Construção e Análise de Algoritmos

aula 18: Códigos de prefixo

1 Introdução

Vocês sabem como o código morse foi inventado?

Bom, dizem que Samuel Morse teve a ideia natural de associar códigos curtos a letras que aparecem bastante, e códigos mais longos para letras que são pouco utilizadas.*

Ao invés de abrir um jornal e contar ele mesmo, ele foi conversar com os responsáveis pela impressão (que, naquela época, era feita com tipos) e obteve estimativas aproximadas da frequência de uso de cada letra.

Dessa maneira, ele codificou a letra `e` *com um ponto (•), a letra* `t` *com um traço (−), a letra* `a` *com um ponto-traço (•−), e assim por diante.*

Dizem também† que os operadores dos telégrafos não tiveram qualquer dificuldade em aprender a coisa, e conseguiam produzir os códigos com muita rapidez.

Mas, o diabo eram as pausas ...

Quer dizer, os operadores deviam fazer uma pausa para demarcar a passagem de uma letra para outra.

Mas, eles se atrapalhavam com elas — as vezes elas eram muito curtas, outras vezes elas eram muito longas, e às vezes eles esqueciam delas por completo.

Para piorar as coisas, se eles tivessem tomado umas no dia anterior (o que era muito comum naquela época), eles ficavam com a mão tremendo e a coisa saía totalmente do controle (mas eles não erravam um código, o que é muito surpreendente mesmo).

No outro lado da linha, o receptor advinhava o que estava acontecendo e fazia o melhor que podia para decifrar a mensagem que estava chegando.

Por exemplo, a sequência $\bullet - \bullet -$ *poderia ser um* `eta`*, um* `aa`*, um* `aet`*, etc.*

Às vezes, o receptor chamava o destinatário da mensagem para ver se o texto fazia sentido, e juntos eles tentavam descobrir as peças e como montá-las.

Mas, um dia, a mensagem de um marido para sua esposa foi decodificada errado e isso acabou dando uma tremenda confusão.

Nesse momento ficou claro que era preciso inventar um novo código ...

---

*KT, p.163

†A partir daqui a história é inventada.

A historinha da aula de hoje apresenta dois problemas:

1. o problema de inventar um código que não precisa de pausas
2. o *problema de otimização* que consiste em descobrir a melhor maneira de associar códigos às letras, de modo a minimizar o tamanho das mensagens.

Para resolver o primeiro problema, considere o seguinte código exemplo

cod(a) = 0	cod(c) = 1	cod(e) = 00	cod(g) = 10
cod(b) = 010	cod(d) = 01	cod(f) = 11	cod(h) = 011

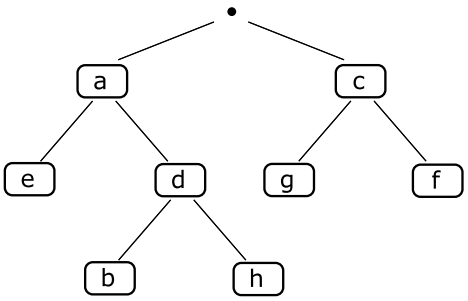
e vamos nos colocar na posição do receptor.

Quando chega um `0` pela linha ele já sabe que a próxima letra é: `a, b, d, e,` ou `h`.

Caso haja uma pausa em seguida, a letra era mesmo um `a` (provavelmente).

Se vier um `0` ela era um `e`, e se vier um 1 ela era `b, d` ou `h`.

Nós podemos imaginar que o receptor possui algo como uma árvore de decodificação na sua cabeça.



Os bits `0` fazem ele descer pela esquerda, os bits `1` fazem ele descer pela direita, e as pausas significam que o local onde ele está indica a letra que foi enviada.

Agora é fácil entender o problema das pausas.

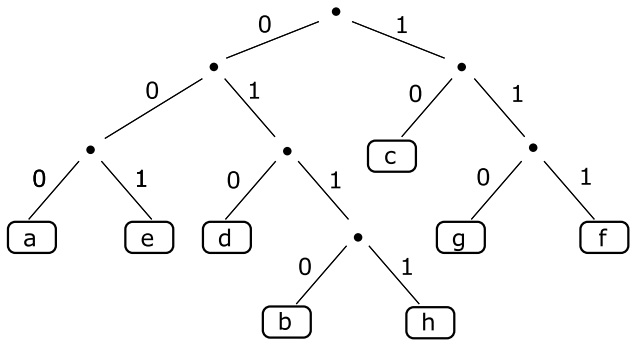
Quando se perde uma pausa e o que vem em seguida não faz sentido — por exemplo, se após um `00` chega imediatamente um `1` — isso não é um grande problema.

Mas, se perdermos a pausa após um `0` e em seguida chegar, digamos, `01`, então nós vamos interpretar como um `b` aquilo que era realmente um `ag`.

Ou seja, o problema acontece quando nós perdemos uma pausa no meio de um caminho na árvore de decodificação, que pode continuar em direção a uma outra letra.

Ora, mas se isso é o caso, então a solução é simples: basta mover todas as letras para as folhas da árvore.

Por exemplo,



Quando temos um código desse tipo, nós não precisamos mais de pausas.

(Porque?)

E isso resolve o nosso primeiro problema.

Esse tipo de código possui uma pequena peculiaridade:

- Nenhuma letra tem um código que é um *prefixo* do código de outra letra

Por esse motivo, esses códigos são conhecidos como *códigos de prefixo* (ou códigos livres de prefixo, o que é mais adequado).

De fato, não é difícil ver que todo código de prefixo possui uma árvore de docodificação com a propriedade acima (i.e., as letras estão todas nas folhas).

(Você consegue ver?)

Essa observação permite reformular o nosso segundo problema da seguinte maneira

2'. Descobrir um código de prefixo que minimiza o tamanho (médio) das mensagens codificadas

Nesse ponto, é conveniente sermos um pouco mais precisos.

Como no caso do código morse, a ideia é que nós vamos ter estimativas das frequências de uso de cada letra.

Por exemplo,

$\mathsf{freq}(\mathrm{a}) = 1/4$	$\mathsf{freq}(\mathrm{c}) = 1/8$	$\mathsf{freq}(\mathrm{e}) = 3/16$	$\mathsf{freq}(\mathrm{g}) = 1/32$
$\mathsf{freq}(\mathrm{b}) = 1/16$	$\mathsf{freq}(\mathrm{d}) = 3/16$	$\mathsf{freq}(\mathrm{f}) = 1/8$	$\mathsf{freq}(\mathrm{h}) = 1/32$

Além disso, nós observamos que o código de cada letra é dado pelo caminho que leva até ela na árvore.

Isso significa que o comprimento do código de cada letra é dado pelo comprimento do seu caminho, ou a sua altura na árvore[‡]

[‡]Em uma árvore binária, a raiz fica em cima e as folhas ficam embaixo. Logo, faz sentido que a altura aumente para baixo.

Agora suponha que nós temos um texto com 1000 caracteres para ser codificado.

De acordo com a estimativa acima, desses 1000 caracteres (aprox.) $1000 \cdot \frac{1}{8} = 125$ são a's.

Cada a é codificado com uma sequência de $\mathsf{alt}(a) = 3$ bits.

Portanto, a contribuição dos a's para o tamanho da mensagem final é de (aprox.)

$$1000 \cdot \mathsf{freq}(a) \cdot \mathsf{alt}(a) \quad = \quad 125 \cdot 3 \quad = \quad 375 \text{ bits}$$

Fazendo a mesma coisa com as outras letras, nós obtemos a seguinte expressão para o tamanho da mensagem codificada:

$$1000 \cdot \underbrace{\sum_{\text{letra l}} \mathsf{freq}(l) \cdot \mathsf{alt}(l)}_{(*)}$$

Note que (*) é a parte significativa dessa expressão (1000 é apenas o tamanho do texto).

O termo (*) corresponde ao número médio de bits por letra que o código utiliza para produzir a mensagem codificada.

Nesse sentido, ele pode ser utilizado como uma medida da eficiência do código

$$\mathsf{Ef} \quad = \quad \sum_{\text{letra l}} \mathsf{freq}(l) \cdot \mathsf{alt}(l)$$

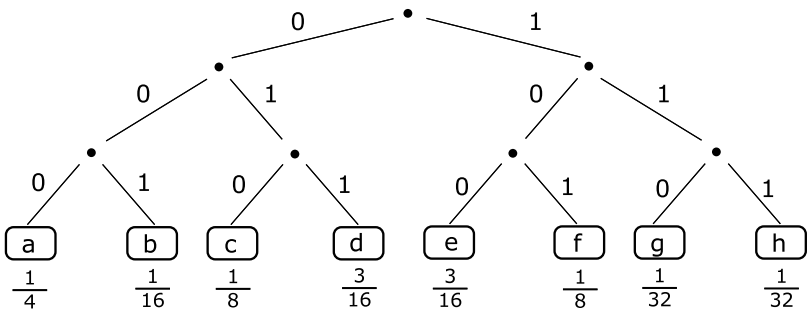
E agora nós podemos formular o nosso problema de otimização de maneira precisa:

2*. Dado um alfabeto A e as frequências de cada símbolo, descobrir um código de prefixo com o menor valor Ef possível

2 Estratégia gulosa

Antes de apresentar uma estratégia gulosa para esse problema, é útil examinar o que faz um código ser mais eficiente do que outro.

Comecemos com o seguinte código



Como se pode ver, esse código associa sequências de 3 bits a todas as letras.

Nesse sentido, ele não se aproveita do fato de que algumas letras possuem frequência maior que outras.

A sua frequência é dada por

$$\mathsf{Ef} \;=\; \sum_{\text{letra l}} \mathsf{freq}(l) \cdot \mathsf{alt}(l) \;=\; \sum_{\text{letra l}} \mathsf{freq}(l) \cdot 3 \;=\; 3 \cdot \sum_{\text{letra l}} \mathsf{freq}(l)$$

isto é, 3 bits por letra.

Agora, vejamos como esse código pode ser melhorado.

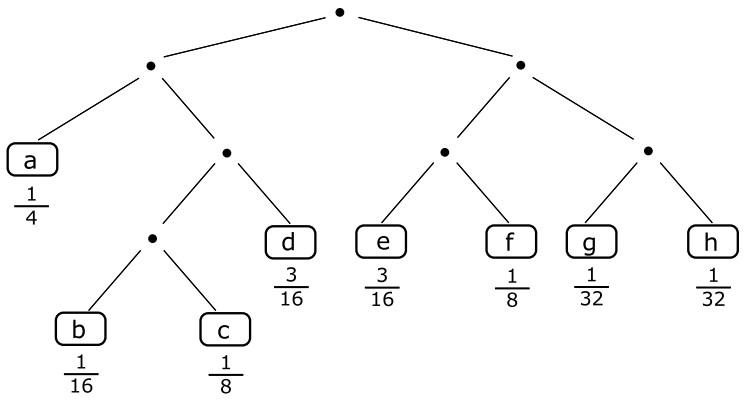
A letra que possui a maior frequência é a letra a, que possui $\mathsf{freq}(a) = 1/4$.

Faz sentido tentar reduzir o tamanho do seu código, o que corresponde a movê-la para cima na árvore.

Suponha que vamos movê-la para o nível imediatamente acima.

Isso significa que nós precisamos encontrar um outro lugar para a letra b — pois o nó onde ela está dependurada irá desaparecer.

Abaixo nós temos uma solução simples



Note que, para mover a letra a para cima foi necessário mover 2 letras para baixo.

Mas, será que isso foi uma boa ideia?

A resposta é: Sim.

A redução de um 1 bit no código de a nos dá um ganho de eficiência de

$$\underbrace{\mathsf{freq}(a) \cdot 3}_{\substack{\text{contribuição de a} \\ \text{antes da mudança}}} \;-\; \underbrace{\mathsf{freq}(a) \cdot 2}_{\substack{\text{contribuição de a} \\ \text{depois da mudança}}} \;=\; \frac{1}{4}$$

Por outro lado, aumentar os códigos de `b` e `c` em 1 bit nos dá uma perda de eficiência de

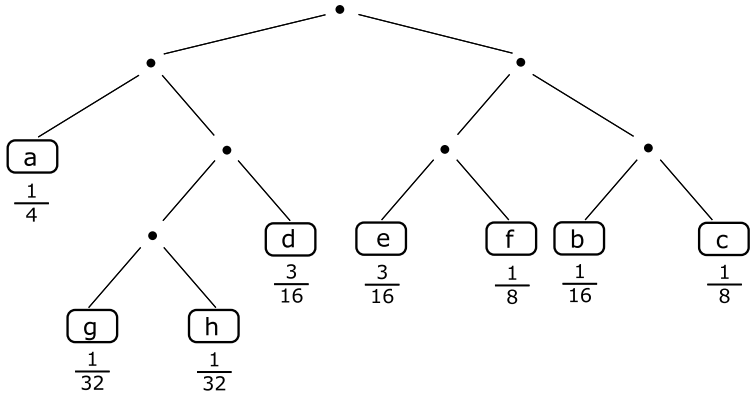
$$\Big(\mathsf{freq}(b) \ + \ \mathsf{freq}(c)\Big) \cdot 3 \quad - \quad \Big(\mathsf{freq}(b) \ + \ \mathsf{freq}(c)\Big) \cdot 4 \quad = \quad -\frac{3}{16}$$

Logo, a mudança proporciona um ganho de eficiência de $\frac{1}{4} - \frac{3}{16} = \frac{1}{16}$.

Mas, o código que nós obtivemos ainda pode ser melhorado.

Notq que as letras `b` e `c`, que possuem frequências 1/16 e 1/8 resp., estão um nível abaixo das letras `g` e `h`, que possuem ambas frequência 1/32.

Isso não faz sentido, e nós podemos trocá-las de lugar.



Essa mudança corresponde a diminuir os códigos de `b` e `c` em 1 bit, e aumentar os códigos de `g` e `h` em 1 bit, o que nos dá um ganho de eficiência de

$$\Big(\mathsf{freq}(b) \ + \ \mathsf{freq}(c)\Big) \quad - \quad \Big(\mathsf{freq}(g) \ + \ \mathsf{freq}(h)\Big) \quad = \quad \frac{1}{8}$$

O que nós aprendemos até aqui?

Bom, duas coisas:

i. é possível mover uma letra para cima na árvore, o que, em geral, requer mover mais de uma letra para baixo

ii. se uma letra de maior frequência está em um nível mais baixo do que uma letra de menor frequência, então trocá-las de lugar aumenta a eficiência do código

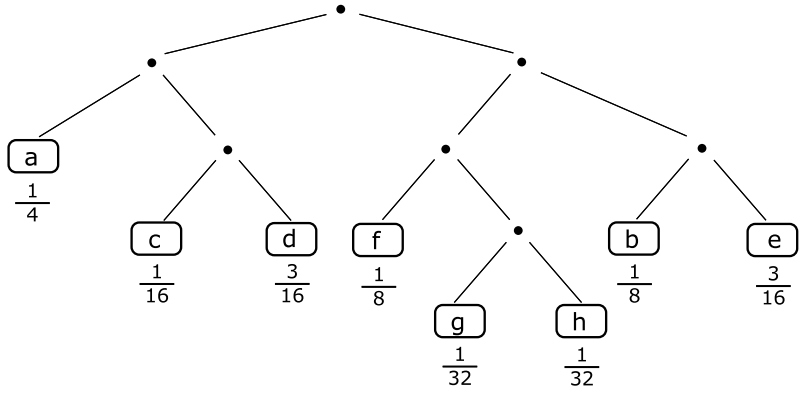
A primeira vista, não há como aplicar essas ideias novamente para melhorar o código ainda mais.

Mas, a seguir nós vamos ver uma pequena esperteza.

É fácil ver que trocar a posição de letras no mesmo nível não afeta a eficiência do código.

Da mesma maneira, trocar a posição de subárvores inteiras, sem alterar a altura das letras, também não afeta a eficiência do código.

Mas, realizando operações desse tipo, nós obtemos a seguinte árvore

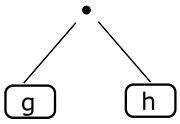


Focando a atenção no extremo direito da árvore, nos vemos a letra `e` com frequência 3/16.

Essa é uma frequência relativamente alta (quase 1/4), e faz sentido querer movê-la um pouco mais para cima — em uma operação semelhante à que fizemos com a letra `a`.

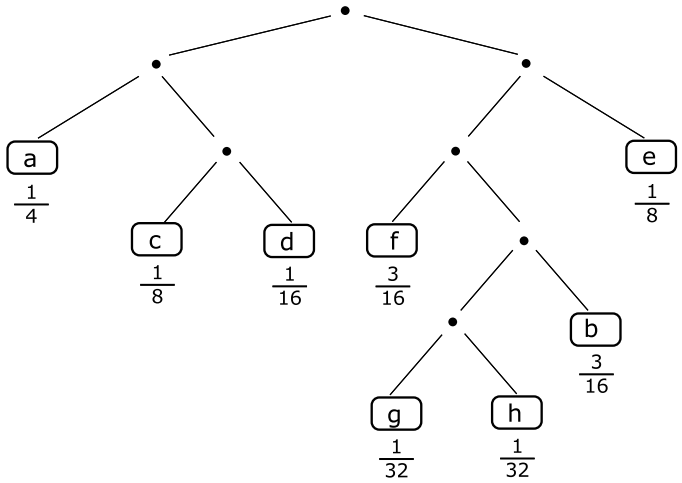
Para fazer isso, a letra `b` terá que ser movida para baixo, juntamente com mais alguém.

A ideia, dessa vez, é movê-la para baixo juntamente com toda subárvore



Isso faz sentido, pois a frequência agregada dessa subárvore é $1/32 + 1/32 = 1/16$, o que é menor do que a frequência de qualquer outra letra.

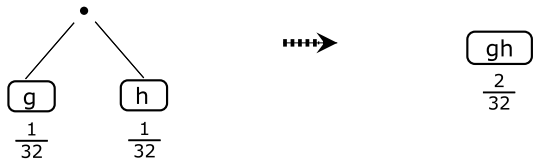
Fazendo isso, nós obtemos o código



e um ganho de eficiência de

$$\mathsf{freq}(e) \;-\; \Big(\mathsf{freq}(b) \;-\; \mathsf{freq}(g) \;-\; \mathsf{freq}(h)\Big) \;=\; \frac{1}{16}$$

A observação chave aqui é que, para todos os efeitos, nós tratamos a subárvore de `g` e `h` como um único nó



para encontrar mais uma oportunidade de melhorar o código.

Essa observação nos dá o elemento que faltava para formularmos a nossa estratégia gulosa.

Quer dizer, nós vimos que as letras são movidas para baixo aos pares (ou em subárvores), para alguém subir.

Faz sentido que esse par seja formado pelas duas letras de menor frequência.

Finalmente, quando esse par é movido para baixo, a sua subárvore pode ser tratada como uma nova letra, com a frequência agregada.

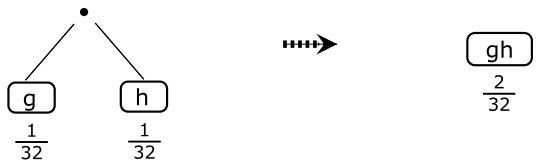
E, nesse ponto, nós temos o mesmo problema novamente — só que um pouquinho menor.

Portanto, a estratégia gulosa natural consiste em ir agrupando os pares de letras de menor frequência em subárvores, e a seguir tratar essas subárvores como uma nova letra.

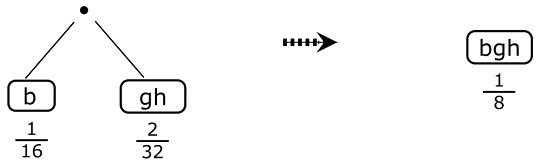
Nós continuamos dessa maneira até que só reste uma letra.

Vejamos como a coisa funciona.

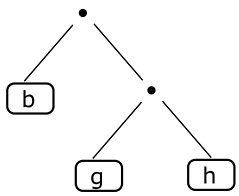
As duas letras de menor frequência no nosso exemplo são `g` e `h`, e portanto nós começamos com elas



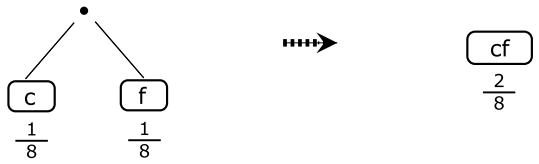
A seguir, o par de menor frequência é `b` e `gh`, o que nos dá



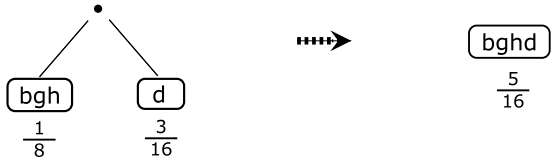
Note que a nova letra `bgh` corresponde à subárvore



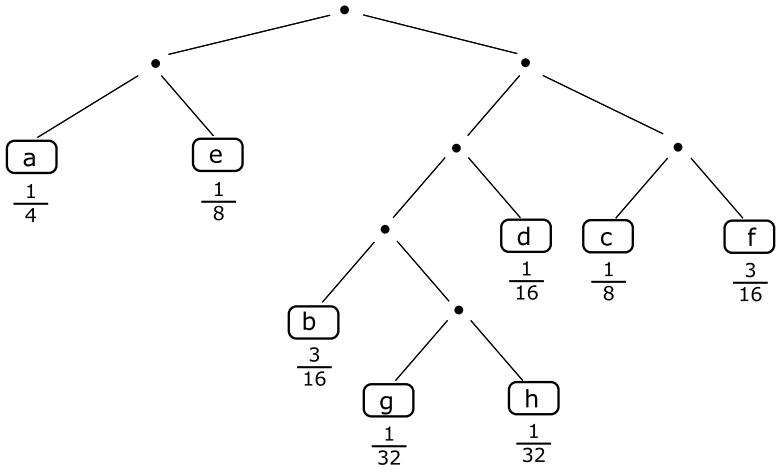
No próximo passo existem várias opções, e nós escolhemos o par `e` e `f`



A seguir, nós escolhemos o par `bgh` e `d`, para formar a subárvore



E, continuando dessa maneira, nós eventualmente produzimos a solução



Esse código tem a mesma eficiência daquele que nós encontramos um pouco mais acima.

E é possível mostrar que essa é a eficiência ótima para esse exemplo.

Mas, será que a mossa estratégia gulosa é capaz de encontrar um código de eficiência ótima em todos os casos?

3 Argumento de otimalidade

Sim.

E, para demonstrar esse fato, nós vamos construir um argumento utilizando a nossa estratégia padrão.

Isto é, nós vamos comparar a solução encontrada pelo algoritmo guloso com uma outra solução qualquer.

Denote por S a árvore solução construída pelo algoritmo, e seja $A = \{a_1, a_2, \ldots, a_n\}$ o alfabeto do problema (com as respectivas frequências).

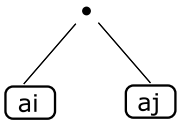
Seja T uma outra árvore solução que alguém encontrou para o problema (sabe-se lá como ...).

A única suposição que nós fazemos a respeito da árvore T é que ela não possui nenhum nó com apenas um filho

pois, nesse caso, o nó pode ser removido, melhorando a eficiência do código

Pronto, agora nós podemos iniciar o argumento.

Na sua primeira escolha, o algoritmo guloso seleciona as duas letras de menor frequência, digamos a_i e a_j, e inicia a construção do código formando a subárvore



A seguir, nós observamos que essa subárvore pode ser parte da árvore T ou não.

Suponha que não.

Nesse caso, a_i e a_j estão em locais diferentes da árvore T; cada um ao lado de uma outra letra ou subárvore

A ideia, então, é que nós podemos trocar as posições de a_j e $?_1$ (ou de a_i e $?_2$), para que a_i e a_j fiquem lado a lado

o que nos dá uma nova árvore T'.

Mas, qual é o impacto dessa operação sobre a eficiência do algoritmo?

Bom, existem dois casos.

O primeiro deles é aquele em que $a_i, a_j, ?_1, ?_2$ estão todos na mesma altura.

Nesse caso, a operação de troca não afeta a eficiência do código, e nós temos

$$\mathsf{Ef}(T') \quad = \quad \mathsf{Ef}(T)$$

No segundo caso, suponha que

$$\mathsf{alt}(a_i) \quad > \quad \mathsf{alt}(a_j)$$

Nesse caso, é importante trocar a_j com $?_1$ (e não a_i com $?_2$).

Porque?

Bom, nós sabemos que a_i e a_j são as duas letras de menor frequência no alfabeto.

Isso significa que

$$\mathsf{freq}(a_j) \quad \geq \quad \mathsf{freq}(?_1)$$

onde $\mathsf{freq}(?_1)$ pode ser uma frequência agregada.

E, nesse caso, a troca de a_j com $?_1$ (que aumenta a altura de a_j e diminui a altura de $?_1$), só pode melhorar a frequência do código:

$$\mathsf{Ef}(T') \quad \leq \quad \mathsf{Ef}(T)$$

A troca entre a_i e $?_2$, por outro lado, corresponde à situação inversa, e ela pode piorar a eficiência do código.

Portanto, em ambos os casos, nós temos que

$$\mathsf{Ef}(T') \quad \leq \quad \mathsf{Ef}(T)$$

A seguir, nós vamos continuar trabalhando com a árvore T'.

Para analisar a segunda escolha do algoritmo guloso, nós podemos fazer uma pequena simplificação.

Nós podemos imaginar que, tanto em S como em T', a subárvore de a_i e a_j é compactada em um único nó:

4 Algoritmo e análise de complexidade

Agora que sabemos que a nossa estratégia gulosa é ótima, só nos resta implementá-la na forma de um algoritmo e analisar o seu tempo de execução.

```
 Procedimento  Cod-Pref-Gul ( A: alfabeto de tamanho n com frequências )
  {
1.   Associar a cada letra  x  de  A  um nó folha

2.   Inserir todas as letras de  A  em um heap  H  (indexado p/ frequência)

3.   Enquanto ( H possui ao menos 2 elementos )
      {
4.       ai,aj  <--  duas letras de  H  com a menor frequência

5.           x  <--  nova letra c/ frequência  freq(ai) + freq(aj)

6.       Associar à letra  x  uma árvore construída com as árvores de ai e aj

7.       Inserir a nova letra  x  no heap  H
      }

8.   z  <--  última letra do heap  H

9.   Retornar a árvore associada à letra  z
  }
```

Análise de complexidade

O tempo de execução desse algoritmo é dominado pelas operações sobre o heap H.

Note que, no início, todas as letras do alfabeto são colocadas no heap.

E que, a cada volta do laço, o tamanho do heap diminui de 1.

Logo, o tamanho do heap é sempre menor ou igual a n.

E isso significa que as operações sobre ele levam tempo $O(\log n)$.

Agora, é fácil ver que

- a linha 1 executa em tempo $O(n)$
- a linha 2 executa em tempo $O(n \log n)$
- o bloco de instruções no interior do laço executa em tempo

$$O(\log n) \;+\; O(1) \;+\; O(1) \;+\; O(\log n) \quad = \quad O(\log n)$$

- o laço realiza $O(n)$ voltas

Portanto, o tempo de execução do algoritmo é $O(n \log n)$.