Análise e Técnicas de Algoritmos
Período 2003.2

Problemas de Otimização e Estratégia Gulosa

Problemas de Otimização

- Problemas que podem apresentar diversas soluções.
- A solução é uma sequência de decisões.
- Um valor pode ser associado para cada solução.
- Achar a solução com valor ótimo.

Exemplo 1: Cálculo do trôco

- **Descrição:** Seja $E = \{e_1, e_2, ..., e_n\}, e_1 \rangle e_2 \rangle ... \rangle e_n$, um conjunto de n denominações de moedas (ou cédulas), e M um valor positivo que representa o trôco.
- **Problema:** Fornecer o montante M com o número mínimo de moedas.
- **Sequência de Decisões:** Escolher r_1, depois r_2, ...

Exemplo 2: Problema da Mochila (Knapsack)

- **Descrição:** Temos n objetos com pesos $w_1, w_2, ..., w_n$ e uma mochila de capacidade M. Se uma fração $x_i (0 \leq x_i \leq 1)$ do objeto i for colocada na mochila, um lucro $p_i x_i$ resulta.
- **Problema:** Maximizar o lucro que pode ser levado na mochila.
- **Sequência de Decisões:** Escolher primeiro objeto, escolher o segundo objeto, ...

Exemplo 3: Intercalação de Arquivos Ordenados

- **Descrição:** Dados n arquivos, onde o arquivo i contém m_i registros. Intercalar os n arquivos.
- **Problema:** Achar uma sequência de intercalação de pares de arquivos de forma a obter um único arquivo ordenado em tempo mínimo.
- **Sequência de Decisões:** Qual o primeiro par a intercalar, qual o segundo par, ...

Exemplo 4: Escalonamento de Tarefas

- **Descrição:** Seja E um conjunto de n tarefas. Associamos à cada tarefa um tempo de execução. Fazer o escalonamento dessas tarefas.
- **Problema:** Minimizar o tempo médio de finalização das tarefas.
- **Sequência de Decisões:** Escolher primeira tarefa, escolher a segunda tarefa, ...

Exemplo 5: Caixeiro Viajante

- **Descrição:** Seja $G = (V, E)$ um grafo direcionado com custo c_{ij} para os arcos. Seja n o número de vértices e v_0 o vértice de origem.
- **Problema:** Achar uma tournê de G de custo mínimo, onde uma tournê é um ciclo direcionado que inclua todos os vértices de V.
- **Sequência de Decisões:** A partir da origem v_0, qual é o primeiro vértice do ciclo, qual é o segundo vértice do ciclo, ...

O Método Guloso (*Greedy*)

- Problema solucionado em diferentes passos.
- As decisões são tomadas de forma isolada, em cada passo.
- Estratégia de *pegar o melhor no momento* (solução ótima local).
- Quando o algoritmo termina, espera-se que tenha ocorrido a melhor solução.
- Com este tipo de algoritmo, podemos achar uma solução ótima para alguns problemas, mas não para todos.

Exemplos

Cálculo do Trôco

- Sejam $E = \{100, 50, 10, 5, 1\}$ e M o montante total.
- **Algoritmo Greedy:** No passo i, escolher $r_i = j$, tal que $e_j \leq M$ e $e_{j-1} > M$ e subtrair e_j de M para o próximo passo.
- Podemos provar que, para este conjunto de moedas E, o algoritmo greedy fornece uma solução ótima.
- Para $E = \{300, 250, 100, 1\}$, **não** funciona (exemplo: $M = 500$).

Alocação de Tarefas (1)

- Seja um conjunto de tarefas $J = j_1, j_2, ..., j_n$ com tempos de execução $t_1, t_2, ..., t_n$, respectivamente.
- Considerar um único processador e alocação não-preemptiva.
- Qual a melhor forma de alocar essas tarefas para minimizar o tempo médio de execução?
- Solução 1: ordem de chegada.

- Solução 2: ordem crescente do tempo de execução.
- Usar $J = \{(j_1, 15), (j_2, 8), (j_3, 3), (j_4, 10)\}$ para ilustrar.
- A solução 2 sempre apresenta solução ótima:
  - Vamos supor que temos a seguinte sequência como solução. $j_{i_1}, j_{i_2}, ..., j_{i_n}$.
  - $C = t_{i_1} + (t_{i_1} + t_{i_2}) + ... + (t_{i_1} + t_{i_2} + ... + t_{i_n})$.
  - $C = \sum_{k=1}^{n} (n - k + 1) t_{i_k}$.
  - $C = (n + 1). \sum_{k=1}^{n} t_{i_k} - \sum_{k=1}^{n} k.t_{i_k}$.
  - Vamos supor que exista uma ordenação em que $x > y$ e $t_{i_x} < t_{i_y}$.
  - Se fizermos a troca, o custo total vai decrescer.

Alocação de Tarefas (2)

- Mesma definição do problema anterior.
- Diferença: considera P processadores.
- Assumir que os tempos estão em ordem crescente de tempo de execução.
- Exemplo: $P = 3$ e $J = \{(j_1, 3), (j_2, 5), (j_3, 6),$ $(j_4, 10), (j_5, 11), (j_6, 14), (j_7, 15), (j_8, 18), (j_9, 20)\}$.
- Solução: começar a alocação em ordem, fazendo um rodízio entre os processadores.
- Outros arranjos podem ser efetuados
- Prova é semelhante.

Alocação de Tarefas (3)

- Mesma definição do problema anterior.
- O objetivo é minimizar o tempo final, ou seja, o tempo em que as tarefas terminam.
- No problema anterior foi 40 e 38.
- Solução: $\{(P_1, j_2, j_5, j_8), (P_2, j_6, j_9), (P_3, j_3, j_4, j_7)\}$.
- O tempo médio não é mínimo mas, a sequência inteira termina mais cedo.
- Esse problema é mais complicado de se resolver.

Alocação de Tarefas (4)

- Seja $S = \{1, 2, ..., n\}$ um conjunto de n atividades que desejam usar um mesmo recurso.
- Cada atividade i tem um tempo de começo (s_i) e um tempo de término (f_i), $s_i \leq f_i$.
- Duas atividades i e j são compatíveis se elas não se sobrepõem, ou seja, se $s_i \geq f_j$ ou $s_j \geq f_i$.
- Assumir que as atividades estão em ordem crescente do tempo de término.

AlgoritmoGreedy (s, f)

$n \leftarrow length(s)$
$A \leftarrow \{1\}$
$j \leftarrow 1$
for i = 2 to n **do**
  if $s_i \geq f_j$ **then**
    $A = A \cup \{i\}$
    $j = i$
return A

- Arrays para representar s e f.
- Prova de que está correto: mostrar que existe uma solução ótima que começa com a atividade 1.

Elementos do Método Guloso

- Um algoritmo guloso obtém uma solução ótima para um problema, fazendo uma sequência de escolhas.
- Nem sempre produz uma solução ótima.
- Não existe uma regra geral que indique que um algoritmo guloso resolva um determinado problema de otimização.
- Bom indício é dado por 2 elementos:
  - Propriedade de escolha gulosa.
  - Sub-estrutura ótima.

Mais Exemplos

Problema da Mochila Fracionária

Considere n objetos com pesos $w_1, w_2, ..., w_n$ e uma mochila de capacidade M. Se uma fração $x_i (0 \leq x_i \leq 1)$ do objeto i for colocada na mochila, um lucro de $p_i x_i$ é conseguido. Como maximizar o lucro.

Vamos usar o seguinte exemplo:

- $n = 3$, $M = 20$.
- $P = (p_1, p_2, p_3) = (25, 24, 15)$ e $W = (w_1, w_2, w_3) = (18, 15, 10)$.

Solução 1: Maximizar o lucro a cada passo

Escolher o objeto com p_i máximo e colocar na mochila ($x_i = 1$). No último passo, quando a mochila está quase cheia, escolher o objeto j, tal que $x_j w_j$ complete a capacidade da mochila e $x_j p_j$ seja máximo.

1. Escolher o de maior lucro ($p_1 = 25, x_1 = 1$). O lucro até então é 25 e a capcidade restante da mochila é 2.
2. O objeto 2 não cabe por inteiro. Usar $x_2 = 2/15$ que dá um lucro de 3,2.
3. A solução $(1, 2/15, 0)$ permite um lucro de 28,2.

Solução 2: Conserva a capacidade

Escolher os objetos em ordem crescente de peso (w_i), maximizando a capacidade restante.

1. $x_3 = 1, p_3 x_3 = 15$. A capacidade restante é 10.
2. $x_2 = 2/3, p_2 x_2 = 16$. Não é possível mais selecionar objetos.
3. A solução $(0, 2/3, 1)$ permite um lucro de 31.

Solução 3: Maximiza o lucro por unidade de peso

Escolher os objetos em ordem decrescente de p_i/w_i.

1. O objeto que apresenta maior lucro por unidade de peso é o 2, seguido do 3 e do 1.
2. Logo: $x_2 = 1, p_2 x_2 = 24$, capacidade restante é 5.
3. $x_3 = 1/2, p_3 x_3 = 7,5$, capacidade restante é 0.
4. A solução $(0, 1, 1/2)$ permite um lucro de 31,5.

O Algoritmo

Primeiro ordenar os objetos em ordem decrescente p_i/w_i.

$cap = M$
$i = 1$

while $w_i \leq cap$ **do**
  $x_i = 1$
  $cap = cap - w_i$
  $i = i + 1$
$x_i = cap/w_i$
for $j = i + 1 ton$ **do**
  $x_j = 0$

É possível provar que o terceiro algoritmo sempre fornece a solução ótima para este problema da mochila.

Prova do Terceiro Algoritmo

O que devemos provar é que a solução fornecida pelo terceiro algoritmo corresponde à solução com o maior lucro.

É importante, entretanto, ressaltar que pode existir várias soluções ótimas.

- Vamos supor que $X = (x_1, x_2, ..., x_n)$ é a solução fornecida pelo algoritmo.

Se, $\forall i, 1 \leq i \leq n, x_i = 1$, essa solução é claramente a ótima. Nesse caso, seria a única solução.

Se não for este o caso, vamos supor que j é o menor número onde $x_j \neq 1$. Do algoritmo temos:

$x_i = 1, \forall i, 1 \leq i \leq j$

$0 \leq x_j < 1$

$x_i = 0, \forall j, j < i \leq n$

Logo, $\Sigma_{i=1}^{j} x_i w_i = M$.

- Seja $Y = (y_1, y_2, ..., y_n)$ uma solução de máximo lucro.

Temos que provar que X tem o mesmo lucro de Y.

Se $X = Y$, não tem o que fazer.

Caso contrário, vamos considerar que k é o menor número onde $x_k \neq y_k$.

Podemos então dizer que:

$Y = x_1 x_2 ... x_{k-1} y_k y_{k+1} ... y_n$

$X = x_1 x_2 ... x_{k-1} x_k x_{k+1} ... x_n$

A estratégia da prova consiste em transformar Y em X, mantendo o lucro. Para tanto, vamos transformar Y em uma solução mais intermediária Z e que se parece mais com X, ou seja,

$Z = x_1 x_2 ... x_{k-1} x_k z_{k+1} ... z_n$

Antes de prosseguirmos com a prova, vamos fazer algumas considerações. vamos investigar os seguintes casos:
Caso 1: $k < j$.

Nesse caso, $x_k = 1$. Portanto, y_k deve ser menor do que x_k pois $x_k \neq y_k$.
Caso 2: $k = j$.

Pela definição de k, $x_k \neq y_k$. Se $y_k > x_k$,

$$\begin{aligned}
M &= \Sigma_{i=1}^{n} y_i w_i \\
&= \Sigma_{i=1}^{k-1} y_i w_i + y_k w_k + \Sigma_{i=k+1}^{n} y_i w_i \\
&= \Sigma_{i=1}^{k-1} x_i w_i + y_k w_k + \Sigma_{i=k+1}^{n} y_i w_i \\
&= \Sigma_{i=1}^{k} x_i w_i + (y_k - x_k) w_k + \Sigma_{i=k+1}^{n} y_i w_i \\
&= \Sigma_{i=1}^{j} x_i w_i + (y_k - x_k) w_k + \Sigma_{i=k+1}^{n} y_i w_i \\
&= M + (y_k - x_k) w_k + \Sigma_{i=k+1}^{n} y_i w_i \\
&> M
\end{aligned}$$

Isso contradiz o fato de que Y é uma solução. Logo, $y_k < x_k$.
Caso 3: $k > j$.

Nesse caso, $x_k = 0$ e $y_k > 0$:

$$\begin{aligned}
M &= \Sigma_{i=1}^{n} y_i w_i \\
&= \Sigma_{i=1}^{j} y_i w_i + \Sigma_{i=j+1}^{n} y_i w_i \\
&= \Sigma_{i=1}^{j} x_i w_i + \Sigma_{i=j+1}^{n} y_i w_i \\
&= M + \Sigma_{i=j+1}^{n} y_i w_i \\
&> M
\end{aligned}$$

Também não é possível, logo o caso 3 nunca acontece.

Portanto, podemos assumir que $y_k < x_k$.

Voltando à prova, para sair de Y para Z temos que aumentar y_k fazendo igual a x_k. e diminuir $y_{k+1}, ..., y_n$ como necessário, para fazer com que o lucro se mantenha. Vamos dizer que a nova solução é $Z = (z_1, z_2, ..., z_n)$.

Portanto,

1. $(z_k - y_k) w_k > 0$
2. $\Sigma_{i=k+1}^{n} (z_i - y_i) w_i < 0$
3. $(z_k - y_k) w_k + \Sigma_{i=k+1}^{n} (z_i - y_i) w_i = 0$

Então,

$$\begin{aligned}
&\Sigma_{i=1}^{n} z_i p_i \\
&= \Sigma_{i=1}^{k-1} z_i p_i + z_k p_k + \Sigma_{i=k+1}^{n} z_i p_i \\
&= \Sigma_{i=1}^{k-1} y_i p_i + z_k p_k + \Sigma_{i=k+1}^{n} z_i p_i \\
&= \Sigma_{i=1}^{n} y_i p_i - y_k p_k - \Sigma_{i=k+1}^{n} y_i p_i + z_k p_k + \Sigma_{i=k+1}^{n} z_i p_i \\
&= \Sigma_{i=1}^{n} y_i p_i + (z_k - y_k) p_k + \Sigma_{i=k+1}^{n} (z_i - y_i) p_i
\end{aligned}$$

$= \Sigma_{i=1}^{n} y_i p_i + (z_k - y_k) w_k p_k / w_k + \Sigma_{i=k+1}^{n} (z_i - y_i) w_i p_i / w_i$
$\leq \Sigma_{i=1}^{n} y_i p_i + (z_k - y_k) w_k p_k / w_k + \Sigma_{i=k+1}^{n} (z_i - y_i) w_i p_k / w_k$
$= \Sigma_{i=1}^{n} y_i p_i + ((z_k - y_k) w_k + \Sigma_{i=k+1}^{n} (z_i - y_i) w_i) p_k / w_k$
$= \Sigma_{i=1}^{n} y_i p_i$

Y e Z têm o mesmo lucro. Z, entretanto, se parece mais com X pois as k primeiras entradas de Z são as mesmas de X.

O procedimento pode ser repetido até transformar Y em X. Logo X também é uma solução ótima.

Códigos de Huffman

Nessa seção vamos considerar uma aplicação de algoritmos gulosos, conhecida como *compressão de arquivos*.

- Vamos considerar um arquivo de 100.000 caracteres, onde apenas os caracteres a, e, i, s, t, b e n ocorrem com freqüência de 15000, 25000, 22000, 7000, 5000, 23000, 3000.
- Usando um código para representar cada caracter:
  1. Se usarmos ASCII Extendido (8 bits), o número total de bits é 800.000.
  2. Se usarmos um código de tamanho fixo, 3 bits para representar 7 caracteres, o número total de bits é 300.000.
  3. Se usarmos um código de tamanho variável: $a = 001$, $e = 01$, $i = 10$, $s = 00000$, $t = 0001$, $b = 11$ e $n = 00001$, o número total de bits é 255.000.
- É possível reduzir 15% da solução 2 e 68% da solução 1.

A redução no número de bits foi possível pois a estratégia utilizada considerou um código de tamanho variável, onde os caracteres com freqüência maior eram codificados com um número menor de bits. Logo, se todos os caracteres ocorrem com a mesma freqüência, é possível que não ocorra nenhuma redução no número de bits utilizados.

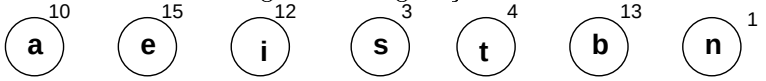
Códigos de Huffman

- Utilizam códigos prefixos:
  - Códigos onde nenhum código de caracter é prefixo de outro.
  - Sempre é possível achar um código ótimo de compressão.
  - Codificação e decodificação são fáceis.
- Utilização de uma árvore binária, com folhas representando os caracteres. 0 significa esquerda e 1 direita.

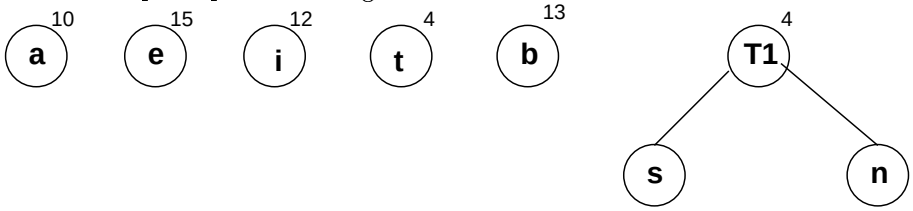
O Algoritmo de Huffman

- Vamos assumir que o número de caracteres é C.
- Manter uma floresta de árvores.
- O peso de uma árvore é a soma das freqüências de suas folhas.
- $C - 1$ vezes, selecionar as duas árvores de menor peso e formar uma nova árvore.
- No início do algoritmo, existem C árvoes de apenas um nó.
- No final temos apenas uma única árvore e á a árvore com o código de Huffman.

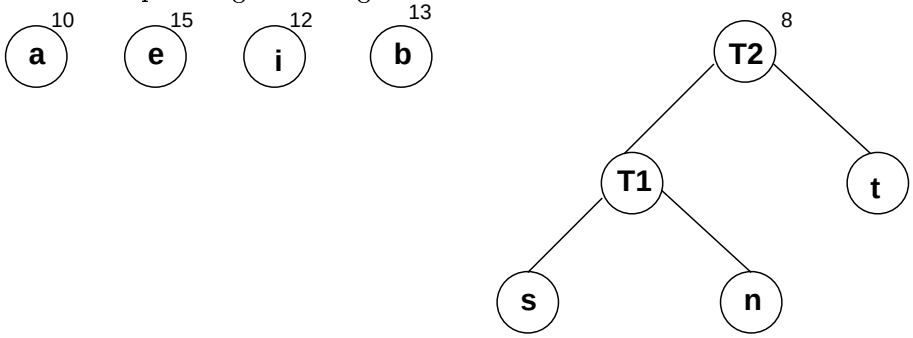
Vamos considerar a seguinte configuração inicial:



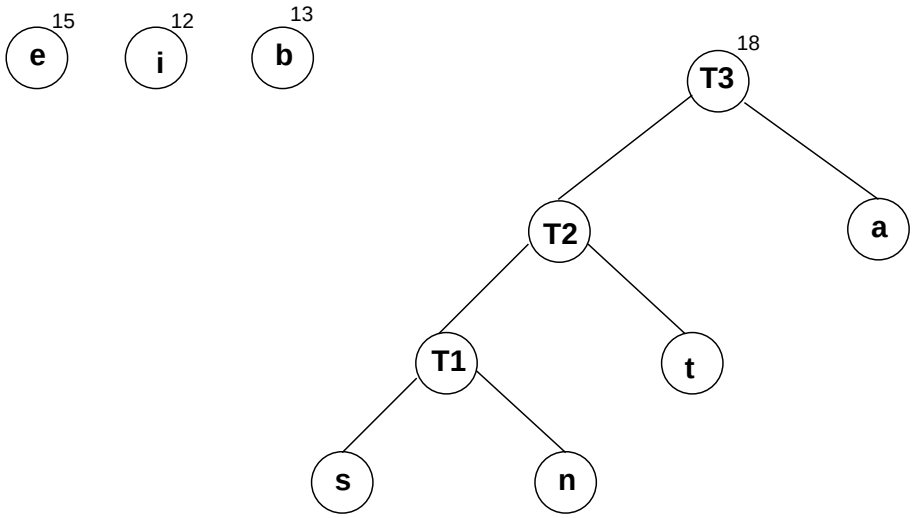
Resultado após o primeiro merge.



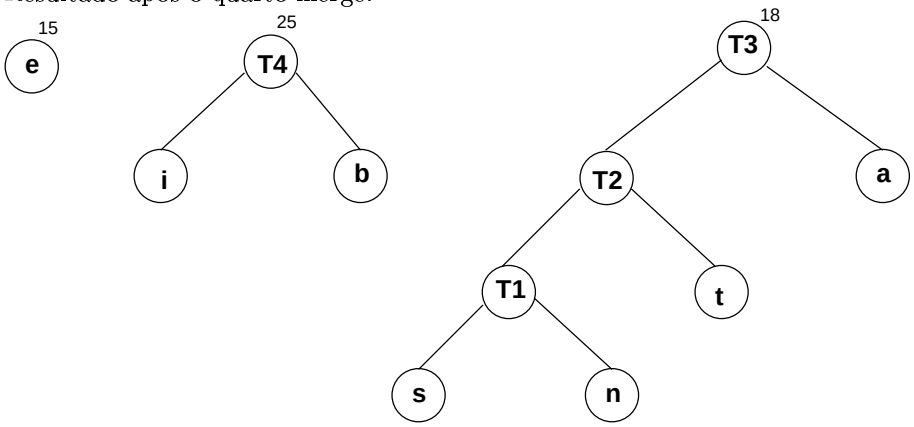
Resultado após o segundo merge.



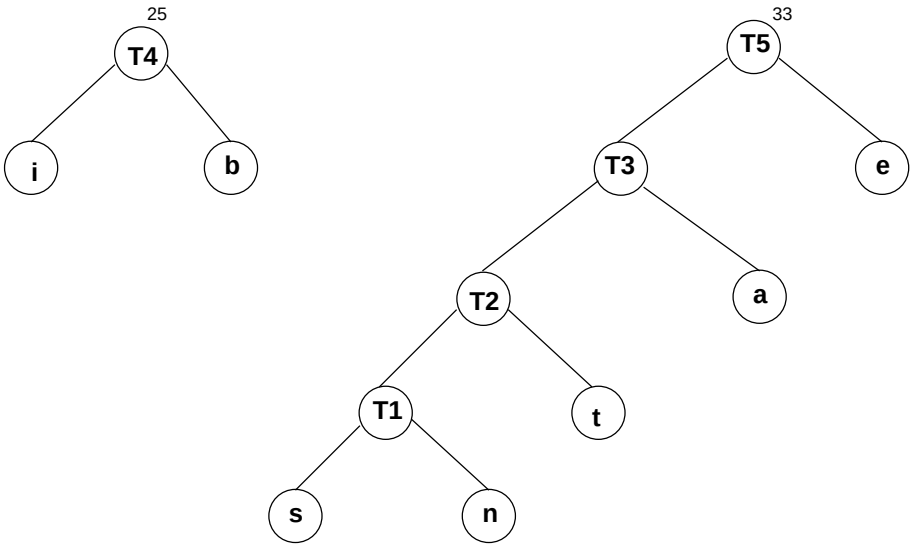
Resultado após o terceiro merge.



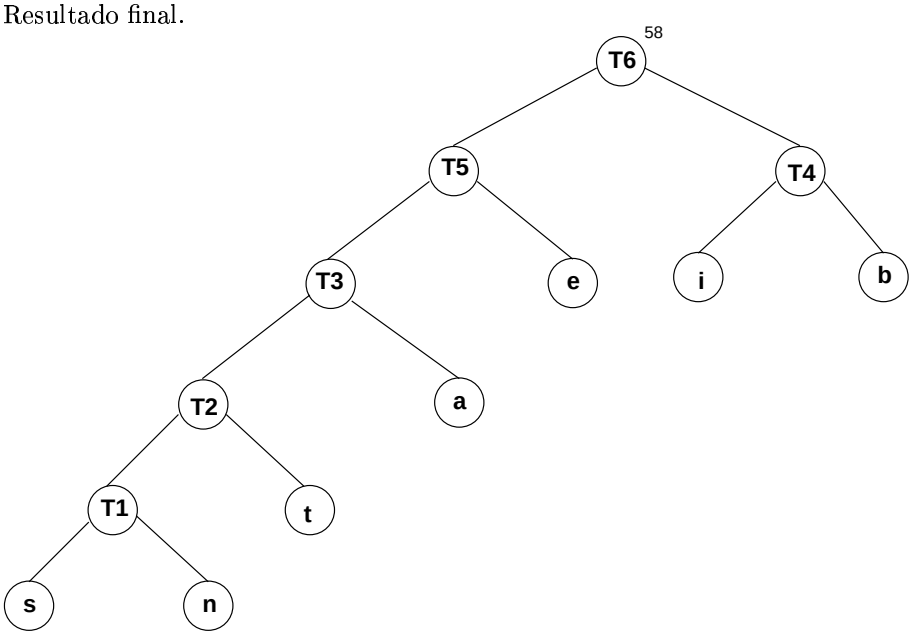
Resultado após o quarto merge.



Resultado após o quinto merge.



Resultado final.



O código prefixo ótimo é:

- $a = 001$
- $e = 01$
- $i = 10$
- $s = 00000$
- $t = 0001$

- $b = 11$
- $n = 00001$

Características Gerais de Algoritmos Gulosos

Como apresentado, algoritmos gulosos são utilizados para resolver problemas de otimização. Os algoritmos que se baseiam em uma estratégia gulosa são, em geral, simples e de fácil implementação. A maioria dos problemas de otimização que são resolvidos utilizando uma estratégia gulosa, tem em geral a seguinte forma:

Instâncias Uma instância consiste de um conjunto de objetos e um relacionamento entre eles.

Soluções para uma instância Uma solução é um subconjunto destes objetos. A estratégia gulosa define qual a forma de escolher os objetos da solução. Vale acrescentar que alguns subconjuntos não são permitidos. Essa restrição é imposta pelo relacionamento existente entre os objetos.

Custo de uma solução Cada subconjunto válido (sem conflito) como solução tem associado um custo. Em geral, o custo é definido pelo número de objetos do subconjunto, pela soma dos custos individuais dos objetos, ou através de uma função mais complexa sobre os elementos do subconjunto.

Objetivo Dada uma instância de um problema, o objetivo é maximizar (minimizar) o custo da solução.

Se uma solução *força bruta* é usada neste caso, é necessário considerar todas as possíveis soluções de uma dada instância, computar o custo de cada solução e escolher a de maior (menor) custo. Em geral, este tipo de algoritmo é de ordem exponencial.

A escolha gulosa A escolha gulosa é aquela que salta aos olhos, ou a que primeiro vem na mente quando imaginamos um algoritmo para o problema. Dado o conjunto de objetos de entrada, a escolha gulosa seleciona um dos objetos, seguindo um critério simples (aquele que parece ser o melhor).

Com base nestas definições, é possível definir elementos que comumente fazem parte de uma solução gulosa, na tentativa de definir uma *cara* genérica para uma solução gulosa.

Vamos considerar o exemplo do cálculo do trôco, em um sistema monetário que apresenta o seguinte conjunto de moedas $C = \{100, 25, 10, 5, 1\}$. Uma possível solução que utiliza estratégia gulosa é apresentada abaixo:

CalculaTroco(M)
$C \leftarrow \{100, 25, 10, 5, 1\}$

$S \leftarrow \emptyset$
$soma \leftarrow 0$
while $soma \neq M$ **do**
  $X \leftarrow$ o maior valor de C em que $soma + X \leq M$
  if não existe o item **then**
    return(solução não encontrada)
  $S \leftarrow S \cup \{$ uma moeda de valor $X\}$
  $soma \leftarrow soma + X$
return S

A solução é obtida partindo-se de um conjunto vazio de moedas e, em cada etapa, escollhendo a maior moeda possível. É possível identificar neste algoritmo os seguintes elementos:

1. Uma lista de candidatos, definido pelo conjunto C.
2. Uma lista de elementos escolhidos, definido pelo conjunto S.
3. Uma função **SELEÇÃO** que seleciona um dos candidatos. No nosso exemplo é a uma função que retorna a maior moeda.
4. Uma função **VIABILIDADE** que verifica a viabilidade da escolha. No exemplo, a função que testa se a soma dos valores da nova moeda escolhida com os valores das moedas previamentes escolhidas não ultrapassa o valor do montante.
5. Uma função **SOLUÇÃO** que verifica se um determinado conjunto de candidatos é uma solução para o problema. Em nosso exemplo, é a função que testa se a soma dos elementos escolhidos é igual ao montante M.

Além dos elementos listados, em algumas soluções é possível identificar mais dois elementos: um conjunto de candidatos rejeitados e uma função que define o valor de uma solução (serve para definir a eficiência de uma solução. No exemplo do cálculo do trôco, essa função retorna o número de moedas utilizadas para fornecer o trôco).

Solução Genérica

Um algoritmo guloso genérico define inicialmente um conjunto vazio de candidatos escolhidos. Em cada etapa um novo elemento é escolhido da lista de candidatos e adicionado ao conjunto de candidatos escolhidos. Esta escolha é guiada pela função **SELEÇÃO** que através do crivo da função **VIABILIDADE** define se o elemento selecionado deve ser considerado ou rejeitado. Se o elemento for rejeitado ele não deve ser considerado novamente. Se o elemento é adicionado ao conjunto dos elementos escolhidos, a função **SOLUÇÃO** verifica se o novo conjunto corresponde a solução do problema.

AlgoritmoGuloso(C)
C é o conjunto de candiadtos

```
S ← ∅
while C ≠ ∅ e S não é solução do
   X ← SELEÇÂO(C)
   C ← C\{X}
   if VIABILIDADE(S ∪ {X}) then
      S ← S ∪ {X}
if SOLUÇÃO(S) then
   return S
else
   return(não tem solução)
```