Projeto e Análise de Algoritmos

Hamilton José Brumatto

Bacharelado em Ciência da Computação - UESC

26 de maio de 2016

Algoritmos Gulosos

Material desenvolvido com base no material de aula desenvolvido por: Cid Carvalho de Souza, Cândida Nunes da Silva e Orlando Lee

Conceitos Básicos

- O paradigma de algoritmo guloso representa uma forma de se resolver problemas de otimização.
- Para poder aplicar este paradigma é necessário que o problema apresente uma subestrutura ótima, tal qual programação dinâmica.
- **Programação Dinâmica:** É feito uma análise em um conjunto de subproblemas para escolher qual a solução ótima.
- **Algoritmos Gulosos:** Primeiro é feita a escolha da solução ótima para depois resolver o subproblema.

Técnica de Programação baseado na Escolha Gulosa

- Um algoritmo guloso sempre faz uma escolha que representa a **melhor** solução utilizando um critério guloso.
- A escolha realizada resolve um subproblema de forma ótima, o critério define uma decisão localmente ótima.
- A cada iteração do problema decide-se localmente por uma solução ótima através da escolha gulosa. O conjunto de ótimos locais no final forma um ótimo global.
- A escolha gulosa é **final**, uma vez escolhido não é feita troca. Não há *backtracking*
- Como o algoritmo depende da escolha, é preciso provar que a escolha resolve um problema local de forma ótima. O algoritmo é simples, mas a prova nem sempre é simples.

Seleção de Atividades

Considere o seguinte problema computacional:

Problema: Seleção de Atividades

Dado um conjunto S de atividades $S = \{a_1, a_2, ..., a_n\}$ onde a atividade a_k tem a duração $[s_k, f_k]$, as atividades a_i e a_j são ditas compatíveis se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ são disjuntos.
O Problema: Encontre no conjunto S um subconjunto de atividades mutuamente compatíveis que tenha tamanho máximo

- Para todo $k = 1, ..., n$, a atividade a_k começa no instante s_k e termina no instante f_k com $0 \leqslant s_k < f_k < \infty$, a atividade a_k é realizada no intervalo semi-aberto $[s_k, f_k)$.

Exemplo para Seleção de Atividades

- As atividades:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- Pares de atividades incompatíveis: (a_1, a_2), (a_1, a_3).
  Pares de atividades compatíveis: (a_1, a_4), (a_2, a_4).
- Subconjunto MAXIMAL de atividades compatíveis: (a_3, a_9, a_{11}).
- Subconjunto MÁXIMO de atividades comptatíveis: (a_1, a_4, a_9, a_{11}).
- Existe outro subconjunto máximo!
- Este conjunto de atividades, em especial, está ordenado por ordem crescente de término.

Abordagem por Programação Dinâmica

- Para aplicarmos a programação dinâmica, devemos identificar:
  - Característica de problema de otimização.
  - Existência de *Subestrutura Ótima*
  - Sobreposição de subproblemas.
- Inicialmente vamos verificar se o problema pode ser resolvido pela programação dinâmica.
- Já identificamos que este é um problema de otimização. Resta saber se existe subestrutura ótima e se existe sobreposição de subproblemas.

Subestrutura Ótima para Seleção de Atividades

- Vamos considerar as atividades ordenadas pelo instante de término, ou seja: $f_1 \leqslant f_2 \leqslant \ldots \leqslant f_k \leqslant \ldots \leqslant f_n$.
- Vamos incluir duas atividades para colocar fechamentos no conjunto: a_0 com $f_0 = 0$ e a_{n+1} com $s_{n+1} = \infty$, ou seja, qualquer subconjunto (não necessariamente próprio) de S pode ser descrito pela seguinte definição:

Definição de S_{ij}

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

- Temos que $S \equiv S_{0,n+1} = \{a_0, a_1, \ldots, a_n\}$ também fica bem definido. Observe que $S_{ij} = \emptyset$ para todo $i \geqslant j$.

Subestrutura Ótima para Seleção de Atividades

- Vamos considerar o subproblema definido pelo conjunto S_{ij} de atividades, suponha que a_k pertença a uma solução ótima de S_{ij}.
- Como $f_i \leqslant s_k < f_k \leqslant s_j$, uma solução ótima para S_{ij} que contenha a_k será composta pelas atividades de uma solução ótima de S_{ik}, por $\{a_k\}$ e pelas atividades de uma solução ótima S_{kj}.
- Logo temos subestruturas ótimas para o problema.
- A solução $c[i,j]$ para o conjunto S_{ij} representa o valor ótimo.
- Para o conjunto inicial, $S = S_{0,n+1}$ é $c[0, n+1]$.

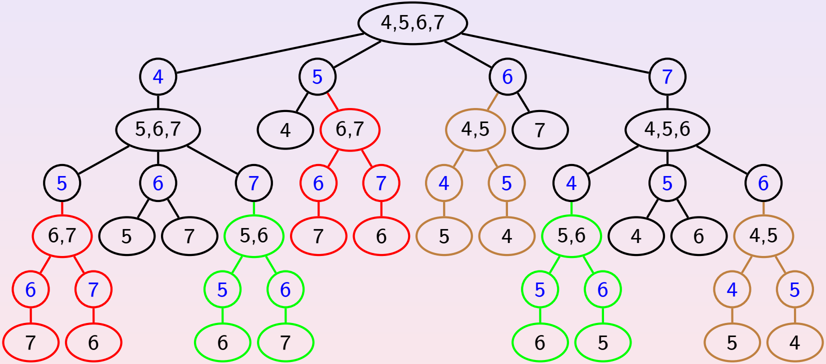
Sobreposição de subproblemas

- Vamos considerar a fórmula de recorrência:

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

- Vamos considerar o subconjunto $S_{3,8}$ do exemplo anterior.
- Vamos traçar uma árvore indicando a escolha de a_k e os subproblemas a serem resolvidos a partir da escolha.

Sobreposição de subproblemas



- Subproblemas formados por um conjunto de duas atividades aparecem resolvidos 2 vezes cada.

Resolvendo por Programação Dinâmica

- Este é um problema que podemos aplicar a programação dinâmica.
- Resolver o problema S_{ij} significa resolver S_{ik} e S_{kj} para todo $k \in (i, j)$.
- Inicialmente resolvemos os problemas S_{ii} para $i = 0, .., n + 1$, obviamente $c[i, i] = 0$.
- Partimos então para conjuntos de tamanho u, fazendo $S_{i,i+u}$ para $i = 0, ..., n + 1 - u$ com u variando de $1 a n + 1$
- Para cada iteração aplicamos a recorrência para encontrar a melhor solução:

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

Algoritmo por programação dinâmica:

Algoritmo para o problema de Seleção de Atividades

Algoritmo
SelecaoAtividadesProgramacaoDinamica(S,n)
  para $i \leftarrow 0$ **até** $n+1$ **faça**
    para $j \leftarrow i$ **até** $n+1$ **faça**
      $c[i,j] \leftarrow 0$
  para $u \leftarrow 1$ **até** $n+1$ **faça**
    para $i \leftarrow 0$ **até** $n+1-u$ **faça**
      $j \leftarrow i+u$
      para $k \leftarrow i+1$ **até** $j-1$ **faça**
        se $s_k \geqslant f_i$ **e** $f_k \leqslant s_j$**e** $c[i,k]+c[k,j]+1 > c[i,j]$
então

$$c[i,j] = c[i,k]+c[k,j]+1$$

  Retorna $c[0,n+1]$

Complexidade do algoritmo em programação dinâmica:

$$\begin{aligned}
T(n) &= \sum_{u=1}^{n+1} \sum_{i=0}^{n+1-u} \sum_{k=i+1}^{i+u-1} \Theta(1) \\
&= \sum_{u=1}^{n+1} \sum_{i=0}^{n+1-u} (u-1)\Theta(1) \\
&= \sum_{u=1}^{n+1} (n+2-u)(u-1)\Theta(1) \\
&= \left[(n+3)\sum_{u=1}^{n+1} u - \sum_{u=1}^{n+1} u^2 - (n+2)\sum_{u=1}^{n+1} 1 \right] \Theta(1) \ldots
\end{aligned}$$

Complexidade do algoritmo em programação dinâmica:

$$
\begin{aligned}
T(n) &= \left[(n+3)\sum_{u=1}^{n+1} u - \sum_{u=1}^{n+1} u^2 - (n+2)\sum_{u=1}^{n+1} 1]\right] \Theta(1)\ldots \\
\sum_{u=1}^{n+1} u^2 &= \frac{(n+1)(n+2)(2n+3)}{6} \\
(n+3)\sum_{u=1}^{n+1} u &= \frac{(n+1)(n+2)(n+3)}{2} \\
T(n) &= \frac{(n+1)(n+2)\left[3(n+3)-(2n+3)-6\right]}{6}\Theta(1) \\
T(n) &= \frac{n(n+1)(n+2)}{6}\Theta(1) = \frac{n^3+3n^2+2n}{6}\Theta(1) \\
T(n) &\in \Theta(n^3)
\end{aligned}
$$

Simulando a solução no exemplo: $u = 0$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0												
1		0											
2			0										
3				0									
4					0								
5						0							
6							0						
7								0					
8									0				
9										0			
10											0		
11												0	
12													0

Simulando a solução no exemplo: $u = 1$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0											
1		0	0										
2			0	0									
3				0	0								
4					0	0							
5						0	0						
6							0	0					
7								0	0				
8									0	0			
9										0	0		
10											0	0	
11												0	0
12													0

Simulando a solução no exemplo: $u = 2$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0										
1		0	0	0									
2			0	0	0								
3				0	0	0							
4					0	0	0						
5						0	0	0					
6							0	0	0				
7								0	0	0			
8									0	0	0		
9										0	0	0	
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: u = 3

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0									
1		0	0	0	0								
2			0	0	0	0							
3				0	0	0	0						
4					0	0	0	0					
5						0	0	0	0				
6							0	0	0	0			
7								0	0	0	0		
8									0	0	0	0	
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: $u = 4$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	1								
1		0	0	0	0	0							
2			0	0	0	0	0						
3				0	0	0	0	0					
4					0	0	0	0	0				
5						0	0	0	0	0			
6							0	0	0	0	0		
7								0	0	0	0	0	
8									0	0	0	0	1
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: $u = 5$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	1	0							
1		0	0	0	0	0	0						
2			0	0	0	0	0	0					
3				0	0	0	0	0	0				
4					0	0	0	0	0	0			
5						0	0	0	0	0	0		
6							0	0	0	0	0	0	
7								0	0	0	0	0	1
8									0	0	0	0	1
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: $u = 6$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	1	0	1						
1		0	0	0	0	0	0	0					
2			0	0	0	0	0	0	1				
3				0	0	0	0	0	0	0			
4					0	0	0	0	0	0	0		
5						0	0	0	0	0	0	1	
6							0	0	0	0	0	0	1
7								0	0	0	0	0	1
8									0	0	0	0	1
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: $u = 7$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	1	0	1	1					
1		0	0	0	0	0	0	0	1				
2			0	0	0	0	0	0	1	1			
3				0	0	0	0	0	0	0	0		
4					0	0	0	0	0	0	0	1	
5						0	0	0	0	0	0	1	2
6							0	0	0	0	0	0	1
7								0	0	0	0	0	1
8									0	0	0	0	1
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: $u = 8$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	1	0	1	1	2				
1		0	0	0	0	0	0	0	1	1			
2			0	0	0	0	0	0	1	1	0		
3				0	0	0	0	0	0	0	0	1	
4					0	0	0	0	0	0	0	1	2
5						0	0	0	0	0	0	1	2
6							0	0	0	0	0	0	1
7								0	0	0	0	0	1
8									0	0	0	0	1
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: u = 9

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	1	0	1	1	2	2			
1		0	0	0	0	0	0	0	1	1	0		
2			0	0	0	0	0	0	1	1	0	2	
3				0	0	0	0	0	0	0	0	1	3
4					0	0	0	0	0	0	0	1	2
5						0	0	0	0	0	0	1	2
6							0	0	0	0	0	0	1
7								0	0	0	0	0	1
8									0	0	0	0	1
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: $u = 10$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	1	0	1	1	2	2	0		
1		0	0	0	0	0	0	0	1	1	0	2	
2			0	0	0	0	0	0	1	1	0	2	3
3				0	0	0	0	0	0	0	0	1	3
4					0	0	0	0	0	0	0	1	2
5						0	0	0	0	0	0	1	2
6							0	0	0	0	0	0	1
7								0	0	0	0	0	1
8									0	0	0	0	1
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: $u = 11$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	1	0	1	1	2	2	0	3	
1		0	0	0	0	0	0	0	1	1	0	2	3
2			0	0	0	0	0	0	1	1	0	2	3
3				0	0	0	0	0	0	0	0	1	3
4					0	0	0	0	0	0	0	1	2
5						0	0	0	0	0	0	1	2
6							0	0	0	0	0	0	1
7								0	0	0	0	0	1
8									0	0	0	0	1
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Simulando a solução no exemplo: $u = 12$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
s_i		1	3	0	5	3	5	6	8	8	2	12	∞
f_i	0	4	5	6	7	8	9	10	11	12	13	14	

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	1	0	1	1	2	2	0	3	4
1		0	0	0	0	0	0	0	1	1	0	2	3
2			0	0	0	0	0	0	1	1	0	2	3
3				0	0	0	0	0	0	0	0	1	3
4					0	0	0	0	0	0	0	1	2
5						0	0	0	0	0	0	1	2
6							0	0	0	0	0	0	1
7								0	0	0	0	0	1
8									0	0	0	0	1
9										0	0	0	0
10											0	0	0
11												0	0
12													0

Algoritmo para recuperar o conjunto de atividades:

Algoritmo RecuperarAtividades(S,c,n)
  Retorna $ConjuntoAtividades(S, c, 0, n + 1)$

Algoritmo ConjuntoAtividades(S,c,i,j)
  $achou \leftarrow$ **falso**; $A \leftarrow \emptyset$
  se $c[i,j] \neq 0$ **então**
    $k \leftarrow i + 1$
    enquanto não $achou$ **e** $k < j$ **faça**
      se $s_k \geqslant f_i$ **e** $f_k \leqslant s_j$ **e** $c[i,k] + c[k,j] + 1 = c[i,j]$ **então**
        $achou \leftarrow$ **verdade**
$A \leftarrow ConjuntoAtividades[S, c, i, k] \cup \{a_k\} \cup ConjuntoAtividades[S, c, k, j]$
      $k \leftarrow k + 1$
  Retorna A

Revendo o problema através de um algoritmo guloso

- Apesar do algoritmo encontrado oferecer uma resposta em tempo polinomial $\Theta(n^3)$, há uma forma de se resolver uma quantidade consideravalmente menor de subproblemas.
- Para evitar de resolver muitos subproblemas, faremos uma ESCOLHA GULOSA, definindo somente uma única escolha de k para um subproblema S_{ij}.
- Com isso não precisamos mais testar todas as opções no intervalo.
- A **Propriedade de Escolha Gulosa** garante que a solução encontrada é ótima.

A escolha gulosa

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

$$f_m = \min\{f_k : a_k \in S_{i,j}\}$$

Então:

1. Existe uma solução ótima para $S_{i,j}$ que contenha a_m e,
2. $S_{i,m}$ 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 $S_{m,j}$ pode não ser vazio.

Precisamos mostrar que conseguimos definir, para o subconjunto $S_{i,j}$, a primeira atividade que pertença a esta solução.

Paradigma de construção de algoritmos pelo método guloso

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

Escolha gulosa no problema de Seleção de Atividades

- Seja a_m uma atividade que pertença a uma solução maximal de $S_{0,n+1}$, com $f_m = \min\{f_k : a_k \in Sij\}$
- Vamos provar que a_m pertence à solução ótima $S_{0,n+1}$:
  - Seja A um conjunto de atividades mutuamente compatíveis de tamanho máximo em $S_{i,j}$. Se $a_m \in A$ então está provado. Vamos considerar então que $a_m \notin A$.
  - Seja $a_k \in A$ com menor f_k. Vamos tomar o conjunto $A' = A - a_k \cup a_m$. Então A' também é conjunto de atividades mutuamente compatíveis de tamanho máximo.
  - A' e A possuem o mesmo número de atividades. A atividade em A, seguinte a a_k também é compatível com a_m, pois se o seu início ocorre em um instante superior a f_k, como $f_k \geqslant f_m$, então também ocorrerá em um instante superior a f_m.

Vamos utilizar a escolha gulosa declarada para construir o algoritmo

Para resolver $S_{i,j}$ precisamos:

1. Determinar a atividade a_m que possua o menor tempo de término em $S_{i,j}$
2. Resolver o subproblema $S_{m,j}$, já que $S_{i,m} = \emptyset$
3. Juntar à solução de $S_{m,j}$ a atividade a_m.
4. Devolver o conjunto de atividades.

O algoritmo guloso

Algoritmo SelecaoAtividadesGuloso(S,n)
    Retorna $ConjuntoAtividadesGuloso(S, 0, n+1)$

Algoritmo ConjuntoAtividadesGuloso(S,i,j)
    $A \leftarrow \emptyset$
    $m \leftarrow i+1$
    enquanto $m < j$ **e** $s_m < f_i$ **faça**
        $m \leftarrow m+1$
    se $m < j$ **e** $f_m \leqslant s_j$ **então**
        $A \leftarrow \{a_m\} \cup ConjuntoAtividadesGuloso(S, m, j)$
    Retorna A

Complexidade do Algoritmo

- A complexidade é dada pela recorrência:

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n-k) + k & n > 1 \text{ e } k < n \end{cases}$$

- Aplicando a recorrência i vezes teremos: $T(n) = T(n - (k_1 + k_2 + \ldots + k_i)) + k_1 + k_2 + \ldots + k_i$
- Chegando a: $n - (k_1 + k_2 + \ldots k_i) = 1$ para encerrar a recursão, ficamos com:
- $T(n) = T(1) + n - 1 = n$.
- $T(n) \in \Theta(n)$.
- Ao longo das chamadas recursivas, a atividade a_k é examinada somente uma única vez no laço das linhas 3..4.
- Este algoritmo pode ser transformado em uma versão não recursiva.

Algoritmo iterativo para Seleção de Atividades

Algoritmo SELECAOATIVIDADESGULOSO(S,n)
  $A \leftarrow a_1$
  $i \leftarrow 1$
  para $m \leftarrow 2$ **até** n **faça**
    se $s_m \geqslant f_i$ **então**
      $A \leftarrow A \cup \{a_m\}$
      $i \leftarrow m$
  Retorna A

Invariantes do laço **while**: Antes da iteração $m+1$ do laço,

- $A \subseteq A'$, sendo A' conjunto máximo de atividades em $S_{0,n+1}$.
- f_i é sempre o maior instante de término de uma atividade em A, ou seja, estamos resolvendo o subproblema $S_{i,n+1}$
- O subproblema $S_{0,i}$ está resolvido de forma ótima e a_i é uma escolha gulosa.

Considerações sobre a abordagem

- Pode-se concluir facilmente que para o algoritmo iterativo, $T(n) \in \Theta(n)$
- A prova de que a escolha gulosa é ótima foi feito considerando-se que existe uma solução ótima que não envolve a escolha gulosa que fizemos. Mostramos que é possível trocar um elemento desta solução ótima pela escolha gulosa, continuamos com uma solução ótima que contém nossa escolha gulosa.
- Este tipo de prova é uma forma mais usada para se provar a corretude do algoritmo.

Códigos de Huffman

- Representa uma técnica de compressão de dados que pode atingir valores entre 20 e 90%.
- É aplicado em arquivos de símbolos (texto) onde existe um distribuição diferenciada na freqüência com que cada símbolo aparece.
- Codifica-se os símbolos com tamanhos distintos de bits. Símbolos mais frequentes recebem menos bits, símbolos menos frequentes recebem mais bits.
- Na média o número de bits do arquivo será menor.

Exemplo do código de Huffman

- Considere o alfabeto $C = \{a, b, c, d, e, f\}$.
- Um dado arquivo possui a freqüência indicada na tabela abaixo para os caracteres do alfabeto.
- Também na tabela estão indicadas duas possíveis codificações para cada objeto, uma de tamanho fixo e outra de tamanho variável.

	a	b	c	d	e	f
Freqüência (milhares)	45	13	12	16	9	5
Código: *Tamanho Fixo*	000	001	010	011	100	101
Código: *Tamanho Variável*	0	101	100	111	1101	1100

- Qual o tamanho do arquivo para cada uma das codificações?

Calculando o custo para cada tipo de codificação

- Codificação com códigos de tamanho fixo:

$$Total de bits = 3 \times 100.000 = 300.000 bits$$

- Codificação com códigos de tamanho variável:

$$\underbrace{1 \times 45}_{a} + \underbrace{3 \times 13}_{b} + \underbrace{3 \times 12}_{c} + \underbrace{3 \times 16}_{d} + \underbrace{4 \times 9}_{e} + \underbrace{4 \times 5}_{f} = 224.000 bits$$

- Há um ganho de aproximadamente 25% se utilizarmos a codificação de tamanho variável.

Problema computacional associado

Problema da codificação

Dadas as freqüências de ocorrência dos caracteres de um arquivo, encontrar as seqüências de bits (códigos) para representar cada caracter de modo que o arquivo comprimido tenha tamanho mínimo.

- Entendemos que qualquer codificação resulta em uma solução para o arquivo, mas buscamos a solução que seja ótima, ou melhor, este é um problema de otimização.

Entendendo a solução

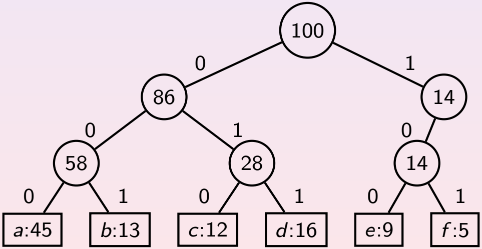
- Vamos apresentar a solução do problema antes de mostrar o processo de construção da solução.
- A solução implica no uso de uma "codificação livre de prefixo".
- Em uma codificação livre de prefixo, para quaisquer símbolos distintos i e j codificados, a codificação de i não é prefixo da codificação de j.
- No exemplo anterior, usando a codificação variável para a palavra "abc" obtemos: 0101100.
  - O único caracter começado com 0 e que portanto utiliza somente um bit é o 'a';
  - A seqüência 101 define o caracter 'b' e não há qualquer outro caracter que inicie com o código 101;
  - O restante 100 representa o caracter 'c'.

Representando o código

- Precisamos identificar uma estrutura que associe um código ao caracter, de forma que na decodificação encontremos facilmente o símbolo utilizando o código fornecido.
- Uma solução é utilizar uma árvore binária:
  - Um filho esquerdo está associado a um bit 0.
  - Um filho direito a um bit 1.
  - Nas folhas se encontram os símbolos.
  - O código lido 0 ou 1 faz com que na navegação na árvore chegue a um símbolo.
  - Ao achar um símbolo o próximo código é aplicado a partir do raiz.

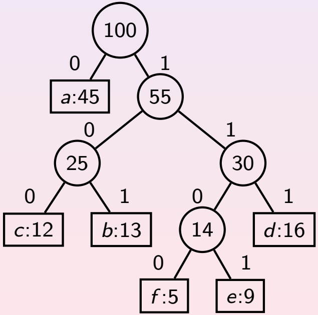
Código de tamanho fixo na forma de árvore

	a	b	c	d	e	f
Freqüência (milhares)	45	13	12	16	9	5
Código	000	001	010	011	100	101



Código de tamanho variável na forma de árvore

	a	b	c	d	e	f
Freqüência (milhares)	45	13	12	16	9	5
Código	0	101	100	111	1101	1100



Propriedades da árvore de código

- Cada código é livre de prefixo: Só há um único caminho para chegar a uma folha, que não passa por outra folha, assim o código de um símbolo não é um prefixo de outro símbolo.
- Uma codificação ótima deve ser representado por uma árvore binária cheia, cada vértice interno tem dois filhos. Seja uma codificação com um vértice interno que só tenha um filho:
    - Se o filho for uma folha. Podíamos colocar esta folha no lugar do vértice e economizaríamos um bit para o código deste símbolo.
    - Se o filho for outro vértice. A partir deste vértice buscamos uma folha, colocamos esta folha como segundo filho do vértice. Economizaríamos no mínimo um bit.
- Buscamos uma árvore binária cheia com $|C|$ folhas (o tamanho do alfabeto) e $|C-1|$ vértices internos.

Prove por indução que uma árvore binária cheia com n folhas possui $n-1$ vértices internos. (Dica: contraia um vértice com duas folhas em uma única folha)

Computando o custo da árvore: Tamanho do arquivo

- Seja T uma árvore de codificação, representando o alfabeto C:
- $d_T(c)$, para $c \in C$ é a profundidade na árvore T do caracter c.
- $f(c)$, para $c \in C$ é a freqüência do caracter c no arquivo.
- O custo da árvore, ou seja, tamanho do arquivo é dado por:

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

Entendendo a proposta de solução

- Começar com $|C|$ árvores folhas isoladas e realizar seqüencialmente $|C-1|$ operações de agregação, agregando duas árvores a um novo vértice raiz comum. O raiz passa a ter como "peso" a soma dos custos de cada árvore agregada.
- A escolha do par de árvores que serão agregadas dependerá do custo de cada árvore. As duas árvores de menor custo serão escolhidas.
- O raiz de uma árvore carrega como informação o custo da árvore.

O algoritmo de Huffman

Entrada: Conjunto de caracteres de C e a freqüências f de cada caracter

Saída: Raiz da árvore binária representando codificação ótima livre de prefixo

Algoritmo HUFFMAN(C)

$n \leftarrow |C|$
$Q \leftarrow C$ ▷ Q é fila de árvores ordenadas por custo
para $i \leftarrow 1$ **até** $n - 1$ **faça**
  $z \leftarrow$ **novo** *Arvore*
  $z.esq \leftarrow Extrai\_Minimo(Q)$
  $z.dir \leftarrow Extrai\_Minimo(Q)$
  $z.info = z.esq.info + z.dir.info$
  $Insere(Q, z)$
Retorna $Extrai\_Minimo(Q)$

Escolha Gulosa

Lema 1: Escolha Gulosa

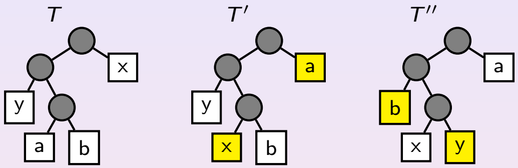
Seja C um alfabeto onde cada caracter $c \in C$ tem freqüência $f(c)$. Sejam x e y dois caracteres em C com as menores freqüências. Então, existe um código ótimo livre de prefixo para C no qual os códigos para x e y têm o mesmo comprimento e diferem apenas no último bit.

A técnica da prova da escolha gulosa segue a mesma idéia geral já passada. Supomos que existe uma solução ótima que não inclui a nossa escolha. A partir desta solução criamos outra que contenha a nossa escolha e mostramos que também é ótima.

Prova da Escolha Gulosa

- Seja T uma árvore **ótima**.
- Sejam a e b duas folhas "irmãs" mais profundas de T e x e y as folhas de T com menor freqüência.
- Se x e y possuem a mesma profundidade de a e b, podemos trocar x com a e y com b que o custo da árvore não se altera, e o lema está provado. Supomos então que x e y não possuam as mesmas profundidades de a e b.
- Vamos criar T' trocando x com a e depois T'' trocando y com b.

Prova da Escolha Gulosa



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

Analogamente, $B(T') - B(T'') \geqslant 0$. Como T é ótima, então T'' também é ótima.

Provando que existe a subestrutura ótima

Lema 2: Subestrutura Ótima

Seja C um alfabeto com freqüência $f(c)$ definida para cada caracter $c \in C$. Sejam x e y dois caracteres de C com as menores freqüências. 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 freqüências 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 por um vértice interno tendo x e y como filhos, representa um código ótimo livre de prefixo para C.

Prova da subestrutura ótima

- Precisamos verificar os custos de T e T':
  - para $c \in T \cap T'$, $f(c)d_T(c) = f(c)d_{T'}(c)$
  - pela propriedade da escolha gulosa, e por construção: $d_T(x) = d_T(y) = d_{T'}(z) + 1$
  - $f(x)d_T(x) + f(y)d_T(y) = (f(x) + f(y))(d_{T'}(z) + 1)$ $f(z)d_{T'}(z) + f(x) + f(y)$.
  - $B(T') = \sum_{c \in C \cap C'} f(c)d_{T'}(c) + f(z)d_{T'}(z)$ $B(T') = B(T) - f(x) - f(y)$
- T' é uma árvore ótima para C', vamos supor por contradição que T não seja ótima, ou seja, existe a árvore T'' com custo menor que T. Seja T''' a árvore obtida de T'' pela substituição de x e y por uma folha z com freqüência $f(z) = f(x) + f(y)$. O custo de T''' é: $B(T''') = B(T'') - f(x) - f(y) < B(T) - f(x) - f(y) = B(T')$, o que contradiz a hipótese de que T' é uma árvore ótima para C'.

Corretude do Algoritmo de Huffman

Teorema:

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

Segue imediatamente dos Lemas 1 e 2.

Revendo os passos da técnica baseada em algoritmos gulosos:

1. Identifique o problema como problema de otimização.
2. Defina uma escolha gulosa.
3. Prove que existe sempre uma solução ótima do problema que atenda à escolha gulosa.
   A técnica aqui é considerar uma solução ótima que não contenha a escolha gulosa, e modificá-la até que ela inclua a escolha gulosa, e mostrar que esta solução também é ótima.
4. Prove que existe uma subestrutura ótima. Ou seja, feita a escolha gulosa, o que resta é um único subproblema a resolver. E a solução ótima deste subproblema mais a escolha gulosa é a solução ótima do problema original.

Atividades baseadas no CLRS.

1. Ler capítulos 16 (prefácio), 16.1 e 16.2.
2. Exercícios: 16.1-2, 16.1-3, 16.2-1, 16.2-4, 16.2-5
3. Resolva a prova por indução do Slide "Propriedades da árvore de código"
4. Ler capítulo 16.3
5. Exercícios: 16.3-1, 16.3-2.
6. Problemas: 16-2.
7. Lista 10.