problema inteiro
 - Não é resolvido usando a técnica gulosa
- Problema fracionário
 - É resolvido usando a técnica gulosa

- Estratégia para resolver o problema fracionário:
 - Calcule o valor por unidade de peso v_i / w_i para cada item
 - Estratégia gulosa é levar tanto quanto possível do item de maior valor por unidade de peso
 - Repita o processo para o próximo item com esta propriedade até alcançar a carga máxima
- Complexidade para resolver o problema fracionário:
 - Ordene os itens i ($i = 1, \dots, n$) pelas frações v_i / w_i
 - $\Theta(n \log n)$

Exemplo:

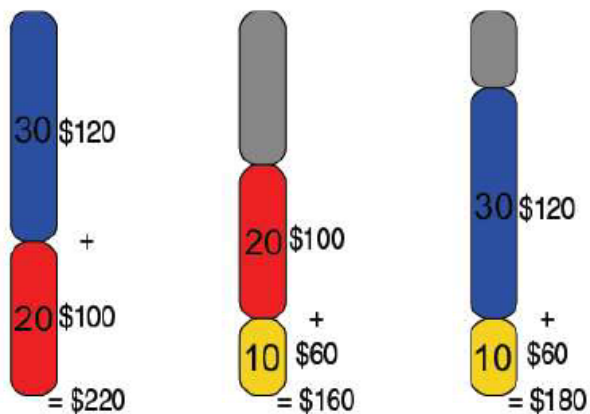
Situação inicial, Problema 0-1



Item	Peso	Valor	V/P
1	10	60	6
2	20	100	5
3	30	120	4

Carga máxima da mochila: 50

Estratégia Gulosa, Problema 0-1



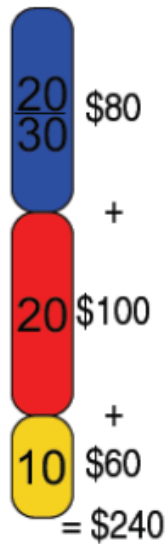
Soluções possíveis:

#	Item (Valor)
1	2 + 3 = 100 + 120 = 220
2	1 + 2 = 60 + 100 = 160
3	1 + 3 = 60 + 120 = 180

→ Solução 2 é a gulosa.

- Considerações:
 - Levar o item 1 faz com que a mochila fique com espaço vazio
 - Espaço vazio diminui o valor efetivo da relação v/w
 - Neste caso deve-se comparar a solução do subproblema quando:
 - ▶ Item é incluído na solução X Item é excluído da solução
 - Passam a existir vários subproblemas
 - Programação dinâmica passa a ser a técnica adequada

Estratégia Gulosa, Problema Fracionário



Item	Peso	Valor	Fração
1	10	60	1
2	20	100	1
3	30	80	2/3

→ Total = 240.

→ Solução ótima!

- O algoritmo é guloso porque, em cada iteração, escolhe o objeto de maior valor específico dentre os disponíveis, sem se preocupar com o que vai acontecer depois. O algoritmo jamais se arrepende do valor atribuído a um componente de x