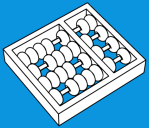
Algoritmos gulosos

MO417 - Complexidade de Algoritmos I

15

05/24



Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP

"Embora a gula seja considerada um dos sete pecados capitais, acontece que algoritmos gulosos frequentemente tem uma performance muito boa."

Stuart Russell.

Algoritmos gulosos

Problemas e subproblemas

Vamos estudar algoritmos gulosos:

- Decompomos um problema em vários subproblemas.
- De novo, um deles corresponde à **subestrutura ótima**.
- Mas podemos escolher a subestrutura eficientemente.

Comparando as estratégias:

- **Algoritmo de programação dinâmica:**
  1. Primeiro resolvemos todos os subproblemas.
  2. Depois decidimos o subproblema ótimo.

- **Algoritmo guloso:**
  1. Primeiro escolhemos o subproblema ótimo.
  2. Depois resolvemos apenas esse subproblema.

Algoritmos gulosos

Ideia:

- Realizamos uma sequência de passos.
- A cada passo, fazemos uma escolha.

Premissas dos algoritmos gulosos:

- A escolha é aquela que parece ser a melhor no momento.
- Essa escolha é denominada **escolha gulosa**.
- É feita de acordo com um **critério guloso**.

Nem sempre um algoritmo guloso encontra uma solução ótima, mas para vários problemas é possível mostrar que sim.

Uma receita para algoritmos gulosos

Vários problemas têm a seguinte estrutura:

- Existe um conjunto de elementos E.
- Uma solução é algum **subconjunto** A^* de E.

Um estratégia genérica:

1. Faça $A \leftarrow \emptyset$.
2. Enquanto A não é solução viável:
   (a) Escolha um elemento $\mathbf{e}$ com algum **critério guloso**.
   (b) Certifique-se de que existe solução A^* contendo $A \cup \{\mathbf{e}\}$.
   (c) Faça $A \leftarrow A \cup \{\mathbf{e}\}$.
3. Devolva A.

Seleção de atividades

Seleção de atividades

Considere n atividades executadas em certo lugar:

- Podem ser palestras, reuniões em um sala etc.
- Denote essas atividades por $S = \{a_1, \dots, a_n\}$.

Duração das atividades:

- A atividade a_i começa no tempo s_i e termina no tempo f_i.
- Assim, ela deve ser realizada no intervalo $[s_i, f_i)$.

Definição

*Duas atividades a_i e a_j são **compatíveis** se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ são disjuntos.*

Problema de seleção de atividades

Problema

- ***Entrada:*** *Conjunto de atividades* $S = \{a_1, \ldots, a_n\}$ *e tempos de início* s *e de termino* f.
- ***Solução:*** *Subconjunto* A *de atividades compatíveis.*
- ***Objetivo: maximizar*** $|A|$.

Uma instância

Os tempos de início e de término são:

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	9	9	10	11	12	14	16

- As atividades estão em ordem de **tempo de término**.
- Iremos usar essa ordem em seguida.

Exemplos de soluções

- $\{a_1, a_2\}$ e $\{a_1, a_3\}$ são pares incompatíveis.
- $\{a_1, a_4\}$ e $\{a_3, a_9, a_{11}\}$ são viáveis, mas não ótimas.
- $\{a_1, a_4, a_8, a_{11}\}$ e $\{a_2, a_4, a_9, a_{11}\}$ são viáveis e **ótimas**.

Definições preliminares

Supomos que $f_1 \leq f_2 \leq \ldots \leq f_n$:

- Ou seja, as atividades estão em ordem de **término**.
- Se não estiverem, podemos ordenar.

Para um par (i, j), defina $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$:

- Atividades que começam depois que a_i termina.
- Além disso, que terminam antes que a_j inicie.

Considere atividades dummies:

- Atividade a_0 com $f_0 = 0$ e atividade a_{n+1} com $s_{n+1} = \infty$.
- Assim S_{ij} está definido para todo $0 \leq i, j \leq n + 1$.
- O conjunto de todas atividades é $S = S_{0,n+1}$.

Subestrutura ótima

Considere uma solução ótima para as atividades em S_{ij}:

- Suponha que a_k esteja nessa solução ótima.
- As demais atividades da solução devem estar:
  - **antes** do início de a_k.
  - **depois** do término de a_k.

Descobrimos uma **subestrutura ótima**:

- Queremos uma solução ótima para o conjunto S_{ik}.
- Além de uma solução ótima para o conjunto S_{kj}.

Mas **NÃO** sabemos qual é a atividade a_k na solução ótima!

Usando programação dinâmica

Definimos o seguinte **subproblema**:

- Considere um par (i, j) para $0 \leq i, j \leq n + 1$.
- Defina $c[i, j]$ o **valor de uma solução ótima** para a instância do problema com atividades S_{ij}.

Podemos computar $c[i, j]$ com a recorrência:

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

Voltando ao problema original:

- O valor ótimo corresponde à entrada $c[0, n + 1]$.
- É fácil calcular usando **programação dinâmica** (exercício).

Simplificando

Algumas observações:

- Na programação dinâmica, testamos **todas** as atividades a_k.
- Em um algoritmo guloso, queremos escolher apenas **uma**.

Como escolher de forma gulosa?

- Queremos uma atividade que consome menos "recursos".
- As atividades estão ordenadas por tempo de término.
- a_1 é a escolha que deixa mais tempo para outras atividades.

Escolha gulosa

Depois de escolher a_1, qual subproblema resta?

- Queremos atividades que começam depois que a_1 termina.
- Definimos um novo subproblema $S_k = \{a_i \in S : s_i \geq f_k\}$.
- Pela definição da dummy a_0, o problema original é $S_0 = S$.

Teorema (Escolha gulosa)

Considere um subproblema S_k não vazio e seja a_m uma atividade em S_k com o ***menor tempo de término****.*

Então existe uma solução ótima para S_k que contém a_m.

Demonstração da escolha gulosa

Seja A^* uma **solução ótima** para S_k:

- Se $a_m \in A^*$, então nada há a fazer.
- Então suponha que $a_m \notin A^*$.

Vamos criar **outra solução ótima** A' contendo a_m:

- Seja $a_i \in A^*$ a atividade com menor f_k.
- Defina $A' = A \setminus \{a_i\} \cup \{a_m\}$.

Temos que provar que A' é solução ótima:

- É claro que $|A^*| = |A'|$.
- Então resta mostrar que A' é viável.
- Nenhuma atividade de $A^* \setminus \{a_k\}$ começa antes de f_i.
- Daí nenhuma atividade de $A^* \setminus \{a_k\}$ começa antes de f_m.
- Assim as atividade de A' são mutualmente compatíveis.

Resolvendo recursivamente

A discussão anterior sugere um algoritmo recursivo:

- Suponha que estamos tentando resolver S_k.
- Seja a_m a atividade em S_k com o menor tempo de término.
- Resolva o subproblema S_m e junte com a_m.

Algoritmo recursivo

- O vetor f está em ordem de tempo de término.
- Queremos atividades que começam **depois** que k termina.

Algoritmo: $\textsc{Seleção-Atividades-Rec}(s, f, k, n)$

▷ acha atividade a_m em S_k que termina primeiro
$m \leftarrow k + 1$
enquanto $m \leq n$ **e** $s_m < f_k$
  $m \leftarrow m + 1$
▷ escolhe a_m e resolve subproblema S_m
se $m \leq n$
  devolva $\{a_m\} \cup \textsc{Seleção-Atividades-Rec}(s, f, m, n)$
senão
  devolva $\emptyset$

Análise:

- Vemos cada elemento apenas uma vez, daí o tempo é $\Theta(n)$.
- Pode ser que precisemos ordenar as atividades.

Algoritmo iterativo

Podemos reescrever de maneira iterativa:

Algoritmo: $\textsc{Seleção-Atividades-Iter}(s, f, n)$

$A \leftarrow \{a_1\}$
$k \leftarrow 1$
para $m \leftarrow 2$ **até** n
  se $s_m \geq f_k$
    $A \leftarrow A \cup \{a_m\}$
    $k \leftarrow m$
devolva A

Codificação de Huffman

Codificações

Queremos representar um **texto**:

- É uma sequência de caracteres de um alfabeto C.
- Cada caractere está associado a uma sequência de bits.

Tipos de codificação:

- Comprimento fixo:
  - Cada sequência tem o mesmo número de bits.
  - Basta que elas sejam distintas.
- Codificação de comprimento variável:
  - As sequências podem ter tamanhos diferentes.
  - São **livres de prefixo**: uma sequência não é prefixo de outra.

Tamanho do texto codificado:

- É o número de bits usados para representar o texto.
- Codificações diferentes podem ter tamanhos diferentes.

Codificação de tamanho variável

Restrição de prefixo:

- O código de um caractere não pode ser prefixo de outro.
- Isso evita que a leitura do texto seja ambígua.
- Chamamos a codificação de **livre de prefixo**.

Exemplo de codificação ruim:

	a	b	c
código	01	1001	100101

- Considere uma sequência de bits 100101.
- A sequência corresponde ao texto ba ou ao texto c?

Exemplo

Considere um texto com 100.000 caracteres de $C = \{a, b, c, d, e, f\}$:

	a	b	c	d	e	f
Frequência (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

Tamanho do texto codificado

- Com a codificação de tamanho fixo, usamos

$$3 \cdot 100.000 = \mathbf{300.000} \text{ bits}$$

- Com a codificação de tamanho variável, usamos

$$(\underbrace{45 \cdot 1}_{a} + \underbrace{13 \cdot 3}_{b} + \underbrace{12 \cdot 3}_{c} + \underbrace{16 \cdot 3}_{d} + \underbrace{9 \cdot 4}_{e} + \underbrace{5 \cdot 4}_{f}) \cdot 1.000 = \mathbf{224.000} \text{ bits}$$

Codificação de Huffman

Codificação de Huffman:

- Problema para a compressão de dados.
- Dependendo da aplicação, reduz de 20 a 90%.

Problema

- ***Entrada:*** *Alfabeto C e tabela de frequências f.*
- ***Solução:*** *Codificação de comprimento variável.*
- ***Objetivo:*** ***minimizar*** *o tamanho do texto codificado.*

Representação de codificação

Uma codificação é representada por uma **árvore binária**:

- O filho esquerdo está associado ao **bit** 0.
- O filho direito está associado ao **bit** 1.
- As folhas representam os caracteres do alfabeto.

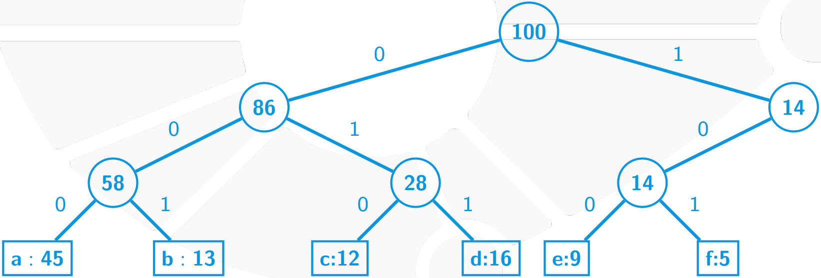
A codificação é livre de prefixo:

- Um código corresponde a um caminho até a folha.
- Prefixos só levam a nós internos.
- Então o código de um caractere não é prefixo de outro.

Arvore binária para codificação fixa

Codificação de comprimento fixo:

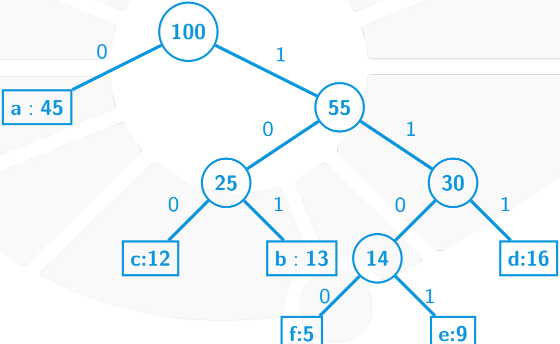
	a	b	c	d	e	f
Frequência	45	13	12	16	9	5
Código fixo	000	001	010	011	100	101



Arvore binária para codificação variável

Codificação de comprimento variável:

	a	b	c	d	e	f
Frequência	45	13	12	16	9	5
Código variável	0	101	100	111	1101	1100



Detalhes da estrutura

Árvores binárias **cheias**:

- São árvores em que cada nó interno tem dois filhos.
- Assim, há $|C|$ folhas e $|C| - 1$ nós internos (exercício).

Sempre existe uma codificação ótima que é cheia:

- Suponha que há nó x com um único filho y.
- Substituímos a subárvore de x pela subárvore de y.
- A nova árvore tem as mesmas folhas.
- O código de cada caractere só pode diminuir.

Detalhes da estrutura

Estrutura de um nó z:

- O filho esquerdo é denotado por $z.esq$.
- O filho direito é denotado por $z.dir$.
- A frequência dos caracteres na **subárvore** é $z.freq$.

Construindo uma árvore ótima

Vamos adotar a seguinte estratégia:

- Caracteres infrequentes estão em folhas mais profundas.
- Dois nós pouco frequentes são unidos por um nó interno.
- Construímos a árvore de maneira **bottom-up**.

Ideia:

- Começar com $|C|$ nós correspondendo aos caracteres.
- Juntar os dois nós menos frequentes.
- Repetir $|C| - 1$ vezes até sobrar apenas um nó.
- O nó restante é a raiz da árvore devolvida.

Algoritmo de Huffman

Algoritmo: $\textsc{Huffman}(C)$

$Q \leftarrow C$
para $i \leftarrow 1$ **até** $|C| - 1$
  alocar novo registro z
  $z.esq \leftarrow x \leftarrow \textsc{Extract-Min}(Q)$
  $z.dir \leftarrow y \leftarrow \textsc{Extract-Min}(Q)$
  $z.freq \leftarrow x.freq + y.freq$
  $\textsc{Insert}(Q, z)$
devolva $\textsc{Extract-Min}(Q)$

- Q é uma fila de prioridades ordenada pela frequência.
- $\textsc{Extract-Min}$ e $\textsc{Insert}$ têm custo $\Theta(\log n)$.
- O algoritmo tem complexidade de tempo $\Theta(n \log n)$.

Custo de uma codificação

Algumas notações:

- T é uma árvore binária representando uma codificação.
- $d_T(c)$ é a distância da raiz até o nó do caractere c.
- $f(c)$ é a frequência do caractere c.

Tamanho do texto codificado:

- O número de bits de um texto codificado por T é

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

- Dizemos que $B(T)$ é o **custo** de T.

Escolha gulosa

Lema (Escolha gulosa)

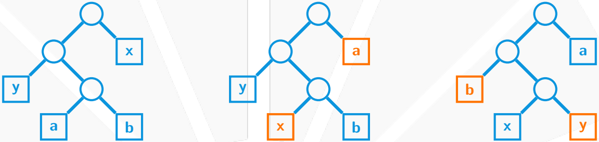
Sejam x e y os nós correspondentes aos dois caracteres com as menores frequências em C. Então:

- *Existe **codificação ótima** em que x e y são folhas irmãs.*
- *Elas estão tão distantes da raiz quanto qualquer folha.*

Demonstração:

- Considere uma árvore ótima T.
- Sejam a e b duas folhas irmãs mais profundas.
- Sejam x e y as duas folhas de menor frequência.
- Construa uma árvore T' trocando a e x.
- Depois uma árvore T'' trocando b por y.
- Vamos mostrar que T'' também é uma **árvore ótima**.

Escolha gulosa



- Aplicando a definição de B e cancelando termos idênticos,

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

- Assim, $B(T) \geq B(T')$ e, analogamente, $B(T') \geq B(T'')$.
- Como T é ótima, T'' também é ótima e o enunciado segue.

Subestrutura ótima

Lema (Subestrutura ótima)

Suponha que:

- x *e* y *são os dois caracteres com menores frequências.*
- $C' = C \setminus \{x, y\} \cup \{z\}$ *é alfabeto com* $f(z) = f(x) + f(y)$*.*
- T' *é uma árvore ótima para o alfabeto* C'*.*
- T *é obtida de* T' *substituindo-se* z *por duas folhas* x *e* y*.*

Então T *é uma árvore ótima para* C*.*

Subestrutura ótima (cont)

Comparando os custos de T e T':

- Se $c \in C \setminus \{x, y\}$, então

$$f(c)d_T(c) = f(c)d_{T'}(c)$$

- Como os códigos de x e y têm um bit a mais que z, sabemos que

$$\begin{aligned} f(x)d_T(x) + f(y)d_T(y) &= (f(x) + f(y))(d_{T'}(z) + 1) \\ &= f(z)d_{T'}(z) + (f(x) + f(y)) \end{aligned}$$

- Portanto, $B(T) = B(T') + f(x) + f(y)$.

Subestrutura ótima

Vamos mostrar que T é uma árvore ótima para C:

- Considere uma árvore ótima T^* para C.
- Pelo lema, supomos que x e y são folhas irmãs em T^*.
- Construa $\hat{T}$ a partir de T^* trocando x e y por z.
- Fazendo $f(z) = f(x) + f(x)$, temos $B(\hat{T}) = B(T^*) - f(x) - f(y)$.
- Como $\hat{T}$ é viável para C' e T' é ótima, $B(T') \leq B(\hat{T})$.
- Logo,

$$\begin{aligned} B(T) &= B(T') + f(x) + f(y) \\ &\leq B(\hat{T}) + f(x) + f(y) \\ &= B(T^*) - f(x) - f(y) + f(x) + f(y) \\ &= B(T^*) \end{aligned}$$

- Então, de fato, T é ótima para C.

Huffman

Algoritmo: $\textsc{Huffman}(C)$

$Q \leftarrow C$

para $i \leftarrow 1$ **até** $|C| - 1$

  alocar novo registro z

  $z.esq \leftarrow x \leftarrow \textsc{Extract-Min}(Q)$

  $z.dir \leftarrow y \leftarrow \textsc{Extract-Min}(Q)$

  $z.freq \leftarrow x.freq + y.freq$

  $\textsc{Insert}(Q, z)$

devolva $\textsc{Extract-Min}(Q)$

Correção do algoritmo de Huffman

Teorema

HUFFMAN *constrói uma codificação ótima.*

Demonstração:

- Seja T a codificação devolvida pelo algoritmo.
- Associamos cada nó de T a um caractere distinto.
- Vamos mostrar que T é ótima por indução em $|C|$.

Correção do algoritmo de Huffman (cont)

Considere $|C| = 1$:

- Nesse caso, o algoritmo devolve um único nó.
- Que é uma solução ótima.

Agora, suponha que $|C| \geq 2$:

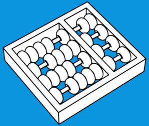
- Sejam x e y os caracteres escolhidos na primeira iteração.
- Seja z um caractere com $f(z) = f(x) + f(z)$.
- Considere um conjunto de caracteres $C' = C \setminus \{x, y\} \cup \{z\}$.
- A árvore devolvida pelo algoritmo para C' é $T' = T - x - y$.
- Como $|C'| < |C|$, por hipótese de indução, T' é ótima para C'.
- Então, pela subestrutura ótima, T é ótima para C.

Algoritmos gulosos

MO417 - Complexidade de Algoritmos I

15

05/24



Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br