Algoritmos Gulosos

Eduardo Camponogara

Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina

DAS-9003: Introdução a Algoritmos

Sumário

Introdução

Algoritmos Gulosos

- Algoritmos para problemas de otimização tipicamente executam uma sequência de passos, com um conjunto de escolhas a cada passo.
- Um *Algoritmo Guloso* sempre escolhe a melhor opção para o momento.
- Ele faz uma escolha ótima local esperando que esta o leve a uma solução ótima global.

Introdução

Algoritmos Gulosos

Este capítulo explora os problemas de otimização que podem ser resolvidos por algoritmos gulosos:

- Problema de seleção de atividades
- Compressão de dados (Huffman)
- Matróides
- Agendamento de tarefas de tempo unitário com *deadlines* e penalidades

Introdução

Algoritmos Gulosos

Outros exemplos:

- Algoritmo para árvores de espalhamento mínima
- Algoritmo de Dijkstra

Sumário

Um problema de seleção de atividades

Definição do problema

- Dado o conjunto $S = \{a_1, a_2, \ldots, a_n\}$ de n atividades propostas que desejam utilizar um dado recurso, como uma sala de aula, que pode ser utilizada por apenas uma atividade por vez.
- Cada atividade a_i possui:
  - *instante de início* s_i
  - *instante de término* f_i
  - tal que a condição a seguir é satisfeita: $0 \leq s_i < f_i < \infty$

Um problema de seleção de atividades

Definição do problema

- Caso selecionada, a atividade a_i posiciona-se no intervalo semi-aberto $[s_i, f_i)$.
- Atividades a_i e a_j são compatíveis se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ não se sobrepõem.

Um problema de seleção de atividades

Problema

Selecione o conjunto tamanho-máximo de atividades mutuamente compatíveis.

Um problema de seleção de atividades

Exemplo

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

- Para este exemplo, $\{a_3, a_9, a_{11}\}$ são atividades mutuamente compatíveis.
- $\{a_1, a_4, a_8, a_{11}\}$ são também compatíveis, que é o maior conjunto de atividades mutuamente compatíveis
- Outro conjunto máximo é $\{a_2, a_4, a_9, a_{11}\}$.

Problema de seleção de atividades

Subestrutura ótima

- Seja $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$ o conjunto das atividades que podem iniciar após a atividade a_i terminar, e antes de a_j iniciar.
- De fato, S_{ij} consiste de todas as atividades que são compatíveis com:
  i) a_i e a_j;
  ii) todas as atividades que terminam antes ou no momento de término de a_i;
  iii) todas as atividades iniciam no momento ou após o início de a_j.

Problema de seleção de atividades

Subestrutura ótima

- Para representar o problema completo, adicionamos atividades fictícias a_0 e a_{n+1} e adotamos a convenção $f_0 = 0$ e $s_{n+1} = \infty$.
- Então, $S = S_{0,n+1}$, e os limites para i e j são dados por $0 \leq i, j \leq n + 1$.
- Assumiremos também que as atividades estão ordenadas em ordem monotônica crescente do instante de término

$$f_0 \leq f_1 \leq f_2 \leq \ldots \leq f_n \leq f_{n+1}$$

Problema de seleção de atividades

Proposição

$S_{ij} = \emptyset$ com $i \geq j$

Demonstração

Suponha que $\exists a_k \in S_{ij}$, então

$$f_i \leq s_k < f_k \leq s_j < f_j \implies f_i < f_j$$

que é uma contradição.

Problema de seleção de atividades

Conclusão

- Concluímos que, assumindo que as atividades estão ordenadas em ordem monotônica crescente do instante de término, o subproblema é selecionar o conjunto tamanho-máximo de atividades mutuamente compatíveis de S_{ij}, para $0 \leq i < j \leq n+1$, sabendo que todos os outros S_{ij} são vazios.
- Considere o subproblema S_{ij} não vazio, e suponha que a solução para S_{ij} inclui alguma atividade a_k, de forma que $f_i \leq s_k < f_k \leq s_j$.

Problema de seleção de atividades

Quebrando o problema

- Usando a atividade a_k, dois subproblemas são gerados:
  - S_{ik}: atividades que iniciam depois de a_i e terminam antes que a_k inicie,
  - S_{kj}: atividades que iniciam depois de a_k e terminam antes que a_j inicie.
- Cada par (S_{ik}, S_{kj}) consiste de subconjuntos das atividades em S_{ij}.
- A solução é a união das soluções para S_{ik} e S_{kj} com a atividade a_k.

Problema de seleção de atividades

Estrutura ótima do problema

- Se uma solução ótima A_{ij} para S_{ij} inclui a atividade a_k, então as soluções A_{ik} para S_{ik} e A_{kj} para S_{kj} também devem ser ótimas.
- Obtemos portanto uma solução ótima A_{ij} fazendo:

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

Problema de seleção de atividades

Solução Recursiva

- Seja $c[i,j]$ o número de atividades em um conjunto tamanho-máximo de atividades mutuamente compatíveis em S_{ij}.
- Temos que $c[i,j] = 0$ sempre que $S_{ij} = \emptyset$; em particular, $c[i,j] = 0$ para $i \geq j$
- Recursivamente:

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

Solução Gulosa

Teorema

Considere qualquer subconjunto S_{ij} não vazio, e seja a_m a atividade em S_{ij} com menor instante de término:

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

Então:

1) A atividade a_m é usada em algum conjunto tamanho-máximo de atividades mutuamente compatíveis em S_{ij}.
2) O subconjunto S_{im} é vazio, e a escolha de a_m faz com que S_{mj} seja o único subproblema não vazio.

Solução Gulosa

Demonstração

- Primeiro, provaremos (2) antes porque é mais simples.
- Suponha S_{im} não vazio, de maneira que exista uma atividade a_k tal que

$$f_i \leq a_k < f_k \leq s_m < f_m$$

- Então a_k está também em S_{ij} e possui um instante de término anterior a a_m, que contradiz com a escolha.
- Concluímos então que S_{im} é vazio.

Solução Gulosa

Demonstração

- Para provar a primeira parte, suponha que A_{ij} é um subconjunto tamanho-máximo de atividades mutuamente compatíveis com S_{ij}
- Ordene as atividades em A_{ij} em ordem monotonicamente crescente de instante de término.
- Seja a_k a primeira atividade em A_{ij}.
- Se $a_k = a_m$, está concluído.

Solução Gulosa

Demonstração

- Se $a_k \neq a_m$, construímos o subconjunto

$$\hat{A}_{ij} = (A_{ij} - \{a_k\}) \cup \{a_m\}$$

- As atividades $\hat{A}_{ij}$ são disjuntas, pois as atividades em A_{ij} são disjuntas, a_k é a primeira atividade em A_{ij} a terminar, e $f_m \leq f_k$.
- Note que $\hat{A}_{ij}$ possui o mesmo número de atividades que A_{ij}, logo $\hat{A}_{ij}$ é um subconjunto tamanho-máximo de atividades mutuamente compatíveis para S_{ij} que inclui a_m.

Problema de seleção de atividades

Por que este teorema é tão valioso?

- Em nossa solução em programação dinâmica, dois subconjuntos são usados na solução ótima, e existem $j - i + 1$ escolhas para resolver S_{ij}.
- O teorema reduz o número de decisões significativamente: apenas um subproblema é utilizado na solução ótima (o outro subproblema é garantidamente vazio)
- Quando buscamos uma solução para S_{ij}, temos que considerar apenas uma escolha: aquela com instante de término mais cedo em S_{ij}.

Problema de seleção de atividades

Por que este teorema é tão valioso?

- O teorema permite que resolvamos o problema de maneira *top-down*
- Para resolver o problema S_{ij}:
  - Escolhemos a atividade a_m em S_{ij} com o instante de conclusão mais cedo.
  - Então resolvemos S_{mj} para obter $A_{m,j}$.
  - Incluímos à solução $\{a_m\}$ o conjunto de atividades A_{mj} utilizadas na solução ótima para o subproblema S_{mj}.
- Em outras palavras,

$$A_{ij} = \{a_m\} \cup A_{mj}$$

Problema de seleção de atividades

Comportamento Guloso

- A atividade a_m que escolhemos quando resolvemos o subproblema é sempre a de menor instante de término que pode ser legalmente agendada.
- A atividade escolhida é portanto a escolha "*gulosa*" no sentido que ela deixa tanto tempo quanto possível para que as atividades restantes sejam agendadas.
- Isto é, uma escolha gulosa é aquela que maximiza a quantidade de tempo restante não agendada.

Problema de seleção de atividades

Um algoritmo guloso recursivo

- Entrada do algoritmo:
  a) vetores s e f
  b) índices i e j do subproblema S_{ij} a resolver
- Retorna o conjunto tamanho-máximo de atividades mutuamente compatíveis em S_{ij}.
- Assumimos que as atividades, de tamanho n, estão ordenadas de maneira crescente pelo instante de término.

Problema de seleção de atividades

Algoritmo recursivo

Recursive_Activity_Selector(s, f, i, j)
    $m \leftarrow i + 1$
    while $m < j$ and $s_m < f_i$
        do $m \leftarrow m + 1$
    if $m < j$
        then return $\{a_m\} \cup$ Recursive_Activity_Selector(s, f, m, j)
        else return $\emptyset$

Problema de seleção de atividades

Algoritmo recursivo

- Chamada: Recursive_Activity_Selector$(s, f, 0, n+1)$
- Assumindo que as atividades já estão ordenadas pelo instante de término, o algoritmo executa em tempo $\Theta(n)$.

Sumário

Elementos dos algoritmos gulosos

Elementos

- Um algoritmo guloso obtém uma solução ótima para um problema fazendo uma sequência de escolhas.
- Para cada ponto de decisão no algoritmo, a escolha que parece ser a melhor para o momento é feita.
- Esta estratégia heurística nem sempre produz a solução ótima, mas como vimos no problema de seleção de atividades, algumas vezes sim.

Elementos dos algoritmos gulosos

Elementos

Para desenvolver um algoritmo guloso para o problema de seleção de atividades, seguimos os seguintes passos:

1) determinar a estrutura ótima do problema,
2) desenvolver uma solução recursiva,
3) provar que em cada estágio da recursão, uma das escolhas ótimas é a gulosa,
4) mostrar que todos menos um dos subproblemas induzidos por ter feito a escolha gulosa é vazio,

Elementos dos algoritmos gulosos

Elementos

Para desenvolver um algoritmo guloso para o problema de seleção de atividades, seguimos os seguintes passos:

5) desenvolver um algoritmo recursivo que implemente a estratégia gulosa,
6) converter o algoritmo recursivo para iterativo.

Elementos dos algoritmos gulosos

Elementos

De maneira mais geral, desenvolvemos algoritmos gulosos de acordo com a seguinte sequência de passos:

1) converta o problema de otimização tal que ao fazermos uma escolha nos resta apenas um subproblema a resolver,
2) prove que existe sempre uma solução ótima para o problema original que faz a escolha gulosa, então ela sempre será segura,
3) demonstre que, tendo feito a escolha gulosa, o que resta é um subproblema cuja solução pode ser combinada à escolha gulosa de maneira a se chegar a uma solução ótima global.

Elementos dos algoritmos gulosos

Observações

Como podemos dizer se um algoritmo guloso resolverá um problema de otimização em particular?

- Não existe uma maneira geral, mas a propriedade *greedy-choice* e a subestrutura ótima são dois ingredientes chaves.

Elementos dos algoritmos gulosos

Propriedade *greedy-choice*

O primeiro elemento chave é a propriedade *greedy-choice*:

- uma solução ótima global pode ser atingida fazendo escolhas ótimas locais (gulosas).
- Em outras palavras, quando consideramos qual escolha a fazer, fazemos a que parece melhor no momento, sem considerar os resultados dos subproblemas.

Elementos dos algoritmos gulosos

DP $\times$ algoritmos gulosos

- Em programação dinâmica, fazemos uma escolha a cada passo, mas a escolha usualmente depende da solução dos subproblemas.
- Consequentemente, resolvemos problemas de programação dinâmica de maneira *bottom-up*, progredindo de subproblemas menores para maiores.

Elementos dos algoritmos gulosos

DP × algoritmos gulosos

- Em algoritmos gulosos, fazemos a escolha que parece ser melhor no momento e então resolvemos o subproblema
- Portanto, ao contrário da programação dinâmica, que resolve o problema de maneira *bottom-up*, a estratégia gulosa usualmente progride de maneira *top-down*, fazendo uma escolha gulosa após a outra, reduzindo cada instância do problema em uma menor.

Elementos dos algoritmos gulosos

Subestrutura ótima

- Um problema possui **subestrutura ótima** se uma solução ótima do problema contém soluções ótimas para os subproblemas
- Esta propriedade é o ingrediente chave para a aplicação de programação dinâmica, assim como algoritmos gulosos.
- **Exemplo:** Se uma solução ótima para o subproblema S_{ij} inclui uma atividade a_k, então ela deve conter soluções ótimas para os subproblemas S_{ik} e S_{kj}.

Elementos dos algoritmos gulosos

O problema da mochila

O problema $0-1$ da mochila é colocado como:

- um ladrão roubando uma loja encontra n itens;
- o i-ésimo item vale v_i reais e pesa w_i quilogramas.
- v_i e w_i são inteiros.
- desejamos carregar o máximo de valor possível, mas podemos carregar no máximo w quilogramas na mochila para algum w inteiro.

Problema

Quais itens devem ser levados?

Elementos dos algoritmos gulosos

O problema da mochila

- O problema da mochila possui subestrutura ótima.
- Para o problema $0-1$, considere que a carga mais valiosa a ser carregada pesa no máximo w quilogramas.
- Se removermos o item j da mochila, o restante deve ser a carga mais valiosa que pese no máximo $w - w_j$ que o ladrão pode levar dos $n-1$ itens originais excluindo j.

Sumário

Códigos Huffman

Indrodução

- Os códigos tipo Huffman são amplamente usados, constituindo uma técnica efetiva para compressão de dados.
- Tais códigos podem economizar de 20% a 90%, dependendo das características dos dados.
- O algoritmo guloso de Huffman utiliza uma tabela de frequência das ocorrências dos caracteres para construir um vetor ótimo que representa cada caractere com um conjunto de bits.

Códigos Huffman

Exemplo

- Suponha que temos um arquivo com 100.000 caracteres que desejamos comprimir.
- Os caracteres no arquivo ocorrem com frequências dadas pela tabela:

	a	b	c	d	e	f
Freqüência	45	13	12	16	9	5
Código fixo	000	001	010	011	100	101
Código variável	0	101	100	111	1101	1100

Códigos Huffman

Exemplo

- Se utilizarmos uma codificação de tamanho fixo, precisaremos de 3 bits para representar 6 caracteres.
- Este método requer 300.000 bits para codificar o arquivo.
- A codificação de tamanho variável é consideravelmente melhor que a de tamanho fixo, designando aos caracteres frequentes códigos curtos, e longos aos infrequentes.

Códigos Huffman

Exemplo

Para a codificação de tamanho variável da tabela, o código requer

$$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 =$$

$$= 224.000 \text{ bits}$$

Uma economia de aproximadamente 25%. De fato, esta é uma codificação ótima.

Códigos Huffman

Prefix Code

- Consideramos códigos onde nenhum código é prefixo de outro.
- Estes códigos são chamados de *prefix codes*
- É possível demostrar que a compressão de dados ótima de uma codificação pode sempre ser gerada por *prefix codes*, então não existe perda de generalidade em se restringir a atenção aos *prefix codes*.

Códigos Huffman

Prefix Code

- Codificar é simples: basta concatenar os códigos que representam cada caractere.

$$\text{“abc”} \Rightarrow [0|101|100] \Rightarrow 0101100$$

Códigos Huffman

Prefix Code

- Códigos prefixo são desejáveis porque eles simplificam a codificação.
- Uma vez que nenhum código é prefixo de outro, o código que inicia um arquivo codificado é único.
- Decodificamos o primeiro código, e repetimos o processo até o final do arquivo.

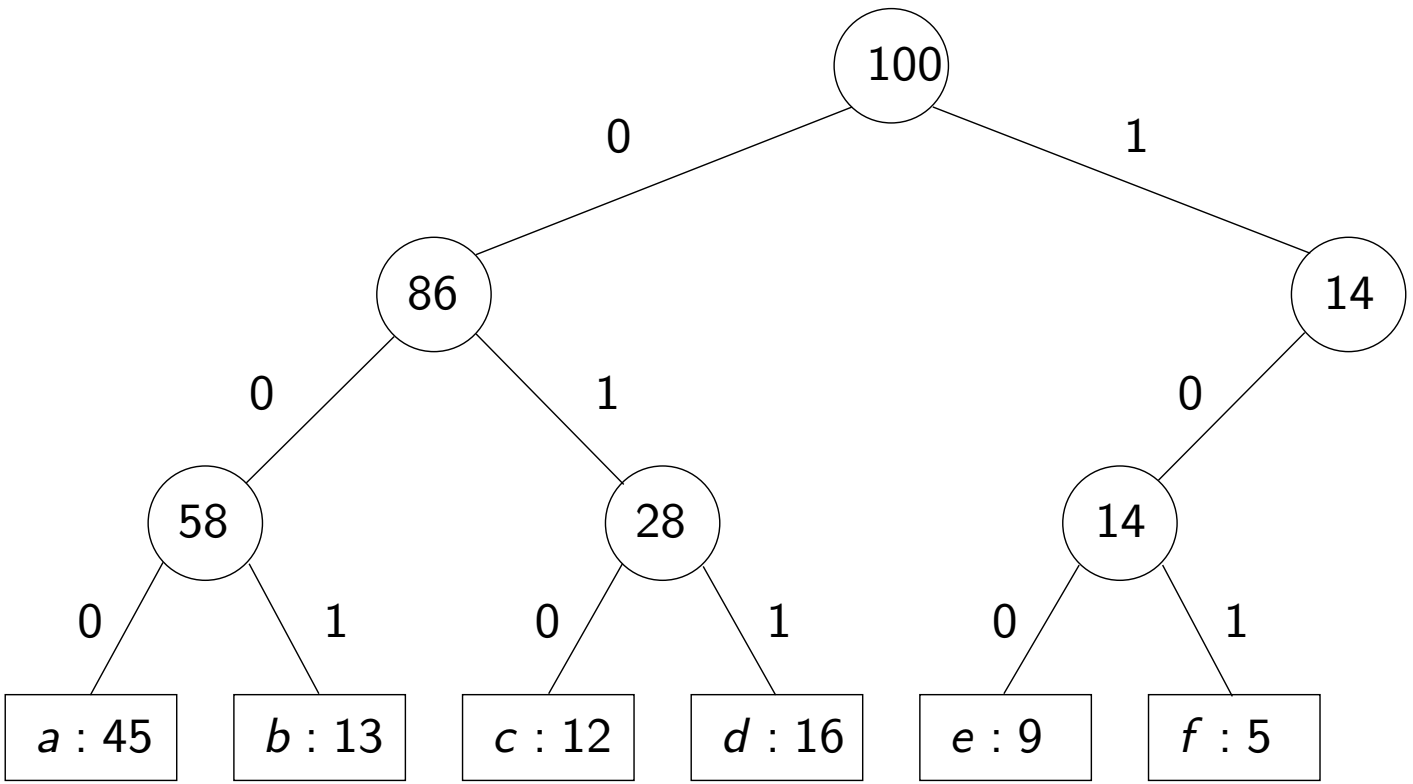
Códigos Huffman

Decodificação

- O processo de decodificação necessita de uma representação conveniente para decodificação, de maneira que o código inicial possa ser facilmente extraído.
- Uma árvore binária, em que as folhas representam os caracteres nos dá uma representação adequada.
- Interpretamos o código de um caractere como o caminho da raiz ao caractere, onde 0 significa "*ir para o filho à esquerda*", e 1 "*ir para o filho à direita*".

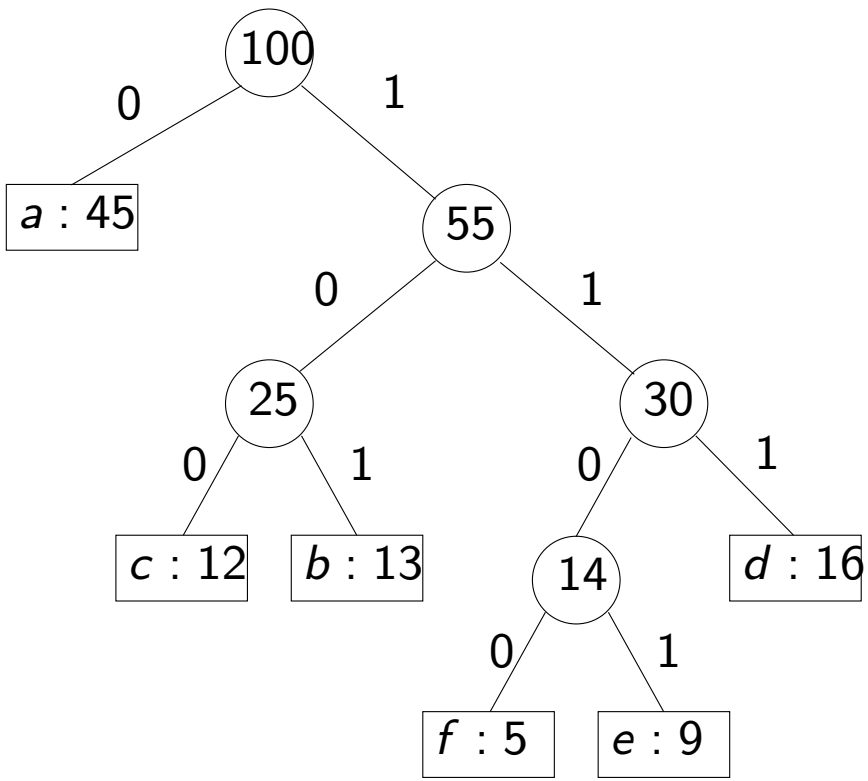
Códigos Huffman

Árvore de tamanho fixo



Códigos Huffman

Árvore de tamanho variável



Códigos Huffman

Observações

- Uma codificação ótima para um arquivo é sempre representada por uma árvore binária completa, em que cada nó que não é uma folha possui dois filhos.
- A codificação de tamanho fixo do nosso exemplo não é ótima uma vez que sua árvore não é completa.
- Restringindo nossa atenção às árvores completas, podemos dizer que se C é o alfabeto em que os caracteres são extraídos, a árvore ótima de codificação tem exatamente $|C|$ folhas, uma para cada letra do alfabeto, e exatamente $|C| - 1$ nós internos.

Códigos Huffman

- Seja T uma árvore código-prefixo.
- Para cada caractere c do alfabeto C, seja $f(c)$ uma função que denota a frequência de c no arquivo.
- Seja $d_T(c)$ a profundidade de c na árvore T.
- Note que $d_T(c)$ é também o comprimento do código para c.
- Portanto, o número de bits necessários para codificar o arquivo é:

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

que é definido como o custo da árvore T.

Códigos Huffman

Construindo o *Huffman code*

- Huffman inventou um algoritmo guloso que constrói uma árvore de prefixos ótima chamada de *Huffman tree*.
- O algoritmo constrói a árvore T correspondente ao código ótimo de maneira *bottom-up*.
- Inicia com um conjunto de $|C|$ folhas e executa uma sequência de $|C| - 1$ operações de união/intercalação (*merge*) para criar a árvore final.

Códigos Huffman

Construindo o *Huffman code*

- No pseudocódigo assumimos que C é um conjunto de n caracteres e que cada caractere $c \in C$ é um objeto com uma frequência definida $f[c]$.
- Uma fila de prioridades Q, indexada por f, é utilizada para identificar os dois objetos de menor frequência a serem intercalados.
- O resultado desta união é um novo objeto, cuja frequência é a soma das frequências dos outros dois.

Códigos Huffman

Algoritmo

Huffman(C)
  $n \leftarrow |C|$
  $Q \leftarrow C$
  for $i \leftarrow 1$ to $n - 1$
    do $z \leftarrow$ Allocate_Node()
      $x \leftarrow left[z] \leftarrow$ Extract_Min(Q)
      $y \leftarrow right[z] \leftarrow$ Extract_Min(Q)
      $f[z] \leftarrow f[x] + f[y]$
      Insert(Q, z)
  return Extract_Min(Q)

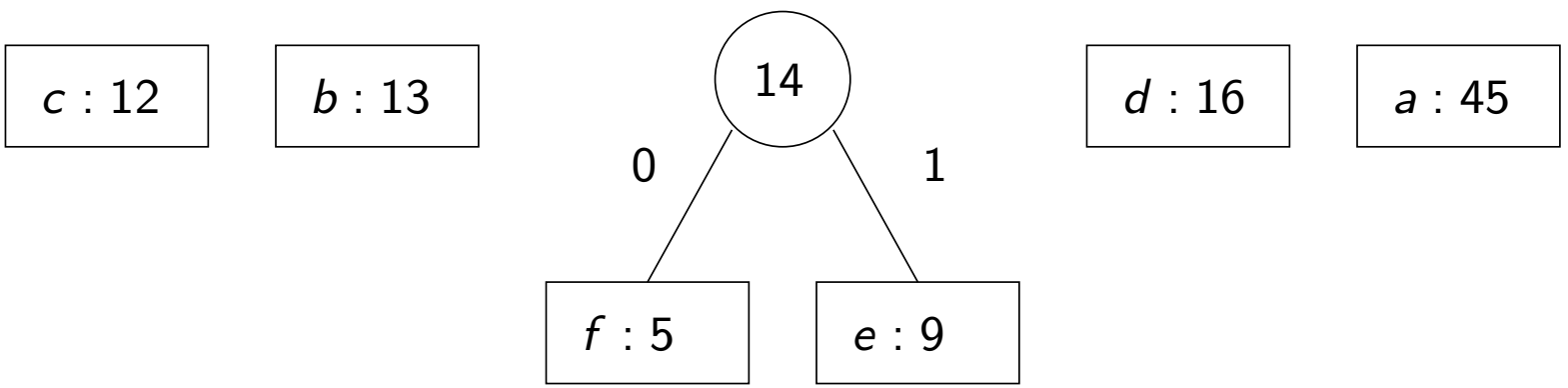
Códigos Huffman

Exemplo

$f : 5$	$e : 9$	$c : 12$	$b : 13$	$d : 16$	$a : 45$

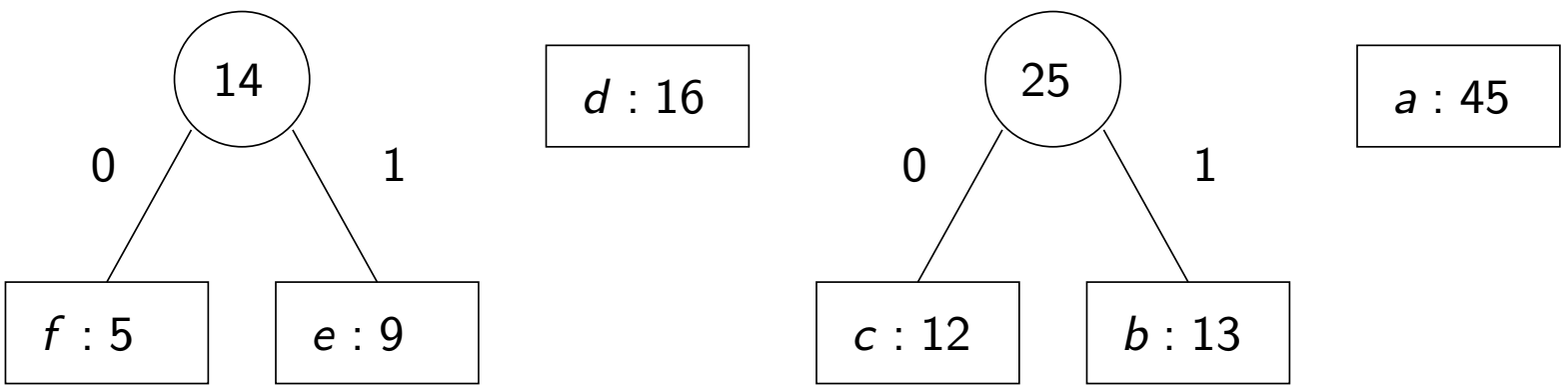
Códigos Huffman

Exemplo



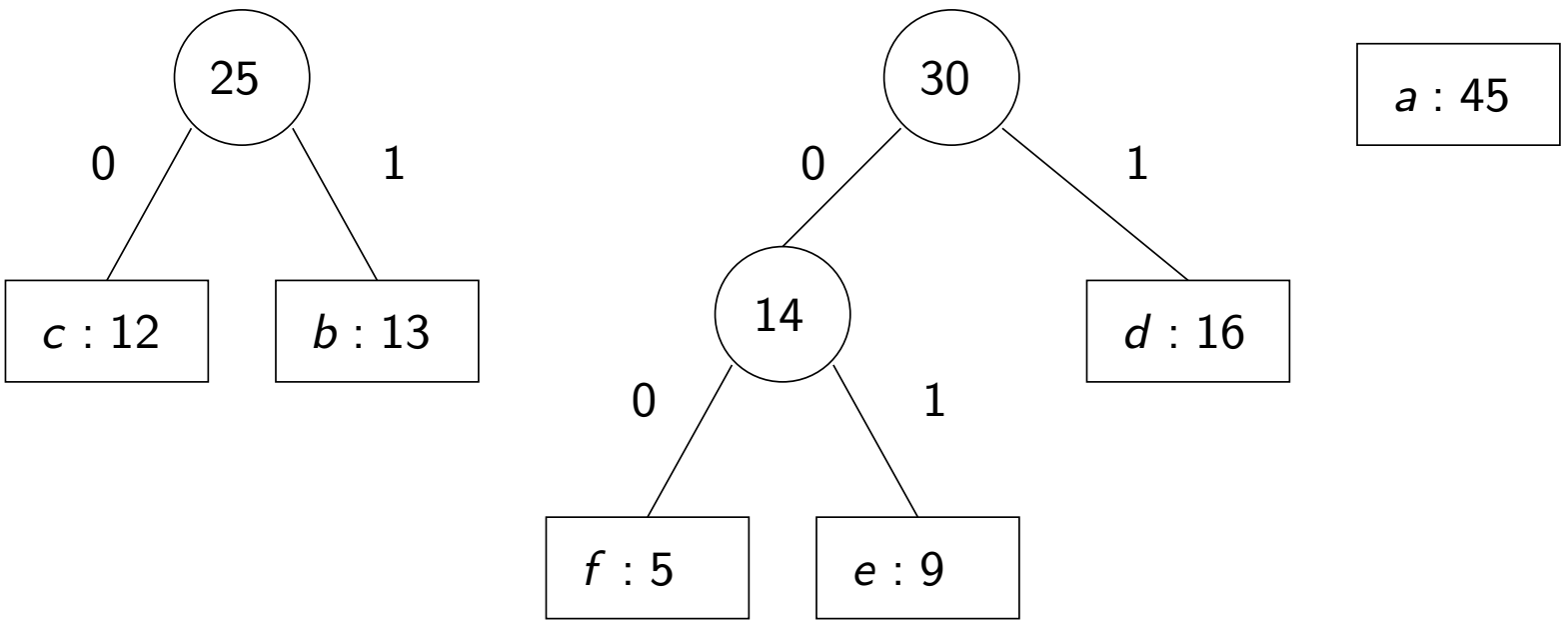
Códigos Huffman

Exemplo



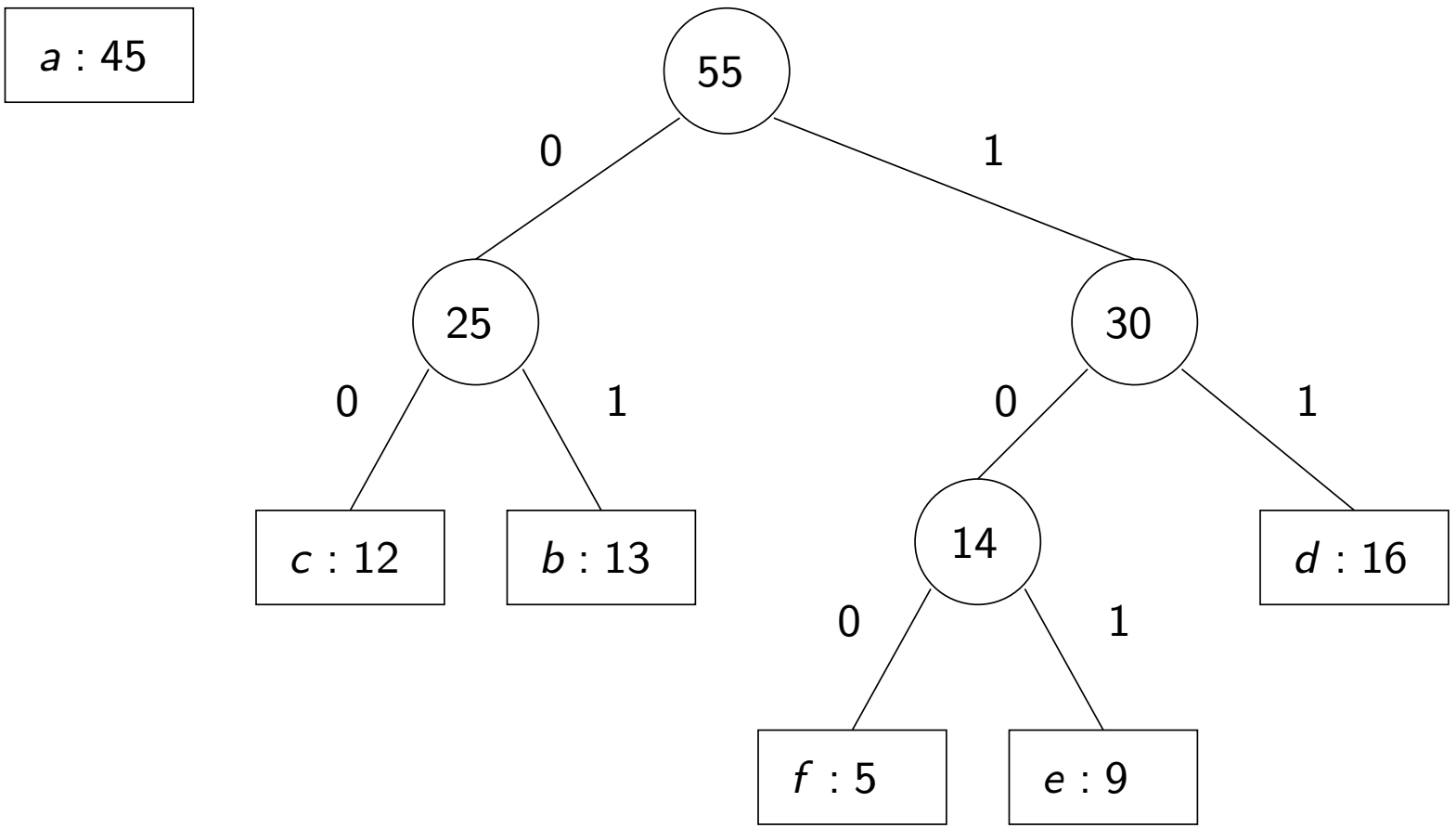
Códigos Huffman

Exemplo



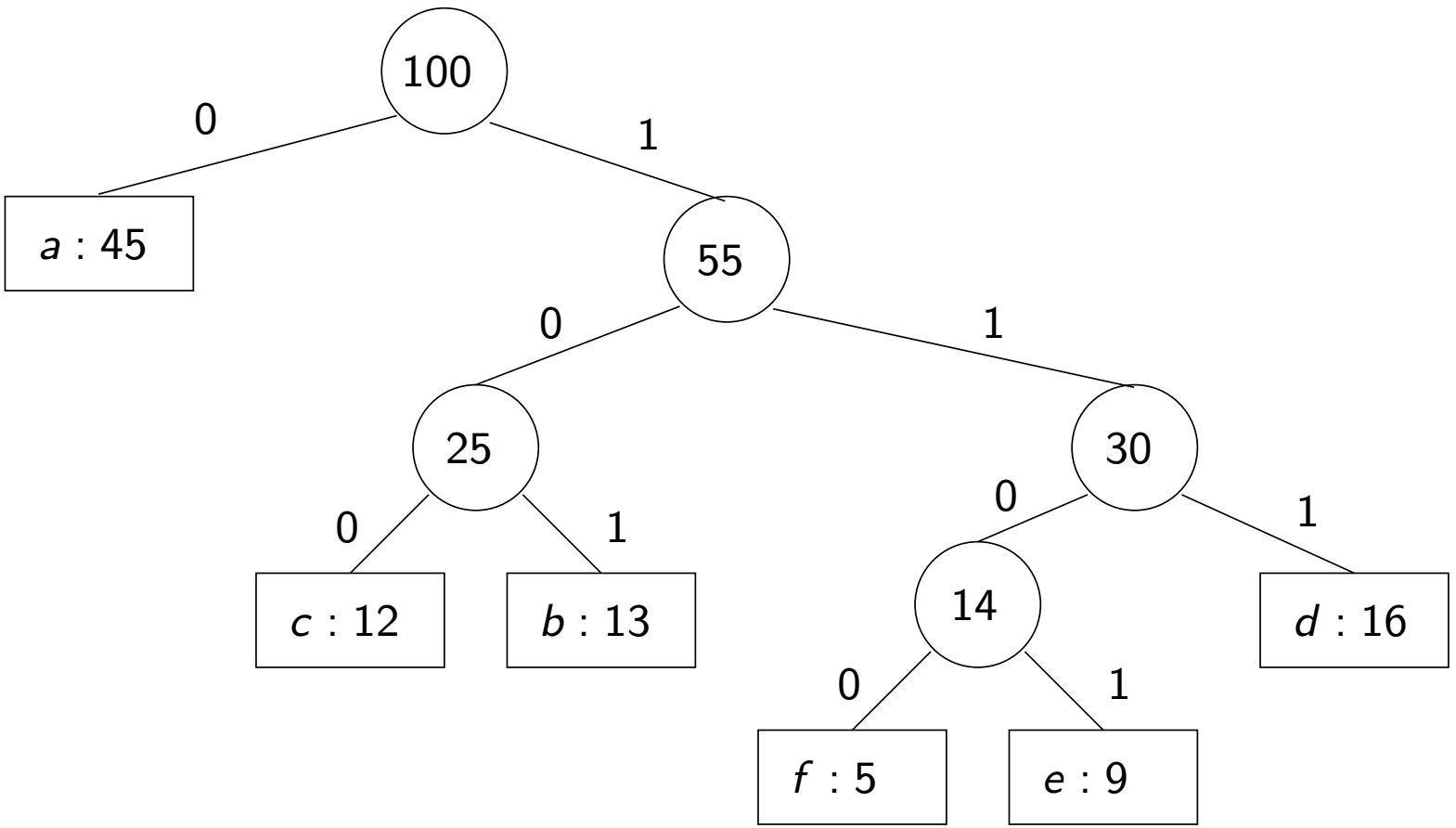
Códigos Huffman

Exemplo



Códigos Huffman

Exemplo



Códigos Huffman

Análise

- Construir Heap $O(n)$
- $(n-1)$ iterações, cada uma com $3O(\lg n)$
- Portanto, o algoritmo executa em tempo $O(n \lg n)$

Corretude do Algoritmo

Lema 2

- Seja C um alfabeto onde cada caractere $c \in C$ possui frequência $f[c]$.
- Sejam x e y dois caracteres em C de menor frequência.
- Então existe um código de prefixo ótimo para C tal que os códigos para x e y têm o mesmo comprimento e diferem apenas no último bit.

Corretude do Algoritmo

Demonstração

- A ideia da prova é pegar uma árvore T que representa um código prefixo ótimo arbitrário.
- Modificar T e obter outra árvore ótima, T', onde os caracteres x e y apareçam como folhas irmãs de profundidade máxima.
- Se conseguirmos, então seus códigos terão o mesmo comprimento e serão diferentes apenas no último bit.

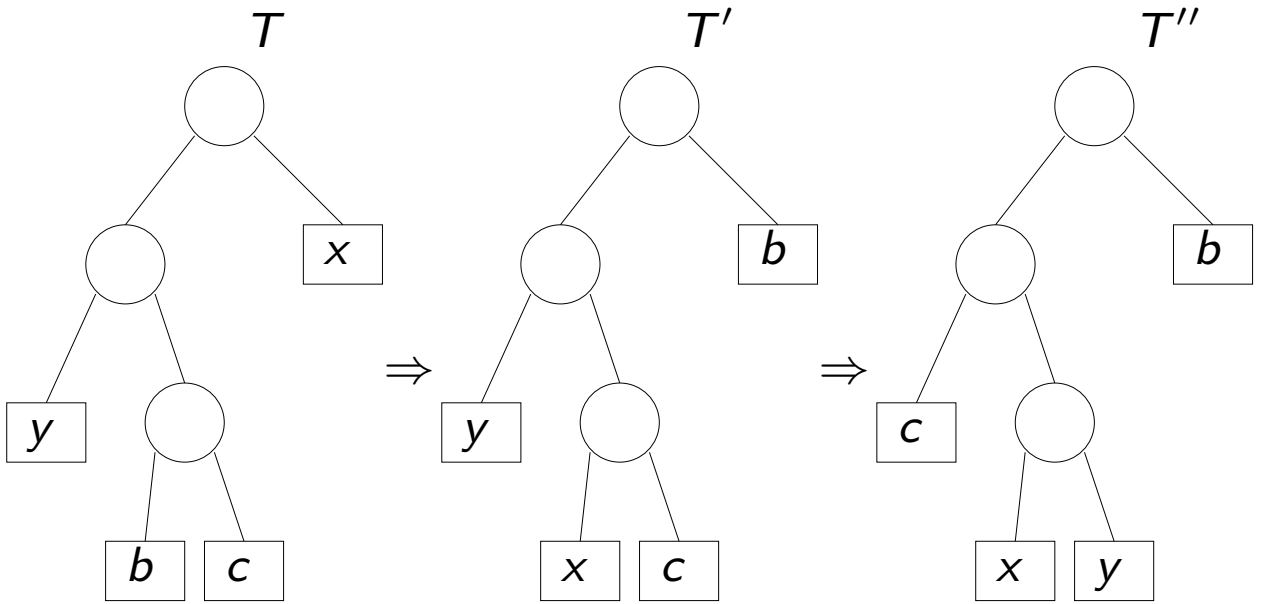
Corretude do Algoritmo

Demonstração

- Sejam b e c dois caracteres que são folhas irmãs de profundidade máxima em T.
- Sem perda de generalidade, assumimos que $f[b] \leq f[c]$ e $f[x] \leq f[y]$.
- Uma vez que $f[x]$ e $f[y]$ são as folhas de menor frequência, em ordem, e $f[b]$ e $f[c]$ são duas frequências arbitrárias, nesta ordem, temos que, $f[x] \leq f[b]$ e $f[y] \leq f[c]$.

Corretude do Algoritmo

Demonstração



trocamos x por b para produzir T', e y por c para T''.

Corretude do Algoritmo

Demonstração

$$
\begin{aligned}
\Delta_1 &= B(T) - B(T') \\
&= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_{T'}(x) - f[b] d_{T'}(b) \\
&= \big(f[x] d_T(x) - f[b] d_{T'}(b)\big) + \big(f[b] d_T(b) - f[x] d_{T'}(x)\big) \\
&= \big(f[x] d_T(x) - f[b] d_T(x)\big) + \big(f[b] d_T(b) - f[x] d_T(b)\big) \\
&= \big(f[x] - f[b]\big) d_T(x) + \big(f[b] - f[x]\big) d_T(b) \\
&= \big(f[b] - f[x]\big)\big(d_T(b) - d_T(x)\big) \\
&\geq 0
\end{aligned}
$$

Corretude do Algoritmo

Demonstração

- A diferença dos custos entre $B(T)$ e $B(T')$ foi calculada como Δ_1.
- Similarmente, como a troca de y por c não aumenta o custo, $\Delta_2 = B(T') - B(T'')$ é não negativo.
- Portanto, $B(T'') \leq B(T') \leq B(T)$, e como T é ótimo, $B(T) \leq B(T')$, que implica $B(T'') = B(T)$.
- Portanto, T'' é uma árvore ótima em que x e y aparecem como folhas irmãs de profundidade máxima, provando o lema.

Corretude do Algoritmo

Observações

O Lema 2 nos diz que a construção de uma árvore ótima com uniões pode, sem perda de generalidade, começar com uma escolha gulosa para unir os caracteres de menor frequência.

Corretude do Algoritmo

Lema 3

- Seja T uma árvore binária completa que representa um código sobre um alfabeto C, onde a frequência $f[c]$ é definida para cada caractere $c \in C$.
- Considere quaisquer dois caracteres x e y que são folhas irmãs de T e seja z o pai.
- Então, considerando z como um caractere com frequência $f[z] = f[x] + f[y]$, a árvore $T' = T - \{x, y\}$ representa um código prefixo ótimo para $C' = (C - \{x, y\}) \cup \{z\}$.

Corretude do Algoritmo

Demonstração

Primeiro mostraremos que o custo $B(T)$ da árvore T pode ser expresso em termos do custo $B(T')$ da árvore T' considerando os custos que compõem a equação:

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

Corretude do Algoritmo

Demonstração

Para cada $c \in C - \{x, y\}$, temos

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

Logo,

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

Corretude do Algoritmo

Demonstração

Uma vez que $d_T(x) = d_T(y) = d_{T'}(z) + 1$, temos

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

Assim concluímos que:

$$B(T) = B(T') + f[x] + f[y]$$

Corretude do Algoritmo

Demonstração

- Se T' representa um código prefixo não ótimo para o alfabeto C', então existe uma arvore T'' cujas folhas são caracteres em C' de maneira que $B(T'') < B(T')$.
- Uma vez que z é tratado como um caractere em C', z aparece como uma folha em T''.

Corretude do Algoritmo

Demonstração

- Se adicionarmos x e y como filhos de z em T'', então obtemos o código prefixo para C com custo

$$\begin{aligned} B(T'') + f[x] + f[y] \quad &< \quad B(T') + f[x] + f[y] = B(T) \\ &\Longrightarrow \quad B(T'') + f[x] + f[y] < B(T) \end{aligned}$$

  contradizendo a otimalidade de T.
- Portanto, T' deve ser ótimo para o alfabeto C'.

Corretude do Algoritmo

Teorema

O procedimento de Huffman produz um código prefixo ótimo.

Algoritmo Gulosos

- Fim!
- Obrigado pela presença