Resumo 4 - EDII

Aluno: *Nathann Zini dos Reis*

Matrícula: *19.2.4007*

Aula 15 - Compressão de textos (Parte 1)

Introdução

Resumidamente, a compressão de textos consiste em representar o texto original de documentos em menos espaço. Basicamente, se substitui alguns caracteres por símbolos que, logicamente, ocupam menos espaço na memória.

Dessa forma, acelera o processo de pesquisa e de transferência entre memórias do documento em questão, uma vez que ele passa a ocupar menos espaço em memória; Em contra partida, para ter-se os ganhos acima mencionado, é aumentado o custo operacional para codificar e decodificar o texto comprimido.

Faz-se, portanto, necessário de um código eficiente no quesito de compressão e descompressão dos texto.

Aspectos importantes para a compressão de texto, além da economia de espaço, são:

- A velocidade de compressão e de descompressão. Salvo que, na maioria das situações, a velocidade de descompressão é mais importante que a de compressão.
- Manter a possibilidade de realizar o casamento de cadeias diretamente no texto comprimido. Possibilitando a busca sequencial dentro de um arquivo de texto comprimido, a fim de aumentar a velocidade, visto que ocupa menos espaço em memória.
- O acesso direto a qualquer parte do texto comprimido, possibilitando o início da descompressão a partir da parte acessada.

A métrica de compressão de texto é calculada com a razão de compressão, que é a porcentagem que representa o tamanho do arquivo comprimido em relação ao arquivo não comprimido.

Método de codificação Huffman

Característica:

Um código único, de tamanho variável, é atribuído a cada símbolo diferente do texto;

Códigos mais curtos são atribuídos a símbolos com frequência altas;

Utiliza caracteres como símbolos.

Utilizando o Huffman Usando Palavras

Considera cada palavra diferente do texto como um símbolo, montando uma tabela com as palavras diferentes do textos (representando o vocabulário do texto), conta as frequências de cada uma delas e gera um código de Huffman para cada uma delas. Então o texto é comprimido substituindo as palavras por seus símbolos.

Importante notar que a compressão é realizada em duas passadas no texto:

Primeiro verifica a frequência de cada símbolo, realizando o vocabulário do texto;

Segundo realiza a compressão do texto.

Os separadores são tratados da seguinte forma:

os espaços em branco são tratados como separador / fim da palavra anterior;

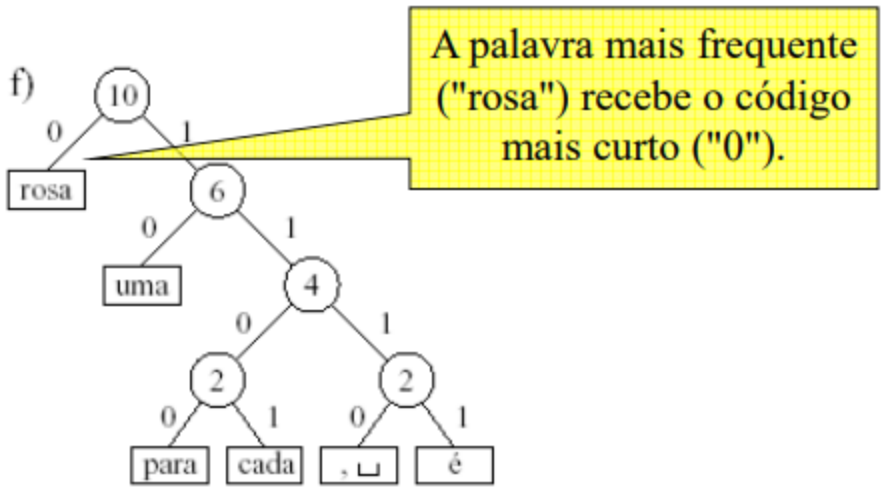
As pontuações (vírgula, ponto...) são tratados como palavras independentes e são atribuídos, portanto, um código para cada uma delas;

Árvore de codificação de texto

Consiste um uma árvore canônica, ou seja, um lado dela sempre será maior que a outra em que as palavras com a maior frequência estarão nos níveis mais a cima e as com menores frequências nos menores níveis.

Cada aresta da árvore representa um bit (0 ou 1) que formará o código da palavra do nó folha por meio do encaminhamento da árvore. É observado que a quantidade de nós externos/folhas da árvore é sempre a quantidade e nós internos + 1.

Exemplo de árvore:



Entretanto, para textos muitos grandes, construir essa árvore tem-se um custo operacional muito grande. Logo, é interessante fazer de uma maneira de simular essa árvore.

Algoritmo de Moffat e Katajainen

Apresenta comportamento linear em tempo e em espaço.

Não encontra os códigos propriamente ditos, porém encontra o comprimento dos códigos e em qual lugar eles estão. Após o cálculo do comprimento do código é feita a codificação e a decodificação do mesmo.

Funcionamento:

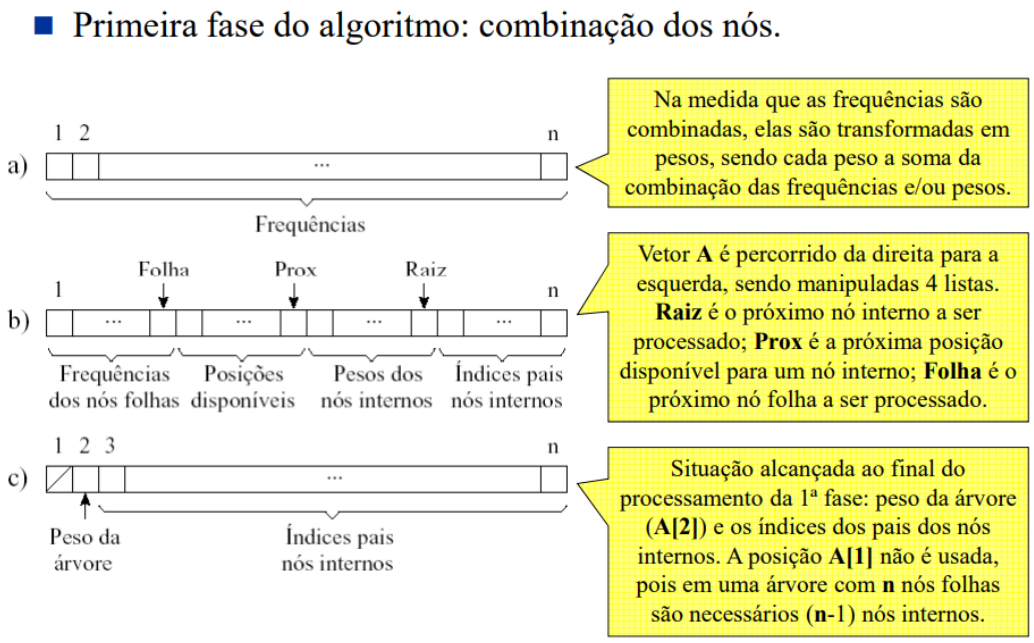
Será criado um vetor A com a frequência de cada palavra do vocabulário em ordem decrescente;

Todo o processo se dá em sub vetores temporário criados dentro do próprio vetor A;

é dividido em 3 etapas:

1- Combinação de nós

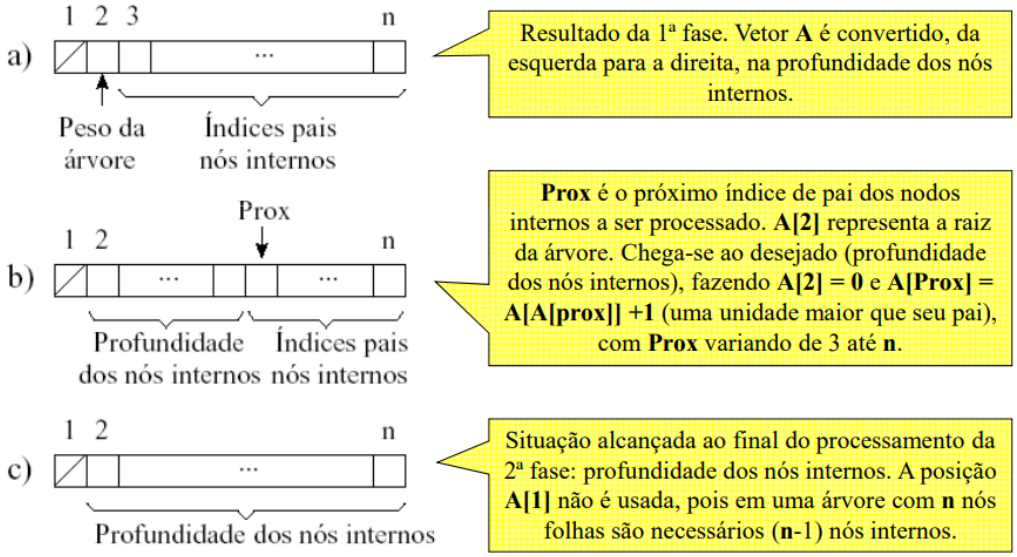
Resumidamente, é responsável por simular a árvore de codificação. Como dado de entrada, tem-se o vetor que guarda, de maneira decrescente, a frequência de cada palavra do vocabulário.



Ao final, será salvo no vetor o peso de cada nó interno da árvore. O primeiro nível, entretanto, terá salvo o peso da árvore. Os demais nós internos serão representados pelo nível em que ele ne encontra na árvore. Os dados são armazenados a partir da segunda posição do vetor, visto que sempre tem um nó interno a menos que nós externo.

2 - Determinação das profundidades dos nós internos

- Segunda fase do algoritmo: profundidade dos nós internos.



Essa fase é responsável por calcular a profundidade dos nós externos. De grosso modo, é sempre incrementado +1 ao valor do nó pai de cada nó e salvo em si mesmo. A primeira posição do vetor é inutilizada, visto que há 1 nó externo a mais que nós internos. A segunda posição de vetor, que anteriormente guarda o peso da árvore, é iniciado com o valor 0. A partir daí, para todos os seus filhos é atribuído o valor do pai + 1, e o mesmo é feito para os filhos dos filhos, até o fim.

```
SegundaFase (A, n)
{ A[2] = 0;
  for (Prox = 3; Prox <= n; Prox++) A[Prox] = A[A[Prox]] + 1;
}
```

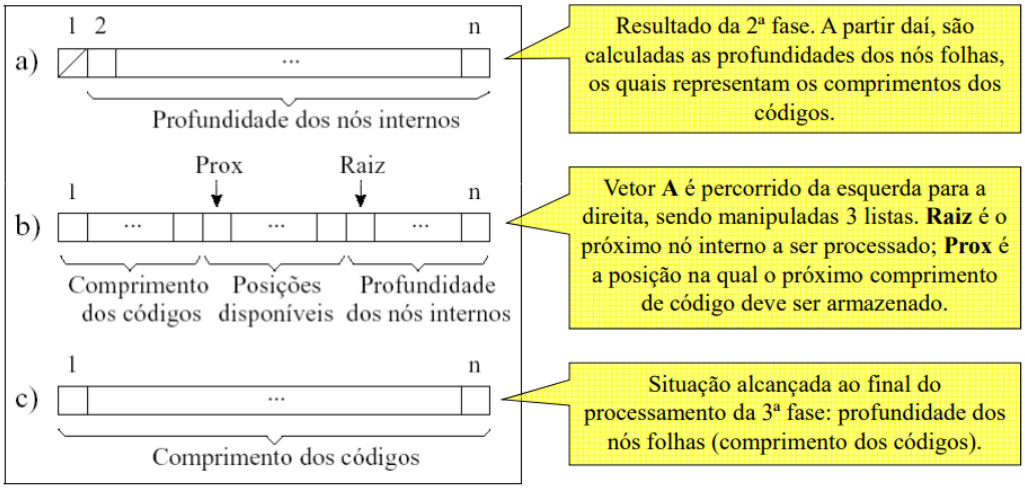
- Resultado da segunda fase:

	0	1	2	3	3

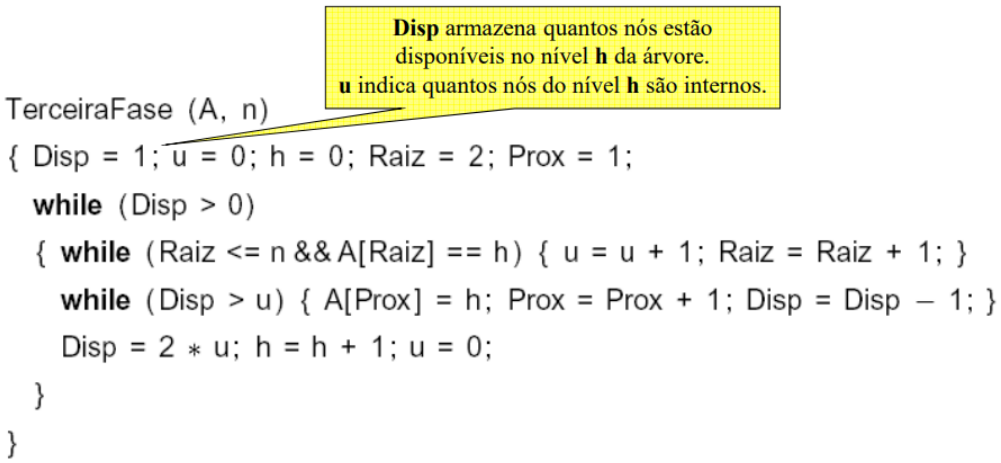
3 - Determinação das profundidades dos nós folhas

Responsável por calcular a profundidade dos nós externos.

Calcula as profundidades dos nós externos a partir dos nós internos. Sabe-se que os nós internos começam da segunda posição do vetor, por tanto a análise começa da segunda posição e o comprimento do código é salvo na posição anterior à do nó interno analisado, inicialmente, é analisado o primeiro nó interno na posição 2 e o comprimento do código é salvo na posição 1, representando o primeiro nó externo.



no final, cada posição do vetor representa uma palavra do vocabulário e é salvo o comprimento de cada palavra representada pela quantidade de bits que possui. Pela estrutura da árvore, será sempre em ordem crescente.



- Resultado da terceira fase:

1	2	4	4	4	4

Aula 16 - Compressão de Textos (parte 2)

Obtenção do Códigos Canônicos

Os comprimentos dos códigos seguem o algoritmo de Huffman e os de mesmo comprimento são inteiros consecutivos.

O primeiro código é composto apenas por zeros. Para os demais, adiciona-se 1 ao código anterior e faz-se um deslocamento à esquerda para obter-se o comprimento adequado quando necessário.

O deslocamento à esquerda é dado pela diferença do comprimento do código atual com o comprimento do código anterior. Exemplo, se o código atual tiver tamanho 4 e o anterior tem tamanho 2, soma-se 1, de forma binaria, ao código anterior, e é feito 4-2 shift à esquerdas, ou seja, é deslocado 2 vezes à esquerda. Analogamente, se o código anterior e o atual tem a mesma posição, é apenas somado 1 ao valor do código anterior e não é feito nenhum shift, pois a diferença de tamanho é 4 - 4 = 0.

i	Símbolo	Código Canônico
1	rosa	0
2	uma	10
3	para	1100
4	cada	1101
5	,␣	1110
6	é	1111

Codificação e Decodificação

- Os algoritmos usam dois vetores com *MaxCompCod* (o comprimento do maior código) elementos:
  - *Base*: indica, para um dado comprimento c, o valor inteiro do 1º código com tal comprimento;

$$\text{Base}[c] = \begin{cases} 0 & \text{se } c = 1, \\ 2 \times (\text{Base}[c-1] + w_{c-1}) & \text{caso contrário,} \end{cases}$$

Nº de códigos com comprimento (**c-1**).

  - *Offset*: indica, para um dado comprimento c, o índice no vocabulário da 1ª palavra de tal comprimento.

É controlado por esses dois vetores destacados na imagem acima, para calcular o código referente a cada palavra;

A tabela gerada será:

c	Base[c]	Offset[c]
1	0	1
2	2	2
3	6	2
4	12	3

Para se calcular os códigos de cada palavras é analisada, portanto, a posição da palavra desejada no vetor, o tamanho do comprimento da palavra e é somado ao código inicial do tamanho daquele comprimento a distância da palavra desejada à primeira palavra daquele tamanho.

Exemplo: Conforme a ultima tabela gerada, O primeiro Inteiro de tamanho 4 é 12, que representa 1100 em binário, e se encontra na posição 3 do vetor gerado pelo algoritmo de **Moffat e Katajainen**. Se quiser uma palavra de tamanho 4 e que esteja na posição 5 do vetor, portanto é calculado a partir do valor inicial de tamanho 4 (que, como mencionado é o inteiro 12, representado por 1100) e é somado à ele a distancia da palavra desejada (posição 5) à primeira palavra daquele tamanho de comprimento (posição 3). Logo, a distância é 2, portanto o código da palavra da posição 5 será 14 (12 + 2) e é representado, em binário, por 1110, que na tabela representa ao símbolo

$$,\sqcup$$

Código para a codificação:

```
Codifica (Base, Offset, i, MaxCompCod)
{ c = 1;
  while ( i >= Offset[c + 1] ) && (c + 1 <= MaxCompCod )
    c = c + 1;
  Codigo = i – Offset[c] + Base[c];
}
```

Parâmetros: vetores **Base** e **Offset**, índice **i** do símbolo a ser codificado e o comprimento **MaxCompCod** dos vetores.

Cálculo do comprimento **c** de código a ser utilizado.

O código corresponde à soma da ordem do código para o comprimento **c** (**i – Offset[c]**) com o valor inteiro do 1º código de comprimento **c** (**Base[c]**).

- Para i = 4 ("cada"), calcula-se que seu código possui comprimento 4 e verifica-se que é o 2º código de tal comprimento.
  - Assim, seu código é 13 (4 – Offset[4] + Base[4]): 1101.

Código para a decodificação:

Parâmetros: vetores **Base** e **Offset**, o arquivo comprimido e o comprimento **MaxCompCod** dos vetores.

```
Decodifica (Base, Offset, ArqComprimido, MaxCompCod)
{ c = 1;
  Codigo = LeBit (ArqComprimido);
  while ((( Codigo << 1 ) >= Base[c + 1]) && ( c + 1 <= MaxCompCod ))
    { Codigo = (Codigo << 1) || LeBit (ArqComprimido);
      c = c + 1;
    }
  i = Codigo – Base[c] + Offset[c];
}
```

Identifica o código a partir de uma posição do arquivo comprimido.

O arquivo de entrada é lido *bit*-a-*bit*, adicionando-se os *bits* lidos ao código e comparando-o com o vetor **Base**.

Esse algoritmo Decodifica retorna o valor i, que representa o índice da palavra dentro do vocabulário criado a partir do código em binário.