CODEC não destrutivo para Texto

Leonor Colaço dos Reis
Departamento de Engenharia Informática
Universidade de Coimbra
Coimbra, Portugal
leonorcolaco19@gmail.com

João Pinto
Departamento de Engenharia Informática
Universidade de Coimbra
Coimbra, Portugal
pintojoao2211@gmail.com

Guilherme Faria
Departamento de Engenharia Informática
Universidade de Coimbra
Coimbra, Portugal
guifaria02@gmail.com

***Abstract*—**Neste artigo exploramos os conceitos de teoria de informação, em particular no que respeita aos conceitos relativos à teoria da compressão. Particularmente, são abordados os principais algoritmos de compressão sem perda para ficheiros de texto. Inicialmente, é abordada a definição de compressão de dados e a sua importância nos dias de hoje. Posteriormente é feita a distinção entre compressão de dados destrutiva e compressão de dados não destrutiva. As várias técnicas de compressão foram divididas entre: técnicas de compressão estatística e técnicas de compressão baseadas em dicionário. Dentro das técnicas de compressão estatística, foram abordadas: códigos de Shannon-Fano; algoritmos de Huffman estático; algoritmos de Huffman estático; Run Length Encoding; códigos Aritméticos. Relativamente às técnicas baseadas em dicionário é feita uma divisão em duas famílias: algoritmos derivados de LZ77 (LZ77, LZSS, LZH e LZB) e algoritmos derivados de LZ78 (LZ78, LZW e LZFG). Por último, são abordados os sistemas.

***Keywords*—** *compressão de dados, compressão destrutiva, compressão não destrutiva, técnicas de compressão estatística, técnicas de compressão baseadas em dicionário, sistemas*

I. Introdução

A compressão de dados refere-se à redução da quantidade de espaço necessário para armazenar informação ou à redução do tempo necessário para a transmitir. O tamanho de informação é reduzido através da remoção da informação redundante. O seu objetivo é representar a fonte com o mínimo número de bits possível usando várias técnicas matemáticas e estatísticas. As quais, neste artigo, serão abordadas em detalhe.

A cada minuto de 2020, o *YouTube* ganhou 500 novas horas de vídeo enviados pelos utulizadores, o *Spotify* incluiu 28 novas músicas ao seu catálogo, 347 mil stories foram publicados no *Instagram*, 147 mil fotos foram publicadas no *Facebook*. Além disso, no ano marcado pela pandemia, estima-se que, em cada minuto, 208 mil pessoas participaram em conferências por meio do *Zoom*. Graças à compressão de dados é possível toda esta circulação de informação. Na internet, a compressão de dados ajuda não só a diminuir a quantidade de tráfego na grande rede, como também a aumentar a velocidade de navegação, realização de downloads de arquivos e visualização de vídeos. Na vida offline, arquivos compactados são preferíveis quando há interesse de armazenamento de maior número de dados possível no menor espaço de memória secundária disponível, como em pen-drives, memmory cards, discos rígidos e demais unidades de memória.

A compressão de texto pode ser: destrutiva ou não destrutiva. Na compressão não destrutiva, os dados obtidos após a compressão são idênticos aos dados originais. Esta forma de compressão é preferível para fontes de dados, como textos, programas de computador, imagens e sons que precisem de ser reproduzidos de forma exata, por exemplo, imagens e gravações para perícias ou impressões digitais. Nestes casos uma pequena perda de dados acarreta o não funcionamento ou torna os dados incompreensíveis. Por outro lado, fontes de informação como fotografias, sons e filmes permitem que perdas de dados poucos significativos ocorram. Algumas perdas não são percebidas pelo olho ou ouvido humano, como por exemplo, sons de frequências muito altas ou muito baixas que os humanos não ouvem, detalhes muito subtis como a diferença de cor entre duas folhas de uma árvore, movimentos muito rápidos que não conseguimos acompanhar num filme. Nestes casos, é preferível a compressão destrutiva, em que detalhes como os falados anteriormente são omitidos. Assim, os dados obtidos após a compressão não são idênticos aos originais, uma vez que perderam as informações irrelevantes, e dizemos então que é um método de compressão com perdas.

II. TÉCNICAS DE COMPRESSÃO ESTATÍSTICA

Os algoritmos descritos nesta secção são baseados em um modelo estatístico, ou seja, num alfabeto e numa distribuição de probabilidade de uma fonte. Em nenhum destes algoritmos é considerado o contexto da fonte.

A. Códigos de Shannon-Fano

O algoritmo de *Shannon-Fano* foi desenvolvido em 1949 por Claude Shannon e Robert Fan. Neste método, é associado a cada símbolo um código de tamanho variável de acordo com a sua probabilidade.

Antes de avançar, é importante de ressalvar que, no código ASCII, para representar um carater são necessários 8 bits (8^2 = 256). Independentemente de um símbolo aparecer mais ou menos vezes, é sempre representado com 8 bits.

Nesta técnica, a símbolos com maior probabilidade de aparecerem é associado um código com menor número de bits e a símbolos com menor probabilidade é associado um código com maior número de bits. Deste modo, é possível diminuir o número de bits utilizados para representar uma fonte de informação.

O algoritmo *Shannon-Fano* segue os seguintes passos:

1. Determina o alfabeto da fonte de informação
2. Conta o número de ocorrências de cada símbolo
3. Determina a probabilidade de cada símbolo, recorrendo ao número de ocorrências

(probabilidade = nº de ocorrências símbolo / nº total de ocorrências)
4. Ordena os símbolos por ordem decrescente de probabilidade
5. Gera um nó para cada símbolo
6. Divide a lista em dois, mantendo a probabilidade do ramo esquerdo aproximadamente igual às da direita
7. Anexe 0 ao nó esquerdo e 1 ao nó direito
8. Aplica as etapas 6 e 7 às subárvores esquerda e direita até que cada nó seja uma folha na árvore.

Este método é anterior ao de *Codificação de Huffman*, e apesar de bastante eficiente e prático, gera resultados sub-ótimos. É mais eficiente quando as probabilidades estão mais próximas de inversos de potências de 2.

B. *Códigos de Huffman*

Duas famílias de codificação de Huffman foram propostas: algoritmos de Huffman estático e algoritmos de Huffman adaptativo.

Os algoritmos de *Huffma*n estático calculam as frequências primeiro e, em seguida, geram uma árvore para os processos de compressão e descompressão. Os detalhes desta árvore devem ser salvos ou transferidos com o arquivo compactado.

Os algoritmos de *Huffman* adaptativo desenvolvem a árvore enquanto calculam as frequências. Nesta abordagem, uma árvore é gerada com a bandeira símbolo no início e é atualizado quando o próximo símbolo é lido.
Ao contrário dos algoritmos de *Huffma*n estático, não é necessário conhecer o modelo estatístico da fonte e não é necessário enviar o dicionário de dados.

Algoritmos de Huffman estático

A codificação Huffman foi desenvolvida em 1952 por David A. Huffman que era, na altura, estudante de doutorado no MIT.
A ideia de Huffman é associar a símbolos mais frequentes códigos com menor comprimento e a símbolos menos frequentes códigos de maior comprimento, tal como, no algoritmo *Shannon-Fano.*

O algoritmo *da Codificação de Huffman* segue os seguintes passos:

1. Varre o texto e determina o alfabeto da fonte de informação
2. Conta o número de ocorrências de cada símbolo
3. Constrói um nó para casa símbolo do alfabeto do texto
4. Enquanto houver mais de uma árvore:
   4.1. Constrói um nó que combine os nós identificados em 3.1 em um novo nó e armazena no novo nó a soma das frequências
   4.2. Atribui como subárvore esquerda do nó criado em 3.2 o nó com menor frequência e o outro como subárvore direita.
5. Atribui 0 para o ramo esquerdo e 1 para o ramo direito.
6. Cria um código de prefixo para cada símbolo do alfabeto percorrendo a árvore binária da raiz ao nó, que corresponde ao símbolo
7. Varre o texto original substituindo cada símbolo por seu código representado na árvore.

Dependendo da distribuição de frequência dos símbolos, o ganho com o método pode ser maior ou menor, ou seja, quanto menos uniforme for a distribuição dos símbolos, maior será o ganho.

Algoritmos de Huffman adaptativo

No algoritmo *Algoritmos de Huffman estático* é necessário conhecer previamente o modelo estatístico da fonte, o que muitas vezes, não é possível. Além disso, não é adequado para casos em que as probabilidades dos símbolos de entrada estão em constante alteração.

Inicialmente a árvore contém apenas o nó NYT (Not Yet Transmitted), um nó especial que representa todos os símbolos que ainda não foram transmitidos. Aqui o

A árvore Huffman inclui um contador para cada símbolo e o contador é incrementado sempre que um símbolo de entrada codificado é igual. Uma árvore sibilante é uma árvore de Huffman. Se a árvore em construção não verificar a propriedade de sibilância, a árvore deve ser reestruturada para garantir esta propriedade.
Geralmente, os algoritmos de Huffman adaptativo são mais eficientes do que os algoritmos de Huffman estático. Ao contrário do que acontece com os algoritmos de Huffman estático, não é necessário armazenar com os códigos a árvore. Mas este algoritmo tem que reconstruir toda a árvore Huffman após codificar cada símbolo, consequentemente torna-se mais lento do que o algoritmo de Huffman estático.

C. *Run Length Encoding*

A ideia principal deste algoritmo é codificar símbolos repetidos como um par: o comprimento do *string* e o símbolo.

O algoritmo *Run Length Encoding* segue os seguintes passos:

1. Determinar uma FLAG que não exista no texto a comprimir
2. Ler um carater. Ler os próximos carateres enquanto forem iguais ao primeiro carater lido
3. Se o número total de carateres lidos for igual ou superior a 4, comprimir essa cadeia de carateres da seguinte forma: FLAG + nº de repetições + carater
4. Se o número total de carateres lidos for inferior a 4, não se efetua a compressão. Logo, essa cadeia de carateres permanece inalterável.

A principal desvantagem do algoritmo RLE é que não atinge elevadas taxas de compressão quando comparado com outros métodos de compressão avançados. No pior

cenário possível o tamanho do arquivo de saída é duas vezes maior do que o tamanho do arquivo de entrada. É de ressalvar que quanto maior for o comprimento da sequência de carateres repetidos, maior será a taxa de compressão. Por outro lado, este algoritmo é fácil de implementar e rápido de executar, o que o torna uma boa alternativa para um algoritmo de compressão complexo.
Este algoritmo só é eficiente em textos com muitos dados repetitivos.
Em geral a técnica de RLE é uma técnica auxiliar que não gera uma compressão muito grande sozinha, mas que aliada a outras técnicas é um instrumento simples e poderoso de compressão.

D. Códigos Aritméticos

Tanto os códigos de Huffman, como os Shannon-Fano códigos têm um overhead de um bit.

Os códigos aritméticos diferem de outras formas de codificação de entropia, como a codificação de Huffman, em que, em vez de separar a entrada em símbolos de componentes e substituir cada um por um código, a codificação aritmética codifica a mensagem inteira em um único número, uma fração de precisão arbitrária q onde $0,0 \leqslant q < 1,0$

A codificação aritmética segue os seguintes passos:

1. Cria-se um intervalo iniciado com [0,1]
2. Para cada elemento da mensagem:
   2.1. Particiona-se o intervalo corrente em sub-intervalos, um para cada símbolo do alfabeto. O tamanho do sub-intervalo associado a uma dada letra é proporcional à probabilidade de que esta letra seja o próximo elemento da mensagem, de acordo com o modelo assumido
   2.2. O sub-intervalo correspondente ao próximo elemento é selecionado como novo intervalo corrente
3. Codifica-se a mensagem com o menor número de bits necessário para distinguir o intervalo corrente final de todos os outros possíveis intervalos correntes finais (isso garantirá que o código resultante seja um código de prefixo).

III. TÉCNICAS DE COMPRESSÃO BASEADAS EM DICIONÁRIO

Os algoritmos descritos nesta secção tiram partido da dependência estatística da fonte.

A. Lempel ZIV algorithms

Os algoritmos Lempe-Ziv inventados pelos cientistas da computação israelitas Abraham Lempel e Jacob Ziv, usam o próprio texto como dicionário, substituindo as ocorrências posteriores de uma string por números que indicam onde ocorreu antes e o seu respetivo comprimento.

B. Algoritmos derivados de LZ77

- LZ77

  LZ77, apresentado por Jacob Ziv e Abraham Lempel em 1977, é uma técnica de compressão "lossless" baseada em dicionários. Como num texto é muito provável a existência de carateres repetidos, LZ77 aproveita este facto para codificar uma repetição como um ponteiro para uma ocorrência anterior.

  Nesta técnica, o "encoder" examina a sequência introduzida usando uma janela deslizante, constituída por duas partes: um "search buffer", que contém uma parte da sequencia recentemente codificada, e um outro "buffer" que contém a próxima porção da sequência a ser codificada.

  A janela de pesquisa tem um tamanho definido e à medida que vai avançando vai eliminando os bytes mais antigos por ter alcançado o seu tamanho limite. Já o "buffer" de "look ahead" é preenchido com os próximos bytes a serem processados. Tendo já estas duas estruturas, é processado a sequência de carateres inicias no "buffer" e verifica qual é o maior "match" que se encontrar na janela de pesquisa. Sendo encontrada, é emitido na saída um tuplo (o,l,c) onde "o" é a posição relativa da sequência dentro da janela (contado de trás para a frente), "l" é o tamanho da sequência e "c" o próximo carater presente no buffer.

  O processo de descompressão é muito mais rápido que a compressão, já que para comprimir é necessário procurar a maior correspondência e para descomprimir é apenas substituir a referência a uma *string* por ela mesma. Uns dos problemas é mesmo este, o de ser bastante lento a comprimir, contudo é de realçar que é muito eficiente para a codificação de textos mais longos, onde existe palavras que são repetidas constantemente.

- *LZSS*

  É uma variante do LZ77, mais eficaz e com uma compressão melhorada. Existem, entre outras, algumas principais alterações: na "janela deslizante" apesar de ainda serem armazenadas as frases como blocos de texto simples e contínuos, ainda cria uma estrutura em árvore de busca binária.[3] O output também difere devido a, neste algoritmo, ser utilizado um byte(carater) ou um par (offset,tamanho). Para serem os dois distinguidos, Storer e Szymanski, criadores deste método, introduziram a utilização de "flags". Caso a "flag" seja 0, o próximo byte é um carater, caso seja 1, o próximo byte é um par (offset,tamanho), que funciona da mesma maneira que no LZ77.

- LZH

  Esta outra variante do LZ77 combina as técnicas de Liv-Lempel e de Huffman. Essencialmente, começa da mesma forma que o LZSS apenas com a posterior utilização de estatística medida nesta primeira fase para codificar os ponteiros e carateres usando a codificação de Huffman.

- LZB

  O objetivo desta técnica de compressão é modificar os ponteiros do LZSS para terem tamanhos diferentes, pois na prática uma compressão melhor é alcançada quando se têm ponteiros com tamanhos diferentes.

C. *Algoritmos derivados de LZ78*

- LZ78

  Os algoritmos LZ78 alcançam a compactação substituindo ocorrências repetidas de dados por referências a um dicionário que é construído com base no fluxo de dados de entrada.

  Cada entrada do dicionário tem a forma de dicionário[…] = {índice, caractere}, onde índice é o índice de uma entrada de dicionário anterior e o caractere é anexado à string representada por dicionário[índice].

  Para cada caractere o fluxo de entrada, é pesquisado no dicionário uma correspondência: {último índice correspondente, caractere}. Se uma correspondência for encontrada, o último índice correspondente será definido como o índice da entrada correspondente e nada será gerado. Se uma correspondência não for encontrada, uma nova entrada de dicionário é criada: dicionário[próximo índice disponível] = { último índice correspondente, caractere} e o algoritmo gera o último índice correspondente, seguido por caractere, em seguida redefine o último índice correspondente = 0 e incrementa o próximo índice disponível.

- LZW

  LZW é um algoritmo baseado em LZ78 que usa um dicionário pré-inicializado com todos os caracteres possíveis(símbolos) ou emulação de um dicionário pré-inicializado. A principal melhoria do LZW é que quando uma correspondência não é encontrada, o caractere do fluxo de entrada atual é considerado o primeiro caractere de uma string existente no dicionário, então apenas a última correspondência índice é a saída.

  -Um grande ficheiro de texto escrito em Inglês pode ser comprimido via LZW para cerca de metade do seu tamanho original.

D. *LZFG*

O *LZFG* é um algoritmo híbrido dos algoritmos *LZ77* e *LZ78*. Basicamente, é semelhante ao *LZ77* mas com a restrição de que a ocorrência anterior de cada frase, deve começar num limite da frase anterior.
Não há qualquer restrição no final da frase, ou seja, cada frase é codificada como um par comprimento/distância, sendo que a distância aponta para uma matriz separada registando as posições dos limites das frases. Nesse aspeto, LZFG é um algoritmo do tipo LZ78.

Usar um grande tamanho de janela ou ilimitado é mais fácil com o LZFG do que com o LZ77, porque os valores da distância são menores e as estruturas de dados para localizar as frases são mais simples.

E. *Transformada de Burrows-Wheeler*

O método de compressão de *Burrows-Wheeler* foi inventado em 1994 por Michael Burrows e David Wheeler. Este algoritmo é considerado um dos melhores algoritmos de compressão global para o texto.

Esta transformação é baseada no contexto. Dado um símbolo presente no bloco de dados, há uma grande probabilidade desse símbolo ser sempre precedido do mesmo conjunto de símbolos. Por exemplo, na língua portuguesa: a letra "N" tem maior probabilidade de ser precedida pela letra "H" do que por qualquer outra letra. Usando este princípio, pretendemos reordenar os símbolos de uma sequência de entrada S, para derivar L, uma nova sequência que irá permitir uma melhor compressão.

O comprimento do array original S, e do array resultante L é o mesmo porque L é apenas uma permutação de S.
O algoritmo de *Burrows-Wheeler* segue os seguintes passos:

1. Dada uma sequência de comprimento N criar N-1 sequências de comprimento N com um Shift cíclico
2. Ordenar lexicograficamente a primeira coluna
3. Transformada da última coluna
4. Guardar a linha onde se encontra a sequência original

IV. SISTEMAS

A. *Deflate*

Este algoritmo, criado por Phillip W. Katz, usa, tal como o LZH, tanto o LZ77 como a codificação de Huffman para comprimir informação. É usado, atualmente em diferentes formatos de ficheiros tal como, GZIP, 7-zip, etc. Primeiramente, o input é dividido em séries de blocos onde posteriormente é aplicado LZ77 para encontrar as strings repetidas e os ponteiros, depois os símbolos são substituídos pelos códigos provenientes da codificação de Huffman aplicada neste caso.

B. *LZMA*

O algoritmo de Lempel-Ziv-Markov foi primeiramente utilizado no formato 7z. Este algoritmo utiliza técnicas de compressão parecidas ao processo LZ77 mas apresenta uma capacidade de compressão superior e um dicionário de compressão variável. O dicionário produz um fluxo de símbolos e referências que depois é codificado com um "range encoder", para fazer uma previsão de probabilidade para cada bit.

C. *BZIP2*

Bzip2 é um software de compressão publicado em 1996 por Julian Seward. Consegue resultados impressionantes devido à sua avançada técnica de compressão de multi-camada. É capaz de dividir a informação em blocos de tamanho entre 100kB até 900kB para depois serem todos individualmente comprimidos usando o algoritmo de compactação de texto de classificação de bloco de Burrows-Wheeler, codificação de Huffman, Run-length encoding e Delta encoding.

Comparado com o Gzip, o Bzip2 apresenta maiores taxas de compressão, mas por outro lado, apresenta um maior tempo de descompressão e um maior uso de memória.

D. *Prediction by partial matching*

PPM é um algoritmo de predição. Atribui um valor de probabilidade a cada símbolo, baseado no número de vezes em que o mesmo ocorreu, e em que contexto. Assim, consegue prever a ocorrência dos mesmos símbolos, numa outra zona idêntica
do ficheiro.

V. Descrição dos algoritmos selecionados

Para a comparação da performance dos variados algoritmos de compressão de texto são usados dois fatores: taxa de compressão e velocidade de compressão.

A taxa de compressão é dada por:

$$\frac{\text{Tamanho do ficheiro original (KB)} - \text{Tamanho do ficheiro comprimido (KB)}}{\text{Tamanho do ficheiro original (KB)}}$$

A velocidade de compressão é dada por:

$$\frac{\text{Tamanho do ficheiro original (KB)}}{\text{Tempo de compressão (s)}}$$

Tendo como base estes dois fatores, iremos descobrir qual o algoritmo que comprime mais eficientemente cada ficheiro.

O limite mínimo teórico para o número médio de bits por símbolo corresponde à entropia da fonte (H) e é dado por:

$$\sum_{i=1}^{c} -p_i log_2 p_i$$

O valor máximo da taxa de compressão que podemos atingir utilizando um código entrópico é dado por:

$$\text{Taxa de compressão} = \frac{\text{H x len(fonte) / 8 (KB)}}{\text{len(fonte\_original) (KB)}}$$

Tabela I. Valores máximos de compressão para códigos entrópicos

	Taxa de compressão (%)
bible.txt	54.28
finance.csv	64.78
jqery-3.6.0.js	63.34
random.txt	74.99

Seguidamente, apresentamos os resultados obtidos para cada algoritmo, pela seguinte ordem: algoritmos entrópicos (Huffman Codec), algoritmos baseados em dicionário (LZ78, LZSS, LZW) e sistemas (LZMA, ZLIB, GZIP, BZIP2).

Tabela II. Valores de compressão com Huffman Codec

Huffman Codec			
	Taxa de compressão (%)	Tempo (s)	Velocidade (KB/s)
bible.txt	45.19	7.268	543
finance.csv	34.83	12.477	460
jqery-3.6.0.js	36.25	0.613	282
random.txt	24.99	0.268	364

Tabela III. Valores de compressão com LZ78

LZ78			
	Taxa de compressão (%)	Tempo (s)	Velocidade (KB/s)
bible.txt	22.97	1.800	2 196
finance.csv	69.45	2.140	2 684
jqery-3.6.0.js	7.80	0.140	700
random.txt	-59.18	0.540	552

Tabela IV. Valores de compressão e descompressão com LZSS

LZSS			
	Taxa de compressão (%)	Tempo (s)	Velocidade (KB/s)
bible.txt	59.50	2.010	1 966
finance.csv	79.20	2.266	2 534
jqery-3.6.0.js	60.20	0.125	2 256
random.txt	-10.70	0.046	2 130

	Tempo de descompressão (s)
bible.txt	0.547
finance.csv	0.686
jquery-3.6.0.js	0.031
random.txt	0.015

Tabela V. Valores de compressão e descompressão com LZW

LZW			
	Taxa de compressão (%)	Tempo (s)	Velocidade (KB/s)
bible.txt	86.44	2.289	1 726
finance.csv	94.76	2.441	2 353
jqery-3.6.0.js	79.46	0.109	2 587
random.txt	49.86	0.047	2 085

	Tempo de descompressão (s)
bible.txt	1.758
finance.csv	2.186
jquery-3.6.0.js	0.046
random.txt	0.046

Tabela VI. Valores de compressão e descompressão com LZMA

LZMA			
	Taxa de compressão (%)	Tempo (s)	Velocidade (KB/s)
bible.txt	78.13	3.346	1 181
finance.csv	96.41	2.371	2 422
jquery-3.6.0.js	74.61	0.167	1 688
random.txt	23.17	0.041	2 390

	Tempo de descompressão (s)
bible.txt	0.109
finance.csv	0.088
jquery-3.6.0.js	0.016
random.txt	0.016

Tabela VII. Valores de compressão e descompressão com ZLIB

ZLIB			
	Taxa de compressão (%)	Tempo (s)	Velocidade (KB/s)
bible.txt	70.55	0.297	13 309
finance.csv	95.00	0.312	18 410
jquery-3.6.0.js	70.50	0.080	3 525
random.txt	24.27	0.016	6 125

	Tempo de descompressão (s)
bible.txt	0.047
finance.csv	0.031
jquery-3.6.0.js	0.016
random.txt	0.016

Tabela VIII. Valores de compressão e descompressão com GZIP

GZIP			
	Taxa de compressão (%)	Tempo (s)	Velocidade (KB/s)
bible.txt	70.91	0.669	5 908
finance.csv	95.33	0.159	52 697
jquery-3.6.0.js	70.64	0.031	9 096
random.txt	24.25	0.001	98 000

	Tempo de descompressão (s)
bible.txt	0.047
finance.csv	0.046
jquery-3.6.0.js	0.015
random.txt	0.015

Tabela IX. Valores de compressão com PPM

PPM			
	Taxa de compressão (%)	Tempo (s)	Velocidade (KB/s)
bible.txt	75.11	153.94	25
finance.csv	90.44	208.72	27
jqery-3.6.0.js	70.92	11.43	24
random.txt	-9.18	9.28	10

	Tempo de descompressão (s)
bible.txt	225.53
finance.csv	317.87
jquery-3.6.0.js	16.90
random.txt	13.01

Tabela X. Valores de compressão com BZIP2

BZIP2			
	Taxa de compressão (%)	Tempo (s)	Velocidade (KB/s)
bible.txt	79.11	0.308	12 834
finance.csv	96.78	0.500	11 488
jqery-3.6.0.js	76.15	0.024	11 750
random.txt	24.32	0.01	9 800

	Tempo de descompressão (s)
bible.txt	0.037
finance.csv	0.031
jquery-3.6.0.js	0.016
random.txt	0.016

VI. ANÁLISE DE RESULTADOS DO DATASET FORNECIDO

Ao analisarmos os resultados obtidos, verificamos que o algoritmo de Huffman é o algoritmo que apresenta os piores resultados: maior tempo de compressão; menor taxa de compressão. O código de Huffman é um código entrópico, ou seja, não tira partido do contexto da fonte. Sendo assim, expectável que apresente os piores valores em ficheiros que o contexto seja relevante.

Como expectável, os algoritmos de compressão baseados em dicionário apresentam taxas de compressão significativamente superiores ao código de Huffman. No entanto, não esperávamos que a taxa de compressão do ficheiro *random.txt* atingida com o código de Huffman fosse inferior do que a atingida com um código baseada em dicionário. Tal como o nome do ficheiro indica, o ficheiro *random.txt* tem carateres completamente aleatórios, ao contrário dos outros ficheiros, que apresentam palavras. Ou seja, neste ficheiro o contexto não é relevante.

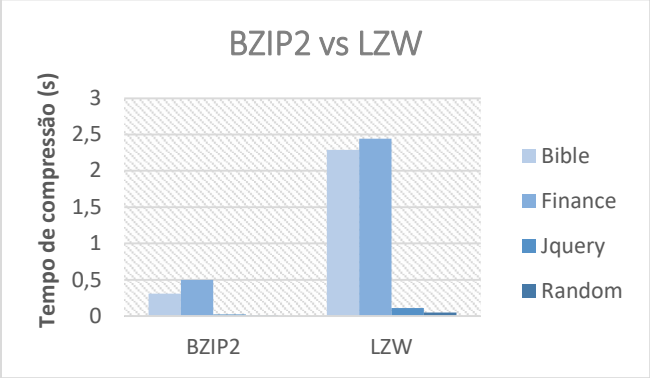
Além disso, nos algoritmos de compressão baseados em compressão verificamos que é o ficheiro *random.txt* que apresenta a menor taxa de compressão, pelas razões ditas anteriormente.

Entre o algoritmo de compressão LZW e LZSS, o LZW apresenta taxas de compressão significativamente superiores às do LZSS, mas por outro lado, apresenta maior tempo de descompressão. Quanto maior é o ficheiro, maior é o desfasamento entre os tempos de descompressão destes dois algoritmos.

Por outro lado, é o algoritmo LZW que apresenta as maiores taxas de compressão de todos os algoritmos analisados. No entanto, este algoritmo apresenta elevados quando comparado aos algoritmos ZLIB, GZIP e BZIP2 apresenta elevados tempos de compressão e descompressão.

O algoritmo LZW é preferível quando é desejável uma maior taxa de compressão, mesmo que isso implique, tempos de compressão significativamente superiores.

Como se pode observar no gráfico, apesar de o algoritmo LZW apresentar taxas de compressão mais elevadas, apresenta um tempo de compressão, em média, 6 vezes superior ao tempo de compressão do algoritmo BZIP2.



Dentro dos algoritmos LZW, ZLIB, GZIP, é o BZIP2 que apresenta não só a melhor relação entre a taxa de compressão e o tempo de compressão, como também, a velocidade de descompressão mais rápida.

Para percebermos os resultados excelentes deste algoritmo iremos analisá-lo melhor. O Bzip2 usa várias camadas de técnicas de compressão empilhadas umas sobre as outras, que ocorrem na seguinte ordem durante a compressão:

1. Run-length encoding (RLE)

   Codifica a sequência para armazenar apenas um único valor e número de ocorrências. Por exemplo, a sequência ABBBBBBBBBCDEEEEF (17 bytes) ao ser comprimida passa a A * 8B C D * 4E F (10 bytes).

2. Burrows–Wheeler transform (BWT)

   Compacta os dados em blocos de tamanho de 100 a 900 kB e converte as sequências de caracteres frequentes em strings de letras idênticas.

3. Move-to-front (MTF) transform

   Cada um dos símbolos é colocado numa matriz. Quando um símbolo é processado, é substituído pela localização (índice) na matriz e esse símbolo é passado para a frente da matriz. O efeito é que os símbolos imediatamente recorrentes são substituídos por símbolos zero (séries longas de qualquer símbolo arbitrário tornam-se séries de símbolos zero), enquanto outros símbolos são remapeados de acordo com sua frequência local.

   Os textos contêm símbolos idênticos que se repetem dentro de um intervalo limitado. Com a transformação MTF, diferentes símbolos de entrada recorrentes podem, na verdade, mapear para o mesmo símbolo de saída. Esses dados podem ser codificados de forma muito eficiente por qualquer método de compactação.

4. Run-length encoding (RLE) of MTF result

   As sequências de zeros resultantes da MTF (que vêm de símbolos repetidos na saída do BWT) são substituídas por uma sequência de dois códigos especiais, RUNA e RUNB.

5. Huffman coding

   Permite a representação em forma binária dos símbolos a partir das probabilidades de ocorrência. Apesar de não ser o principal contribuidor para a compressão, é um algoritmo simples capaz de aumentar a taxa de compressão.

6. Delta encoding (Δ) of Huffman-code bit lengths

Concluímos, que a combinação destes vários algoritmos por esta ordem resulta no algoritmo mais eficiente para a compressão do dataset fornecido.

VII. Conclusão

Através do desenvolvimento deste projeto, fomos elucidados sobre o poder da compressão de ficheiros de texto e o quão vantajoso pode ser nas grandes empresas que necessitam de armazenar imensos dados. Apesar da pouca velocidade e eficiência de alguns métodos, temos a ideia de que a grande evolução que estamos a presenciar a nível da inteligência artificial possa vir a melhorar estes aspetos com recurso a algoritmos de compressão bastante avançados.

Com a realização deste trabalho, foi também possível fomentar o nosso pensamento crítico devido ao facto de nem todos os algoritmos disponíveis pela internet estarem completos e ter de concluir qual deles seria o aconselhável para cada tipo de ficheiro, obrigando-nos assim de pôr os nossos conhecimentos de programação à prova.

Referências

[1] S, S., & L, R. (2011). Text Compression Algorithms - a Comparative Study. ICTACT Journal on Communication Technology, 02(04), 444–451.

[2] https://pt.wikipedia.org/wiki/LZ77, 20/11/2021

[3]http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-baseada-em-dicionarios/lzss/ 20/11/2021

[4] Gupta, A., Bansal, A., & Khanduja, V. (2017). Modern lossless compression techniques: Review, comparison and analysis. Proceedings of the 2017 2nd IEEE International Conference on Electrical, Computer and Communication Technologies, ICECCT 2017, February.

[5]https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Markov_chain_algorithm 20/11/2021

[6] https://criadoresid.com/dados-do-youtube-2020/ 18/11/2021

[7]https://www2.unitins.br/BibliotecaMidia/Files/Documento/AVA_633682985082488750aula_6.pdf 22/11/2021

[8] http://multimedia.ufp.pt/codecs/compressao-sem-perdas/supressao-de-sequencias-repetitivas/run-length-encoding/ 20/11/2021

[9]https://pt.wikipedia.org/wiki/Compress%C3%A3o_de_dados#Com_perdas_e_sem_perdas

[10] http://multimedia.ufp.pt/codecs/compressao-sem-perdas/supressao-de-sequencias-repetitivas/metodo-de-burrows-wheeler/
[11] https://www.tutorialspoint.com/compression-using-the-lzma-algorithm-using-python-lzma
[12] https://en.wikipedia.org/wiki/Run-length_encoding
[13] http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-baseada-em-dicionarios/lzss/
[14] https://github.com/DyakoVlad/python-LZ78
[15] https://en.wikipedia.org/wiki/Bzip2
[16]https://www.stringology.org/DataCompression/ppmc/index_en.html