Codificação Huffman

Método Estatístico

De acordo com Salomon (2002, p. 9), o método estatístico de compressão de dados utiliza uma propriedade estatística para atribuir códigos de tamanhos variáveis a cada dado de um arquivo a ser compactado.

Essa propriedade estatística é o número de ocorrências que um certo dado possui dentro de um arquivo. O que faremos é classificar os dados do arquivo por ordem de ocorrência.

Sabendo o número de ocorrências de cada dado e sabendo a quantidade total de dados no arquivo é possível calcular a probabilidade individual de ocorrência de cada dado dentro do arquivo. Para isso, basta dividir a ocorrência individual de um dado pelo total de dados do arquivo:

$$P(dado) = \frac{número\ de\ ocorrências\ do\ dado}{total\ de\ dados\ do\ arquivo}$$

Então $P(dado)$ é a probabilidade de um certo dado ocorrer no arquivo.

Essa é a informação mais importante a ser utilizada no método estatístico de compressão.

Código de tamanho variável

A codificação *ASCII* utiliza 7 *bits* de dados para codificar cada um dos 128 caracteres possíveis, isto é, cada caractere dessa codificação possui o mesmo tamanho de 7 *bits,* tornando-a uma codificação de tamanho fixo. A codificação de tamanho variável, por sua vez, permite que a quantidade de *bits*

que representará cada dado varie de acordo com alguma regra. No caso da codificação pelo método estatístico, essa regra é a probabilidade de ocorrência deste dado em um determinado arquivo.

Na atribuição de código de tamanho variável pela regra de probabilidade de ocorrência, a ideia é que quanto mais ocorrências de um dado houver, melhor será (para a compressão dos dados) atribuir ao dado um código de tamanho menor. Em contrapartida, quanto menos ocorrências houver de um dado, maior o tamanho do seu código.

Visão geral da codificação *Huffman*

Segundo Salomon (2002, p.12-13), a codificação *Huffman*, elaborada por *David Huffman* em 1952, está entre as codificações mais utilizadas em softwares de compressão de dados. É muito utilizada como método único de compressão e, também, como um passo intermediário de compressão em softwares que combinam métodos diferentes de compressão, além de utilizar o método estatístico e a codificação de tamanho variável para compactar os dados.

Dessa forma, ao combinar o método estatístico com a codificação de tamanho variável, passa a ser um requisito que a codificação *Huffman* siga duas regras importantes: primeiro, a atribuição de códigos menores aos dados mais frequentes e, segundo, a codificação atribuída deve seguir a “propriedade do prefixo”, que explicaremos a seguir.

A atribuição de códigos menores aos dados mais frequêntes é simples. Suponha que haja três caracteres a serem codificados seguindo essa regra de

menor tamanho, a, b e c. Suponha ainda que o caractere a apareça 1000 vezes no arquivo original, o b apareça 30 vezes e o c 5 vezes. Se tivermos à disposição três códigos para serem atribuidos a esses caracteres $(00, 010, 1000)$, é lógico pensar que vamos atribuir o código 00 ao caractere a (pois este é o caractere mais frequente), o código 010 ao caractere b e, por fim, o código 1000 ao caractere c (pois este é o menos frequente).

A propriedade do prefixo, por sua vez, diz que um código já atribuído não pode ser prefixo de outro código. Quando há incosistência de prefixos, há um problema de ambiguidade na decodificação pois ao ser lido da esquerda para a direita, *bit* a *bit*, um código em que houver um mesmo prefixo para caracteres diferentes, o decodificador não saberá determinar a qual caractere essa sequência de *bits* pertence.

Por exemplo, digamos que o caractere a esteja associado ao código binário 01, o caractere b ao código 010 e, o caractere c, ao código 001: nesse caso o caractere a possui um código que é prefixo do caractere b, uma vez que ambos iniciam com a mesma codificação 01. Porém, entre os caracteres a e c, ou, b e c, não há inconsistências pois nenhum é prefixo do outro.

Assim, uma vez que o decodificador receba a sequência de *bits* 010 para ser decodificada, ao ler da esquerda para a direita *bit* a *bit* não saberá dizer se a sequência corresponde ao caractere a, e sobra o último *bit* 0 ou se os três *bits* correspondem ao caractere b. Por isso há a necessidade de uma codificação de tamanho variável seguir estritamente a regra do prefixo.

No tópico seguinte será mostrado como a codificação Huffman gera os códigos de tamanhos variáveis, que é a ideia central da compressão Huffman.

Bem como nos demais algoritmos estudados nesta pesquisa, a codificação *Huffman* será utilizada para compactar dados de arquivos de texto puro.

Algoritmo de compressão pela codificação Huffman

A compressão pelo método *Huffman* pode ser compreendida por dois importantes momentos em que o primeiro se dá pela obtenção da frequência de ocorrência de cada caractere e sua respectiva codificação de tamanho variável e, o segundo, pela compressão do arquivo original utilizando a codificação gerada anteriormente.

Conforme Salomon (2002, pg. 12-13), há três maneiras (mais relevantes) de se obter a frequência dos dados para que seja feita a compressão.

A primeira forma, que foi muito utilizada nas máquinas de *fax*, analisou documentos que poderiam representar muito bem a ocorrência de letras no alfabeto inglês obtendo, com isso, a frequência de cada letra do alfabeto e criando uma tabela padrão de frequências e códigos.

A segunda maneira é realizando a compressão de um arquivo em duas etapas distintas. Na primeira o compressor irá ler todo o arquivo e obter os dados estatísticos necessários de cada caractere e, na segunda, a compressão será realizada com base nesses dados. Este tipo de compressão, segundo Salomon (2002, p. 12), resulta em uma excelente taxa de compressão, muito embora seja um processo significativamente mais lento que o normal. Como a finalidade deste trabalho é a demonstração do funcionamento de cada

algoritmo, será esta abordagem a ser utilizada na demonstração da compressão *Huffman*.

Por fim, a terceira abordagem relevante de compressão dos dados é realizando uma compressão adaptativa, na qual a etapa de obtenção de dados estatísticos e de compressão ocorrem simultaneamente. Nesse modelo, o compressor inicia o processo sem conhecimento prévio do arquivo a ser comprimido e, conforme o arquivo é lido, o compressor vai obtendo dados estatísticos para as análises e compressões seguintes. Devido a isso, a compactação inicial do arquivo é menos eficaz, tornando-se mais expressiva conforme o arquivo original vai sendo compactado, isto é, a codificação vai adaptando-se conforme ela vai progredindo.

Uma vez que os dados estatísticos foram obtidos, ou seja, quando o compressor possui as frequência de ocorrência de cada caractere, então ele iniciará a atribuição dos códigos de tamanho variável a cada caractere e, por fim, compactará o arquivo original.

Para exemplificar o processo de codificação de tamanho variável dos caracteres de um texto, será usada a seguinte cadeia de caracteres

$$cabacebdcaddcbfcbdaaafcabaabcd$$

O processo de codificação inicia-se com a obtenção da frequência de cada caractere no texto. Vamos usar a notação $\varphi(caractere)$ para denotar a frequência de um dado caractere. No caso, $\varphi(a) = 9, \varphi(b) = 6, \varphi(c) = 7, \varphi(d) = 5, \varphi(e) = 1$ *e* $\varphi(f) = 2$.

Em seguida, obtem-se o número total n de caracteres no texto, isto é, $n = 30$. Com isso, temos que as propabiliddes de ocorrência de cada caractere no texto são dadas por

$$P(caractere) = \frac{\varphi(caractere)}{n}$$

Logo,

$$P(a) = \frac{\varphi(a)}{n} = \frac{9}{30} = 0{,}30$$

$$P(b) = \frac{\varphi(b)}{n} = \frac{6}{30} = 0{,}20$$

$$P(c) = \frac{\varphi(c)}{n} = \frac{7}{30} = 0{,}23$$

$$P(d) = \frac{\varphi(d)}{n} = \frac{5}{30} = 0{,}17$$

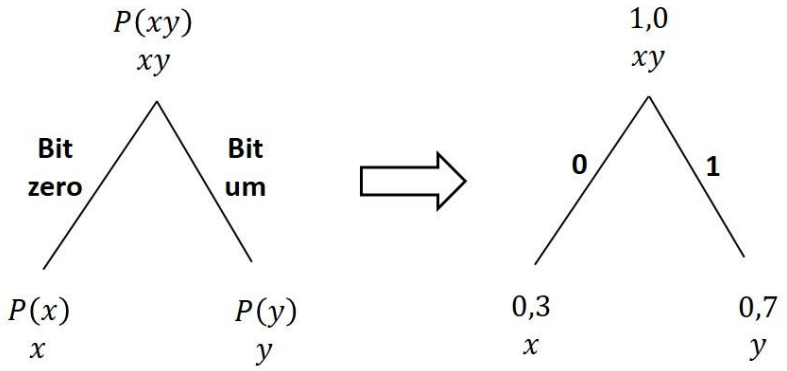
$$P(e) = \frac{\varphi(e)}{n} = \frac{1}{30} = 0{,}03$$

$$P(f) = \frac{\varphi(f)}{n} = \frac{2}{30} = 0{,}07$$

O passo seguinte, após a obtenção das probabilidades, é a atribuição de códigos a cada caractere, devendo-se respeitar as regras do prefixo e do menor código para os símbolos mais frequentes.

Para isso, vamos construir uma árvore binária que terá uma estrutura semelhante ao exemplo abaixo. Neste exemplo, temos a árvore da esquerda,

como modelo teórico, e a árvore da direita, como modelo prático em uma situação de exemplo:



Árvore na computação é um dos diversos tipos de estrutura de dados. Uma árvore que não é vazia possui nós (ou vértices), que armazenam as informações. Um nó particular de uma árvore é a raiz, que é um nó que dá origem às ramificações da árvore. Qualquer nó pode ser considerado raiz e uma árvore pode ser representada como na imagem acima, com a raiz no topo e suas ramificações abaixo, muito embora isso não seja regra. Todo nó que não for a raiz e for uma extremidade da árvore, é chamado de folha e a ligação entre dois nós é chamado de ramificação.

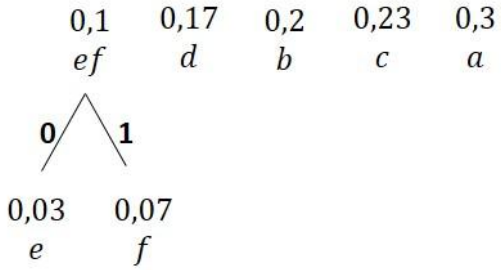
No caso da árvore do exemplo acima, o ponto mais alto corresponde a um nó, raiz. Abaixo de um nó seguem suas respectivas ramificações, à esquerda e à direita. Cada ramificação de um mesmo nó receberá sempre um *bit*, zero ou um. Vamos convencionar o *bit* zero à ramificação da esquerda e o *bit* um à ramificação da direita. Por fim, cada ramificação resultará em uma folha, isto é, a extremidade da árvore. Perceba que cada nó irá gerar duas folhas e, por isso, essa estrutura é chamada de árvore binária.

Na ávore binária para codificação Huffman, o nó ou a raiz irá exibir a soma das probabilidades dos caracteres das respectivas folhas e cada folha irá exibir o caractere que ela armazena e sua probabilidade, sendo a folha da esquerda aquela que conterá sempre a menor probabilidade e a da direita sempre a maior probabilidade. No exemplo prático acima, a raiz da ávore mostrará os caracteres ab com probabilidade total igual a $1{,}0$. Cada folha exibirá, individualmente, esses caracteres e suas respectivas probabildiades que no caso são, para o caractere x, $0{,}3$ e, para o caractere y, $0{,}7$.

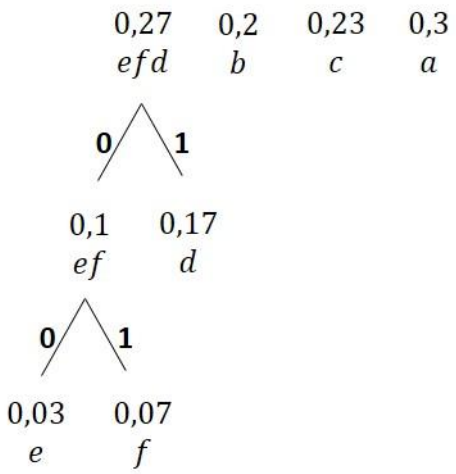
Com esta explicação conseguimos construir, portanto, a árvore binária para o exemplo que estávamos desenvolvendo. Começaremos alinhando, em uma mesma linha, as probabilidades e seus respectivos caracteres em ordem crescente de probabilidade.

0,03	0,07	0,17	0,2	0,23	0,3
e	f	d	b	c	a

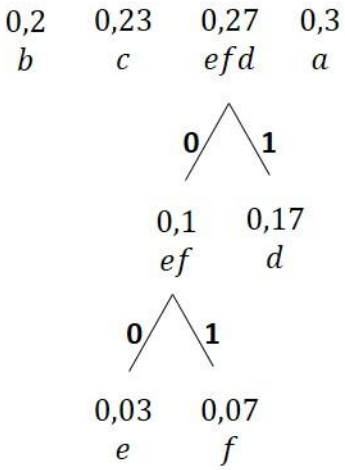
Após a ordenação crescente, a criação de um novo nó será sempre feita utilizando os dois elementos de menor probabilidade. No caso, o primeiro nó será originado à partir dos caracteres e e f, pois ambos possuem as menores probabilidades. Para isso, o novo nó será a união deste caracteres em um mesmo nó cujas folhas serão os próprios caracteres individualmente, seguindo a regra da menor probabilidade à esquerda, como na imagem abaixo:



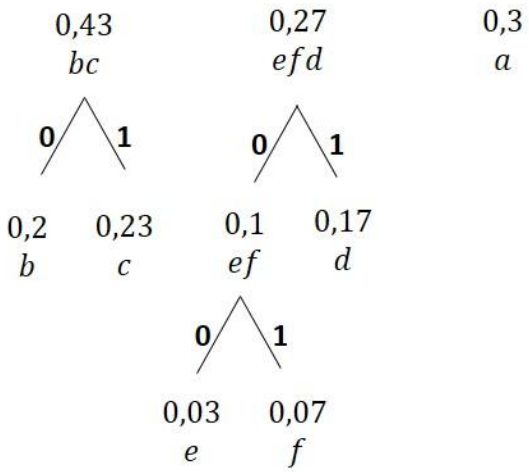
Após a criação de um nó, devemos reordenar novamente a primeira linha de elementos e, então, prosseguir com a criação de nós até que haja apenas um elemento na primeira linha e este seja a raiz da árvore. Dessa forma, como a primeira linha continua ordenada, prosseguiremos com a criação do próximo nó (efd):



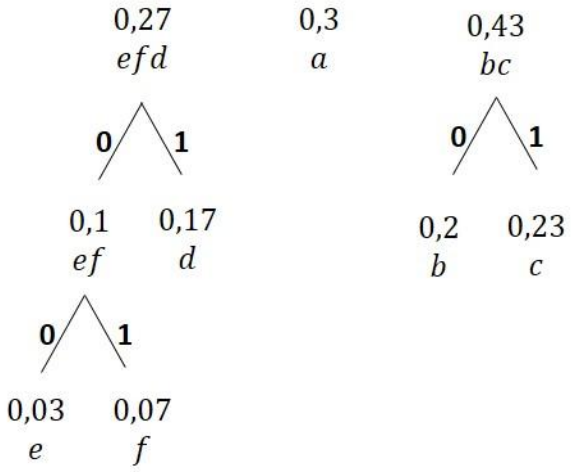
Novamente, reordenaremos a primeira linha por ordem crescente de probabilidades e o resultado será como na imagem a seguir:



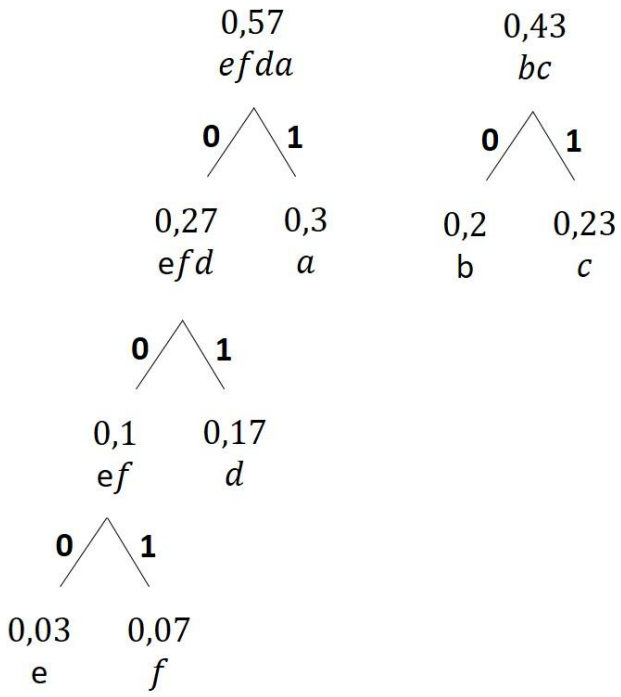
Agora, a criação do novo nó se dará pela união dos dois novos caracteres de menor probabilidade, b e c:



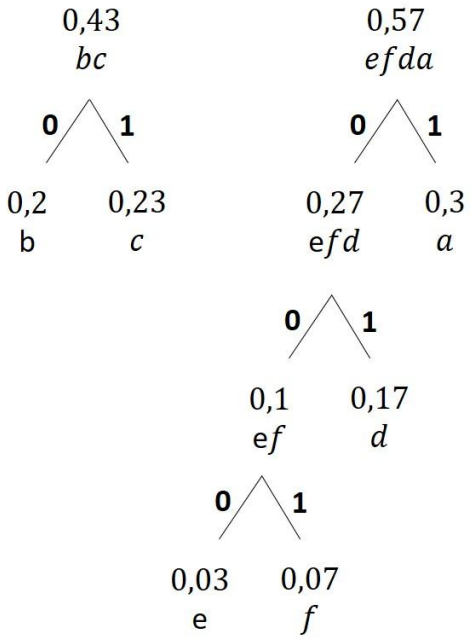
Abaixo, segue a reordenação:



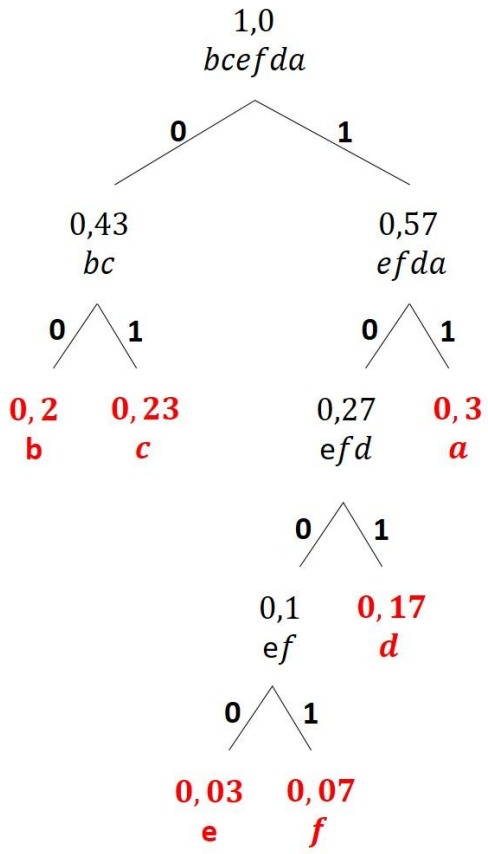
Novamente, a criação do próximo com elementos de menor probabilidade, efd e a:



Após a reordenação ficara como a imagem a seguir:



Por fim, chegamos ao nó raiz com a união dos dois últimos elementos, bc e $efda$:

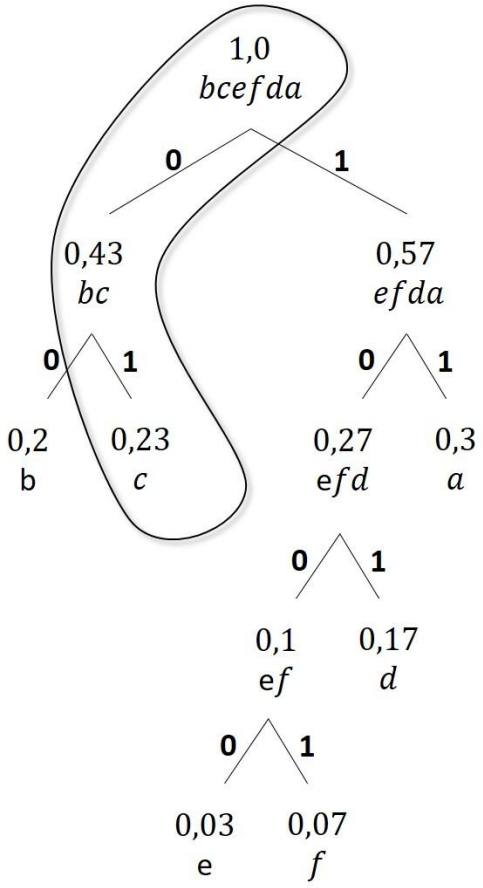


Se notarmos, todos os caracteres para os quais calculamos as probabilidades inicialmente estão em folhas, como na imagem acima:

É exatamente isso que deveria acontecer.

Agora, com a ávore binária de Huffman completamente estruturada (com apenas uma raiz e todos os caracteres em folhas), podemos montar a tabela de caracteres e códigos. Para descobrir o respectivo código de Huffman de um caractere, basta percorrer da raiz até a folha, e ir concatenando os *bits* de cada ramificação passada na sequência em que forem lidos.

Por exemplo, para encontrar o código do caractere c, observe o caminho destacado da imagem abaixo:



O código referente ao caractere c, portanto, é 01. Analogamente todos os outros códigos serão encontrados da mesma forma, originando a tabela abaixo:

Caractere	Probabilidade	Código Huffman
a	0,30	11
c	0,23	01
b	0,20	00
d	0,17	101
e	0,07	1000
f	0,03	1001

É possível perceber, com este método de árvore binária, que os caracteres mais recorrentes são aqueles com códigos de menor tamanho,

enquanto os menos recorrentes possuem os maiores códigos, pois a atribuição de códigos é cumulativa, ou seja, os códigos atribuídos crescem a cada etapa e, pelo fato de as etapas de codificação operarem sempre sobre os caracteres menos frequentes, ao final a acumulação de *bits* será sempre maior nos caracteres menos repetidos. Além disso nenhum dos códigos gerados são prefixos de outros, isto é, não há ambiguidade na codificação por este método porque, como pode-se observar na árvore de codificação, todos os caracteres estão localizados em folhas, garantindo assim a não ambiguidade de códigos, uma vez que para havê-la seria necessário que o caractere não estivesse em uma folha, mas em um nó. Portanto, a atribuição de código por meio da árvore binária obedece às duas regras citadas no início do tópico.

Por fim, com a tabela de codificação construída, basta apenas que o compressor leia o arquivo original, caractere por caractere, e reescreva o arquivo substituindo o caractere lido pela sua respectiva codificação Huffman.

Retomando o texto usado como exemplo para a construção dessa tabela,

$$cabacebdcaddcbfcbdaaafcabaabcd$$

A compressão desta sequência de caracteres utilizando os códigos gerados será a seguinte:

$$01|11|00|11|01|1000|00|101|01|11|101|101|01|00|1001$$

$$|01|00|101|11|11|11|1001|01|11|00|11|11|00|01|101$$

O uso das barras verticais é meramente didático, isto é, não faz parte da compressão de fato. Assim, considere o arquivo final somente com o uso dos *bits* zeros e uns, sem as barras.

A análise da compressão é feita da seguinte maneira: considerando que o texto original possui 30 caracteres ao todo e que cada caractere ocupa um byte de dados, então o texto original ocupa um total de 240 *bits*. A sequência compactada, por sua vez (desconsiderando as barras), ocupa um total de 71 *bits* de dados. Como são necessários 8 *bits* para se obter um byte, então os 71 *bits* compactados ocupam ao todo 9 *bytes* de dados. Portanto, a taxa de compactação para este exemplo foi de 96,3%.

Porém, como foi optado por realizar a compactação em duas etapas distintas (construção da codificação e, em seguida, compactação), a tabela de codificação se faz, logicamente, necessária também para o descompressor. Com isso, é imprescindível que essa tabela, ou apenas os dados estatisticos, sejam gravados junto ao arquivo compactado para que o descompressor consiga decodificar corretamente a sequência binária do arquivo compactado. Isso significa um incremento da ordem de poucos *bytes* (a depender de como o programador grave essas informações no arquivo), não afetando significativamente o tamanho final da compactação.

Algoritmo de Descompressão *Huffman*

A descompressão de um arquivo compactado pela codificação Huffman é simples e exige, neste caso de estudo (em que codificação e compressão ocorrem em etapas distintas), o conhecimento prévio de alguma informação da codificação usada para compactar os dados, como as probabilidades ou os códigos atribuídos aos caracteres.

Se o descompressor receber como dados prévios as probabilidades de cada caractere, então ele terá que montar a árvore de codificação da mesma maneira que foi feita na etapa de compressão vista anteriormente. Do contrário, se receber diretamente a lista de caracteres associados aos códigos, então o descompressor terá o único trabalho de ler o arquivo compactado e encontrar os caracteres correspondentes a esses códigos. Não importa a maneira como o programador irá desenvolver o algoritmo de descompressão, o essencial é que a codificação dos caracteres na compressão seja idêntica à codificação dos caracteres na descompressão.

Assim, como já foi demonstrado no tópico de compressão como a árvore de codificação é construída, partiremos do princípio, então, de que o descompressor já recebeu uma lista de caracteres com suas respectivas probabilidades e que a mesma árvore de codificação Huffman foi recriada.

Com isso, resta apenas que o descompressor leia o arquivo com dados binários (compactado) e o transforme novamente no arquivo original. Para isso, o arquivo compactado será lido *bit* a *bit.* Para cada nova leitura realizada, o descompressor irá concatenar aos *bits* já lidos o novo *bit*. Em seguida, irá verificar se a sequência concatenada de *bits* é prefixo de algum código na lista de codificação de caracteres. Sempre que a sequência de *bits* for prefixo de mais de um código, o descompressor irá ler e concatenar à sequência um novo *bit*, repetindo em seguida a verificação na lista. No momento em que um *bit* concatenado fizer com que a sequência de *bits* seja prefixo de apenas um código na lista de codificação, então o descompressor buscará na tabela o caractere correspondente a esta sequência e o escreverá no arquivo descompactado. Em seguida, a sequência utilizada será descartada e uma

nova sequência de *bits* será iniciada. O processo de verificação de prefixo, concatenação de *bits* e obtenção do caractere continuará até que todos os *bits* do arquivo compactado sejam lidos e o arquivo descompactado seja restaurado.

O processo descrito acima pode ser sintetizado da seguinte forma: a descompressão de uma sequência de *bits* é realizada de tal modo que esses *bits* são lidos e concatenados, um a um, formando uma nova sequência até que esta seja prefixo de apenas uma sequência na tabela que contém os caracteres codificados.

Como exemplo, vamos demonstrar a descompressão da sequência de *bits* gerada anteriormente no tópico de compressão. Visando facilitar a visualização das etapas de descompressão, os bits serão postos dentro de uma tabela (enfileirados em uma mesma linha) e, na linha abaixo, haverá uma enumeração para identificar a posição de cada *bit*.

bits	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**
posição	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*	*33*	*34*	*35*	*36*
bits	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	
posição	*37*	*38*	*39*	*40*	*41*	*42*	*43*	*44*	*45*	*46*	*47*	*48*	*49*	*50*	*51*	*52*	*53*	*54*	*55*	*56*	*57*	*58*	*59*	*60*	*61*	*62*	*63*	*64*	*65*	*66*	*67*	*68*	*69*	*70*	*71*	

Antes de iniciar a descompressão, segue abaixo a tabela de caracteres com seus códigos correspondentes:

Caractere	Probabilidade	Código Huffman
a	0,30	11
c	0,23	01
b	0,20	00
d	0,17	101
e	0,07	1000

f	0,03	1001

A demonstração feita etapa por etapa, da descompressão, será feita apenas para os primeiros caracteres. Os demais que virão na sequência serão descompactados com um menor detalhadamento para que o trabalho não fique extremamente extenso apenas para demonstrar uma repetição grande de passos simples.

Inicia-se a descompressão lendo o *bit* que está na posição um, isto é, o *bit* 0. Após a leitura, verifica-se, na coluna Código Huffman, se este *bit* sozinho é prefixo, ou seja, se algum código é iniciado com ele ou igual a ele. Dentre os seis códigos possíveis há dois códigos que se iniciam pelo *bit* zero, que são os códigos 01 e 00. Portanto, como o *bit* zero sozinho é prefixo, o descompressor irá ler o próximo *bit* do arquivo compactado (na posição 2) e concatená-lo ao *bit* zero já lido. Dessa forma surge uma sequência com dois *bits* resultantes, 01.

Novamente o descompressor irá verificar na coluna código Huffman se a sequência 01 é prefixo de algum código. Dessa vez haverá apenas uma combinação, que é o código 01.

Como explicado anteriormente, o compressor ficarará concatenando *bits* à sequencia de verficação até o momento que o último *bit* concatenado fizer com que a sequência não seja mais encontrada na coluna código Huffman. Devido a isso, então, mais um *bit* do arquivo compactado (posição 3) será lido e concatenado à sequencia 01, resultando portanto na sequência 011. Dessa vez a verificação dessa nova sequência à coluna código Huffman não irá gerar correspondência. Com isso o compressor agora sabe que a sequência anterior,

01, possuia uma correspondência, enquanto a nova sequência 011 não possui nenhuma. Logo, ele irá desconsiderar o último *bit* lido, um, retomará a sequência 01 anterior e recuperará o caracatere que esse sequência codifica na tabela, que é o caractere c. Por fim, escreverá este caractere no arquivo descompactado:

Arquivo descompactado
$\boldsymbol{c}$

Para continuar, o algoritmo irá descartar a sequência 01, que já foi decodificada e irá retomar uma nova sequência à partir do último *bit* lido (aquele que foi descartado no passo anterior), o *bit* 1 da posição 3. Assim, irá verificar se este *bit* 1 é prefixo de algum código na coluna Código de Huffman. Há quatro codificações iniciadas por 1, portanto o proximo *bit* do arquivo compactado (posição 4) será lido e concatenado, resultando na sequência 11. Como resultado de uma nova verificação na coluna Código de Huffman o compressor terá que a sequência 11 também é prefixo de um código e, portanto, uma nova leitura (posição 5) e concatenação irá gerar a sequência 110. Esta, por sua vez, não codificará ninguém na tabela e, como ja foi demonstrado, o último *bit* será descartado temporariamente, retomando a sequência 11. Em seguida, o compressor recuperará o caractere a que é codificado por esta sequência 11 e o escreverá ao final do arquivo compactado:

Arquivo descompactado
$\boldsymbol{ca}$

Continuando, o compressor descartará a sequência 11, já utilizada, e iniciará uma nova sequência a partir do *bit* 0 (posição 5), descartado temporariamente no passo anterior. Enfim, esse processo irá se repetir até que todos os *bits* do arquivo compactado sejam lidos e decodificados na seguinte sequência de caracteres:

Arquivo descompactado
$\boldsymbol{cabacebdcaddcbfcbdaaafcabaabcd}$

Esta sequência de caracteres é exatamente igual à sequência original, ilustrando assim que o algoritmo de codificação Huffman é um compressor sem perdas.

Implementação em Python do algoritmo Codificação *Huffman*

Compressão *Huffman*

Resumidamente, o algoritmo de compressão *Huffman* realizará a compressão de um arquivo por meio de três processos bem definidos: primeiro irá contar as ocorrências de cada caractere no arquivo; em seguida, codificará cada caractere com base em sua recorrência no arquivo, atribuindo códigos menores aos caracteres mais repetidos; por fim, irá escrever no arquivo compactado as informações da codificação dos caracteres (necessárias para a descompressão) e os caracteres do arquivo original de forma codificada.

Até a linha 15 o código está explicado no tópico linhas de código gerais, pois são linhas comuns a outros algoritmos. O programa, nessas linhas, recebe

o arquivo a ser compactado, abre-o e verifica se está vazio. Se não estiver o programa abre um novo arquivo de saída que receberá o conteúdo compactado. Note no algoritmo a seguir que, para uma boa leitura, o código foi marcado com caracteres de comentários (#) a fim de evidenciar os três blocos de códigos que representam os três processos acima descritos.

```
1.  #include: utf-8
2.  import sys
3.  def leCaractereArquivo():
4.      return (arquivo_entrada.read((1))).decode("latin1")
5.  num_argv = 0
6.  for elemento in sys.argv:
7.      num_argv += 1
8.  if num_argv > 1:
9.      arquivo_entrada = open(sys.argv[1], "r")
10. else:
11.     sys.exit("Programa executado sem arquivo a ser compactado")
12. char = leCaractereArquivo()
13. if len(char) == 0:
14.     sys.exit("Arquivo Vazio!")
15. arquivo_saida = open(sys.argv[1] + ".huf", "w")
16. ############# Bloco para sumarizar as ocorrencias #############
17. arquivo_entrada.seek(0)
18. caracteresOcorrencias = {}
19. char= leCaractereArquivo()
20. while char:
21.     if  char in caracteresOcorrencias:
22.         caracteresOcorrencias[char] += 1
```

```
23.      else:
24.          caracteresOcorrencias[char] = 1
25.      char = leCaractereArquivo()
26.  ocorrenciasOrdenadas =  sorted(caracteresOcorrencias.iteritems(),
     key=lambda (chave,valor): (valor,chave))
27.  ############## FIM - Bloco para sumarizar as ocorrencias #############
28.  ############## Bloco para codificar os caracteres #############
29.  caracteresCodificados = {}
30.  for ocorrencia in ocorrenciasOrdenadas:
31.      caracteresCodificados[ocorrencia[0]] = ""
32.  while len(ocorrenciasOrdenadas) > 1:
33.      parOrdenado0 = ocorrenciasOrdenadas.pop(0)
34.      parOrdenado 1 = ocorrenciasOrdenadas.pop(0)
35.      itemAgrupado = []
36.      for caractere in parOrdenado0[0]:
37.          itemAgrupado.append(caractere)
38.      for caractere in parOrdenado1[0]:
39.          itemAgrupado.append(caractere)
40.      ocorrenciasOrdenadas.insert(0,  (itemAgrupado, parOrdenado0[1]+
         parOrdenado1[1]))
41.      ocorrenciasOrdenadas.sort(key=lambda x: x[1])
42.      for caractere in parOrdenado0[0]:
43.          caracteresCodificados[caractere] = '0' +
             caracteresCodificados[caractere]
44.      for caractere in parOrdenado1[0]:
45.          caracteresCodificados[caractere] = '1' +
             caracteresCodificados[caractere]
46.  ############## FIM - Bloco para codificar os caracteres #############
47.  ############## Bloco para compactar o arquivo #############
48.  for (caractere, codigo) in sorted(caracteresCodificados.iteritems(),
     key=lambda (caractere, codigo): (len(codigo))):
49.      arquivo_saida.write((caractere).encode('latin1') + '_' + codigo +
         '|')
```

```
50.  arquivo_saida.write("}")
51.  arquivo_entrada.seek(0)
52.  char = ""
53.  char = leCaractereArquivo()
54.  while char :
55.      charCodificado = caracteresCodificados[char]
56.      arquivo_saida.write((charCodificado).encode('latin1') )
57.      char = leCaractereArquivo()
58.  ############# FIM - Bloco para compactar o arquivo #############
59.  arquivo_entrada.close()
60.  arquivo_saida.close()
```

Entre as linhas 16 e 27 está contido o código destinado a sumarizar as ocorrências dos caracteres do arquivo original e esse processo é feito através da leitura completa do arquivo de entrada a ser compactado, caractere por caractere, realizando uma contagem do número de ocorrências de cada caractere e armazenando estes dados em uma estrutura de dados do tipo dicionário, em que cada registro do dicionário é composto de dois atributos: a chave e o valor. Neste caso, a chave é o caractere e o valor é o número de ocorrências deste caractere no arquivo de entrada, por exemplo $\{"a" : 10, "b" : 15, "c" : 20\}$. Uma observação a ser feita sobre as estruturas de dicionários é que elas possuem comportamentos semelhantes aos de um vetor, pois é possível percorrer e realizar uma busca em um dicionário do mesmo modo que em um vetor (linha 18), onde a chave é o índice (mesmo sendo caractere ou *string*) e o valor é o conteúdo do índice, além de ser possível declará-lo incialmente vazio, como na linha 18.

Na linha 17 o programa faz com que o cursor de leitura do arquivo seja posicionado no início do arquivo de entrada.

Na linha 18 a estrutura de dicionário para armazenar a contagem dos caracteres é criada como um vetor de nome $caracteresOcorrencias$, cujo conteúdo inicial é vazio.

O primeiro caractere do arquivo é lido, na linha 19, e armazenado na variável $char$.

Da linha 20 à 25 um laço é iniciado e percorrerá o arquivo de entrada, *byte* a *byte,* até que o final do arquivo seja encontrado. Isso ocorrerá quando o conteúdo da variável $char$ for igual a uma string vazia, isto é, $char = ""$. Quando isso ocorrer, a condição da instrução $while$ será falsa e o laço será encerrado.

A cada caractere lido, dentro do laço acima, será verificado se esse caractere já fora inserido na estrutura de contagem $caracteresOcorrencias$, na linha 21. Se o caractere já existir, então sua contagem será incrementada de uma unidade, na linha 22. Do contrário, se o caractere ainda não existir, então ele será inserido e sua contagem inicial será 1 (linha 24).

Por fim, o próximo caractere do arquivo de entrada é lido, na linha 25, e o laço é repetido novamente.

Na linha 26, ao término de execução do laço, uma variável do tipo lista denominada $ocorrenciasOrdenadas$ é criada e seu conteúdo é uma lista de pares ordenados (extraídos da variável $caracteresOcorrencias$) de caracteres e números de ocorrências, ordenados de forma crescente pelo número de

ocorrência. Por exemplo: $ocorrenciasOrdenadas = [("a", 10), ("b", 15), ("c", 20)]$.

Ainda na linha 26, para realizar a ordenação das ocorrências, na função *sorted*, um recurso muito interessante do *Python* foi utilizado, que é a função *lambda.* Função *lambda* é como uma função comum, porém anônima, isto é, não possui um nome particular. Recebe valores como argumento, possui uma restrição de suportar apenas uma expressão de código e retorna um valor após sua execução. Esse tipo de função é muito utilizada como argumento de outras funções, como é o caso de sua utilização dentro da função *sorted.* Sua estrutura é do tipo *lambda (argumentos) : expressão de retorno*, onde a palavra *lambda* é obrigatória, *argumentos* são os valores que se deseja passar à função e, por fim, *expressão de retorno* é a operação que a função irá executar e retornar a quem a chamou. No caso específico da linha 26, a função está definida como *lambda (chave, valor) : (valor, chave)*, isto é, a função recebe como parâmetro um par ordenado $(chave, valor)$ e retorna um novo par ordenado invertido, $(valor, chave)$.

Esta variável $ocorrenciasOrdenadas$ servirá de base para a atribuição de códigos de menor tamanho aos caracteres mais recorrentes, processo este que será detalhado a seguir.

O processo de codificação dos caracteres será realizado entre as linhas 28 e 46 do programa. De forma sintética, o programa varrerá a lista ordenada obtida anteriormente, $ocorrenciasOrdenadas$, e, repetidamente, realizará o seguinte processo: irá retirar da lista os dois primeiros pares ordenados e, em seguida, os agrupará em um novo par ordenado cuja primeira posição será

uma lista contendo a união dos caracteres de cada par ordenado retirado e a segunda posição será a soma entre os números de ocorrências. Em seguida, adicionará esse novo par ordenado agrupado de volta à lista $ocorrenciasOrdenadas$ e a ordenará novamente seguindo o mesmo critério da ordenação anterior. Por exemplo: a variável $ocorrenciasOrdenadas = [("a", 10), ("b", 15), ("c", 20)]$ após passar pelo processo descrito resultará em $ocorrenciasOrdenadas = [("c", 20), (["a", "b"], 25)]$. Por último, retomando os dois pares ordenados incialmente retirados, varrerá a lista de caracteres de cada par e irá atribuir o *bit* 0 aos caracteres do primeiro par ordenado (aquele com menor número de ocorrencias) e o *bit* 1 aos caracteres do segundo par ordenado. Por exemplo: o primeiro par ordenado é $("a", 10)$ e $("b", 15)$. Portanto, o *bit* 0 é designado ao caractere a e, o *bit* 1, ao b.

Para isso, na linha 29 uma nova estrutura de dados do tipo dicionário é criada sem conteúdo e com o nome $caracteresCodificados$. Essa estrutura armazenará os caracteres como chave e os códigos gerados para cada caractere como valor, isto é, $\{caractere : código\}$.

Nas linhas 30 e 31 um laço percorrerá cada par ordenado da lista $ocorrenciasOrdenadas$ e irá inserir cada caractere desses pares ordenados na estrutura $caracteresCodificados$, porém com seus respectivos valores no formato de *string* vazia, isto é ("").

Entre as linhas 32 e 45 um novo laço irá realizar todo o processo de codificação descrito qualitativamente acima. As linhas 33 e 34 atribuem os nomes de variáveis $parOrdenado0$ e $parOrdenado1$ aos pares ordenados retirados da lista $ocorrenciasOrdenadas$. As linhas 35 a 41 realizam todo o

processo de agrupamento dos pares ordenados, a adição do par agrupado novamente à lista $ocorrenciasOrdenadas$ e a ordenação dessa lista. Por fim, as linhas 42 a 45 atribuem os *bits* 0 e 1 aos caracteres de cada par ordenado na estrutura $caracteresCodificados$.

O último bloco de código, composto pelas linhas 48 a 60, é o responsável pela escrita das informações no arquivo compactado. Primeiramente o programa irá gravar todos os caracteres e seus respectivos códigos para, em seguida, ler novamente todo o arquivo de entrada a ser compactado, caractere a caractere, e escrever no arquivo de saída seu rescpectivo código.

Para isso, na linha 48 o programa iniciará um laço percorrendo todos os caracteres codificados da variável $caracteresCodificados$. Para cada caractere, o programa irá gravar a estrutura "$caractere\_código|$" no arquivo de saída. De forma mais genérica, a estrutura gravada no arquivo será "$caractere1\_código1|caractere2\_código2|...|caractereN\_códigoN|$".

Ao final da gravação de todos os caracteres e seus códigos será gravado o caractere "}" (linha 50), que terá a finalidade de informar ao descompresssor que ali encerram-se as informações da codificação dos caracteres e iniciam-se as informações do arquivo compactado.

Em seguida, entre as linhas 54 e 57 o programa irá ler um caractere por vez do arquivo original e encontrar seu código correspondente em $caracteresCodificados$, escrevendo o no arquivo de saída, até que o fim do arquivo original seja encontrado.

Por último, o programa fecha o arquivo de entrada e o arquivo de saída, nas linhas 59 e 60, e é encerrado.

Descompressão *Huffman*

Bem como a compressão *Huffman,* o algoritmo de descompressão é constituído de duas importantes etapas: a primeira é a rescontrução da estrutura que armazena os caracteres codificados e a segunda é a descompactação em si.

Para as linhas 1 e 2 e de 5 a 15 o código está explicado no tópico linhas de código gerais, pois são linhas comuns a outros algoritmos. O programa, nessas linhas, recebe o arquivo a ser compactado, abre-o e verifica se está

```
1.  #include: utf-8
2.  import sys
3.  def leCaractereArquivo():
4.      return (arquivo_entrada.read((1))).decode("latin1")
5.  num_argv = 0
6.  for elemento in sys.argv:
7.      num_argv += 1
8.  if num_argv > 1:
9.      arquivo_entrada = open(sys.argv[1], "r")
10. else:
11.     sys.exit("Programa executado sem arquivo a ser compactado")
12. char = leCaractereArquivo()
13. if len(char) == 0:
14.     sys.exit("Arquivo Vazio!")
15. arquivo_saida = open(sys.argv[1] + "uhuf", "w")
16. ########## Bloco para remontar a estrutura de codificacao ############
17. tempDadosCodificacao = ""
18. while  char != '}':
19.     tempDadosCodificacao += char
```

á

```
20.      char = leCaractereArquivo()
21.  tempDadosCodificacao = tempDadosCodificacao[:len(tempDadosCodificacao)
     - 1]
22.  tempDadosCodificacao = tempDadosCodificacao.split('|')
23.  caracteresCodificados = []
24.  for dado in tempDadosCodificacao:
25.      caracteresCodificados.append( (dado.split('_')[0],
         dado.split('_')[1]) )
26.  ######## FIM - Bloco para remontar a estrutura de codificacao ########
27.  ##################### Bloco para descompactar ##########################
28.  char = leCaractereArquivo()
29.  codigoMontado = char
30.  while char:
31.      listaCodigosEncontrados = []
32.      codigoEncontrado = ""
33.      for codigo in caracteresCodificados:
34.          if codigoMontado == codigo[1][:len(codigoMontado)]:
35.              listaCodigosEncontrados.append(codigo)
36.      if len(listaCodigosEncontrados) == 1:
37.          codigoEncontrado = listaCodigosEncontrados[0][0]
38.          arquivo_saida.write(codigoEncontrado.encode('latin1'))
39.          codigoMontado = ""
40.      char = leCaractereArquivo()
41.      codigoMontado += char
42.  ################## FIM - Bloco para descompactar ###################
43.  arquivo_entrada.close()
44.  arquivo_saida.close()
```

As linhas 3 e 4 definem a função $leCaractereArquivo()$ que será utilizada para encapsular o código de leitura de caracteres do arquivo e facilitar a leitura do programa.

Entre as linhas 17 e 25 encontra-se o bloco de código designado a reconstruir a estrutura de dados que armazena a relação entre os caracteres e seus respectivos códigos binários.

Na linha 17 é criada uma variável denominada $tempDadosCodificacao$, inicialmente com uma *string* vazia, que receberá uma *string* única, lida *byte* a *byte* do arquivo compactado, *correspondendo* a codificação dos caracteres.

As linhas 18, 19 e 20 correspondem ao laço que o programa executará para ler *byte* por *byte* do arquivo compactado, concatenando-os à variável $tempDadosCodificacao$, até que seja encontrado o caractere que sinaliza o fim da leitura dos dados da codificação, o caractere “}”.

Ao final do laço, a *string* $tempDadosCodificacao$ possuirá a seguinte estrutura genérica: "$caractere1\_código1|\ldots|caractereN\_códigoN|$".

A linha 21 retira apenas a última barra vertical da estrutura acima para que, na linha 22, a própria variável transforme a única *string* acima em uma lista através do separador barra vertical, que servirá como referência para a quebra da *string* em itens. Assim, a variável $temDadosCodificacao = [caractere1\_código1\ ,\ caractere2\_código2\ ,\ldots\ ,\ caractereN\_códigoN]$.

Na linha 23 uma lista com conteúdo inicial vazio é criada com o nome $caracteresCodificados$.

Nas linhas 24 e 25, por fim, um laço percorrerá cada posição da variável $tempDadosCodificacao$ a fim de obter os pares ordenados "$(caractere, código)$" e inserindo-os na lista $caracteresCodificados$. Dessa

forma, ao final do laço, a lista $caracteresCodificados$ conterá $[(caractere1, código1),\ (caractere2,$

$código2), \ldots, (caractereN, códigoN)]$.

A descompactação do arquivo original, de fato, está contida no bloco final de código entre as linhas 28 e 41. Neste bloco o programa irá realizar um laço de leituras sucessivas do arquivo a ser descompactado, composto apenas por *bits*, e a cada leitura irá buscar na lista $caracteresCodificados$ todos os códigos que são iniciados pelo *bit* lido (individual) ou pela sequência de *bits* lidos (o último *bit* lido concatenado aos bits lidos anteriormente). Sempre que encontrar na lista de $caracteresCodificados$ mais de uma correspondência que seja iniciada pela sequência de verificação, o programa realziará uma nova leitura, concatenando um novo *bit* à sequência atual, e realizará uma nova consulta. Esse processo será realizado até que a consulte retorne apenas uma correspondência, significando então que foi encontrado o caractere correto que descompacta esta sequência de *bits.* E este processo será realizado até o fim do arquivo a ser descompactado.

A linha 28 lê o primeiro caractere a ser descompactado, que é um *bit.*

A linha 29 inicializa uma variável chamada $codigoMontado$ com este primeiro *bit* lido. Esta variável receberá as sequências de *bits* lidos citada na explicação acima.

A linha 30 inicia o laço de leitura dos dados do arquivo compactado.

As linhas 30 e 31 inicializam duas variáveis inicialmente vazias denominadas $listaCodigosEncontrados$ e $codigoEncontrado$.

As linhas 33, 34 e 35 varrem a lista $caracteresCodificados$ e verificam se algum código dessa lista é iniciado pelo conteúdo de $codigoMontado$. Se for, adiciona na lista $listaCodigosEncontrados$ esta correspondência. Todas as correspondências encontradas serão adicionadas nessa lista.

A linha 36 verifica quantos códigos iniciados por $codigoMontado$ foram encotrados. Se foram encontrados mais de um, então o bloco das linhas 37, 38 e 39 não é executado e mais um *bit* é lido e concatenado à variável $codigoMontado$, reiniciando o processo de consulta. Porém, se apenas um código é encontrado, então o caractere correspondente a este código é gravado no arquivo descompactado (linha 38) e, em seguida, a o conteúdo da variável $codigoMontado$ é esvaziado para que um nova sequência de *bits* seja descompactada.

Este processo irá ocorrer até que o fim do arquivo compactado seja encontrado. Ao final, as linhas 43 e 44 fecham os arquivos de entrada e de saída.

Transformação de *Burrows* e *Wheeler*

Visão geral da transformação

A transformação de *Burrows e Wheeler* não configura, de fato, uma técnica de compressão de dados. Segundo Burrows e Wheeler (1994), a transformação, na verdade, faz com que dados sejam manipulados com a finalidade de se obter uma nova disposição que favoreça o uso de determinadas técnicas de compressão com mais eficácia, como a *run-length encoding*, a *move-to-front* e a codificação *Huffman*. Por isso chama-se

transformação, pois altera o estado de agregação dos dados buscando reorganizá-los de uma maneira que se atinja um potencial maior de compressão em relação ao estado de agregação anterior.

De acordo com Salomon (2007, p. 853), a ideia fundamental da transformação de *Burrows e Wheeler* é, à partir de uma *string* inicial **S**, obter uma nova *string* final **L** com os mesmos caracteres e a mesma quantidade, diferindo apenas na posição deles. A principal característica dessa nova *string* **L** é a tendência em concentrar em uma mesma região da string os caracteres iguais, além de ser possível, a partir da dela**,** reconstruir a *string* **S** em sua totalidade**.** Em suma, a transformação de BW apenas transforma uma sequência de caracteres em outra e, a partir desta outra sequência gerada, retoma a sequência original sem perdas.

Como dito, a transformação BW não é um processo de compressão, porém, se necessário, um algoritmo de compactação pode ser encaixado justamente no momento em que a *string* **S** é transformada na *string* **L.** A transformação de BW com o uso da compressão se dá realizando a compactação da *string* **L** por meio de uma ou mais técnicas distintas de compressão.

Algoritmo da transformação Burrows e Wheeler

Esta subseção é baseada nos autores Sayood (2006, p. 152 - 156) e Salomon (2007, pg. 853 - 856).

A transformação BW trabalha com blocos de dados (no caso deste trabalho, caracteres). O algoritmo de transformação lê blocos de caracteres

(isto é, uma *string*) e, para cada *string* $\boldsymbol{S}$ lida, transforma-a em uma nova *string* $\boldsymbol{L}$ através de etapas a serem descritas a seguir.

O processo se inicia com uma *string* $\boldsymbol{S}$ composta de $\boldsymbol{n}$ caracteres. Em seguida, criam-se $\boldsymbol{n-1}$ novas *strings,* a partir da string $\boldsymbol{S}$, onde cada uma dessas $\boldsymbol{n-1}$ sequências sejam uma permutação circular da *string* $\boldsymbol{S}$.

Por exemplo, suponha uma *string* $\boldsymbol{S}$ cujo conteúdo seja $paralelepipedo$, com $\boldsymbol{n = 14}$**.** As outras 13 *strings* derivadas são obtidas apenas retirando o primeiro caractere de sua posição e colocando-o ao final. Assim, a partir de $paralelepipedo$ surge a primeira permutação, $aralelepipedop$, que, por sua vez, dá origem à segunda permutação, $ralelepipedopa$. E assim até a décima terceira permutação, cuja *string* é $oparalelepiped$.

Vamos reunir todas essas *strings* em uma tabela de $\boldsymbol{n}$ linhas:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
0	**p**	**a**	**r**	**a**	**l**	**e**	**l**	**e**	**p**	**i**	**p**	**e**	**d**	**o**
1	a	r	a	l	e	l	e	p	i	p	e	d	o	p
2	r	a	l	e	l	e	p	i	p	e	d	o	p	a
3	a	l	e	l	e	p	i	p	e	d	o	p	a	r
4	l	e	l	e	p	i	p	e	d	o	p	a	r	a
5	e	l	e	p	i	p	e	d	o	p	a	r	a	l
6	l	e	p	i	p	e	d	o	p	a	r	a	l	e
7	e	p	i	p	e	d	o	p	a	r	a	l	e	l
8	p	i	p	e	d	o	p	a	r	a	l	e	l	e
9	i	p	e	d	o	p	a	r	a	l	e	l	e	p
10	p	e	d	o	p	a	r	a	l	e	l	e	p	i
11	e	d	o	p	a	r	a	l	e	l	e	p	i	p
12	d	o	p	a	r	a	l	e	l	e	p	i	p	e
13	o	p	a	r	a	l	e	l	e	p	i	p	e	d

Após gerar as $\boldsymbol{n-1}$ permutações circulares e dispor todas em uma tabela de $\boldsymbol{n}$ linhas e $\boldsymbol{n}$ colunas, é preciso ordenar lexicograficamente a tabela por inteira.

Ordenar lexicograficamente a tabela significa colocar cada linha em ordem alfabética levando em conta todas as suas colunas. Por exemplo, a linha 0 e a linha 1 possuem, respectivamente, as *strings* $paralelepipedo$ e $aralelepipedop$. Neste caso, a ordem lexica (alfabética) entre as duas *strings* é $aralelepipedop$ e, depois, $paralelepipedo$. Porém quando o primeiro caractere de duas ou mais *strings* são iguais, a comparação lexica deve prosseguir aos demais caracteres em sequência, como é o caso das linhas 1 e 3 em que $aralelepipedop$ e $alelepipedopar$ começam com o mesmo caractere. Neste caso a ordenação será determinada pelo segundo caractere de cada *string,* que é r na linha 1, e l na linha 3. Logo, a ordem lexica entre essas duas *strings* é $alelepipedopar$ e, depois, $aralelepipedop$.

Após a ordenação lexicográfica, a tabela possuirá a seguinte disposição:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
0	a	l	e	l	e	p	i	p	e	d	o	p	a	**r**
1	a	r	a	l	e	l	e	p	i	p	e	d	o	**p**
2	d	o	p	a	r	a	l	e	l	e	p	i	p	**e**
3	e	d	o	p	a	r	a	l	e	l	e	p	i	**p**
4	e	l	e	p	i	p	e	d	o	p	a	r	a	**l**
5	e	p	i	p	e	d	o	p	a	r	a	l	e	**l**
6	i	p	e	d	o	p	a	r	a	l	e	l	e	**p**
7	l	e	l	e	p	i	p	e	d	o	p	a	r	**a**
8	l	e	p	i	p	e	d	o	p	a	r	a	l	**e**
9	o	p	a	r	a	l	e	l	e	p	i	p	e	**d**
10	**p**	**a**	**r**	**a**	**l**	**e**	**l**	**e**	**p**	**i**	**p**	**e**	**d**	**o**
11	p	e	d	o	p	a	r	a	l	e	l	e	p	**i**
12	p	i	p	e	d	o	p	a	r	a	l	e	l	**e**
13	r	a	l	e	l	e	p	i	p	e	d	o	p	**a**

Por definição, a *string* **L** (transformada à partir da *string* **S**) é dada pela sequência de caracteres que compõem a última coluna da tabela ordenada. No exemplo dado, a *string* **L** é a coluna 14, sendo a sequência $rpepllpaedoiea.$

Conforme Burrows e Wheeler (1994, p. 5), na *string* $\boldsymbol{L}$**,** a probabilidade de um caractere qualquer ocorrer em um certo ponto é alta se ele certamente ocorrer próximo a este ponto. Do contrário, se esse caractere não ocorrer em uma região próxima então a probabilidade dele ocorrer neste ponto é baixa. Este é o motivo pelo qual a string $\boldsymbol{L}$ é utilizada como resultado da transformação.

Por fim, a útlima informação necessária a ser guardada ($\boldsymbol{I}$) é o número da linha, na tabela ordenada, em que a sequência original (*string* $\boldsymbol{S}$) aparece. No exemplo dado a linha da tabela ordenada que contém a *string* $paralelepipedo$ é a linha de número 10, isto é, $\boldsymbol{I} = 10$.

Até aqui temos, então, a *string* $\boldsymbol{S} = paralelepipedo$, a *string* $\boldsymbol{L} = rpepllpaedoiea$ e o índice $\boldsymbol{I} = 10$. Esse é o resultado do processo de transformação de *Burrows* e *Wheeler*.

Se houver a necessidade de compactação, ela será realizada sobre a *string* $\boldsymbol{L}$. Como este tópico é focado na transformação BW apenas, não entraremos em detalhes de como compactar a *string* $\boldsymbol{L}$, isso ficará a cargo do próximo tópico, algoritmo bzip2, que usa a transformação BW associada a técnicas de compressão.

Voltando à transformação BW, os passos que serão descritos a seguir representam o processo inverso da transformação ou, melhor dizendo, representam a retomada da *string* $\boldsymbol{S}$ a partir da *string* $\boldsymbol{L}$ obtida.

Para realizar o processo inverso as informações necessárias são a *string* $\boldsymbol{L}$*, o* índice $\boldsymbol{I}$ e a *string* representada pela primeira coluna da tabela ordenada, que vamos chamar de *string* $\boldsymbol{F}$*.* A *string* $\boldsymbol{F}$ pode ser obtida à partir da primeira coluna (coluna 0) da tabela ordenada ou, caso a tabela não esteja mais acessível, basta ordenar lexicograficamente a própria *string* $\boldsymbol{L}$*. Portanto* $\boldsymbol{F} = aadeeeillopppr$

Note que as *strings* $\boldsymbol{L}, \boldsymbol{F}$ e $\boldsymbol{S}$ possuem uma relação entre si, pois uma é permutação da outra. Dado isso, podemos usar $\boldsymbol{L}$ e $\boldsymbol{F}$ para recriar a *string* $\boldsymbol{S}$**.** Sabendo que a sequência original encontra-se na linha número $\boldsymbol{I}$ da tabela ordenada, iniciaremos a reconstrução da *string* $\boldsymbol{S}$ a partir da posição $\boldsymbol{I}$ na *string* $\boldsymbol{F}$*.*

Temos que:

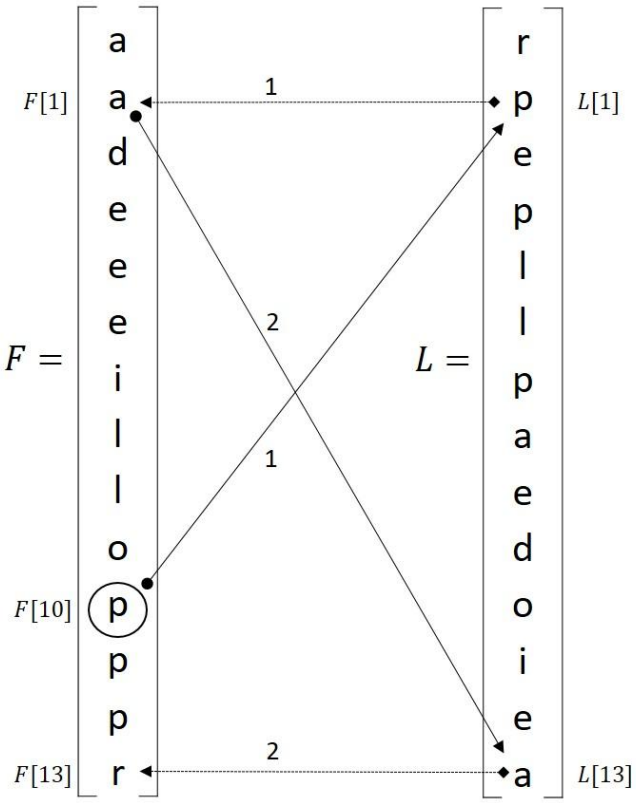
$$F = \begin{bmatrix} a \\ a \\ d \\ e \\ e \\ e \\ i \\ l \\ l \\ o \\ p \\ p \\ p \\ r \end{bmatrix} \qquad L = \begin{bmatrix} r \\ p \\ e \\ p \\ l \\ l \\ p \\ a \\ e \\ d \\ o \\ i \\ e \\ a \end{bmatrix}$$

Os caracteres de restauração da *string* $\boldsymbol{S}$ virão de $\boldsymbol{F}$ e a posição de busca desses caracteres da *string* $\boldsymbol{S}$ virá de $\boldsymbol{L}$.

O processo é iniciado obtendo-se o primeiro caractere da *string* $\boldsymbol{S}$, que está localizado na posição $F[I]$, ou seja, $F[10]$. Com isso, até o momento $\boldsymbol{S} = \mathrm{p}$. Para encontrar o caractere seguinte ao caractere p, buscamos as posições em que o caractere p aparece na *string* $\boldsymbol{L}$. Há 3 posições para p na *string* $\boldsymbol{L}$, em $L[1], L[3]$ e $L[6]$. Deve-se escolher a mesma ordem relativa de p em $\boldsymbol{L}$ que o p em $\boldsymbol{F}$, ou seja, na string $\boldsymbol{F}$ o caratere p que estamos usando é o primeiro dos três $p's$ possíveis e, por isso, devemos escolher o primeiro dos três $p's$ na *string* $\boldsymbol{L}$, logo o obtemos $L[1]$. Por fim, o próximo caractere da sequência em restauro, $\boldsymbol{S}$, está na sequência $\boldsymbol{F}$ na mesma posição da sequência $\boldsymbol{L}$ que acabamos de encontrar, isto é, como encontramos $L[1]$, então em $\boldsymbol{F}$ é posição $F[1] = a$, e agora $\boldsymbol{S} = pa$.

Novamente, reiniciamos o processo acima buscando na *string* $\boldsymbol{L}$ o caractere a (o último obtido em $\boldsymbol{S}$). Há duas posições para a na *string* $\boldsymbol{L}$, $L[7]$ e $L[13]$. Como feito anteriormente, deve-se escolher o caractere em $\boldsymbol{L}$ de mesma posição relativa que ocorre em $\boldsymbol{F}$. Como em $\boldsymbol{F}$ o caractere a que está em uso é o último dos dois possíveis, devemos escolher o último caractere a em $\boldsymbol{L}$, isto é, na posição 13. Com isso, o próximo caractere da sequência $\boldsymbol{S}$ localiza-se na posição 13 da *string* $\boldsymbol{F}$, ou seja, $F[13] = r$. Assim, a string $\boldsymbol{S}$, até o momento, possui o conteúdo par.

A imagem a seguir mostra o caminho feito, a partir do início, até a obtenção do terceiro caractere, como descrito acima:



Esse processo deve ser repetido até que o último caractere disponível na *string* $\boldsymbol{F}$ seja lido. Quando isso ocorrer, a *string* $\boldsymbol{S}$ armazenará o conteúdo $paralelepipedo$.

Assim, fica exemplificado como a transformação de *Burrows* e *Wheeler* opera e que, também, ela por si só não representa uma compressão de dados, apenas uma transformação de dados de um estado para outro.

Implementação em Python do algoritmo de transformação *Burrows e Wheeler*

Transformação *Burrows e Wheeler* (*BWT*)

O algoritmo de transformação *Burrows e Wheeler,* de forma simplificada, irá realizar leituras de sequências S de caracteres do arquivo de entrada a ser

transformado. O tamanho dessas sequências será previamente definido (sendo adotado neste trabalho como 1024 caracteres) e as leituras do arquivo resultarão em sequências com no máximo este tamanho. Quando a sequência S atingir este tamanho máximo ou quando não atingir o tamanho máximo mas for o fim do arquivo, então o processo de transformação será executado sobre esta sequência. Uma matriz de permutações será criada contendo todas as N permutações circulares de S e, em seguida, essa matriz será ordenada lexicograficamente. Depois uma nova sequência L será criada contendo apenas os útlimos caracteres da matriz ordenada e o índice I será obtido com a posição em que a sequência S aparece na matriz de permutações. Por fim, o alogoritmo irá gravar tanto o índice I quanto a sequencia L no arquivo de saída, transformado.

Até a linha 14 o código está explicado no tópico linhas de código gerais, pois são linhas comuns a outros algoritmos. O programa, nessas linhas, recebe o arquivo a ser compactado, abre-o e verifica se está vazio. Se não estiver o programa abre um novo arquivo de saída que receberá o conteúdo compactado.

```
1.  #include: utf-8
2.  import sys
3.  num_argv = 0
4.  for elemento in sys.argv:
5.      num_argv += 1
6.  if num_argv > 1:
7.      arquivo_entrada = open(sys.argv[1], "r")
8.  else:
9.      sys.exit("Programa executado sem arquivo a ser compactado")
```

```
10.  char = arquivo_entrada.read(1).decode("latin1")
11.  if len(char) == 0:
12.      sys.exit("Arquivo Vazio!")
13.  arquivo_saida = open(sys.argv[1] + "bwt", "w")
14.  arquivo_entrada.seek(0)
15.  tamanhoString = 1024
16.  stringS = ""
17.  i = 1
18.  stringS = arquivo_entrada.read(1).decode("latin1")
19.  while stringS:
20.      if i == tamanhoString:
21.          matrizPermutacoes = []
22.          stringL = ""
23.          for j in range(len(stringS)):
24.              permutaStringS = stringS[1:] + stringS[0]
25.              matrizPermutacoes.append(permutaStringS)
26.              stringS = permutaStringS
27.          matrizPermutacoes.sort()
28.          indiceI = matrizPermutacoes.index(stringS)
29.          for permutacao in matrizPermutacoes:
30.              stringL += permutacao[len(permutacao) - 1]
31.          arquivo_saida.write("{" + str(indiceI) + "}" +
             (stringL).encode('latin1'))
32.          stringS = ""
33.          i = 0
34.      stringS +=  arquivo_entrada.read(1).decode("latin1")
35.      i += 1
36.  arquivo_entrada.close()
37.  arquivo_saida.close()
```

A linha 15 estabelece o tamanho máximo da *string* criando a variável $tamanhoString$.

A linha 16 inicializa a variável $stringS$ vazia.

A linha 17 inicializa uma variável contadora i, inicialmente igual a 1, que serivrá para dizer se $stringS$ já atingiu o tamanho máximo ou se não atingiu mas o fim do arquivo foi encontrado.

A linha 18 inicializa a variável $stringS$ com o primeiro caractere do arquivo a ser transformado.

Entre as linhas 19 e 35 o laço de leitura do arquivo a ser transformado é executado.

As linhas 20 a 33 somente serão executadas caso a variável contadora i tenha atingido o tamanho máximo. Se não atingiu, o bloco é ignorado e a linha 34 concatena mais um caractere à $stringS$ e i é incrementeada de uma unidade na linha 35.

Porém, caso tenha atingido o tamanho máximo, a linha 21 e 22 inicializam duas variáveis vazias: $matrizPermutacoes$ e $stringL$.

O laço corresponde às linhas 23 a 26 será executado um número de vezes igual ao tamanho da variável $stringS$. Esse laço realizará a permutação circular da $stringS$ e armazenará cada permutação na $matrizPermutacoes$.

Em seguida, na linha 27, a $matrizPermutacoes$ será ordenada lexicograficamente.

A linha 28 armazenará na variável $indiceI$ a posição da matriz em que ocorre o conteúdo da variável $stringS$.

As linhas 29 e 30 irão montar a $stringL$ varrendo a $matrizPermutacoes$ e obtendo o ultimo caractere de cada linha da matriz.

Por fim, será gravado (linha 31) no arquivo de saida, transformado, a estrutura genérica "$\{indiceI\}stringL$". O índice I é necessário ser fornecido para que o processo de reversão da $stringL$ à $stringS$ seja efetuado corretamente e, para isso, foi preciso utilizar os colchetes ("{" e "}") para delimitar o valor do índice para que não fosse confundido com o conteúdo da $stringL$ no momento da reversão.

Reversão da Transformação de *Burrows e Wheeler*

O algoritmo de reversão do processo de transformação, resumidamente, é realizado lendo o arquivo de entrada (a ser revertido), caractere por caractere, sendo estes concatenados formando a $stringL$ até um tamanho máximo estipulado ou, caso não atinja o tamanho máximo, até o fim do arquivo. A cada $stringL$ formada, seu conteúdo, inicialmente, será composto pelo índice I e pela própria $stringL$. Portano, será preciso separar o índice I da $stringL$. Em seguida é preciso, também, obter uma terceira *string* chamada $stringF$, que é a $stringL$ ordenada de forma lexicográfica (ordem crescente de caractere). Depois de obtidos o índice I e as *strings* L e F , será realizado o processo de montagem da $string$ original S. Para melhor compreendê-lo, em vez de reexplicá-lo, recomendamos acompanhar a descrição teórica do processo de reversão da Transformação de *Burrows e Wheeler*.

Os caracteres da $stringS$ virão da $stringF$ e, a posição desses caracteres na $stringF$, virão da $stringL$. Com isso, o processo de reversão é iniciado obtendo-se o caractere da $stringF$ localizado na posição do índice I. A

$stringS$ então armazena esse primeiro caractere. Para encontrar os demais, um laço, que se repetirá tantas vezes quanto for o tamanho da $stringL - 1$, irá verificar qual a ordem de aparição do caractere atual na $stringF$ em relação aos demais caracteres iguais da própria $stringF$. Sabendo a ordem de aparição em relação aos seus iguais na $stringF$, o programa irá buscar na $stringL$ a posição geral na *string* em que este caractere ocorre com a mesma ordem de aparição. Sabendo a posição geral na $stringL$, agora o programa consegue descobrir qual é o próximo caractere da $stringS$, pois este se encontra na $stringF$ localizado na posição geral encontrada. Este processo é repetido até que a $stringS$ esteja toda restaurada.

Para as linhas 1 e 2 e 23 a 35 o código está explicado no tópico linhas de código gerais, pois são linhas comuns a outros algoritmos. O programa, nessas linhas, recebe o arquivo a ser compactado, abre-o e verifica se está vazio. Se não estiver o programa abre um novo arquivo de saída que receberá o conteúdo compactado.

```
1.   #include: utf-8
2.   import sys
3.   def ordemAparicaoStringF (stringF, posicao):
4.       ordemAparicao = 0
5.       i = 0
6.       char = stringF[posicao]
7.       for elemento in stringF:
8.            if elemento == char:
9.                ordemAparicao += 1
10.               if i == posicao:
11.                   break
12.           i += 1
13.      return ordemAparicao
```

```
14. def posicaoNaStringL (stringL, charStringF, ordemAparicaoStringF):
15.     i = 0
16.     ordemAparicao = 0
17.     for elemento in stringL:
18.         if elemento == charStringF:
19.             ordemAparicao += 1
20.             if ordemAparicao == ordemAparicaoStringF:
21.                 return i
22.         i += 1
23. num_argv = 0
24. for elemento in sys.argv:
25.     num_argv += 1
26. if num_argv > 1:
27.     arquivo_entrada = open(sys.argv[1], "r")
28. else:
29.     sys.exit("Programa executado sem arquivo a ser compactado")
30. char = arquivo_entrada.read(1).decode('latin1')
31. if len(char) == 0:
32.     sys.exit("Arquivo Vazio!")
33. arquivo_entrada.seek(0)
34. arquivo_saida = open(sys.argv[1] + "unbwt", "w")
35. tamanhoString = 1024
36. stringL = ""
37. i = 1
38. stringL = arquivo_entrada.read(1).decode('latin1')
39. while stringL:
40.     if '{' in stringL and '}' in stringL:
41.         if i - len( stringL[ : stringL.index('}') + 1] ) ==
            tamanhoString:
42.             indiceI = int(stringL[1 : stringL.index('}')])
43.             stringL = stringL[stringL.index('}') + 1 : ]
44.             stringF = "".join(sorted(stringL))
45.             stringS = stringF[indiceI]
46.             posicaoEmF = indiceI
```

```
47.          for j in range(1, len(stringL)):
48.                  posicaoProxChar = posicaoNaStringL (stringL,
                     stringF[posicaoEmF], ordemAparicaoStringF (stringF,
                     posicaoEmF))
49.                  stringS += stringF[posicaoProxChar]
50.                  posicaoEmF = posicaoProxChar
51.              arquivo_saida.write((stringS).encode('latin1'))
52.              stringL = ""
53.              i = 0
54.      stringL += arquivo_entrada.read(1).decode('latin1')
55.      i += 1
56.  arquivo_entrada.close()
57.  arquivo_saida.close()
```

A linha 3 define a função $ordemAparicaoStringF$ e a linha 14 define a função $posicaoNaStringL$. Ambas funções foram explicadas no início desta seção.

A linha 35 define o tamanho máximo da $string$, com a variável $tamanhoString$. Este tamanho deve ser igual ao do processo de transformação.

A linha 36 define a $stringL$ inicialmente vazia e a linha 38 inicializa a variável contadora i com o valor 1.

A linha 38 obtem o primeiro caractere do arquivo e armazena na variavel $stringL$.

Nas linhas 39 a 55 um laço é executado até o fim do arquivo.

As linhas 40 e 41 verificam se a variável $stringL$ já está completa, com a informação do índice I e da própria $stringL$. Se não estiver, as linhas 54 e 55 são executadas incrementando as variáveis $stringL$ com caracteres do arquivo e i de uma unidade. Do contrário, caso $stringL$ já esteja completa, então a o índice I é extraído da $stringL$ e armazenado na variável $indiceI$ (linha 42). A $stringL$, na linha 43, é limpa, restanto apenas os caracteres realmente pertencentes à *string.* Na linha 44 a $stringF$, que é a $stringL$ ordenada, é criada. Em seguida, na linha 45 o primeiro caractere da $stringS$ é obtido a partir da posição I na $stringF$. A linha 46 inicializa a variável $posicaoEmF$ com o valor do $indiceI$.

As linhas 47, 48, 49 e 50 realizarão o processo descrito no início desta seção de varrer as *strings* F *e* L em busca dos próximos caracteres da $stringS$.

Quando a $stringS$ estiver restaurada, a linha 51 gravará seu conteúdo no arquivo de saída (arquivo restaurado). Em seguida, na linha 52, a variável $stringL$ é esvaziada para que seja utilizada para o próximo processo de restauração, até que todo o arquivo a ser restaurado seja lido.

Por fim, as linhas 56 e 57 fecham os arquivos de entrada e saída.

Testes e resultados com os algoritmos implementados

Esta seção apresentará os resultados obtidos com a compressão do livro Os Lusíadas (os_lusiadas.txt) utilizando o algoritmo da codificação *Huffman* implementado. Como o algoritmo de *Burrows e Wheeler* não configura

uma compressão, somente uma transformação, será verificado para este caso apenas se o arquivo restaurado é idêntico ao arquivo original.

Para o algoritmo de compressão *Huffman* serão mostrados o tamanho final dos arquivos, a taxa de compressão e os tempos de compressão e descompressão. Para o algoritmo de *Burrows e Wheeler* será mostrado apenas o tempo de transformação e restauração.

Os testes serão realizados em ambiente Linux Mint 17.2 64-bit, processador Intel Pentium Dual-Core T440 2.20Ghz e memória RAM 1.9Gb.

Arquivos compactados e transformados

A seguir, nesta seção, serão apresentados os arquivos gerados após as execuções dos algoritmos de compressão e transformação para que o leitor consiga visualizar os resultados produzidos pelos algoritmos.

A imagem a seguir exibe o arquivo compactado gerado pelo método *Huffman.* Como explicado nas seções teóricas e de implementação, o arquivo compactado é composto de um cabeçalho - que são os dados relacionados à codificação dos caracteres - e dos dados do arquivo original compactados.

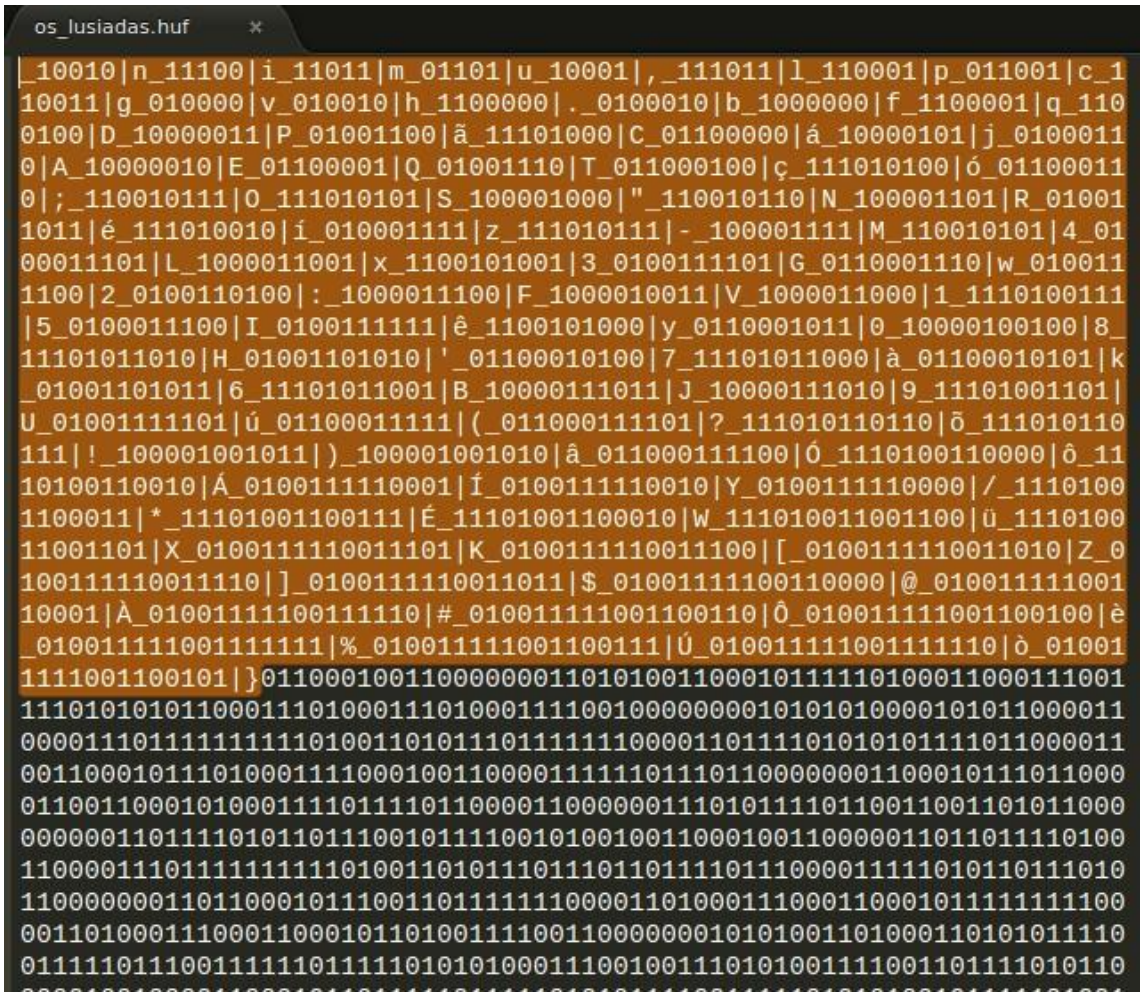


Figura 1:Arquivo compactado pelo algoritmo Huffman

O cabeçalho, na imagem acima, é a parte destacada em laranja enquanto o arquivo original compactado é o restante do arquivo. Note que o último caractere do cabeçalho é o caractere chave “}”, que é um *byte* adicional para informar ao descompressor que este ponto separa os dados relacionados à codificação do arquivo original compactado. Consequentemente o restante do arquivo refere-se ao arquivo original compactado.

Já a imagem que segue abaixo refere-se ao resultado produzido pelo algoritomo de transformação *Burrows e Wheeler.*

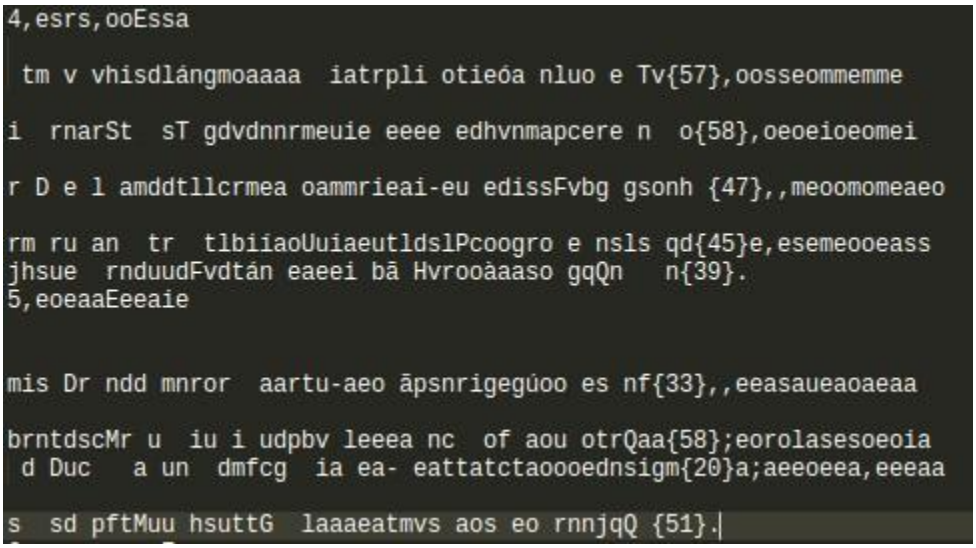


Figura 2: Arquivo gerado pela transformação de Burrows e Wheeler

Como descrito na seção de implementação, o arquivo produzido pela transformação de *Burrows e Wheeler* é um arquivo composto de sequências de caracteres de tamanho pré-definido (pelo algoritmo) e que seguem a estrutra $\{índice\}sequência\ de\ caracteres\ transformados$. Na imagem acima é possível encontrar essas estruturas que, em particular, referem-se às duas estrofes de Os Lusíadas retratadas na imagem abaixo.

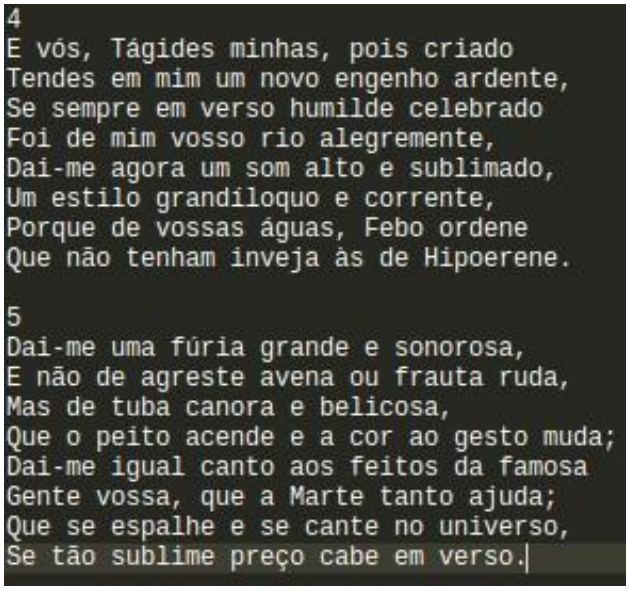


Figura 3: Estrofes 4 e 5 de Os Lusíadas

É possível deduzir então, empiricamente, que a transformação de *Burrows e Wheeler* realmente favorece um rearranjo dos caracteres de tal modo que aproxime em uma mesma região os caracteres iguais.

Tamanho compactado e taxa de compactação

Neste caso específico da compressão pela codificação *Huffman,* pelo fato de produzir um arquivo cujo conteúdo original foi integralmente convertido em caracteres zeros e uns (binários), será realizada uma análise do seu resultado de compactação sob duas óticas diferentes: do arquivo físico gerado, sem levar em conta seu conteúdo, e apenas do seu conteúdo, sem levar em conta seu espaço em disco. Para facilitar o entendimento das análises de cada situação, a primeira situação será denominada como compactação real enquanto, a segunda, compactação teórica.

Compactação real em disco		
Tamanho do arquivo Os Lusíadas original: 337.704 bytes		
Compressor (em Python)	**Tamanho após compactação (bytes)**	**Taxa de compactação**
Huffman	1.575.484	4,67

Figura 4:Tabela de compressão real Huffman

A tabela acima mostra o resultado da compactação real, ou seja, exibe o tamanho que o arquivo produzido ocupa no disco rígido. Neste caso, como é possível observar na seção de arquivos compactados e transformados, cada caractere do arquivo original (que ocupa um byte na mémoria) foi substituído por uma sequências de caracteres zeros e uns que possuem no mínimo três

dígitos, ou seja, no mínimo 3 *bytes.* Devido a isso, o arquivo final é maior que o arquivo original. Portanto, do ponto de vista da compactação efetiva, esta implementação é ineficaz.

Porém, uma outra abordagem para a análise dos resultados é enxergar a parte estritamente de caracteres binários do arquivo compactado, que se refere ao arquivo original compactado, como um potencial de compactação a ser atingido caso esse *script*, ao invés de produzir um arquivo de texto, trabalhasse e produzisse um arquivo compactado em nível binário efetivamente, bem como a maioria das ferramentas de compressão mais usuais fazem atualmente.

Sob este ponto de vista, nosso arquivo binário teria um resultado de compactação próximo ao que segue na tabela abaixo. Este resultado desconsidera o trecho de cabeçalho porque o tamanho dele, se fosse tratado em nível binário, causaria uma alteração ínfima no tamanho final do arquivo, uma vez que como texto ele representa apenas 0,08% (1.382 *bytes* de 1.575.484) da totalidade do arquivo compactado.

Compactação teórica				
Tamanho do arquivo Os Lusíadas original: 337.704 bytes				
Compressor (em Python)	**Tamanho do cabeçalho (bytes)**	**Total de caracteres binários**	**Tamanho teórico final (Total de carac. binários / 8) (bytes)**	**Taxa de compactação**
Huffman	1.382	1.574.102	196.763	0,58

Figura 5: Tabela de compressão teórica Huffman

Dessa forma, os caracteres binários do arquivo compactado produzido, isto é, a parte do arquivo que diz respeito ao arquivo original, contabiliza um

total de 1.574.102 caracteres. Se ao invés de caracteres este resultado fosse obtido trabalhando em nível binário teríamos um total, então, de 1.574.102 *bits,* correspondendo a 196.763 *bytes*, que seria a aproximação do tamanho teórico final desta compressão *Huffman,* atingindo uma taxa de compressão de 58%.

Tempos de compactação

Apuração dos tempos de compactação do arquivo original de 337.704 bytes	
Tempo	**Huffman**
Real	0m1.431s
User	0m1.353s
Sys	0m1.036s
Total (user + Sys)	**0m1.389s**

Figura 6: Tempos de compressão Huffman

Tempos de descompactação

Apuração dos tempos de descompactação do arquivo compactado	
Tempo	**Huffman**
Real	0m58.195s
User	0m57.968s
Sys	0m00.116s
Total (user + Sys)	**0m58.084s**

Figura 7:Tempos de descompressão Huffman

Tempos de transformação

Apuração dos tempos de transformação do arquivo original de 337.704 bytes	
Tempo	**Burrows e Wheeler**
Real	0m1.009s
User	0m0.990s
Sys	0m0.012s
Total (user + Sys)	**0m1.002s**

Figura 8: Tempos de transformação BWT

Tempos de restauração

Apuração dos tempos de restauração do arquivo transformado	
Tempo	**Burrows e Wheeler**
Real	0m3.678s
User	0m3.605s
Sys	0m0.048s
Total (user + Sys)	**0m3.653s**

Figura 9: Tempos de restauração BWT

Comparação entre os arquivos

Como realizado em testes anteriores, o comando **cmp** compara dois arquivos e, se forem iguais, não retorna nenhum tipo mensagem ou alerta. Neste caso, então, observamos que tanto o arquivo descompactado quanto o arquivo restaurado são iguais aos seus respectivos originais pois as comprações não resultaram em nenhum tipo de mensagem ou alerta.

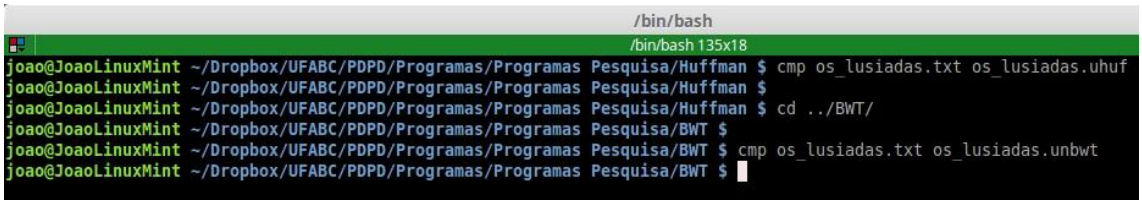


Figura 10 Comparação dos arquivos descompactado e restaurado com seus originais

Ambos os algoritmos alcançaram a meta de compactar e transformar os arquivos originais, sendo possível realizar suas operações inversas sem que houvessem perdas de dados, como eram originalmente.

Referências

BURROWS, M.; WHEELER, D. A block sorting lossless data compression algorithm. *Technical Report*, n. 124. Digital Equipment Corporation, 1994.

SALOMON, D. *A guide to data compression methods*. New York: Springer, 2002.

SALOMON, D. *Data compression – the Complete Reference*. New York: Springer, 2007.

SAYOOD, K. *Introduction to data compression*. 3. ed. Amsterdam: Morgan Kauf

mann, 2006.