Técnicas de Desenho de Algoritmos

Ana Paula Tomás

Desenho e Análise de Algoritmos

Novembro 2017

Técnicas de desenho de algoritmos

- Pesquisa exaustiva (*exhaustive search*)
- **Estratégias ávidas**, gananciosas, gulosas (*greedy*)
- **Programação Dinâmica** (*dynamic programmimng*)
- Divisão-e-conquista (*Divide-and-conquer*)
- ...

Programação Dinâmica (DP)

- Obtém a solução à custa de soluções de subproblemas (ou de problemas relacionados).
- A construção é muitas vezes realizada por fases (como nos algoritmos de Floyd-Warshall, Bellman-Ford, Dijkstra, Prim).
- **Cada subproblema só é resolvido uma vez** e a sua solução é memorizada para utilização futura, se necessária.
- DP torna-se particularmente **eficiente** quando a partilha de subproblemas entre os subproblemas é significativa (não se reduz a "divide-and-conquer").
- Em **problemas de otimização**: utilizado quando as soluções ótimas têm subestrutura ótima (por exemplo, caminho mínimo de s para t num grafo), mas não só. Habitualmente, começamos por definir uma recorrência para caraterizar o valor ótimo e uma estratégia ótima em função de valores e estratégias para os subproblemas.

Algoritmo de Floyd-Warshall - aplicação de DP

Problema:
Determinar o comprimento do caminho mínimo de s para t, para **todos os pares** $(s, t) \in V \times V$, $s \neq t$.

- Pode ser resolvido usando o algoritmo de Dijkstra
  - Para cada nó v_i (origem), aplicar o algoritmo de Dijkstra para determinar $D^{\star}_{ij}$, para todo j. Complexidade: $O(|V|(|E| + |V|) \log_2 |V|)$.
  - Para grafos densos, com $|E| \in \Theta(|V|^2)$, seria $O(n^3 \log_2 n)$.
- Mas, o **algoritmo de Floyd-Warshall** (1962), tem complexidade $\Theta(n^3)$.

  Inicialmente: $D_{ii} = 0$, $D_{ij} = d(i, j)$, se $i \neq j$ e $(i, j) \in E$; se não, $D_{ij} = \infty$.

```
AlgoritmoFloyd-Warshall(D, n)
  Para k ← 1 até n fazer
      Para i ← 1 até n fazer
          Para j ← 1 até n fazer
              Se D[i,j] > D[i,k] + D[k,j] então D[i,j] ← D[i,k] + D[k,j];
```

Algoritmo de Floyd-Warshall - aplicação de DP

Problema:
Determinar o comprimento do caminho mínimo de s para t, para **todos os pares** $(s,t) \in V \times V$, $s \neq t$.

- Pode ser resolvido usando o algoritmo de Dijkstra
  - Para cada nó v_i (origem), aplicar o algoritmo de Dijkstra para determinar $D^\star_{ij}$, para todo j. Complexidade: $O(|V|(|E|+|V|)\log_2|V|)$.
  - Para grafos densos, com $|E| \in \Theta(|V|^2)$, seria $O(n^3 \log_2 n)$.
- Mas, o **algoritmo de Floyd-Warshall** (1962), tem complexidade $\Theta(n^3)$.

  Inicialmente: $D_{ii} = 0$, $D_{ij} = d(i,j)$, se $i \neq j$ e $(i,j) \in E$; se não, $D_{ij} = \infty$.

```
AlgoritmoFloyd-Warshall(D, n)
  Para k ← 1 até n fazer
     Para i ← 1 até n fazer
        Para j ← 1 até n fazer
           Se D[i,j] > D[i,k] + D[k,j] então D[i,j] ← D[i,k] + D[k,j];
```

Algoritmo de Floyd-Warshall (cont.)

Seja $G = (V, E, d)$ um grafo dirigido finito, com $d(e) \in \mathbb{R}^+$, para todo $e \in E$. Suponhamos que os nós estão **numerados de** 1 **a** $n = |V|$

- Seja $D_{ij}^{(k)}$ o valor da distância mínima de i para j em G se os percursos só puderem ter os nós $1, 2, \ldots, k$ como nós intermédios, para cada $k \geq 0$, fixo.
- Para todo $(i, j) \in V \times V$,

$$D_{ij}^{(k)} = \min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}), \quad \text{se } k \geq 1$$

$$D_{ij}^{(0)} = \begin{cases} d(i,j), & \text{se } i \neq j \wedge (i,j) \in E \\ \infty & \text{se } i \neq j \wedge (i,j) \notin E \\ 0 & \text{se } i = j \end{cases}$$

- O algoritmo de Floyd-Warshall baseia-se nesta recorrência e no facto de $D_{ij}^{(k+1)} \leq D_{ij}^{(k)}$, o que permite dispensar a construção de matrizes auxiliares.
- A matriz das distâncias mínimas é $D_{ij}^{(n)}$, sendo $n = |V|$. Notar que, nesse caso, qualquer nó pode ser nó intermédio.

Algoritmo de Bellman-Ford (aplicação de DP)

- $G = (V, E, d)$ **pode ter pesos negativos**. O algoritmo determina percursos com peso mínimo de um nó **origem** s para cada nó $v \in V = \{1, 2, \ldots, n\}$.
- Baseia-se no facto de o número de ramos de um percurso com peso mínimo não exceder $n - 1$, a menos que o percurso inclua ciclos com peso negativo. Nesse caso, **não existiria** um percurso com peso mínimo (podia diminuir o peso quanto quiser).
- Inclui um passo para verificar se existem ciclos com peso negativo (nesse caso, as distâncias finais não estariam corretas).

```
ALGORITMO BELLMAN-FORD(s, n)
    Para cada v ∈ V fazer dist[v] ← ∞
    dist[s] ← 0;
    Para r ← 1 até n − 1 fazer
        Para cada (u, v) ∈ E fazer
            Se dist[v] > dist[u] + d(u, v) então dist[v] ← dist[u] + d(u, v)
    Para cada (u, v) ∈ E fazer
        Se dist[v] > dist[u] + d(u, v) então retorna false;    /* ciclos com peso negativo */
    retorna true;    /* sem ciclos com peso negativo */
```

Algoritmo de Bellman-Ford (adaptado)

Para $d(e) \in \mathbb{R}^+$, a matriz das distâncias mínimas $\tilde{D}_{ij}^{(n-1)}$ para **todos os** (i,j) pode ser definida pela recorrência

$$\begin{aligned}
\tilde{D}_{ij}^{(1)} &= \begin{cases} d(i,j) & \text{se } i \neq j \text{ e } (i,j) \in E \\ \infty, & \text{se } i \neq j \text{ e } (i,j) \notin E \\ 0, & \text{se } i = j. \end{cases} \\
\tilde{D}_{ij}^{(r)} &= \min\{\tilde{D}_{ik}^{(r-1)} + \tilde{D}_{kj}^{(1)} \mid 1 \leq k \leq n\} \\
&= \min\{\tilde{D}_{ik}^{(1)} + \tilde{D}_{kj}^{(r-1)} \mid 1 \leq k \leq n\}, \quad \text{se } r \geq 2
\end{aligned}$$

onde $\tilde{D}_{ij}^{(r)}$ é a distância mínima de i para j se o percurso não puder ter mais do que r ramos, para cada $r \geq 1$. Calcula-se como um **produto de matrizes** $\otimes$ em $(\mathbb{R}, \min, +)$, sendo um **método multiplicativo**. Os percursos mínimos têm subestrutura ótima, pelo que $\tilde{D}^{(r+s)} = \tilde{D}^{(r)} \otimes \tilde{D}^{(s)}$ e é dada por

$$(\tilde{D}^{(r)} \otimes \tilde{D}^{(s)})_{ij} = \min_{1 \leq k \leq n} (\tilde{D}_{ik}^{(r)} + \tilde{D}_{kj}^{(s)})$$

Para reduzir o número de multiplicações $\otimes$ de $\Theta(n)$ para $\Theta(\log_2 n)$ podemos usar **o método binário para cálculo de potências**: $x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n\%2} = \prod_{t=0}^{\lfloor \log_2 n \rfloor} (x^{2^t})^{b_t}$, onde b_t é o bit t da representação de n em binário. Por exemplo, $\tilde{D}^{19} = \tilde{D}^{16} \otimes \tilde{D}^2 \otimes \tilde{D}$.

Cálculo do fecho transitivo de uma relação binária

- Um grafo dirigido $G = (V, E)$ representa uma **relação binária** R definida no conjunto V. O conjunto de ramos corresponde ao conjunto de pares ordenados que definem R (recordar que $R \subseteq V \times V$, por definição).
- R é transitiva se $((x, y) \in R \wedge (y, z) \in R) \Rightarrow (x, z) \in R$, para todo (x, y, z).
- O fecho transitivo de R denota-se por R^+ e é a menor relação binária definida em V que é transitiva e contém R. **Menor** para $\subseteq$.
- Usando a composta de relações, define-se $R^1 = R$ e $R^{i+1} = R^i R = RR^i$. É conhecido que:
  - $(x, y) \in R^+$ **sse existir um percurso de x para y no grafo de R**.
  - $(x, y) \in R^i$ sse existir um percurso de x para y com i ramos no grafo de R, para $i \geq 1$. Prova-se que $R^+ = \bigcup_{i=1}^{n} R^i$, sendo $n = |V|$.

Algoritmo de Warshall para cálculo do fecho transitivo

- A matriz da relação binária R é uma **matriz de booleanos** dada por

$$M_{ij} = \begin{cases} 1 & \text{se } (i,j) \in R \\ 0 & \text{se } (i,j) \notin R \end{cases}$$

- À semelhança do algoritmo de Floyd-Warshall, seja $M_{ij}^{(k)} = 1$ se existir algum **percurso no grafo de R do nó i para o nó j que, quando muito, use nós numerados até k como nós intermédios**.
- Então, $M_{ij}^{(0)} = M_{ij}$ e $(i,j) \in R^+$ sse $M_{ij}^+ = M_{ij}^{(n)} = 1$, para $V = \{1, \ldots, n\}$.

Algoritmo Warshall(M, n)

1 Para $k \leftarrow 1$ até n fazer
2   Para $i \leftarrow 1$ até n fazer
3   Para $j \leftarrow 1$ até n fazer
4             $M[i,j] \leftarrow M[i,j] \vee (M[i,k] \wedge M[k,j])$;

(Linha 4) explora propriedades de R^+; mais eficiente do que $M^{(k)}[i,j] \leftarrow M^{(k-1)}[i,j] \vee (M^{(k-1)}[i,k] \wedge M^{(k-1)}[k,j])$;

Outra abordagem DP para cálculo do fecho transitivo

- *Podemos adaptar ideia do algoritmo de Bellman-Ford.*
- Definimos $\tilde{M}_{ij}^{(r)} = 1$ se existir algum **percurso no grafo de R do nó i para o nó j com <u>até r ramos</u>, para $r \geq 1$, fixo**.
- Recorrência: para todos os pares (i, j) tem-se

$$\begin{aligned} \tilde{M}_{ij}^{(1)} &= M_{ij} \\ \tilde{M}_{ij}^{(r)} &= \tilde{M}_{ij}^{(r-1)} \vee \left(\bigvee_{k=1}^{n} (\tilde{M}_{ik}^{(r-1)} \wedge M_{kj})\right), \qquad \text{para } r \geq 2 \end{aligned}$$

- Podemos avaliar usando o método multiplicativo e adaptar o método binário para reduzir o número de multiplicações, pois

$$\tilde{M}^{(r+s)} = \tilde{M}^{(r)} \otimes \tilde{M}^{(s)}$$

onde o produto de matrizes $\otimes$ é considerado em $(\{0, 1\}, \vee, \wedge)$.

- **Pontos fixos:** Se $\tilde{M}^{(r)} = \tilde{M}^{(r+1)}$, então $\tilde{M}^{(r)} = M^{+}$. Também, no algoritmo de Bellman-Ford (adaptado), se $\tilde{D}^{(r)} = \tilde{D}^{(r+1)}$, então $\tilde{D}^{(r)} = \tilde{D}^{(n)}$.

Aplicação de DP para obter expressões regulares para AFs – Método de Kleene

Dado um autómato finito $A = (S, \Sigma, \delta, s_1, F)$, com estados numerados de 1 a n, seja $r_{ij}^{(k)}$ a expressão que descreve a linguagem determinada pelos percursos de i para j que passam quando muito por estados intermédios *etiquetados* com números não superiores a k.

$$r_{ii}^{(0)} = \begin{cases} \varepsilon & \textit{sse} \quad \text{não existe qualquer lacete em } i \\ \varepsilon + a_1 \ldots + a_p & \textit{sse} \quad \text{os lacetes em } i \text{ estão etiquetados com } a_1, \ldots, a_p \end{cases}$$

$$r_{ij}^{(0)} = \begin{cases} \emptyset & \textit{sse} \quad \text{não existe qualquer arco } (i,j) \\ a_1 + \ldots + a_p & \textit{sse} \quad a_1, \ldots, a_p \text{ etiquetam os arcos } (i,j) \end{cases}$$

Define-se agora $r_{ij}^{(k)}$, para $k \geq 1$, recursivamente assim:

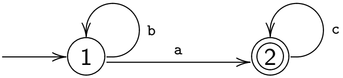
$$r_{ij}^{(k)} = r_{ij}^{(k-1)} + r_{ik}^{(k-1)} (r_{kk}^{(k-1)})^{\star} r_{kj}^{(k-1)}$$

onde $\star$ é o (habitual) fecho de Kleene. A expressão que define a linguagem reconhecida pelo autómato é dada por:

$$\sum r_{1s}^{(n)}$$

Método de Kleene para obter expressões regulares para AFs

Muito trabalhoso...



$$r_{11}^{(0)} = \varepsilon + \mathtt{b} \qquad r_{22}^{(0)} = \varepsilon + \mathtt{c} \qquad r_{12}^{(0)} = \mathtt{a} \qquad r_{21}^{(0)} = \emptyset$$

$$\begin{aligned}
r_{11}^{(1)} &= r_{11}^{(0)} + r_{11}^{(0)}(r_{11}^{(0)})^\star r_{11}^{(0)} = \varepsilon + \mathtt{b} + (\varepsilon + \mathtt{b})(\varepsilon + \mathtt{b})^\star(\varepsilon + \mathtt{b}) = \mathtt{b}^\star \\
r_{22}^{(1)} &= r_{22}^{(0)} + r_{21}^{0}(r_{11}^{(0)})^\star r_{12}^{(0)} = \varepsilon + \mathtt{c} + \emptyset(\varepsilon + \mathtt{b})^\star \mathtt{a} = \varepsilon + \mathtt{c} \\
r_{12}^{(1)} &= r_{12}^{(0)} + r_{11}^{(0)}(r_{11}^{(0)})^\star r_{12}^{(0)} = \mathtt{a} + (\varepsilon + \mathtt{b})(\varepsilon + \mathtt{b})^\star \mathtt{a} = \mathtt{b}^\star \mathtt{a} \\
r_{21}^{(1)} &= r_{21}^{(0)} + r_{21}^{(0)}(r_{11}^{(0)})^\star r_{11}^{(0)} = \emptyset
\end{aligned}$$

$$\begin{aligned}
r_{11}^{(2)} &= r_{11}^{(1)} + r_{12}^{(1)}(r_{22}^{(1)})^\star r_{21}^{(1)} = r_{11}^{(1)} = \mathtt{b}^\star \\
r_{12}^{(2)} &= r_{12}^{(1)} + r_{12}^{(1)}(r_{22}^{(1)})^\star r_{22}^{(1)} = \mathtt{b}^\star \mathtt{a} + \mathtt{b}^\star \mathtt{a}(\varepsilon + \mathtt{c})^\star(\varepsilon + \mathtt{c}) = \mathtt{b}^\star \mathtt{a} \mathtt{c}^\star \\
r_{22}^{(2)} &= r_{22}^{(1)} + r_{22}^{(1)}(r_{22}^{(1)})^\star r_{22}^{(1)} = \mathtt{c}^\star \\
r_{21}^{(2)} &= \emptyset
\end{aligned}$$

Conclusão: a expressão que descreve a linguagem aceite pelo AF é $r_{12}^{(2)}$ ou seja, $b^\star a c^\star$.

Se 1 e 2 fossem estados finais seria $r_{11}^{(2)} + r_{12}^{(2)} = \mathtt{b}^\star + \mathtt{b}^\star \mathtt{a} \mathtt{c}^\star$.

Algoritmo CYK para decidir se $x \in \mathcal{L}(G)$, para GIC G na forma normal de Chomsky, e $x \in \Sigma^\star$

- Para $G = (V, \Sigma, P, S)$ fixa, a complexidade temporal do algoritmo é $O(|x|^3)$, ou seja, é cúbica no comprimento da palavra que se pretende analisar.
- Seja $N[i, i+s]$ o conjunto de variáveis em V que geram a subpalavra $x_i \ldots x_{i+s}$ de x, isto é, $N[i, i+s] = \{A \mid \ A \in V,\ A \Rightarrow^\star_{\mathcal{G}} x_i \ldots x_{i+s}\}$.
- **Algoritmo CYK:**
  - $N[i,i] := \{A \mid A \in V,\ A \rightarrow x_i\}$, para $1 \leq i \leq n$ e $N[i,j] := \emptyset$, para todo (i,j), com $i \neq j$.
  - Para cada s entre 1 e $n-1$ fazer

    Para cada i entre 1 e $n-s$, considerar $N[i,k]$ e $N[k+1, i+s]$, para todo k com $i \leq k \leq (i+s)-1$. Se existir $(A \rightarrow BC) \in P$ com $B \in N[i,k]$ e $C \in N[k+1, i+s]$, acrescentar A a $N[i, i+s]$.
  - A palavra x está em $\mathcal{L}(\mathcal{G})$ se e só se $S \in N[1, n]$.

Algoritmo CYK – aplicação de DP

É usual usar uma matriz, com $N[t, t+s]$ na coluna t e linha $\#s{+}1$.

$\#n$	$N[1,n]$					
$\#n{-}1$	$N[1,n{-}1]$	$N[2,n]$				
$\vdots$	$\vdots$	$\vdots$	$\vdots$			
$\#3$	$N[1,3]$	$N[2,4]$	$\cdots$	$N[n{-}2,n]$		
$\#2$	$N[1,2]$	$N[2,3]$	$\cdots$	$N[n{-}2,n{-}1]$	$N[n{-}1,n]$	
$\#1$	$N[1,1]$	$N[2,2]$	$\cdots$	$N[n{-}2,n{-}2]$	$N[n{-}1,n{-}1]$	$N[n,n]$
	x_1	x_2	$\cdots$	x_{n-2}	x_{n-1}	x_n

A entrada $N[t, t+s]$ da tabela apresenta o conjunto das categorias possíveis para a subpalavra $x_t \cdots x_{t+s}$ de x. Portanto, carateriza as categorias das subpalavras indicadas na matrix seguinte:

$\#n$	$x_1 \cdots x_n$					
$\#n{-}1$	$x_1 \cdots x_{n-1}$	$x_2 \cdots x_n$				
$\vdots$	$\vdots$	$\vdots$	$\vdots$			
$\#3$	$x_1x_2x_3$	$x_2x_3x_4$	$\cdots$	$x_{n-2}x_{n-1}x_n$		
$\#2$	x_1x_2	x_2x_3	$\cdots$	$x_{n-2}x_{n-1}$	$x_{n-1}x_n$	
$\#1$	x_1	x_2	$\cdots$	x_{n-2}	x_{n-1}	x_n

Algoritmo CYK - Exemplo

Para GIC G, com $V = \{E, T, F, E_1, E_2, T_1, T_2, T_3, M, S, X, Q, A, B\}$, símbolo inicial E, e seguintes produções:

$$
\begin{array}{lcl}
E & \rightarrow & TE_1 \mid TE_2 \mid FT_1 \mid FT_2 \mid AT_3 \mid \mathtt{n} \\
T & \rightarrow & \mathtt{n} \mid FT_1 \mid FT_2 \mid AT_3 \\
F & \rightarrow & AT_3 \mid \mathtt{n} \\
E_1 & \rightarrow & ME \\
E_2 & \rightarrow & SE \\
T_1 & \rightarrow & XT \\
T_2 & \rightarrow & QT
\end{array}
\qquad\qquad
\begin{array}{lcl}
M & \rightarrow & \texttt{+} \\
S & \rightarrow & \texttt{-} \\
X & \rightarrow & \texttt{*} \\
Q & \rightarrow & \texttt{/} \\
A & \rightarrow & \texttt{(} \\
B & \rightarrow & \texttt{)} \\
T_3 & \rightarrow & EB
\end{array}
$$

tem-se $(\mathtt{n} + \mathtt{n}) * \mathtt{n} \in \mathcal{L}(\mathcal{G})$? Sim, porque E ocorre no topo da tabela:

#7	$\{T, E\}$						
#6	$\emptyset$	$\emptyset$					
#5	$\{F, T, E\}$	$\emptyset$	$\emptyset$				
#4	$\emptyset$	$\{T_3\}$	$\emptyset$	$\emptyset$			
#3	$\emptyset$	$\{E\}$	$\emptyset$	$\emptyset$	$\emptyset$		
#2	$\emptyset$	$\emptyset$	$\{E_1\}$	$\{T_3\}$	$\emptyset$	$\{T_1\}$	
#1	$\{A\}$	$\{E, T, F\}$	$\{M\}$	$\{E, T, F\}$	$\{B\}$	$\{X\}$	$\{E, T, F\}$
	(	n	+	n	)	*	n

Contagem de percursos em grafos

Problema:
Dado um grafo dirigido finito $G = (V, E)$, com n nós numerados de 0 a $n-1$, determinar o número de percursos de v_i para v_j, para todos os pares (v_i, v_j) de nós do grafo. Esses valores devem ser guardados numa matriz M, fazendo $M[i,j] = -1$ se existir uma infinidade de percursos de v_i para v_j.

Resolução:
Seja C_{ij}^k o número de percursos de i para j que apenas podem ter como nós intermédios os numerados até k, com $k \geq -1$ fixo. Define-se pela recorrência

$$C_{ij}^{-1} = \begin{cases} 1, \text{ se } (i,j) \in E \\ 0, \text{ se } (i,j) \notin E \end{cases}$$

$$C_{ij}^{k} = C_{ik}^{k-1} \times (C_{kk}^{k-1})^\star \times C_{kj}^{k-1} + C_{ik}^{k-1}$$

em que $\times$ e $+$ (extensão das operações habituais a $\mathbb{R}_0^+ \cup \{\infty\}$) e $\star$ satisfazem:

$0^\star = 1$
$\infty \times 0 = 0 \times \infty = 0$
$\infty \times y = y \times \infty = \infty, \;$ se $y \neq 0$

$y^\star = \infty, \;$ se $y \neq 0$
$\infty + y = y + \infty = \infty, \;$ para todo y

Contagem de percursos – Implementação em C

Se $M_{ij}^{k-1} = -1$ ou se $M_{ik}^{k-1} \times M_{kj}^{k-1} \neq 0$ e algum dos valores M_{ik}^{k-1}, M_{kj}^{k-1} e M_{kk}^{k-1} for -1, então $M_{ij}^{k} = -1$. Nos outros casos, $M_{ij}^{k} = M_{ik}^{k-1} \times M_{kj}^{k-1} + M_{ij}^{k-1}$.

```
void contacaminhos(int n,int M[][MAX])
{ int aux[MAX][MAX], k,  i, j;

  for(k=0; k < n ; k++) {
    // copia M para aux
    for (i=0; i < n; i++)
       for (j=0; j < n; j++) aux[i][j] = M[i][j];
    // atualiza a contagem
    for (i=0; i < n; i++)
      for (j=0; j < n; j++)
       if (a[i][j] != -1) {
         if (aux[i][k]*aux[k][j])
           if(aux[k][k] || aux[i][k]== -1 || aux[k][j]== -1) M[i][j] = -1;
           else M[i][j] += aux[i][k]*aux[k][j];
      }
  }
}
```

Caixotes de Morangos - aplicação de DP

O dono de uma pequena cadeia de ($L \geq 1$) mercearias adquiriu ($C \geq 1$) caixotes de morangos e tem que decidir quantos caixotes enviar para cada uma das suas lojas, de forma a maximizar o lucro. Devido às características específicas de cada loja (localização, capacidade de armazenamento, número médio de clientes, etc.), o lucro esperado com a venda dos morangos varia, não só de loja para loja, como, também, consoante o número de caixotes enviados para cada loja. É conhecido o lucro do envio de n caixotes para cada uma das lojas, para cada $n \in [0, C]$. Naturalmente, é nulo se não enviar nenhum caixote. Por razões administrativas, cada caixote é indivisível (i.e., o seu conteúdo não pode ser repartido por várias lojas). Não é necessário enviar caixotes para todas as lojas. Como efectuar a distribuição?

(Margarida Mamede, UNL, adaptado)

Caixotes de Morangos

Exemplo de dados:

```
3 5
1.50 2.50 2.00
3.50 5.00 3.00
4.50 5.50 5.50
6.00 5.50 6.00
6.50 5.50 6.00
```

Na coluna j tem os lucros v_{ij} do envio de i caixotes para a loja j, com $i = 1, 2, \ldots, C$, e $j = 1, 2, \ldots, L$.

Admitimos ainda que $v_{0j} = 0$, para todo j.

Neste exemplo, tem $L = 3$ lojas e $C = 5$ caixotes.

Seja $z_{k,j}$ o lucro ótimo se enviar no total k caixotes para as j primeiras lojas, com k e j fixos. Então, $z_{k,j}$ é definido pela recorrência:

$$\begin{array}{rcl} z_{0,j} & = & 0, \quad \text{para } 1 \leq j \leq L \\ z_{k,1} & = & v_{k,1}, \quad \text{para } 1 \leq k \leq C \\ z_{k,j} & = & \max\limits_{0 \leq t \leq k} (v_{k-t,j} + z_{t,j-1}), \quad \text{para } 1 \leq k \leq C, \text{ e } 2 \leq j \leq L , \end{array}$$

Para $j \geq 2$, calculam-se os lucros das soluções que enviam $k - t$ caixotes à loja j e distribuem *otimamente os restantes* t pelas lojas numeradas até $j - 1$, para $0 \leq t \leq k$. A solução de maior valor define $z_{k,j}$.

Caixotes de Morangos (cont.)

$$
\begin{aligned}
z_{0,j} &= 0, \quad \text{para } 1 \leq j \leq L \\
z_{k,1} &= v_{k,1}, \quad \text{para } 1 \leq k \leq C \\
z_{k,j} &= \max_{0 \leq t \leq k} (v_{k-t,j} + z_{t,j-1}), \quad \text{para } 1 \leq k \leq C, \text{ e } 2 \leq j \leq L
\end{aligned}
$$

- **Algoritmos baseados em programação dinâmica podem gastar muita memória**. É necessário evitar, se possível, gastos de memória excessivos.
- Neste caso, não precisamos de uma matriz $(C+1) \times L$ para guardar $z_{k,j}$, pois $z_{k,j}$ só depende dos valores de $z_{t,j-1}$.
- Bastariam dois *arrays* com $C+1$ posições, para guardar os valores de $z_{k,j-1}$ e de $z_{k,j}$, para todo k.
- Se analisarmos com mais cuidado, podemos concluir que, de facto, **basta um array** $Z[\cdot]$, sendo $Z[k] = z_{k,j}$, pois $z_{k,j}$ só depende dos valores de $z_{t,j-1}$, para $t \leq k$. Para tal, **na atualização** de $Z[k]$ **para um novo** j, tem de se **começar pelo valor mais alto de** k, tomando $k = C, C-1, \ldots, 2, 1$.

Caixotes de Morangos – *DP construção "bottom-up"*

$\textsc{CaixotesMorangos}(V, L, C, Z)$

```
0  Z[0] ← 0;
1  Para k ← 1 até C fazer Z[k] ← V[k,1];
2  Para j ← 2 até L fazer
3    Para k ← C até 1 com decremento de 1 fazer
4      Para t ← 0 até k − 1 fazer      /* NB: inicialmente Z[k] é já V[0,j] + Z[k] */
5        Se V[k − t, j] + Z[t] > Z[k] então
6          Z[k] ← V[k − t, j] + Z[t];
```

Complexidade:
Passando V e Z por referência e C e L por valor, a **complexidade temporal** é $\Theta(LC^2)$ e a **espacial** (adicional) é $\Theta(C)$. "Adicional" porque não contabiliza o espaço $\Theta(LC)$ ocupado pela matriz de dados V, mas apenas Z.

Justificação (sucinta): A complexidade temporal do **ciclo 4-6** é $\Theta(k)$. Logo, para o **ciclo 3-6** é $\Theta(\sum_{k=1}^{C} k) = \Theta(C(C+1)/2) = \Theta(C^2)$ e, portanto, para o **ciclo 2-6** é $\Theta(LC^2)$. Assim, o **bloco 1-6** tem complexidade $\Theta(C + LC^2) = \Theta(LC^2)$.

Se os dados fossem lidos de um ficheiro e se desse V^T (a transposta de V) em vez de V, o espaço total podia ser $\Theta(C)$.

Ver problema da aula prática: Caixotes de Morangos II (não passar V; ler lucro da loja j dentro da função e atualizar Z)

Caixotes de Morangos – *DP construção "bottom-up"*

```
CaixotesMorangos(V, L, C, Z)
0  Z[0] ← 0;
1  Para k ← 1 até C fazer Z[k] ← V[k, 1];
2  Para j ← 2 até L fazer
3    Para k ← C até 1 com decremento de 1 fazer
4      Para t ← 0 até k − 1 fazer     /* NB: inicialmente Z[k] é já V[0, j] + Z[k] */
5        Se V[k − t, j] + Z[t] > Z[k] então
6          Z[k] ← V[k − t, j] + Z[t];
```

Complexidade:
Passando V e Z por referência e C e L por valor, a **complexidade temporal** é $\Theta(LC^2)$ e a **espacial** (adicional) é $\Theta(C)$. "Adicional" porque não contabiliza o espaço $\Theta(LC)$ ocupado pela matriz de dados V, mas apenas Z.

Justificação (sucinta): A complexidade temporal do **ciclo 4-6** é $\Theta(k)$. Logo, para o **ciclo 3-6** é $\Theta(\sum_{k=1}^{C} k) = \Theta(C(C+1)/2) = \Theta(C^2)$ e, portanto, para o **ciclo 2-6** é $\Theta(LC^2)$. Assim, o **bloco 1-6** tem complexidade $\Theta(C + LC^2) = \Theta(LC^2)$.

Se os dados fossem lidos de um ficheiro e se desse V^T (a transposta de V) em vez de V, o espaço total podia ser $\Theta(C)$.

Ver problema da aula prática: Caixotes de Morangos II (não passar V; ler lucro da loja j dentro da função e atualizar Z)

Problemas de trocos com número de moedas limitado

- O problema "Não lhes dês troco" usa uma **estratégia ávida** (*greedy*) para dar o troco, que nem sempre permite obter o montante pretendido.
- De quantas formas conseguiria obter uma quantia Q dada se não usar essa estratégia? Seja d_k **o número de moedas disponíveis de valor** v_k, para $1 \leq k \leq m$. Admita-se que $v_k < v_{k+1}$, para todo $k < m$.
- Se puder usar apenas moedas de valor $v_1, \ldots, v_k$, o número de formas $N_{q,k}$ de obter q pode ser definido recursivamente assim:

$N_{0,k} = 1$, para $1 \leq k \leq m$ (não dar moeda nenhuma se $q = 0$)

$$N_{q,1} = \begin{cases} 1 & \text{se } q > 0 \wedge q\% v_1 = 0 \wedge d_1 \geq \frac{q}{v_1} \\ 0 & \text{se } q > 0 \wedge (q\% v_1 \neq 0 \vee d_1 < \frac{q}{v_1}) \end{cases}$$

$N_{q,k} = \sum_{r=0}^{\min(d_k, \lfloor q/v_k \rfloor)} N_{q-rv_k,k-1}$, para todo $q > 0$ e $1 < k \leq m$.

O valor procurado é $N_{Q,m}$. Dependendo de Q e dos valores das moedas disponíveis, pode acontecer que nem todos os pares (q, k), com $q \leq Q$, precisem de ser calculados.

Problemas de trocos com número de moedas limitado

Abordagem "Top-Down" com memoização

$\textsc{ContaSols}(v, d, q, k)$ /* chamar $\textsc{ContaSols}(v, d, Q, m)$ para obter $N_{Q,m}$ */

Se $q = 0$ então retorna 1;
Se $k = 1$ então
  Se $v[1]\%q \neq 0 \vee d[1] < q/v[1]$ então retorna 0;
  retorna 1;
Se $N[q, k]$ já calculado então retorna $N[q, k]$;
$rmax \leftarrow \min(d[k], \lfloor q/v[k] \rfloor)$;
$conta \leftarrow 0$;
Para $r \leftarrow 0$ até $rmax$ fazer
  $conta \leftarrow conta + \textsc{ContaSols}(v, d, q - r * v[k], k - 1)$;
$N[q, k] \leftarrow conta$; /* memoriza para uso futuro se necessário */
retorna $conta$;

Implementação: Definir a tabela N por **dicionário** (*hash-table*)

*Dicionários/Tabelas de dispersão/*Arrays *associativos – coleção de pares* **(Chave,Valor)**.

Java: Map, HashMap, TreeMap C++: std::unordered_map, std::map

Problemas de trocos com número de moedas limitado

Abordagem "Top-Down" com memoização

$\text{ContaSols}(v, d, q, k)$ /* chamar $\text{ContaSols}(v, d, Q, m)$ para obter $N_{Q,m}$ */

Se $q = 0$ então retorna 1;
Se $k = 1$ então
    Se $v[1]\%q \neq 0 \vee d[1] < q/v[1]$ então retorna 0;
    retorna 1;
Se $N[q, k]$ já calculado então retorna $N[q, k]$;
$rmax \leftarrow \min(d[k], \lfloor q/v[k] \rfloor)$;
$conta \leftarrow 0$;
Para $r \leftarrow 0$ até $rmax$ fazer
    $conta \leftarrow conta + \text{ContaSols}(v, d, q - r * v[k], k - 1)$;
$N[q, k] \leftarrow conta$; /* memoriza para uso futuro se necessário */
retorna $conta$;

Implementação: Definir a tabela N por **dicionário** (*hash-table*)

*Dicionários/Tabelas de dispersão/*Arrays *associativos – coleção de pares* **(Chave,Valor)**.

Java: `Map`, *HashMap*, *TreeMap* *C++: std::unordered_map*, *std::map*

Problemas de trocos com número de moedas ilimitado

Problema: Supondo que se tem um número não limitado de moedas de valores 200, 100, 50, 20, 10, 5, 2, e 1, qual é o **número mínimo** de moedas necessário para formar uma quantia Q?

- Abordagem de programação dinâmica é **ineficiente**.
- Prova-se que a **estratégia greedy** que consiste em começar por **usar a moeda de valor mais alto $v_k \geq Q$ o número máximo de vezes que puder** (isto é, $n_k = \lfloor Q/v_k \rfloor$ vezes) e aplicar a mesma estratégia para obter a quantia $Q - n_k v_k$ restante, determina a **solução ótima**, em $O(m)$, sendo m o número de tipos de moedas existentes.

Para garantir $O(m)$, é importante usar $Q - n_k v_k$ em vez de dar uma moeda v_k e aplicar a estratégia a $Q - v_k$. Note que $O(Q)$ é $O(2^{\log_2 Q})$ e, portanto, é exponencial no tamanho da representação de Q (input) em binário (assumido no modelo RAM para análise assintótica).

Problemas de trocos com número de moedas ilimitado

Demonstração de que a estratégia greedy determina a solução ótima

- Seja $x^\star$ uma solução ótima para a quantia Q. Seja $x^\star_v$ é o número de moedas que usa de valor v.
- Se $x^\star_{100} > 1$, a solução não seria ótima (podia reduzir o número de moedas se substituir duas de 100 por uma de 200). Portanto, $x^\star_{100} \leq 1$.
Analogamente se conclui que: $x^\star_{50} \leq 1$, $x^\star_{10} \leq 1$, e $x^\star_1 \leq 1$.
- Se $x^\star_{20} > 2$ então a solução não seria ótima porque podia trocar três moedas de 20 por uma de 50 e uma de 10. Portanto, $x^\star_{20} \leq 2$. Analogamente, $x^\star_2 \leq 2$.
- Não pode ter simultaneamente $x^\star_2 = 2$ e $x^\star_1 = 1$, pois a solução não seria ótima (podia substituir essas três moedas por uma de 5). Portanto $2x^\star_2 + x^\star_1 \leq 4$.
Também não tem simultaneamente $x^\star_{20} = 2$ e $x^\star_{10} = 1$.
- Como $2x^\star_2 + x^\star_1 \leq 4$, $x^\star_5 \leq 1$ e $x^\star_{10} \leq 1$ então $5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 9$ e $10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 19$. Analogamente, se deduz que
$20x^\star_{20} + 10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 49$, $50x^\star_{50} + 20x^\star_{20} + 10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 99$.
$100x^\star_{100} + 50x^\star_{50} + 20x^\star_{20} + 10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 199$.
- Tem-se $\sum_{i=1}^{k} v_i x^\star_{v_i} < v_{k+1}$, para todo k. Portanto, $x^\star$ tem exatamente o mesmo valor que a solução greedy. □

NB: A estratégia *greedy* apresentada não seria correta para, por exemplo, $V = \{1, 300, 1000\}$, $Q = 1200$.

Problemas de trocos com número de moedas ilimitado

Demonstração de que a estratégia greedy determina a solução ótima

- Seja $x^\star$ uma solução ótima para a quantia Q. Seja $x_v^\star$ é o número de moedas que usa de valor v.
- Se $x_{100}^\star > 1$, a solução não seria ótima (podia reduzir o número de moedas se substituir duas de 100 por uma de 200). Portanto, $x_{100}^\star \leq 1$.
  Analogamente se conclui que: $x_{50}^\star \leq 1$, $x_{10}^\star \leq 1$, e $x_1^\star \leq 1$.
- Se $x_{20}^\star > 2$ então a solução não seria ótima porque podia trocar três moedas de 20 por uma de 50 e uma de 10. Portanto, $x_{20}^\star \leq 2$. Analogamente, $x_2^\star \leq 2$.
- Não pode ter simultaneamente $x_2^\star = 2$ e $x_1^\star = 1$, pois a solução não seria ótima (podia substituir essas três moedas por uma de 5). Portanto $2x_2^\star + x_1^\star \leq 4$.
  Também não tem simultaneamente $x_{20}^\star = 2$ e $x_{10}^\star = 1$.
- Como $2x_2^\star + x_1^\star \leq 4$, $x_5^\star \leq 1$ e $x_{10}^\star \leq 1$ então $5x_5^\star + 2x_2^\star + x_1^\star \leq 9$ e $10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 19$. Analogamente, se deduz que $20x_{20}^\star + 10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 49$, $50x_{50}^\star + 20x_{20}^\star + 10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 99$. $100x_{100}^\star + 50x_{50}^\star + 20x_{20}^\star + 10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 199$.
- Tem-se $\sum_{i=1}^{k} v_i x_{v_i}^\star < v_{k+1}$, para todo k. Portanto, $x^\star$ tem exatamente o mesmo valor que a solução greedy. □

NB: A estratégia *greedy* apresentada não seria correta para, por exemplo, $V = \{1, 300, 1000\}$, $Q = 1200$.

Problemas de trocos com número de moedas ilimitado

Demonstração de que a estratégia greedy determina a solução ótima

- Seja $x^\star$ uma solução ótima para a quantia Q. Seja $x_v^\star$ é o número de moedas que usa de valor v.
- Se $x_{100}^\star > 1$, a solução não seria ótima (podia reduzir o número de moedas se substituir duas de 100 por uma de 200). Portanto, $x_{100}^\star \leq 1$.
  Analogamente se conclui que: $x_{50}^\star \leq 1$, $x_{10}^\star \leq 1$, e $x_1^\star \leq 1$.
- Se $x_{20}^\star > 2$ então a solução não seria ótima porque podia trocar três moedas de 20 por uma de 50 e uma de 10. Portanto, $x_{20}^\star \leq 2$. Analogamente, $x_2^\star \leq 2$.
- Não pode ter simultaneamente $x_2^\star = 2$ e $x_1^\star = 1$, pois a solução não seria ótima (podia substituir essas três moedas por uma de 5). Portanto $2x_2^\star + x_1^\star \leq 4$.
  Também não tem simultaneamente $x_{20}^\star = 2$ e $x_{10}^\star = 1$.
- Como $2x_2^\star + x_1^\star \leq 4$, $x_5^\star \leq 1$ e $x_{10}^\star \leq 1$ então $5x_5^\star + 2x_2^\star + x_1^\star \leq 9$ e $10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 19$. Analogamente, se deduz que $20x_{20}^\star + 10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 49$, $50x_{50}^\star + 20x_{20}^\star + 10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 99$. $100x_{100}^\star + 50x_{50}^\star + 20x_{20}^\star + 10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 199$.
- Tem-se $\sum_{i=1}^{k} v_i x_{v_i}^\star < v_{k+1}$, para todo k. Portanto, $x^\star$ tem exatamente o mesmo valor que a solução greedy. □

NB: A estratégia *greedy* apresentada não seria correta para, por exemplo, $V = \{1, 300, 1000\}$, $Q = 1200$.

Problemas de trocos com número de moedas ilimitado

Demonstração de que a estratégia greedy determina a solução ótima

- Seja $x^\star$ uma solução ótima para a quantia Q. Seja $x_v^\star$ é o número de moedas que usa de valor v.
- Se $x_{100}^\star > 1$, a solução não seria ótima (podia reduzir o número de moedas se substituir duas de 100 por uma de 200). Portanto, $x_{100}^\star \leq 1$.
  Analogamente se conclui que: $x_{50}^\star \leq 1$, $x_{10}^\star \leq 1$, e $x_1^\star \leq 1$.
- Se $x_{20}^\star > 2$ então a solução não seria ótima porque podia trocar três moedas de 20 por uma de 50 e uma de 10. Portanto, $x_{20}^\star \leq 2$. Analogamente, $x_2^\star \leq 2$.
- Não pode ter simultaneamente $x_2^\star = 2$ e $x_1^\star = 1$, pois a solução não seria ótima (podia substituir essas três moedas por uma de 5). Portanto $2x_2^\star + x_1^\star \leq 4$.
  Também não tem simultaneamente $x_{20}^\star = 2$ e $x_{10}^\star = 1$.
- Como $2x_2^\star + x_1^\star \leq 4$, $x_5^\star \leq 1$ e $x_{10}^\star \leq 1$ então $5x_5^\star + 2x_2^\star + x_1^\star \leq 9$ e $10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 19$. Analogamente, se deduz que $20x_{20}^\star + 10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 49$, $50x_{50}^\star + 20x_{20}^\star + 10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 99$. $100x_{100}^\star + 50x_{50}^\star + 20x_{20}^\star + 10x_{10}^\star + 5x_5^\star + 2x_2^\star + x_1^\star \leq 199$.
- Tem-se $\sum_{i=1}^{k} v_i x_{v_i}^\star < v_{k+1}$, para todo k. Portanto, $x^\star$ tem exatamente o mesmo valor que a solução greedy. $\square$

NB: A estratégia *greedy* apresentada não seria correta para, por exemplo, $V = \{1, 300, 1000\}$, $Q = 1200$.

Problemas de trocos com número de moedas ilimitado

Demonstração de que a estratégia greedy determina a solução ótima

- Seja $x^\star$ uma solução ótima para a quantia Q. Seja $x^\star_v$ é o número de moedas que usa de valor v.
- Se $x^\star_{100} > 1$, a solução não seria ótima (podia reduzir o número de moedas se substituir duas de 100 por uma de 200). Portanto, $x^\star_{100} \leq 1$.
  Analogamente se conclui que: $x^\star_{50} \leq 1$, $x^\star_{10} \leq 1$, e $x^\star_1 \leq 1$.
- Se $x^\star_{20} > 2$ então a solução não seria ótima porque podia trocar três moedas de 20 por uma de 50 e uma de 10. Portanto, $x^\star_{20} \leq 2$. Analogamente, $x^\star_2 \leq 2$.
- Não pode ter simultaneamente $x^\star_2 = 2$ e $x^\star_1 = 1$, pois a solução não seria ótima (podia substituir essas três moedas por uma de 5). Portanto $2x^\star_2 + x^\star_1 \leq 4$.
  Também não tem simultaneamente $x^\star_{20} = 2$ e $x^\star_{10} = 1$.
- Como $2x^\star_2 + x^\star_1 \leq 4$, $x^\star_5 \leq 1$ e $x^\star_{10} \leq 1$ então $5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 9$ e
  $10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 19$. Analogamente, se deduz que
  $20x^\star_{20} + 10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 49$, $50x^\star_{50} + 20x^\star_{20} + 10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 99$.
  $100x^\star_{100} + 50x^\star_{50} + 20x^\star_{20} + 10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 199$.
- Tem-se $\sum_{i=1}^{k} v_i x^\star_{v_i} < v_{k+1}$, para todo k. Portanto, $x^\star$ tem exatamente o mesmo valor que a solução greedy. □

NB: A estratégia *greedy* apresentada não seria correta para, por exemplo, $V = \{1, 300, 1000\}$, $Q = 1200$.

Problemas de trocos com número de moedas ilimitado

Demonstração de que a estratégia greedy determina a solução ótima

- Seja $x^\star$ uma solução ótima para a quantia Q. Seja $x^\star_v$ é o número de moedas que usa de valor v.
- Se $x^\star_{100} > 1$, a solução não seria ótima (podia reduzir o número de moedas se substituir duas de 100 por uma de 200). Portanto, $x^\star_{100} \leq 1$.
  Analogamente se conclui que: $x^\star_{50} \leq 1$, $x^\star_{10} \leq 1$, e $x^\star_1 \leq 1$.
- Se $x^\star_{20} > 2$ então a solução não seria ótima porque podia trocar três moedas de 20 por uma de 50 e uma de 10. Portanto, $x^\star_{20} \leq 2$. Analogamente, $x^\star_2 \leq 2$.
- Não pode ter simultaneamente $x^\star_2 = 2$ e $x^\star_1 = 1$, pois a solução não seria ótima (podia substituir essas três moedas por uma de 5). Portanto $2x^\star_2 + x^\star_1 \leq 4$.
  Também não tem simultaneamente $x^\star_{20} = 2$ e $x^\star_{10} = 1$.
- Como $2x^\star_2 + x^\star_1 \leq 4$, $x^\star_5 \leq 1$ e $x^\star_{10} \leq 1$ então $5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 9$ e
  $10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 19$. Analogamente, se deduz que
  $20x^\star_{20} + 10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 49$, $50x^\star_{50} + 20x^\star_{20} + 10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 99$.
  $100x^\star_{100} + 50x^\star_{50} + 20x^\star_{20} + 10x^\star_{10} + 5x^\star_5 + 2x^\star_2 + x^\star_1 \leq 199$.
- Tem-se $\sum_{i=1}^{k} v_i x^\star_{v_i} < v_{k+1}$, para todo k. Portanto, $x^\star$ tem exatamente o mesmo valor que a solução greedy. $\square$

NB: A estratégia *greedy* apresentada não seria correta para, por exemplo, $V = \{1, 300, 1000\}$, $Q = 1200$.

Problema da mochila (*knapsack problem*)

a) *Knapsack binário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \{0,1\} \end{cases}$$

b) *Knapsack inteiro*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \mathbb{Z}_0^+ \end{cases}$$

c) *Knapsack fracionário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \leq u_i \wedge x_i \in \mathbb{R}_0^+ \end{cases}$$

Exemplo

p	peso (Kg)	42	32	12	20	27
v	valor (u.m.)	90	82	37	61	70
u	máximo (Kg)	35	60	30	25	20

(a) Para um limite de carga $L = 80Kg$, dados p_i e v_i para cada cada objeto i, que objetos transporta para maximizar o valor total? ($x_2 = 1$, $x_4 = x_5 = 1$) (b) E, se puder transportar vários idênticos? ($x_3 = 5$, $x_4 = 1$) (c) E, se puder fracioná-los, sendo o valor e o peso proporcionais à fracção que leva, não podendo exceder um limite máximo dado para cada tipo? ($x_3 = 30$, $x_4 = 25$, $x_5 = 20$, $x_2 = 5$)

Problema da mochila (*knapsack problem*)

a) *Knapsack binário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \{0,1\} \end{cases}$$

b) *Knapsack inteiro*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \mathbb{Z}_0^+ \end{cases}$$

c) *Knapsack fracionário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \leq u_i \wedge x_i \in \mathbb{R}_0^+ \end{cases}$$

Exemplo

p	peso (Kg)	42	32	12	20	27
v	valor (u.m.)	90	82	37	61	70
u	máximo (Kg)	35	60	30	25	20

(a) Para um limite de carga $L = 80Kg$, dados p_i e v_i para cada cada objeto i, que objetos transporta para maximizar o valor total? ($x_2 = 1$, $x_4 = x_5 = 1$) (b) E, se puder transportar vários idênticos? ($x_3 = 5$, $x_4 = 1$) (c) E, se puder fracioná-los, sendo o valor e o peso proporcionais à fracção que leva, não podendo exceder um limite máximo dado para cada tipo? ($x_3 = 30$, $x_4 = 25$, $x_5 = 20$, $x_2 = 5$)

Problema da mochila (*knapsack problem*)

a) *Knapsack binário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \{0,1\} \end{cases}$$

b) *Knapsack inteiro*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \mathbb{Z}_0^+ \end{cases}$$

c) *Knapsack fracionário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \leq u_i \wedge x_i \in \mathbb{R}_0^+ \end{cases}$$

Exemplo

p	peso (Kg)	42	32	12	20	27
v	valor (u.m.)	90	82	37	61	70
u	máximo (Kg)	35	60	30	25	20

(a) Para um limite de carga $L = 80Kg$, dados p_i e v_i para cada cada objeto i, que objetos transporta para maximizar o valor total? ($x_2 = 1$, $x_4 = x_5 = 1$) (b) E, se puder transportar vários idênticos? ($x_3 = 5$, $x_4 = 1$) (c) E, se puder fracioná-los, sendo o valor e o peso proporcionais à fracção que leva, não podendo exceder um limite máximo dado para cada tipo? ($x_3 = 30$, $x_4 = 25$, $x_5 = 20$, $x_2 = 5$)

Problema da mochila (*knapsack problem*)

a) *Knapsack binário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \{0,1\} \end{cases}$$

b) *Knapsack inteiro*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \mathbb{Z}_0^+ \end{cases}$$

c) *Knapsack fracionário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \leq u_i \wedge x_i \in \mathbb{R}_0^+ \end{cases}$$

Exemplo

p	peso (Kg)	42	32	12	20	27
v	valor (u.m.)	90	82	37	61	70
u	máximo (Kg)	35	60	30	25	20

(a) Para um limite de carga $L = 80Kg$, dados p_i e v_i para cada cada objeto i, que objetos transporta para maximizar o valor total? ($x_2 = 1$, $x_4 = x_5 = 1$) (b) E, se puder transportar vários idênticos? ($x_3 = 5$, $x_4 = 1$) (c) E, se puder fracioná-los, sendo o valor e o peso proporcionais à fracção que leva, não podendo exceder um limite máximo dado para cada tipo? ($x_3 = 30$, $x_4 = 25$, $x_5 = 20$, $x_2 = 5$)

Problema da mochila (*knapsack problem*)

a) *Knapsack binário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \{0,1\} \end{cases}$$

b) *Knapsack inteiro*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \mathbb{Z}_0^+ \end{cases}$$

c) *Knapsack fracionário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \leq u_i \wedge x_i \in \mathbb{R}_0^+ \end{cases}$$

Exemplo

p	peso (Kg)	42	32	12	20	27
v	valor (u.m.)	90	82	37	61	70
u	máximo (Kg)	35	60	30	25	20

(a) Para um limite de carga $L = 80Kg$, dados p_i e v_i para cada cada objeto i, que objetos transporta para maximizar o valor total? ($x_2 = 1$, $x_4 = x_5 = 1$) (b) E, se puder transportar vários idênticos? ($x_3 = 5$, $x_4 = 1$) (c) E, se puder fracioná-los, sendo o valor e o peso proporcionais à fracção que leva, não podendo exceder um limite máximo dado para cada tipo? ($x_3 = 30$, $x_4 = 25$, $x_5 = 20$, $x_2 = 5$)

Problema da mochila (*knapsack problem*)

a) *Knapsack binário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \{0,1\} \end{cases}$$

b) *Knapsack inteiro*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \in \mathbb{Z}_0^+ \end{cases}$$

c) *Knapsack fracionário*

maximizar $\sum_{i=1}^{n} v_i x_i$
sujeito a
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \quad x_i \leq u_i \land x_i \in \mathbb{R}_0^+ \end{cases}$$

Exemplo

p	peso (Kg)	42	32	12	20	27
v	valor (u.m.)	90	82	37	61	70
u	máximo (Kg)	35	60	30	25	20

(a) Para um limite de carga $L = 80Kg$, dados p_i e v_i para cada cada objeto i, que objetos transporta para maximizar o valor total? ($x_2 = 1$, $x_4 = x_5 = 1$) (b) E, se puder transportar vários idênticos? ($x_3 = 5$, $x_4 = 1$) (c) E, se puder fracioná-los, sendo o valor e o peso proporcionais à fracção que leva, não podendo exceder um limite máximo dado para cada tipo? ($x_3 = 30$, $x_4 = 25$, $x_5 = 20$, $x_2 = 5$)

Problema da mochila fracionário (*linear knapsack problem*)

$$\text{maximizar} \sum_{i=1}^{n} v_i x_i$$
$$\text{sujeito a}$$
$$\begin{cases} \sum_{i=1}^{n} p_i x_i \leq L \\ \forall i \;\; x_i \leq u_i \wedge x_i \in \mathbb{R}_0^+ \end{cases}$$

Algoritmo *greedy* que calcula uma solução ótima para *knapsack fracionário:*

> *Assumindo que os itens estão ordenados por ordem decrescente de* **valor por unidade de recurso despendida** *(ou seja, por v_i/p_i), levar a maior quantidade possível do primeiro item (isto é, $x_1 = \min(u_1, L/p_1)$) e aplicar a mesma estratégia para $i \geq 2$, com peso máximo $L - x_1$.*

Exemplo ($L = 80$, solução ótima: $x_3 = 30$, $x_4 = 25$, $x_5 = 20$, $x_2 = 5$)

p	peso (Kg)	42	32	12	20	27
v	valor (u.m.)	90	82	37	61	70
u	máximo (Kg)	35	60	30	25	20
v/p	rendimento (u.m/Kg)	2.14	2.56	3.08	3.05	2.59

Matróides pesados e Algoritmos Greedy

Seja S um conjunto **finito** e $\mathcal{F}$ uma família de subconjuntos de S tal que $\mathcal{F} \neq \emptyset$.
O par $(S, \mathcal{F})$ designa-se por **matróide** sse satisfizer para todo A e B:

- (*Hereditariedade*) se $B \in \mathcal{F}$ e $A \subseteq B$ então $A \in \mathcal{F}$.
- (*Extensão*) Se $A, B \in \mathcal{F}$ e $|A| < |B|$ então $A \cup \{x\} \in \mathcal{F}$, para algum $x \in B$.

Os elementos de $\mathcal{F}$ designam-se por *subconjuntos independentes*.

Propriedade: Os conjuntos independentes **maximais** (para $\subseteq$) têm o mesmo cardinal.

Um matróide pesado é um matróide $(S, \mathcal{F})$ com uma função de peso $w : S \to \mathbb{R}^+$, sendo $w(A) = \sum_{a \in A} w(a)$, para todo $A \subseteq S$.

Exemplos de matróides:

- $S = \{$colunas da matriz de coeficientes de um sistema $AX = b\}$,
  $\mathcal{F} = \{$subconjuntos de colunas de A linearmente independentes$\}$.
- Para $G = (V, E)$ grafo finito não dirigido, $S = E$ e $\mathcal{F} = \{$florestas de $E\}$

O problema da **determinação de $A \in \mathcal{F}$ com peso $w(A)$ máximo** pode ser resolvido pelo **"algoritmo greedy trivial"**: partir de $A = \emptyset$ e, tomando os elementos $x \in S$ por ordem decrescente de peso, inserir $x \in A$ se $A \cup \{x\} \in \mathcal{F}$.

Exemplos de Aplicação - Otimização em matróide pesados

- **Exemplo 1:** Determinar árvore geradora de peso máximo/mínimo – Algoritmo de Kruskal

- **Exemplo 2:** Localizar observadores em rotundas para determinar os volumes de tráfego q_{ij}, da entrada i para a saída j, para todos os pares (i, j)

  São dados os volumes totais O_i e D_j e ainda o que passa frontalmente a uma entrada F_1. Assume-se que os veículos não estacionam no anel de circulação. Se colocar observador para q_{ij} tem um custo c_{ij}. Minimizar o custo total.

- **Exemplo 3:** Dado um conjunto finito de **tarefas unitárias cada uma com um prazo limite** (*deadline*) d_j e uma **penalização** c_j se ultrapassar esse prazo, determinar a ordem pela qual as tarefas serão realizadas de forma a minimizar o custo (penalização) total.

Exemplos de Aplicação - Otimização em matróide pesados

- **Exemplo 1:** Determinar árvore geradora de peso máximo/mínimo – Algoritmo de Kruskal

- **Exemplo 2:** Localizar observadores em rotundas para determinar os volumes de tráfego q_{ij}, da entrada i para a saída j, para todos os pares (i, j)

  São dados os volumes totais O_i e D_j e ainda o que passa frontalmente a uma entrada F_1. Assume-se que os veículos não estacionam no anel de circulação. Se colocar observador para q_{ij} tem um custo c_{ij}. Minimizar o custo total.

- **Exemplo 3:** Dado um conjunto finito de **tarefas unitárias cada uma com um prazo limite** (*deadline*) d_j e uma **penalização** c_j se ultrapassar esse prazo, determinar a ordem pela qual as tarefas serão realizadas de forma a minimizar o custo (penalização) total.

Exemplos de Aplicação - Otimização em matróide pesados

- **Exemplo 1:** Determinar árvore geradora de peso máximo/mínimo – Algoritmo de Kruskal

- **Exemplo 2:** Localizar observadores em rotundas para determinar os volumes de tráfego q_{ij}, da entrada i para a saída j, para todos os pares (i, j)

  São dados os volumes totais O_i e D_j e ainda o que passa frontalmente a uma entrada F_1. Assume-se que os veículos não estacionam no anel de circulação. Se colocar observador para q_{ij} tem um custo c_{ij}. Minimizar o custo total.

- **Exemplo 3:** Dado um conjunto finito de **tarefas unitárias cada uma com um prazo limite** (*deadline*) d_j e uma **penalização** c_j se ultrapassar esse prazo, determinar a ordem pela qual as tarefas serão realizadas de forma a minimizar o custo (penalização) total.

Para recordar...

$$\begin{cases} x_1 - 4x_2 + 5x_3 - 10x_4 & = & 2 \\ -x_1 + 7x_2 + 2x_3 - 4x_4 & = & 1 \\ \quad 3x_2 + 7x_3 - 14x_4 & = & 3 \\ x_1, x_2, x_3, x_4 \in \mathbb{R} \end{cases} \qquad A = \begin{bmatrix} 1 & -4 & 5 & -10 \\ -1 & 7 & 2 & -4 \\ 0 & 3 & 7 & -14 \end{bmatrix}$$

- A matriz A tem característica 2 porque a terceira equação é redundante. $car(A)$ é igual também à dimensão do espaço pelas colunas de A.
- Dos $\binom{n}{2} = \binom{4}{2} = 6$ subconjuntos $\{A_i, A_j\}$, com $i \neq j$, cinco são **bases** do espaço gerado pelas colunas de A. Cada base define uma *"forma resolvida"*.

$$\begin{cases} x_1 & = & 6 - 43/3x_3 + 86/3x_4 \\ x_2 & = & 1 - 7/3x_3 + 14/3x_4 \end{cases} \qquad \begin{cases} x_1 & = & -1/7 + 43/7x_2 \\ x_3 & = & 3/7 - 3/7x_2 + 2x_4 \end{cases}$$

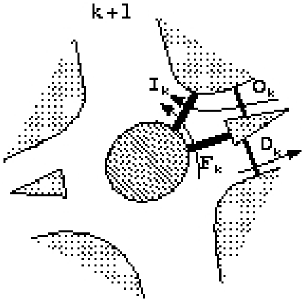
$$\begin{cases} x_1 & = & -1/7 + 43/7x_2 \\ x_4 & = & -3/14 + 3/14x_2 + 1/2x_3 \end{cases} \qquad \begin{cases} x_2 & = & 1/43 + 7/43x_1 \\ x_3 & = & 18/43 - 3/43x_1 + 2x_4 \end{cases}$$

$$\begin{cases} x_2 & = & 1/43 + 7/43x_1 \\ x_4 & = & -9/43 + 3/86x_1 + 1/2x_3 \end{cases} \qquad \begin{cases} x_3 & = & \ldots.A_3 \text{ e } A_4 \text{ linearmente} \\ x_4 & = & \ldots..\text{dependentes} \end{cases}$$

- Sistema **indeterminado**. Se se atribuir valores às $n - car(A)$ **variáveis livres**, admite uma única solução para as restantes $car(A)$ variáveis.

Exemplo 2: Contagem de tráfego em rotundas

Obter os volumes direcionais q_{ij}, para todos os (i, j), com custo total mínimo.



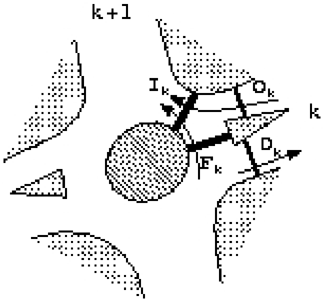
$$
\begin{aligned}
\sum_{j \in \mathcal{D}} q_{ij} &= O_i, \text{ para } i \in \mathcal{O} \\
\sum_{i \in \mathcal{O}} q_{ij} &= D_j, \text{ para } j \in \mathcal{D} \\
\sum_{i \in \mathcal{O} \setminus \{k\}} \; \sum_{j \in \mathcal{D},\, k \prec j \preceq i} q_{ij} &= F_k, \text{ para } 1 \le k \le n \\
I_k &= F_k + O_k, \text{ para } k \in \mathcal{O} \\
I_k &= F_k, \text{ para } k \notin \mathcal{O}
\end{aligned}
$$

O_i: total na entrada i; D_j: total na saída j;
F_k: total na secção frontal a k; I_k: total na secção intermédia k.

O número de variáveis q_{ij} é $|\mathcal{O}| \times |\mathcal{D}|$ mas a **caraterística da matriz** do sistema é $|\mathcal{O}| + |\mathcal{D}|$ ou $|\mathcal{O}| + |\mathcal{D}| - 1$. **Quais dos q_{ij} não serão obtidos por observação, sendo calculados por resolução do sistema?**

Exemplo 2: Contagem de tráfego em rotundas

Obter os volumes direcionais q_{ij}, para todos os (i, j), com custo total mínimo.



- Qualquer uma das equações para os O_i's e D_j's é redundante face às restantes. Também, apenas um dos F_k's poderá ser não redundante face aos O_i's e D_j's.
- A caraterística da matriz do sistema é $|\mathcal{O}| + |\mathcal{D}|$ ou $|\mathcal{O}| + |\mathcal{D}| - 1$.
- É $|\mathcal{O}| + |\mathcal{D}| - 1$ sse a equação que define F_k é $F_k = 0$, para algum k. Isto acontece se a rotunda for do tipo $S^\star(D + SE)E^\star$, onde S designa saída, E entrada e D sentido duplo.

A.P Tomás, M. Andrade and A. Pires da Costa (2001) Obtaining Origin-Destination Data at Optimal Cost at Urban Roundabouts. In CSOR - EPIA'01.
A.P. Tomás (2002). Solving Optimal Location of Traffic Counting Points at Urban Intersections in CLP(FD). In Proc. MICAI'2002. LNAI 2313, 242-251. `http://www.dcc.fc.up.pt/~apt/onlinepapers/micai02.pdf`

Exemplo 2: Contagem de tráfego em rotundas

Sendo r a caraterística da matriz do sistema, queremos escolher os r volumes direcionais **independentes** que **não serão** obtidos por contagem mas deduzidos dos restantes q_{ij}'s e dos volumes totais em seccção (O_i's, D_j's e F_k's), de forma a minimizar o **custo da recolha**.

Exemplos: $custo(q_{ij}) =$ número de ramos de i para j

		volumes a observar
SEDE	r=3+2=5 vars = $3 \times 2 = 6$	q(4,1)
DDDE	r=4+3=7 nvars =$4 \times 3 = 12$	q(1,2) q(2,3) q(3,1) q(4,1) q(4,2)
DDSE	r= 3+3 = 6 nvars =$3 \times 3 = 9$	q(1,2) q(2,3) q(4,1)
SSSD	r= 1+4-1 = 4 nvars =$1 \times 4 = 4$	

SSSD é do tipo $S^\star(D+SE)E^\star$

Exemplo 2: Contagem de tráfego em rotundas

Se custo total for dado pela soma dos custos da contagem de cada um dos q_{ij}'s, o problema corresponde à determinação da solução de peso máximo num matróide pesado. A solução ótima pode ser determinada pelo "algoritmo greedy trivial".

A rotunda SDSDEE com c_{ij} = número de ramos de i para j

p'_{21}	$\boxed{p'_{22}}$	p'_{23}	p'_{24}
p'_{41}	p'_{42}	p'_{43}	$\boxed{p'_{44}}$
p'_{51}	p'_{52}	p'_{53}	$\boxed{p'_{54}}$
p'_{61}	p'_{62}	p'_{63}	$\boxed{p'_{64}}$

$\boxed{p'_{21}}$	$\boxed{p'_{22}}$	p'_{23}	p'_{24}
p'_{41}	$\boxed{p'_{42}}$	$\boxed{p'_{43}}$	$\boxed{p'_{44}}$
p'_{51}	p'_{52}	p'_{53}	$\boxed{p'_{54}}$
p'_{61}	p'_{62}	p'_{63}	$\boxed{p'_{64}}$

$\boxed{p'_{21}}$	$\boxed{p'_{22}}$	p'_{23}	$\boxed{p'_{24}}$
p'_{41}	$\boxed{p'_{42}}$	$\boxed{p'_{43}}$	$\boxed{p'_{44}}$
p'_{51}	p'_{52}	p'_{53}	$\boxed{p'_{54}}$
p'_{61}	p'_{62}	p'_{63}	$\boxed{p'_{64}}$

Ordem decrescente de custos: $c_{22} = c_{44} = 6$, $c_{54} = c_{21} = c_{43} = 5$, $c_{64} = c_{53} = c_{42} = 4$, $c_{41} = c_{52} = c_{63} = 3$, $c_{24} = c_{51} = c_{62} = 2$, $c_{23} = c_{61} = 1$

- $p'_{ij} = \mathrm{e}_i + \mathrm{e}_{m+j} + \theta_{ij}\mathrm{e}_{m+n+1}$: coluna de q_{ij} no sistema formado pelas equações nas entradas $\mathcal{O} = \{2, 4, 5, 6\}$, saídas $\mathcal{D} = \{1, 2, 3, 4\}$ e secção F_1. Os e_t definem a base canónica de $\mathbb{R}^{n+m+1}$, θ_{ij} é 1 ou 0 (indica se q_{ij} passa em F_1) e $m = |\mathcal{O}|$, $n = |\mathcal{D}|$.
- Para *SDSDEE*, a dimensão da base é $|\mathcal{O}| + |\mathcal{D}| = 8$. $\boxed{p'_{ij}}$ denota p'_{ij} escolhido.

Exemplo 3: Unit task scheduling

Problema:
Dado um conjunto $\mathcal{T}$ de tarefas com duração 1, cada uma com um **prazo** d_j (*deadline*) e uma **penalização** c_j, se ultrapassar esse prazo, por que ordem as realizar de forma a minimizar a penalização total?

- Um conjunto A de tarefas é **independente** se todas as tarefas em A podem ser executadas até ao seu *deadline*. Prova-se que tal acontece se, para todo $k \geq 0$, o número de tarefas em A com *deadline* até k é menor ou igual a k.
- O "Algoritmo greedy trivial" para obter uma solução ótima: ordenar as tarefas por ordem decrescente de penalização (supor que $t_1, t_2, \ldots, t_n$ traduz essa ordem). No início, $S = \emptyset$. Para j de 1 até n, colocar t_j em S desde que $S \cup \{t_j\}$ seja *independente*.
- As tarefas em S podem ser realizadas sem penalização (por exemplo, se as realizar por ordem crescente de *deadline*). As tarefas em $\mathcal{T} \setminus S$ são realizadas por qualquer ordem (têm sempre penalização).

Exemplo 3: Unit task scheduling

Problema:
Dado um conjunto $\mathcal{T}$ de tarefas com duração 1, cada uma com um **prazo** d_j (*deadline*) e uma **penalização** c_j, se ultrapassar esse prazo, por que ordem as realizar de forma a minimizar a penalização total?

- Um conjunto A de tarefas é **independente** se todas as tarefas em A podem ser executadas até ao seu *deadline*. Prova-se que tal acontece se, para todo $k \geq 0$, o número de tarefas em A com *deadline* até k é menor ou igual a k.
- O "Algoritmo greedy trivial" para obter uma solução ótima: ordenar as tarefas por ordem decrescente de penalização (supor que $t_1, t_2, \ldots, t_n$ traduz essa ordem). No início, $S = \emptyset$. Para j de 1 até n, colocar t_j em S desde que $S \cup \{t_j\}$ seja *independente*.
- As tarefas em S podem ser realizadas sem penalização (por exemplo, se as realizar por ordem crescente de *deadline*). As tarefas em $\mathcal{T} \setminus S$ são realizadas por qualquer ordem (têm sempre penalização).

Exemplo 3: Unit task scheduling

Problema:
Dado um conjunto $\mathcal{T}$ de tarefas com duração 1, cada uma com um **prazo** d_j (*deadline*) e uma **penalização** c_j, se ultrapassar esse prazo, por que ordem as realizar de forma a minimizar a penalização total?

- Um conjunto A de tarefas é **independente** se todas as tarefas em A podem ser executadas até ao seu *deadline*. Prova-se que tal acontece se, para todo $k \geq 0$, o número de tarefas em A com *deadline* até k é menor ou igual a k.
- O "Algoritmo greedy trivial" para obter uma solução ótima: ordenar as tarefas por ordem decrescente de penalização (supor que $t_1, t_2, \ldots, t_n$ traduz essa ordem). No início, $S = \emptyset$. Para j de 1 até n, colocar t_j em S desde que $S \cup \{t_j\}$ seja *independente*.
- As tarefas em S podem ser realizadas sem penalização (por exemplo, se as realizar por ordem crescente de *deadline*). As tarefas em $\mathcal{T} \setminus S$ são realizadas por qualquer ordem (têm sempre penalização).

Exemplo 3: Unit task scheduling

Problema:
Dado um conjunto $\mathcal{T}$ de tarefas com duração 1, cada uma com um **prazo** d_j (*deadline*) e uma **penalização** c_j, se ultrapassar esse prazo, por que ordem as realizar de forma a minimizar a penalização total?

- Um conjunto A de tarefas é **independente** se todas as tarefas em A podem ser executadas até ao seu *deadline*. Prova-se que tal acontece se, para todo $k \geq 0$, o número de tarefas em A com *deadline* até k é menor ou igual a k.
- O "Algoritmo greedy trivial" para obter uma solução ótima: ordenar as tarefas por ordem decrescente de penalização (supor que $t_1, t_2, \ldots, t_n$ traduz essa ordem). No início, $S = \emptyset$. Para j de 1 até n, colocar t_j em S desde que $S \cup \{t_j\}$ seja *independente*.
- As tarefas em S podem ser realizadas sem penalização (por exemplo, se as realizar por ordem crescente de *deadline*). As tarefas em $\mathcal{T} \setminus S$ são realizadas por qualquer ordem (têm sempre penalização).

Exemplo (não matróide pesado): Interval scheduling

Problema: $\mathcal{T}$ é um conjunto de n tarefas. A tarefa t_j teria forçosamente de decorrer no intervalo $[a_j, b_j[$, ou seja, começar no instante a_j e terminar em b_j, para $1 \leq j \leq n$ (notar que $b_j \notin [a_j, b_j[$). Em cada instante, só uma tarefa pode estar a decorrer. Pretende-se **maximizar o número de tarefas realizadas**.

- Não se modela por um matróide pesado.
- A solução ótima pode ser obtida em tempo $O(n \log n)$ por um algoritmo *greedy*, o qual usa a estratégia "earliest finish first".

```
Ordenar T por ordem crescente de tempo de finalização;
 /* Supor que t1, t2, ..., tn traduz essa ordem */
S ← ∅;  f ← 0;
Para j ← 1 até n fazer
    Se a[j] ≥ f então
        S ← S ∪ {tj};  f ← b[j];
```

- **Correção:** Seja $S^\star$ uma solução ótima distinta de S. Sejam k e j as primeiras duas tarefas que as distinguem. Então, t_j (a escolha greedy) pode substituir t_k em $S^\star$, ou seja, $(S^\star \setminus \{t_k\}) \cup \{t_j\}$ é também uma solução ótima (t_j não pode criar conflitos pois $b[j] \leq b[k]$). Assim, repetindo, acabamos por conseguir transformar qualquer solução ótima $S^\star$ na solução greedy S, pelo que S é ótima.

Exemplo (não matróide pesado): Interval scheduling

Problema: $\mathcal{T}$ é um conjunto de n tarefas. A tarefa t_j teria forçosamente de decorrer no intervalo $[a_j, b_j[$, ou seja, começar no instante a_j e terminar em b_j, para $1 \leq j \leq n$ (notar que $b_j \notin [a_j, b_j[$). Em cada instante, só uma tarefa pode estar a decorrer. Pretende-se **maximizar o número de tarefas realizadas**.

- Não se modela por um matróide pesado.
- A solução ótima pode ser obtida em tempo $O(n \log n)$ por um algoritmo *greedy*, o qual usa a estratégia "earliest finish first".

```
Ordenar 𝒯 por ordem crescente de tempo de finalização;
 /* Supor que t1, t2, ..., tn traduz essa ordem */
S ← ∅;  f ← 0;
Para j ← 1 até n fazer
    Se a[j] ≥ f então
        S ← S ∪ {tj};  f ← b[j];
```

- **Correção:** Seja $S^\star$ uma solução ótima distinta de S. Sejam k e j as primeiras duas tarefas que as distinguem. Então, t_j (a escolha greedy) pode substituir t_k em $S^\star$, ou seja, $(S^\star \setminus \{t_k\}) \cup \{t_j\}$ é também uma solução ótima (t_j não pode criar conflitos pois $b[j] \leq b[k]$). Assim, repetindo, acabamos por conseguir transformar qualquer solução ótima $S^\star$ na solução greedy S, pelo que S é ótima.

Exemplo (não matróide pesado): Interval scheduling

Problema: $\mathcal{T}$ é um conjunto de n tarefas. A tarefa t_j teria forçosamente de decorrer no intervalo $[a_j, b_j[$, ou seja, começar no instante a_j e terminar em b_j, para $1 \leq j \leq n$ (notar que $b_j \notin [a_j, b_j[$). Em cada instante, só uma tarefa pode estar a decorrer. Pretende-se **maximizar o número de tarefas realizadas**.

- Não se modela por um matróide pesado.
- A solução ótima pode ser obtida em tempo $O(n \log n)$ por um algoritmo *greedy*, o qual usa a estratégia "earliest finish first".

```
Ordenar T por ordem crescente de tempo de finalização;
 /* Supor que t1, t2, ..., tn traduz essa ordem */
S ← ∅;  f ← 0;
Para j ← 1 até n fazer
     Se a[j] ≥ f então
          S ← S ∪ {tj};  f ← b[j];
```

- **Correção:** Seja $S^\star$ uma solução ótima distinta de S. Sejam k e j as primeiras duas tarefas que as distinguem. Então, t_j (a escolha greedy) pode substituir t_k em $S^\star$, ou seja, $(S^\star \setminus \{t_k\}) \cup \{t_j\}$ é também uma solução ótima (t_j não pode criar conflitos pois $b[j] \leq b[k]$). Assim, repetindo, acabamos por conseguir transformar qualquer solução ótima $S^\star$ na solução greedy S, pelo que S é ótima.