Universidade Federal
de Ouro Preto

ENYA LUÍSA GOMES DOS SANTOS
19.2.4201

RESUMO IV - COMPRESSÃO DE TEXTOS

Resumo apresentado por exigência da disciplina BCC203 - Estrutura de Dados II, da Universidade Federal de Ouro Preto.
Professor: Guilherme Tavares Assis

Enya Luísa Gomes dos Santos
19.2.4201

COMPRESSÃO DE TEXTOS

A compreensão de texto consiste em comprimir textos, ou seja, na redução do espaço consumido por um texto. Para alcançar esse objetivo deve-se substituir os símbolos do texto original por outros que ocupam um número menor de bits ou bytes. Esses símbolos podem ser um caractere ou até mesmo uma palavra inteira.

Em suma, a ideia geral da compreensão de textos, segue a ideia de gerar códigos, que ocupem espaços menores, para os diferentes símbolos do texto original e por fim substitui cada símbolo do texto justamente pelo seu código.

O maior ganho desse processo é o ganho na redução de espaço ocupado pelo texto, também há outros ganhos, como menor tempo para se pesquisar uma cadeia desejada em um texto, ou seja, o casamento de cadeia ocorre de forma mais rápida em um texto comprimido. A leitura também ocorre de forma mais rápida.

Já o preço a se pagar por esses ganhos é o custo computacional para **codificar** e **decodificar** o texto.

Bons algoritmos para realizar a compreensão de texto devem considerar:

- Velocidade de **compressão** e de **descompressão**, onde, em muitas situações, a velocidade de descompressão é mais importante que a de compreensão.
- Possibilidade de realizar casamento de cadeias diretamente no texto comprimido.
- Acesso direto a qualquer parte do texto comprimido, possibilitando que a descompressão possa ocorrer em qualquer parte do arquivo.

Razão de compressão corresponde à porcentagem que o arquivo comprimido representa em relação ao tamanho do arquivo não comprimido. A ideia é que, através dessa métrica, o melhor algoritmo apresentará uma razão maior, que comprima mais o texto original. Exemplo: se o arquivo não comprimido possui 100 bytes e o arquivo comprimido possui 30 bytes, a razão é de 30%.

Atualmente tem-se um dos melhores métodos de compreensão que se chama **Compressão de Huffman**, proposto em 1952.

Enya Luísa Gomes dos Santos
19.2.4201

COMPRESSÃO DE HUFFMAN

- Um código único, de **tamanho variável**, é atribuído a cada símbolo diferente do texto.
- **Códigos mais curtos são atribuídos a símbolos com frequências altas.**
- As implementações tradicionais do método de Huffman consideram caracteres como símbolos, ou seja, cada caractere era um símbolo. Hoje as implementações não utilizam mais os caracteres como símbolos, e sim as **palavras**, pois foi verificado que, na maioria dos casos, quando se usa caractere como símbolo tem-se uma razão de compressão na faixa de 60%, já com as palavras tem-se 25%.

Primeiramente, a leitura é feita no texto original com a ideia de gerar o vocabulário para determinar a frequência de cada palavra do vocabulário, ou seja, o número de vezes que cada palavra aparece no texto. A seguir, ocorre uma nova passada no arquivo de texto substituindo cada palavra pelo seu código correspondente. As palavras com frequências altas possuem códigos menores.

A compressão é realizada em duas passadas sobre o texto:

1. Obtenção da frequência de cada palavra diferente.
2. Realização da compressão.

Sabemos que em um texto com linguagem natural existem os separadores em meio às palavras, esses **caracteres separadores** são: caracteres que aparecem entre palavras, como espaço, vírgula, ponto, etc. Para lidar com esses separadores, o algoritmo de Huffman trata de duas formas diferentes, uma é, os espaços em branco determinam o fim de uma palavra e cada caractere separador será tratado como uma palavra com código separado, já a outra forma consiste em tratar os separadores junto com a palavra, como por exemplo “como?” que seria tratado como uma palavra. Porém a segunda forma pode reduzir a eficiente, visando que a mesma palavra será visualizada como diferente pelo algoritmo por estar em conjunto com um caractere separador.

Enya Luísa Gomes dos Santos
19.2.4201

Árvore de codificação

Considerando o seguinte texto: “para cada rosa rosa, uma rosa é uma rosa”.

No passo ***a*** ocorre a parte de identificação do vocabulário e a contagem de frequência em que cada uma palavra aparece no texto. Cada palavra constituem um nodo folha da árvore de codificação, a partir dos nodos folhas a árvore é construída, ou seja, de baixo para cima, da seguinte forma, pega-se os nodos folha que apresentam menor frequência e os combinam em uma subárvore onde o nodo pai recebe a soma (chamado de **peso**) da frequência dos nodos folha considerado, como ocorre no passo ***b***, e esse processo se repete e dessa forma a árvore é construída.

A ideia da árvore de codificação é que ela seja uma **árvore canônica**, ou seja, um dos lados, para cada subárvore, será maior ou igual ao outro em termos de altura da árvore, sendo assim desbalanceada, também a relação entre a quantidade de nós internos e nós externos e que os nós externos serão sempre (m+1), onde m é a quantidade de nós internos. Vejamos o passo ***c***, nesse passo, em teoria, poderíamos escolher tanto as duas subárvore quanto o node uma, pois todos apresentam o mesmo número de frequência 2 e esse sendo o menor, mas para manter essa propriedade de árvore canônica, o algoritmo prioriza unir subárvore já formadas.

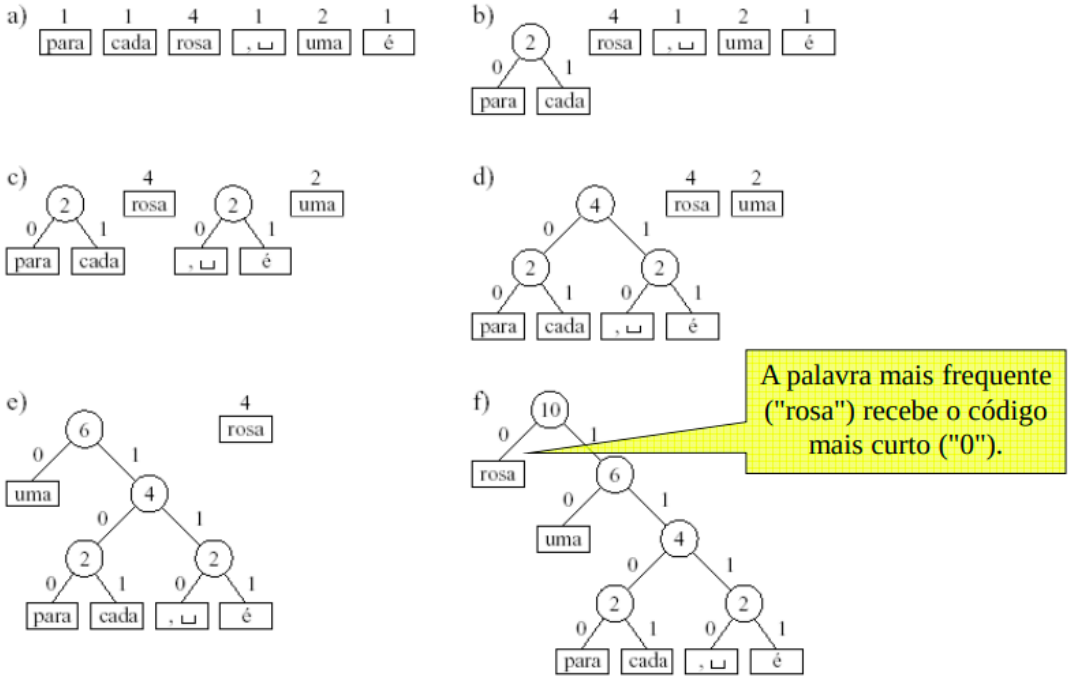

Figura 01 - Representação da construção da árvore de codificação

Essa árvore foi construída de forma a ser uma árvore binária, onde a aresta do lado esquerdo representa o *bit* 0 do código a ser gerado e a direita o *bit* 1 desse código. Logo o código será a sequência de bits que leva até a palavra ao percorrer a árvore. Por exemplo, na imagem acima, a palavra ***para*** será representada pelos bits **1100**. E ainda considerando a imagem acima, a palavra ***rosa*** será representada pelo **bit 0**, ou seja, o menor código em relação aos outros, e ***rosa*** é a palavra que aparece mais vezes no texto, ou seja, tem uma frequência maior em termo de presença.

Após a montagem da árvore de codificação, o próximo passo é reescrever o texto usando seus códigos. E para encontrar uma determinada palavra dentro da árvore a única forma é caminhar a árvore por completo a partir da raiz, o que torna um processo custoso, a ideia para resolver esse problema é fazer o caminhamento apenas uma vez guardando os código referente a cada palavra, e após isso construir uma estrutura que seja possibilita uma busca binária, por exemplo uma árvore TRIE ou uma árvore B.

Já a decodificação/descompressão do texto pode ser feita por meio da árvore de codificação, visto que há como fazer uma busca binária pelos bits 0 ou 1, o que será um processo rápido.

Entretanto, a construção dessa árvore, dependendo do tamanho do texto, pode ser muito custosa em termos de memória e tempo, e para isso o algoritmo de Moffat e Katajainen **simula** a construção dessa árvore por meio de vetor.

Algoritmo de Moffat e Katajainen

Baseado na codificação canônica para, a partir de um texto original descobrir quais são os códigos referente a cada palavra do vocabulário sem a necessidade de construir a árvore de codificação e apenas simulando. Apresenta comportamento linear em tempo e em espaço.

O algoritmo não calcula os códigos e sim **calcula os comprimentos** dos códigos em lugar dos códigos propriamente ditos. Após o cálculo dos comprimentos, há uma forma elegante e eficiente para a codificação e a decodificação.

Esse algoritmo possui três etapas distintas:

1. Combinação dos nós.
2. Determinação das profundidades dos nós internos.
3. Determinação das profundidades dos nós folhas (comprimentos dos códigos).

Usando como exemplo o texto: “para cada rosa rosa, uma rosa é uma rosa”, a primeira coisa s se fazer é criar um vetor de taxa de frequência do vocabulário em ordem decrescente tem-se vetor:

4	2	1	1	1	1

Onde cada posição do vetor representa uma palavra do vocabulário e dentro do vetor guarda-se a taxa de frequência da palavra com ordem decrescente. Logo as três fases para calcular o comprimento dos códigos é iniciada e feita nesse mesmo vetor e quando preciso, com subvetores.

Primeira fase do algoritmo: combinação dos nós.

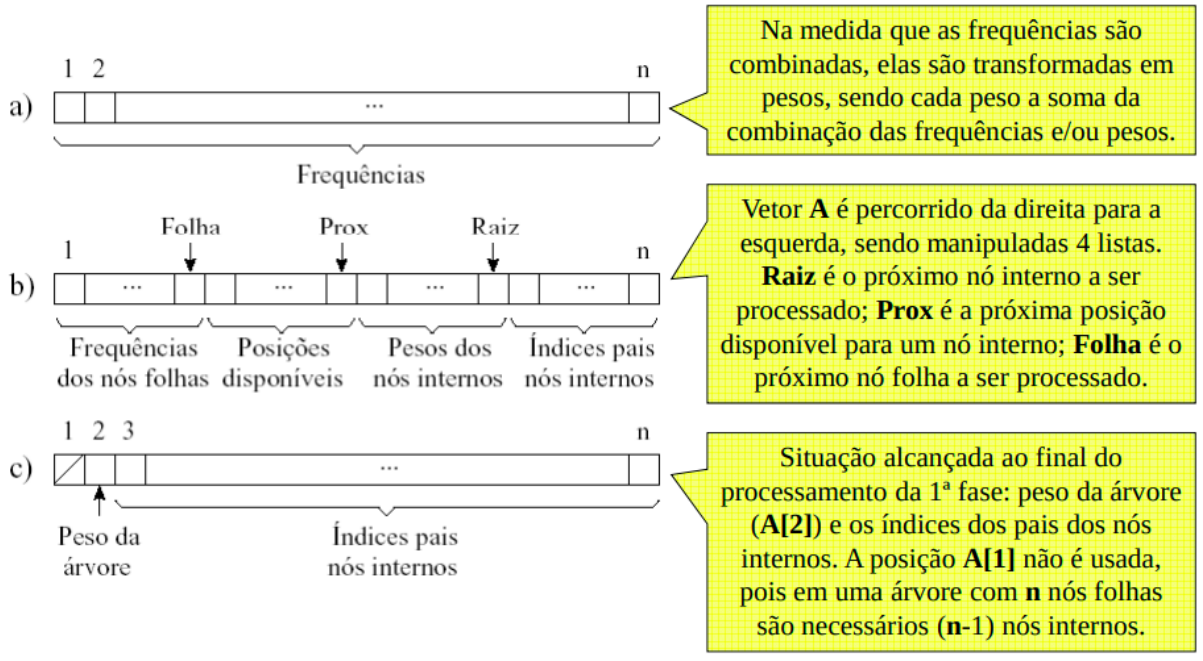

Figura 02 - Primeira fase do algoritmo: combinação dos nós.

Enya Luísa Gomes dos Santos
19.2.4201

Índices dos pais nós internos: representa o nível dentro da árvore onde aquele nó interno se encontra, começando de 1, a primeira posição do vetor no passo ***c*** fica vazia pois o número de nós externos é uma unidade maior que o número de nós internos.

A ideia é que, a partir do primeiro passo apresentado acima, gera-se a tabela abaixo, onde ***a*** é o início desse algoritmo e ***k*** o resultado final, com a primeira posição vazia, a segunda posição o peso da árvore e as demais os índices dos nós internos, tudo utilizando o mesmo vetor. Na figura abaixo, em ***k*** tem-se o índice (nível) de cada subárvore da árvore de codificação simulada:

	1	2	3	4	5	6	Prox	Raiz	Folha
a)	4	2	1	1	1	1	6	6	6
b)	4	2	1	1	1	1	6	6	5
c)	4	2	1	1	1	2	5	6	4
d)	4	2	1	1	1	2	5	6	3
e)	4	2	1	1	2	2	4	6	2
f)	4	2	1	2	2	4	4	5	2
g)	4	2	1	4	4	4	3	4	2
h)	4	2	2	4	4	4	3	4	1
i)	4	2	6	3	4	4	2	3	1
j)	4	4	6	3	4	4	2	3	0
k)	/	10	2	3	4	4	1	2	0

Figura 03 - Exemplo da primeira fase do algoritmo. Prox, Raiz e Folha são inicializados com 6 pois esse é o tamanho do vetor inicial.

```
PrimeiraFase (A, n)
{ Raiz = n; Folha = n;
  for (Prox = n; n >= 2; Prox--)
  { /* Procura Posicao */
    if ((nao existe Folha) || ((Raiz > Prox) && (A[Raiz] <= A[Folha])))
    { A[Prox] = A[Raiz]; A[Raiz] = Prox; Raiz = Raiz - 1; /* No interno */ }
    else { A[Prox] = A[Folha]; Folha = Folha - 1; /* No folha */ }
    /* Atualiza Frequencias */
    if ((nao existe Folha) || ((Raiz > Prox) && (A[Raiz] <= A[Folha])))
    { /* No interno */
      A[Prox] = A[Prox] + A[Raiz]; A[Raiz] = Prox; Raiz = Raiz - 1;
    }
    else { A[Prox] = A[Prox] + A[Folha]; Folha = Folha - 1; /* No folha */ }
  }
}
```

Figura 04 - Algoritmo da primeira fase do algoritmo. *No for não é n >= 2 e sim **Prox >=2.**

Segunda fase do algoritmo: profundidade dos nós internos

Dado o vetor da primeira fase em ***a*** calcula-se a profundidade dos nós internos, ou seja, a altura (a distância do nó interno em relação a raiz), cujo os níveis já se sabe, em ***c***. A primeira posição do vetor continua não sendo utilizada.

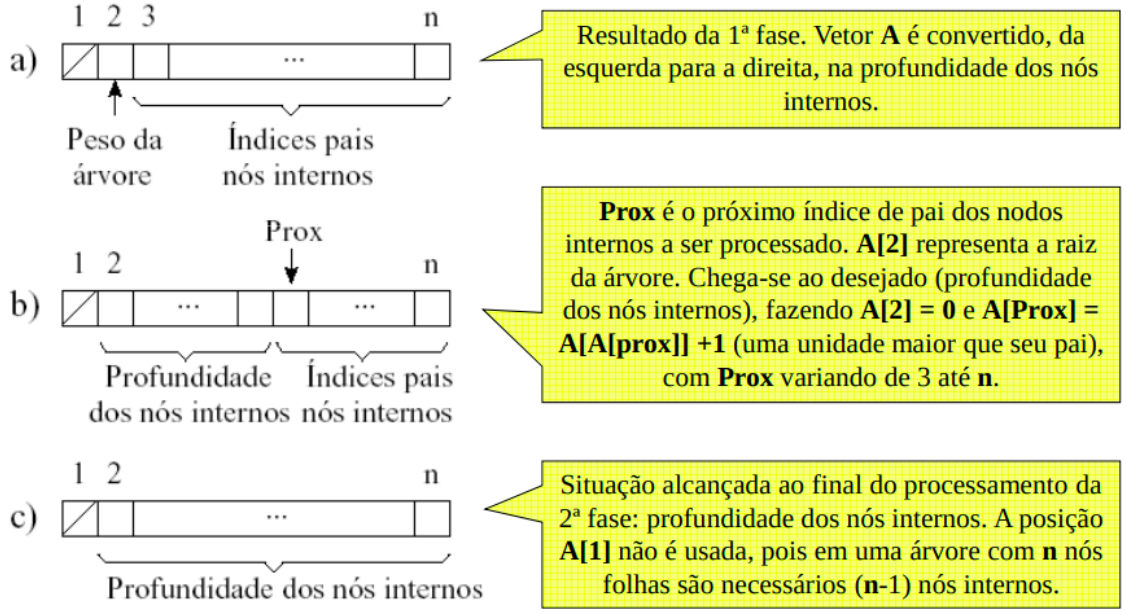

Figura 05 - Profundidade dos nós internos.

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

Figura 06 - Algoritmo da fase de profundidade dos nós internos.

Tabela de representação da execução do algoritmo

1	2	3	4	5	6	A[Prox]	
	10	**2**	**3**	**4**	**4**		
	0	1	3	4	4		**Pos: 2.** A profundidade da raiz é **simplesmente 0.**
	0	**1**	3	4	4	2	**Pos: 3.** Será a profundidade anterior + 1, ou seja, A[A[Prox]] + 1 => 0 + 1 = 1
	0	1	**2**	4	4	3	**Pos: 4.** Será a profundidade anterior + 1, ou seja, 1 + 1 = 2
	0	1	2	**3**	4	4	**Pos: 5.** Será a profundidade anterior + 1, ou seja, A[A[6]] => A[4] => 2 + 1 = 3
	0	1	2	3	**3**	4	**Pos: 6.** Será a profundidade anterior + 1, ou seja, A[A[6]] => A[4] => 2 + 1 = 3
	0	**1**	**2**	**3**	**3**	**4**	

Terceira fase do algoritmo: profundidade dos nós folhas

Agora, a partir do vetor obtido na fase dois, calcula-se a profundidade dos nós folhas, que é justamente o comprimento de cada código referente à árvore de codificação.

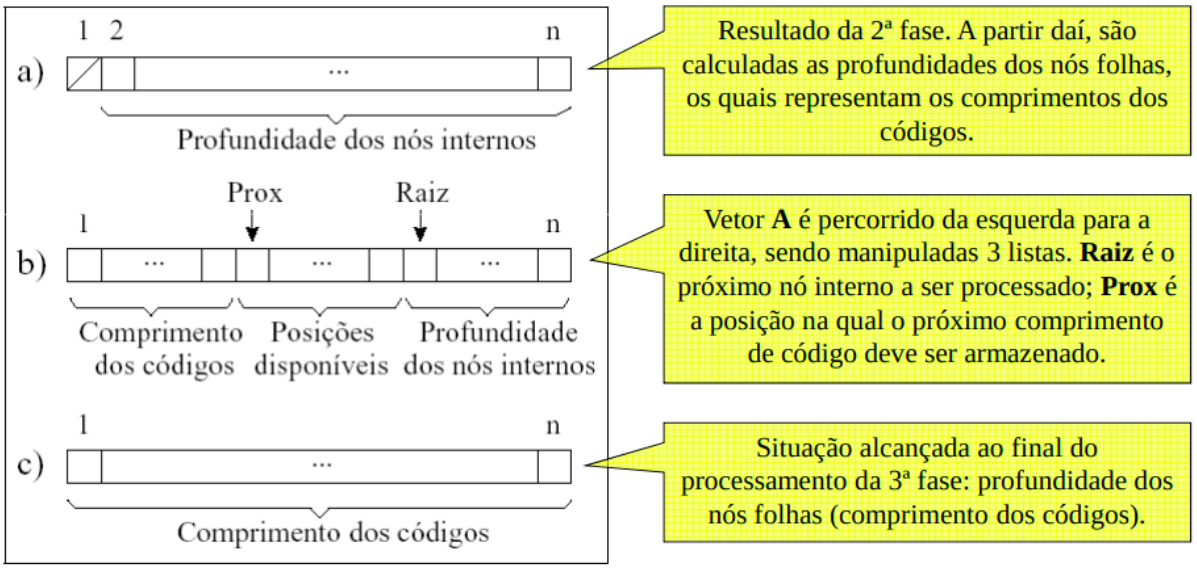

Figura 07 - Profundidade dos nós folhas.

Disp armazena quantos nós estão disponíveis no nível **h** da árvore.
u indica quantos nós do nível **h** são internos.

```
TerceiraFase (A, n)
{ Disp = 1; u = 0; h = 0; Raiz = 2; Prox = 1;
  while (Disp > 0)
  { while (Raiz <= n && A[Raiz] == h) { u = u + 1; Raiz = Raiz + 1; }
    while (Disp > u) { A[Prox] = h; Prox = Prox + 1; Disp = Disp – 1; }
    Disp = 2 * u; h = h + 1; u = 0;
  }
}
```

- Resultado da terceira fase:

1	2	4	4	4	4

Figura 08 - Algoritmo da fase três, profundidade dos nós folhas.

No algoritmo acima:

Disp representa -> Quantos nodos há na altura atual da árvore.

h representa -> Altura da árvore a ser tratada, ou seja a altura atual.

u representa -> Quantidade de nodos **internos** entre os **disp** no nível **h**. Logo os nodos externos são iguais a `Disp - u`.

O resultado apresenta, em cada posição, o **comprimento dos códigos** do vocabulário do texto original.

Logo, o algoritmo completo para **calcular o comprimento dos códigos** a partir de um vetor de frequências é o seguinte:

```
CalculaCompCodigo (A, n)
{ A = PrimeiraFase (A, n);
  A = SegundaFase (A, n);
  A = TerceiraFase (A, n);
}
```

Figura 09 - Algoritmo completo para cálculo do comprimento dos códigos.

É sobre esse vetor de comprimento dos códigos obtido que será calculado os códigos em si referente a cada palavra do vocabulário.

OBTENÇÃO DO CÓDIGOS CANÔNICOS

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

A partir dos **comprimentos** obtidos pelo algoritmo de Moffat e Katajainen, o cálculo dos códigos ocorre de maneira simples:

- Primeiro código é composto apenas por zeros;
- Para os demais, adiciona-se 1 (**soma binária**) ao código anterior e faz-se um deslocamento (*shift*) à esquerda, de quantas posições forem necessárias, para obter-se o comprimento adequado **quando necessário**;

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

Figura 10 - Tabela dos códigos canônicos

Analisando com a árvore de codificação representada na figura 01, pode-se perceber que os códigos da tabela são os mesmos dos códigos obtidos por meio do caminhamento na árvore.

Com os códigos referentes a cada palavra do vocabulário, agora é possível codificar o texto original substituindo as palavras pelo seu código correspondente para comprimi-lo.

Porém, após a compreensão do texto, em algum momento será necessário a **decodificação** desse, para isso é preciso ter conhecimento sobre a relação entre os códigos e as palavras, e para que isso ocorra a primeira coisa guardada dentro do arquivo comprimido, como cabeçalho, é justamente algo como a tabela da figura 10, esse cabeçalho acaba que ocupa um grande espaço, há também a possibilidade de se guardar somente as posições das palavra do vocabulário e essas palavras, onde os códigos canônicos serão gerados automaticamente no momento que for necessário.

Codificação e Decodificação

Outra forma de gerar os códigos sem aplicar as duas regras acima para criar os códigos, onde os códigos serão gerados de forma dinâmica no momento que for necessário e para somente as palavras necessárias, com objetivo de tirar a necessidade de guardar os códigos juntamente com as palavras no arquivo de texto comprimido para diminuir o uso de espaço dentro do arquivo.

Enya Luísa Gomes dos Santos
19.2.4201

Os algoritmos são baseados no fato de que os códigos de mesmo comprimento são inteiros consecutivos. Geram-se duas tabelas a partir do vetor gerado pelo algoritmo de Moffat e Katajainen.

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

***c*:** indica o comprimento de código

Base: indica, para um dado comprimento *c*, o valor inteiro do 1º código com tal comprimento. A regra para a base está apresentada abaixo:

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

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

Offset: indica, para um dado comprimento *c*, o índice no vocabulário da 1ª palavra de tal comprimento. Caso a palavra não exista, o offset recebe o valor do índice anterior.

Com essas tabelas é possível gerar os códigos referente a cada palavra de forma dinâmica da seguinte forma, através do comprimento da palavra em ***c,*** a **primeira** palavra de tamanho ***c*** possui do valor da **Base** e está na posição do **Offset**, logo, se necessário, lembrando que as palavras com o mesmo comprimento têm valores consecutivos, subtrai o índice ***i*** de **Offset** e soma com o valor da **Base**, e por fim esse valor é convertido em binário. Seguindo o exemplo:

Enya Luísa Gomes dos Santos
19.2.4201

1	2	3	4	5	6
1	2	4	4	4	4

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

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

Iremos encontrar o código da palavra **cada**, o índice ***i*** de casa é 4, como apresentado na tabela à direita e através da tabela à esquerda sabemos que seu comprimento é 4, então se busca na tabela do meio a Base e o Offset de *c* = 4, que é 12 e 3 respectivamente, esses dados nos dizem que a primeira palavra de código com comprimento 4 tem valor igual a 12 e está na posição 3, mas queremos a palavra da posição 4, sabemos que as palavras com o mesmo comprimento têm valores consecutivos, logo o seguinte cálculo é feito `i - Offset => 4 - 3 = 1`, e somamos o resultado à Base, ou seja `12 + 1 = 13` e por fim esse valor é convertido para binário **`1101`** resultando no código da palavra **cada**. Por meio desse processo é possível calcular o código de forma dinâmica através do cálculo (**`i - Offset + Base`**) que resultará em um inteiro e esse é convertido para binário.

```
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]**).

No arquivo se guarda a tabela com o **`i`** e os símbolos e também a tabela gerada com **`c, Base e Offset`**.

Como se sabe a forma para se codificar uma palavra a decodificação ocorre de forma mais simples.

Enya Luísa Gomes dos Santos
19.2.4201

Decodificação

No algoritmo abaixo o comprimento de código é calculado, pois parte do princípio que o vetor gerado pelo algoritmo de Moffat e Katajainen não está no arquivo de texto codificado, logo se torna necessário calcular para decodificar os códigos.

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**.

Execução do ***while*** algoritmo de decodificação da sequência de bits "1101" da palavra **cada** representada na tabela a seguir:

c	LeBit	Codigo	Codigo $<< 1$	Base$[c+1]$
1	1	1	-	-
2	1	10 **or** 1 = 11	10	10
3	0	110 **or** 0 = 110	110	110
4	1	1100 **or** 1 = 1101	1100	1100

Ao final da execução obtivemos ***c* = 4** ou seja, o comprimento da palavra referente a sequência de bits "1101". E calcula-se `i = Codigo – Base[c] + Offset[c] => i = 13(1101) - 12 + 3 =` **4** valor que corresponde a posição ***i*** do vocabulário que indica o símbolo **cada**.

Enya Luísa Gomes dos Santos
19.2.4201

COMPRESSÃO

O processo de compressão é realizado por meio de 3 etapas.

1ª etapa, o arquivo texto é percorrido e o vocabulário é gerado juntamente com a frequência de cada palavra.

- Uma tabela hash com tratamento de colisão é utilizada para que as operações de inserção e pesquisa no vetor de vocabulário sejam realizadas com custo O(1).

```
Compressao (ArqTexto, ArqComprimido)
{ /* Primeira etapa */
  while (!feof (ArqTexto))
    { Palavra = ExtraiProximaPalavra (ArqTexto);
      Pos = Pesquisa (Palavra, Vocabulario);
      if Pos é uma posicao valida
      Vocabulario[Pos].Freq = Vocabulario[Pos].Freq + 1
      else Insere (Palavra, Vocabulario);
    }
```

2ª etapa:

- O vetor Vocabulario é ordenado pelas frequências de suas palavras.
- Calcula-se o comprimento dos códigos (algoritmo de Moffat e Katajainen).
- Os vetores Base, Offset e Vocabulario são construídos e gravados no início do arquivo comprimido.
- A tabela hash é reconstruída a partir da leitura do vocabulário no disco, como preparação para a 3ª etapa.

```
/* Segunda etapa */
Vocabulario = OrdenaPorFrequencia (Vocabulario);
Vocabulario = CalculaCompCodigo (Vocabulario, n);
ConstroiVetores (Base, Offset, ArqComprimido);
Grava (Vocabulario, ArqComprimido);
LeVocabulario (Vocabulario, ArqComprimido);
```

3ª etapa:

- O arquivo texto é novamente percorrido.
- As palavras são extraídas e codificadas.
- Os códigos correspondentes são gravados no arquivo comprimido.

```
/* Terceira etapa */
PosicionaPrimeiraPosicao (ArqTexto);
while (!feof(ArqTexto))
  { Palavra = ExtraiProximaPalavra (ArqTexto);
    Pos = Pesquisa (Palavra, Vocabulario);
    Codigo = Codifica (Base, Offset,
                       Vocabulario[Pos].Ordem, MaxCompCod);
    Escreve (ArqComprimido, Codigo);
  }
```

O processo de **decompressão** é mais simples do que o de compressão:

- Leitura dos vetores Base, Offset e Vocabulario gravados no início do arquivo comprimido.
- Leitura dos códigos do arquivo comprimido, descodificando-os e gravando as palavras correspondentes no arquivo texto.

```
Descompressao (ArqTexto, ArqComprimido)
{ LerVetores (Base, Offset, ArqComprimido);
  LeVocabulario (Vocabulario, ArqComprimido);
  while (!feof(ArqComprimido))
    { i = Decodifica (Base, Offset, ArqComprimido, MaxCompCod);
      Grava (Vocabulario[i], ArqTexto);
    }
}
```