

## MC458 — Projeto e Análise de Algoritmos I

C.C. de Souza C.N. da Silva O. Lee P.J. de Rezende

1º Semestre de 2018

### Algoritmos gulosos

## Algoritmos Gulosos: Conceitos Básicos

- Tipicamente algoritmos gulosos são utilizados para resolver problemas de **otimização**.
- Uma característica comum dos problemas onde se aplicam algoritmos gulosos é a existência **subestrutura ótima**, **semelhante a programação dinâmica!**
- **Algoritmo por programação dinâmica**: tipicamente os subproblemas são resolvidos à otimalidade **antes** de se proceder à **escolha** de um elemento que irá compor uma solução ótima.
- **Algoritmos gulosos**: primeiramente é feita a escolha de um elemento que irá compor uma solução ótima e só **depois** um subproblema é resolvido.

## Algoritmos Gulosos: Conceitos Básicos

- Um algoritmo guloso sempre faz uma **escolha** identificada como a “melhor” a cada iteração segundo um **critério guloso** local. **Essa é uma decisão localmente ótima.**
- **Propriedade que a escolha gulosa deve ter: 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 *backtracking*.
- Em geral, é **fácil “pensar”** em critérios gulosos. O **difícil é provar** que eles funcionam e que levam ao projeto de algoritmos gulosos!

## 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  $0 \leq s_i < f_i < \infty$ .  
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**.

## Seleção de Atividades

- **Exemplo:**

|       |   |   |   |   |   |   |    |    |    |    |    |
|-------|---|---|---|---|---|---|----|----|----|----|----|
| $i$   | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  | 9  | 10 | 11 |
| $s_i$ | 1 | 3 | 0 | 4 | 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:  $(a_1, a_4)$ ,  $(a_4, a_8)$
- Conjunto **maximal** de atividades compatíveis:  $(a_3, a_9, a_{11})$ .
- Conjunto **máximo** de atividades compatíveis:  $(a_1, a_4, a_8, a_{11})$ .

## Seleção de Atividades

- Vimos que tanto os algoritmos gulosos quanto aqueles que usam programação dinâmica valem-se da existência da **propriedade de subestrutura ótima**.
- Inicialmente verificaremos que o problema da seleção de atividades tem esta propriedade e, então, projetaremos um algoritmo por **programação dinâmica**.
- 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**.
- Este processo auxiliará no entendimento da diferença entre estas duas **técnicas de projeto de algoritmos**.

## Seleção de Atividades

Suponha que  $f_1 \leq f_2 \leq \dots \leq f_n$ , ou seja, as atividades estão **ordenadas em ordem crescente de instantes de término**.

### Definição

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

- **Atividades 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  $0 \leq i, j \leq n + 1$ .
- Note que  $S_{ij} = \emptyset$  para todo  $i \geq j$ .  
**Por quê?**

## Seleção de Atividades

- **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  $a_k$ .

**Por quê?**

## Seleção de Atividades

- **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:**

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

Agora é fácil escrever o algoritmo de programação dinâmica. (**Exercício.**)

## Seleção de Atividades

Podemos “converter” o algoritmo de programação dinâmica em um algoritmo guloso se notarmos que o primeiro resolve subproblemas desnecessariamente.

### 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.

## Seleção de Atividades

### Método geral para provar que um algoritmo guloso funciona

- Mostre que o problema tem subestrutura ótima.
- Mostre que se  $a$  foi a primeira escolha do algoritmo, então **existe** alguma **solução ótima** que contém  $a$ . Segue então por indução e pela subestrutura ótima que o algoritmo sempre faz escolhas corretas.

## Seleção de Atividades

Vou mostrar que existe uma solução ótima para  $S = S_{0,n+1}$  que contém  $a_m$ .

Seja  $A$  um conjunto de atividades mutuamente compatíveis de tamanho máximo em  $S_{ij}$ . Se  $a_m \in A$  então nada há a fazer. Suponha então que  $a_m \notin A$ .

Seja  $a_k \in A$  com menor  $f_k$ . Seja  $A' = A - \{a_k\} \cup \{a_m\}$ . Então  $A'$  também é conjunto de atividades mutuamente compatíveis de tamanho máximo. (Por quê?)

Este parece ser um truque importante: modificar uma solução ótima "genérica" e obter uma solução ótima com a(s) escolha(s) gulosa(s).

## Seleção de Atividades

Usando o teorema anterior, um modo simples de projetar um algoritmo seria o seguinte:

- Suponha que estamos tentando resolver  $S_{ij}$ .
- Determine a atividade  $a_m$  com menor tempo de término em  $S_{ij}$ .
- Resolva o subproblema  $S_{mj}$  e junte  $a_m$  à solução obtida na recursão. Devolva este conjunto de atividades.

## Seleção de Atividades

### SelecAtivGulRec( $s, f, i, j$ )

▷ **Entrada:** vetores  $s$  e  $f$  com instantes de início e término das atividades  $a_i, a_{i+1}, \dots, a_j$ , sendo  $f_i \leq \dots \leq f_j$ .

▷ **Saída:** conjunto de tamanho máximo de índices de atividades mutuamente compatíveis.

1.  $m \leftarrow i + 1$ ;  
▷ Busca atividade com menor tempo de término que está em  $S_{ij}$
2. enquanto  $m < j$  e  $s_m < f_i$  faça  $m \leftarrow m + 1$ ;
3. se  $m \geq j$  então devolva  $\emptyset$ ;
4. senão
5. se  $f_m > s_j$  então devolva  $\emptyset$ ;      ▷  $a_m \notin S_{ij}$
6. senão devolva  $\{a_m\} \cup \text{SelecAtivGulRec}(s, f, m, j)$ .

## Seleção de Atividades

- A chamada inicial será  $\text{SelecAtivGulRec}(s, f, 0, n + 1)$ .
- **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 do mesmo.

### SelecAtivGullter( $s, f, n$ )

▷ **Entrada:** vetores  $s$  e  $f$  com instantes de início e término das  $n$  atividades com os instantes de término em ordem crescente.

▷ **Saída:** um conjunto  $A$  de tamanho máximo contendo atividades mutuamente compatíveis.

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

- Observe que na linha 3,  $i$  é o índice da última atividade colocada em  $A$ . Como as atividades estão ordenadas pelo instante de término, tem-se que:

$$f_i = \max\{f_k : a_k \in A\},$$

ou seja,  $f_i$  é sempre o maior instante de término de uma atividade em  $A$ .

- Pode-se concluir que o algoritmo faz as mesmas escolhas de *SelecAtivGulRec* e portanto, está correto.
- **Complexidade:**  $\Theta(n)$ .

## Códigos de Huffman

- **Códigos de Huffman:** técnica de compressão de dados.
- Reduções no tamanho dos arquivos dependem das características dos dados contidos nos mesmos. Valores típicos oscilam entre 20 e 90%.
- **Exemplo:** arquivo texto contendo 100.000 caracteres no alfabeto  $\Sigma = \{a, b, c, d, e, f\}$ . As freqüências de cada caracter no arquivo são indicadas na tabela abaixo.

|                            | $a$ | $b$ | $c$ | $d$ | $e$  | $f$  |
|----------------------------|-----|-----|-----|-----|------|------|
| Freqüência (em milhares)   | 45  | 13  | 12  | 16  | 9    | 5    |
| Código de tamanho fixo     | 000 | 001 | 010 | 011 | 100  | 101  |
| Código de tamanho variável | 0   | 101 | 100 | 111 | 1101 | 1100 |

- **Codificação do arquivo:** representar cada caracter por uma seqüência de *bits*
- **Alternativas:**
  - 1 seqüências de **tamanho fixo**.
  - 2 seqüências de **tamanho variável**.

## Códigos de Huffman

- Qual o tamanho (em *bits*) do arquivo comprimido usando os códigos acima ?
- **Códigos de tamanho fixo:**  $3 \times 100.000 = 300.000$
- **Códigos de tamanho variável:**

$$\underbrace{(45 \times 1)}_a + \underbrace{(13 \times 3)}_b + \underbrace{(12 \times 3)}_c + \underbrace{(16 \times 3)}_d + \underbrace{(9 \times 4)}_e + \underbrace{(5 \times 4)}_f \times 1.000 = 224.000$$

Ganho de  $\approx 25\%$  em relação à solução anterior.

### Problema da Codificação:

Dadas as freqüências de ocorrência dos caracteres de um arquivo, encontrar as seqüências de *bits* (códigos) para representá-los de modo que o arquivo comprimido tenha tamanho mínimo.

### Definição:

Códigos livres de prefixo são aqueles onde, dados dois caracteres quaisquer  $i$  e  $j$  representados pela codificação, a seqüência de *bits* associada a  $i$  **não** é um *prefixo* da seqüência associada a  $j$ .

### Importante:

Pode-se provar que sempre **existe** uma solução ótima do problema da codificação que é dado por um código *livre de prefixo*.

O **processo de codificação**, i.e, de geração do arquivo comprimido é sempre fácil pois reduz-se a concatenar os códigos dos caracteres presentes no arquivo original em seqüência.

**Exemplo:** usando a codificação de tamanho variável do exemplo anterior, o arquivo original dado por `abc` seria codificado por `0101100`.

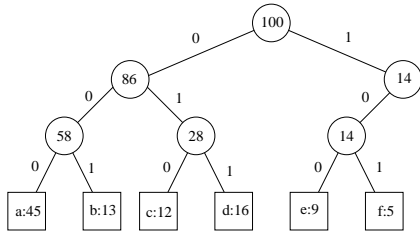
- **A vantagem dos códigos livres de prefixo se torna evidente quando vamos decodificar o arquivo comprimido.**
- Como nenhum código é prefixo de outro código, o código que se encontra no início do arquivo comprimido não apresenta ambigüidade. Pode-se simplesmente identificar este código inicial, traduzi-lo de volta ao caracter original e repetir o processo no restante do arquivo comprimido.
- **Exemplo:** usando a codificação de tamanho variável do exemplo anterior, o arquivo comprimido contendo os *bits* `001011101` divide-se de **forma unívoca** em `0 0 101 1101`, ou seja, corresponde ao arquivo original dado por `aabe`.

- Como representar de maneira conveniente uma codificação livre de prefixo de modo a facilitar o processo de decodificação?
- **Solução:** usar uma árvore binária.  
O **filho esquerdo** está associado ao *bit ZERO* enquanto o **filho direito** está associado ao *bit UM*. Nas **folhas** encontram-se os caracteres presentes no arquivo original.

## Códigos de Huffman

Vejam como ficam as árvores que representam os códigos do exemplo anterior.

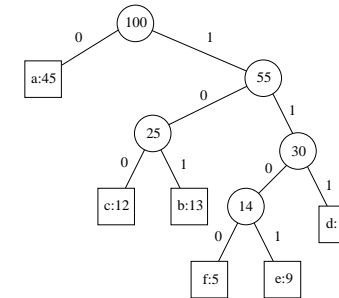
|             | <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> | <i>e</i> | <i>f</i> |
|-------------|----------|----------|----------|----------|----------|----------|
| Frequência  | 45       | 13       | 12       | 16       | 9        | 5        |
| Código fixo | 000      | 001      | 010      | 011      | 100      | 101      |



## Códigos de Huffman

Vejam como ficam as árvores que representam os códigos do exemplo anterior.

|                 | <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> | <i>e</i> | <i>f</i> |
|-----------------|----------|----------|----------|----------|----------|----------|
| Frequência      | 45       | 13       | 12       | 16       | 9        | 5        |
| Código variável | 0        | 101      | 100      | 111      | 1101     | 1100     |



## Códigos de Huffman

- Pode-se mostrar (**Exercício!**) que uma **codificação ótima** sempre pode ser representada por uma árvore binária **cheia**, na qual cada vértice interno tem exatamente **dois** filhos.
- Então podemos restringir nossa atenção às árvores binárias cheias com  $|C|$  folhas e  $|C| - 1$  vértices internos (**Exercício!**), onde  $C$  é o conjunto de caracteres do alfabeto no qual está escrito o arquivo original.

## Códigos de Huffman

Computando o tamanho do arquivo comprimido:

Se  $T$  é a árvore que representa a codificação,  $d_T(c)$  é a profundidade da folha representado o caracter  $c$  e  $f(c)$  é a sua frequência, o tamanho do arquivo comprimido será dado por:

$$B(T) = \sum_{c \in C} f(c)d_T(c).$$

Dizemos que  $B(T)$  é o **custo** da árvore  $T$ . Isto é exatamente o tamanho do arquivo codificado.

## Códigos de Huffman

- **Idéia do algoritmo de Huffman:** Começar com  $|C|$  folhas e realizar sequencialmente  $|C| - 1$  operações de “coalescência” de dois vértices da árvore. Cada uma destas coalescências dá origem a um novo vértice interno, que será o pai dos vértices que participaram da coalescência.
- A escolha do par de vértices que dará origem à coalescência em cada passo depende da soma das freqüências das folhas das subárvores com raízes nos vértices que ainda não participaram de coalescências.

## Corretude do algoritmo de Huffman

### Lema 1: (escolha gulosa)

Seja  $C$  um alfabeto onde cada caracter  $c \in C$  tem freqüência  $f[c]$ . Sejam  $x$  e  $y$  dois caracteres em  $C$  com as **menores** freqüências. Então, existe **um** código ótimo livre de prefixo para  $C$  no qual os códigos para  $x$  e  $y$  tem o mesmo comprimento e diferem apenas no último bit.

### Prova do Lema 1:

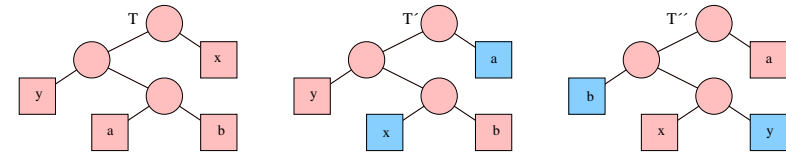
- Seja  $T$  uma árvore **ótima**.
- Sejam  $a$  e  $b$  duas folhas “irmãs” (i.e. usadas em uma coalescência) **mais profundas** de  $T$  e  $x$  e  $y$  as folhas de  $T$  de **menor freqüência**.
- **Idéia:** a partir de  $T$ , obter uma outra árvore **ótima**  $T'$  com  $x$  e  $y$  sendo duas folhas “irmãs”.

## Algoritmo de Huffman

### Huffman( $C$ )

- ▷ **Entrada:** Conjunto de caracteres  $C$  e as freqüências  $f$  dos caracteres em  $C$ .
- ▷ **Saída:** raiz de uma árvore binária representando uma codificação ótima livre de prefixos.
1.  $n \leftarrow |C|$ ;  
▷  $Q$  é fila de prioridades dada pelas freqüências dos vértices ainda não coalescidos
  2.  $Q \leftarrow C$ ;
  3. **para**  $i \leftarrow 1$  **até**  $n - 1$  **faça**
  4.     **alocar novo registro**  $z$ ;     ▷ vértice de  $T$
  5.      $z.esq \leftarrow x \leftarrow \text{EXTRAI\_MIN}(Q)$ ;
  6.      $z.dir \leftarrow y \leftarrow \text{EXTRAI\_MIN}(Q)$ ;
  7.      $z.f \leftarrow x.f + y.f$ ;
  8.      $\text{INSERE}(Q, z)$ ;
  9. **retorne**  $\text{EXTRAI\_MIN}(Q)$ .

## Corretude do algoritmo de Huffman



$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0 \end{aligned}$$

Assim,  $B(T) \geq B(T')$ .

Analogamente  $B(T') \geq B(T'')$ .

Como  $T$  é ótima,  $T''$  é ótima e o resultado vale.  $\square$



### Lema 2: (subestrutura ótima)

Seja  $C$  um alfabeto com frequência  $f[c]$  definida para cada caracter  $c \in C$ . Sejam  $x$  e  $y$  dois caracteres de  $C$  com as menores frequências. Seja  $C'$  o alfabeto obtido pela remoção de  $x$  e  $y$  e pela inclusão de um **novo** caracter  $z$ , ou seja,  $C' = C \cup \{z\} - \{x, y\}$ . As frequências dos caracteres em  $C' \cap C$  são as mesmas que em  $C$  e  $f[z]$  é definida como sendo  $f[z] = f[x] + f[y]$ .

Seja  $T'$  uma árvore binária representado um código ótimo livre de prefixo para  $C'$ . Então a árvore binária  $T$  obtida de  $T'$  substituindo-se o vértice (folha)  $z$  pela por um vértice interno tendo  $x$  e  $y$  como filhos, representa uma código ótimo livre de prefixo para  $C$ .

### Teorema:

O algoritmo de Huffman constrói um código ótimo (livre de prefixo).

Segue imediatamente dos Lemas 1 e 2.

### Prova do Lema 2:

- Comparando os custos de  $T$  e  $T'$ :
  - Se  $c \in C - \{x, y\}$ ,  $f[c]d_T(c) = f[c]d_{T'}(c)$ .
  - $f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) = f[z]d_{T'}(z) + (f[x] + f[y])$ .
- Logo,  $B(T') = B(T) - f[x] - f[y]$ .
- Por **contradição**, suponha que existe  $T''$  tal que  $B(T'') < B(T)$ .

Pelo lema anterior, podemos supor que  $x$  e  $y$  são folhas “irmãs” em  $T''$ . Seja  $T'''$  a árvore obtida de  $T''$  pela substituição de  $x$  e  $y$  por uma folha  $z$  com frequência  $f[z] = f[x] + f[y]$ . O custo de  $T'''$  é tal que

$$B(T''') = B(T'') - f[x] - f[y] < B(T) - f[x] - f[y] = B(T'),$$

contradizendo a hipótese de que  $T'$  é uma árvore ótima para  $C'$ .  $\square$

- 1 Formule o problema como um **problema de otimização** no qual uma escolha é feita, restando-nos então um único subproblema a resolver.
- 2 Provar que existe sempre uma solução ótima do problema que atende à **escolha gulosa**, ou seja, a escolha feita pelo algoritmo guloso é segura.
- 3 Demonstrar que, uma vez feita a escolha gulosa, o que resta a resolver é um subproblema tal que se combinarmos a resposta ótima deste subproblema com o(s) elemento(s) da escolha gulosa, chega-se à solução ótima do problema original.

Esta é a parte que requer mais engenhosidade!

Normalmente a prova começa com uma solução ótima *genérica* e a modificamos até que ela inclua o(s) elemento(s) identificados pela escolha gulosa.