CODEC *lossless* de compressão de texto

Botelho João
Departamento de Eng. Informática
Universidade de Coimbra
Coimbra, Portugal
uc2019155348@student.uc.pt

Branco Guilherme
Departamento de Eng. Informática
Universidade de Coimbra
Coimbra, Portugal
mbranco@student.dei.uc.pt

Félix Dário
Departamento de Eng. Informática
Universidade de Coimbra
Coimbra, Portugal
dario@student.dei.uc.pt

I. Introdução

Os algoritmos de compressão *lossless* (não destrutivos) permitem representar a mesma quantidade de informação com menor ocupação de espaço de armazenamento ou com melhor capacidade de transmissão. Habitualmente aplicam transformadas aos dados para tornar mais evidentes redundâncias na sua informação, e em seguida exploram-nas codificando probabilidades ou através de mapeamento. O trabalho prático descrito neste relatório foca a exploração destes conceitos de teoria da compressão e o desenvolvimento de uma solução para compressão *lossless* de texto.

Este relatório está organizado da seguinte forma: a Secção II expõe alguns dos principais *encoders* e transformadas usados para compressão *lossless* de texto, bem como programas de uso comum que os implementam; a Secção III apresenta o *dataset*, os algoritmos e conceitos utilizados na criação de propostas para compressão *lossless*, e as *baselines* e referências para a sua testagem; a Secção IV apresenta e discute os resultados do programa desenvolvido; a Secção V sumaria as principais conclusões deste trabalho, sugere possíveis melhorias ao programa concebido e propõe objetivos para trabalho futuro.

II. Estado da Arte

Apresentam-se em seguida alguns dos principais algoritmos usados para compressão *lossless* de texto e programas que os implementam.

A. *Encoders*

1) Entrópicos

Algoritmos que geram um modelo estatístico dos dados que permita mapeá-los para uma cadeia de *bits*, representando os dados mais frequentes com menos *bits*. [1]

a) Codificação de Huffman

Constrói uma árvore binária dos caracteres a partir das suas probabilidades de ocorrência *P(simbolo)*, usando-a em seguida para gerar um código de prefixo para cada caracter. A árvore deve ser associada ao ficheiro comprimido para permitir a sua descodificação. A codificação de Huffman aproxima obrigatoriamente o número de *bits* necessários para codificar um símbolo, $-log_2 P(simbolo)$, a um número inteiro. Símbolos com altas probabilidades de ocorrência podem ter um valor de *bits*/símbolo inferior a 1, pelo que a sua codificação é pouco eficiente. O impacto desta desvantagem pode ser reduzido por agrupamento de símbolos, mas o alfabeto mais extenso que resulta desse agrupamento leva a outros problemas como uma maior ocupação de memória. [1], [2], [3]

b) Codificação aritmética

Codifica o input com um único número *floating point*, a partir das probabilidades de ocorrência dos caracteres. Os caracteres com maior probabilidade de ocorrência passam a ocupar menos *bits*, enquanto aqueles com menor probabilidade de ocorrência passam a ocupar mais *bits*. A codificação final da totalidade dos dados ocupa um menor número de *bits*. A codificação aritmética tende a obter melhores resultados de compressão que a codificação de Huffman, pois codifica símbolos com probabilidade p num número de bits arbitrariamente próximo de $-log_2 p$. [1], [3]

c) Range Encoding

Generaliza a codificação aritmética, atuando em dígitos de qualquer base e não apenas em bits (base 2). Ao ser aplicado ao nível do *byte*, *Range Encoding* pode ser quase duas vezes mais rápida que codificação aritmética, mas resulta em maior ocupação de memória durante o processamento e numa menor taxa de compressão caso se pretenda a mesma precisão de probabilidades. [4]

2) Dicionário

Algoritmos que tiram partido da repetição do mesmo símbolo ou grupo de símbolos ao longo dos dados, substituindo ocorrências seguintes por uma referência à sua entrada num dicionário. [5]

a) LZ77

Explora a probabilidade de repetição de palavras ou frases. Quando isto acontece, os dados podem ser codificados com um ponteiro para uma ocorrência anterior dentro de um *buffer* ou janela deslizante, acompanhado pelo número de caracteres iguais encontrados. Tem a vantagem de ser muito simples e não requerer informação sobre as características da fonte *a priori*. Os algoritmos derivados do LZ77 compõem a família LZ77. [1], [2]

B. *Transformadas*

Algoritmos que transformam os dados de forma reversível, tornando-os mais favoráveis à aplicação de técnicas de compressão. [6]

1) Transformada de Burrows-Wheeler (BWT)

Reorganiza uma sequência de caracteres em conjuntos de caracteres semelhantes. É útil antes da aplicação de outras técnicas que comprimam sequências de caracteres repetidos. Tem como desvantagem a impossibilidade da sua aplicação a um *stream* de frases enviadas separadamente, pois a transformada é aplicada a um bloco inteiro. [6]

2) Move To Front (MTF)

Cada símbolo é substituído pelo seu índice numa pilha de "símbolos usados recentemente". No final, os dados foram transformados numa sequência de inteiros, que tendem a ser pequenos se a fonte tiver muitas correlações locais. [6]

C. Outros

1) Run Length Encoding (RLE)

Codifica símbolos repetidos e contíguos em pares "comprimento da cadeia, símbolo". Cada par pode ser posteriormente codificado com outro algoritmo. Apenas é eficiente em ficheiros com muitos dados repetidos, podendo aumentar o seu tamanho se isto não se verifica. [1], [2]

D. Sistemas

1) bzip2

Este programa realiza múltiplas camadas de compressão, começando pela aplicação de RLE à fonte, seguida de BWT, MTF e uma RLE do resultado. Segue-se codificação de Huffman e termina com codificação unária, codificação Delta e disposição dos símbolos usados num *array*. A descodificação é realizada pela ordem inversa. A *performance* do bzip2 é assimétrica, dado que a codificação tem grandes custos computacionais e de memória pelo uso de BWT, mas a descodificação é muito rápida [7]. O uso inicial de RLE na fonte tende a não ser eficiente e é um ponto de criticismo do *bzip2*. [8]

2) rzip

Este programa codifica inicialmente dados duplicados usando um dicionário com um *buffer* de 900MB, várias ordens de magnitude maior que o de outros programas, procedendo à compressão usando bzip2 como *backend*. A capacidade de aproveitar redundância a longas distâncias leva a uma compressão mais rápida, pois a informação processada por bzip2 foi previamente reduzida. O rzip atinge taxas de compressão elevadas para ficheiros grandes, sendo a sua principal desvantagem a elevada ocupação de memória durante o processamento. [9]

3) Deflate

Aplica codificação de Huffman e codificação por dicionário estilo LZ77. A construção do dicionário pode procurar e criar entradas para sequências longas de caracteres se a taxa de compressão for valorizada, ou minimizar o número de entradas se a velocidade for valorizada. Algumas implementações de Deflate têm a opção de usar árvores de Huffman pré-definidas do próprio algoritmo, em vez de criar árvores que requerem espaço de armazenamento adicional. [10], [11]

4) Algoritmo Lempel-Ziv-Markov (LZMA)

É composto por um algoritmo de dicionário com janela deslizante baseado no LZ77, seguido de *Range Encoding* e árvores binárias para codificação entrópica. Os dicionários usados têm tamanho até 4GiB, muito superior aos de outros algoritmos baseados em LZ. LZMA atinge altas taxas de compressão e tem *performance* assimétrica, dado que a descodificação é mais rápida que a codificação. [12], [13]

III. Material e Métodos

O *dataset* para testagem é composto pelos seguintes ficheiros:

- *bible.txt* – texto integral da bíblia, mais de 4 milhões de caracteres ASCII;
- *finance.csv* – ficheiro CSV, mais de 5 milhões de caracteres organizados em frases com grande repetição separadas por vírgulas;
- *jquery-3.6.0.js* – biblioteca jQuery, cerca de 300 mil caracteres;
- *random.txt* – coleção aleatória de 100 mil caracteres.

O sistema operativo Ubuntu foi escolhido como suporte para o programa devido às ferramentas de compressão que apresenta nativamente, facilitando o *benchmarking*, e às implementações de algoritmos encontradas, maioritariamente compatíveis com Linux.

Foram explorados os seguintes sistemas e as seguintes componentes cujo código está disponível na Internet:

- LZMA, [14]
- bzip2, [15]
- gzip (algoritmo Deflate), [16]
- rzip, [17]
- Codificação aritmética (ARITH) em C++, [18]
- BWT em Rust, [19]
- MTF em Java, [20]
- RLE em C, [21]
- LZ77 em C, [22]

A. Critérios de benchmarking

Para avaliação do desempenho foram consideradas a taxa de compressão e as velocidades de compressão e descompressão.

A taxa de compressão, calculada pela seguinte fórmula, teve o maior peso:

$$Taxa\ de\ compressão\ = \frac{tamanho\ antes\ da\ compressão}{tamanho\ após\ compressão} \quad (1)$$

De acordo com o Teorema de Codificação da Fonte (TCF), um sinal de tamanho *S* e entropia *H* não pode ser comprimido para menos de *S*×*H bits* com um código perfeito sem ocorrer perda de informação [23]. A melhor taxa de compressão alcançável com um código entrópico que seja perfeito e unicamente descodificável é assim calculável e inversamente proporcional a *H*:

$$Taxa\ de\ compressão\ máxima(TCF) = \frac{S\ bytes}{S \times H\ \div\ 8\ bytes} \quad (2)$$

Nas TABELA I. , TABELA II. e TABELA III. estão representadas as taxas de compressão e os tempos médios de compressão e descompressão atingidos pelas ferramentas do Ubuntu para os ficheiros do *dataset*. As linhas na TABELA I. indicam *baselines* calculadas através de (2) a partir das entropias dos ficheiros originais, bem como das suas entropias após aplicação de transformadas. As entropias foram obtidas usando a ferramenta ent [24].

Destaca-se que nenhum dos sistemas testados atingiu taxas de compressão superiores aos máximos teóricos do ficheiro *random.txt* original ou alterado com poucas transformadas. De

facto, a aplicação de transformadas aumenta a sua entropia já elevada, reduzindo a compressão teoricamente possível.

B. *Proposta genérica*

Tendo em conta que o bzip2 apresenta as melhores taxas de compressão para três dos quatro ficheiros avaliados, bem como boas velocidades de compressão/descompressão, foi seguido o seu modelo descrito na secção II.D.1) para o desenvolvimento de uma solução genérica.

TABELA I. TAXAS DE COMPRESSÃO DO ESTADO DA ARTE E BASELINES

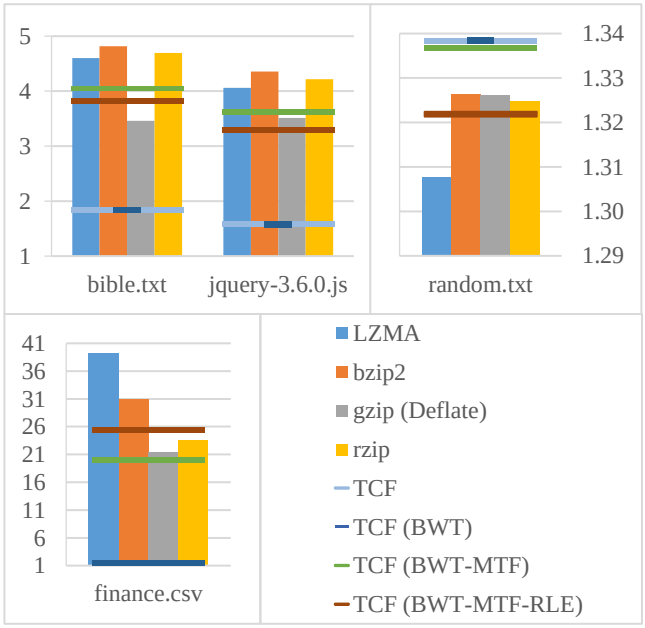


TABELA II. TEMPO MÉDIO DE COMPRESSÃO DO ESTADO DA ARTE

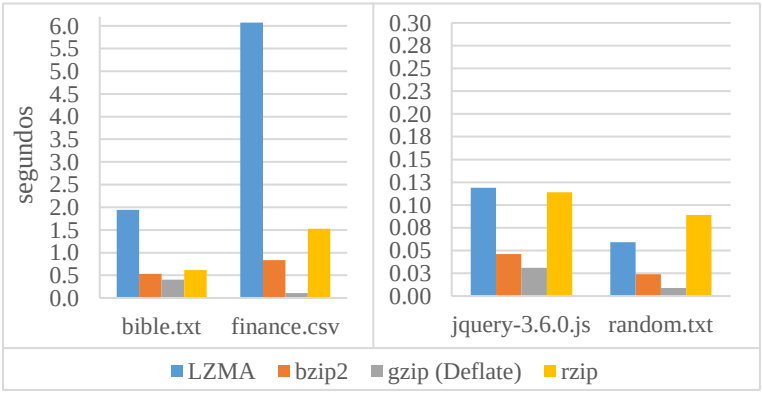
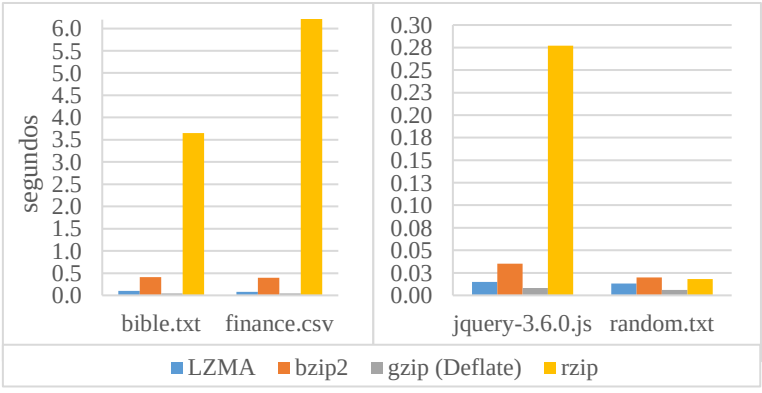


TABELA III. TEMPO MÉDIO DE DESCOMPRESSÃO DO ESTADO DA ARTE



substituição da codificação de Huffman usada pelo bzip2, de forma a negar o *overhead* da árvore binária e da codificação de símbolos com valores de *bits*/símbolo inferiores a 1.

Como RLE realiza alguma compressão, a sua aplicação após as transformadas leva a um aumento drástico da entropia e, consequentemente, uma queda no máximo teórico de compressão (TABELA I.). É expectável que a posterior codificação aritmética seja ineficiente, pelo que foi também avaliada uma segunda versão do *codec* generalista, denominada CODEC1, que exclui a fase de RLE.

Resumindo, as seguintes sequências de módulos compõem os algoritmos genéricos:

CODEC0: BWT – MTF – RLE – ARITH;

CODEC1: BWT – MTF – ARITH.

C. *Proposta para o ficheiro finance.csv*

O ficheiro *finance.csv* apresenta longas sequências de símbolos repetidos que são vantajosas para compressão por dicionário, o que se verifica nos resultados significativamente superiores do LZMA (TABELA I.). Assim, foi procurada uma solução específica para este ficheiro, denominada CODEC2, baseada no modelo descrito na secção II.D.4).

O *encoder* de dicionário escolhido foi LZ77 nas suas configurações máximas: um *look-ahead* de 255 *bytes* e uma *search window* de 65535 *bytes*. A codificação aritmética foi também utilizada nesta implementação, substituindo o *Range encoding* usado pelo LZMA, por obter melhores taxas de compressão sacrificando velocidade [4].

Assim, a solução específica para *finance.csv* é a seguinte sequência:

CODEC2: LZ77 – ARITH.

D. *Implementação como programa*

Foi criado um ficheiro executável na *shell* que implementa as componentes em sequência. O *output* de cada módulo do algoritmo é escrito em ficheiros temporários que são eliminados no final do processo. O resultado é um ficheiro que contém os dados do original, comprimidos sem perda de informação. A descompressão envolve utilizar o modo de descompressão de cada módulo na ordem inversa.

IV. RESULTADOS E DISCUSSÃO

Durante o desenvolvimento dos *codecs* genéricos verificou-se que a Transformada de Burrows-Wheeler aplicada a *finance.csv* não era invertida corretamente, apesar do erro não se verificar para outros ficheiros CSV de tamanho semelhante. Assim, apenas os resultados de CODEC2 foram considerados para este ficheiro.

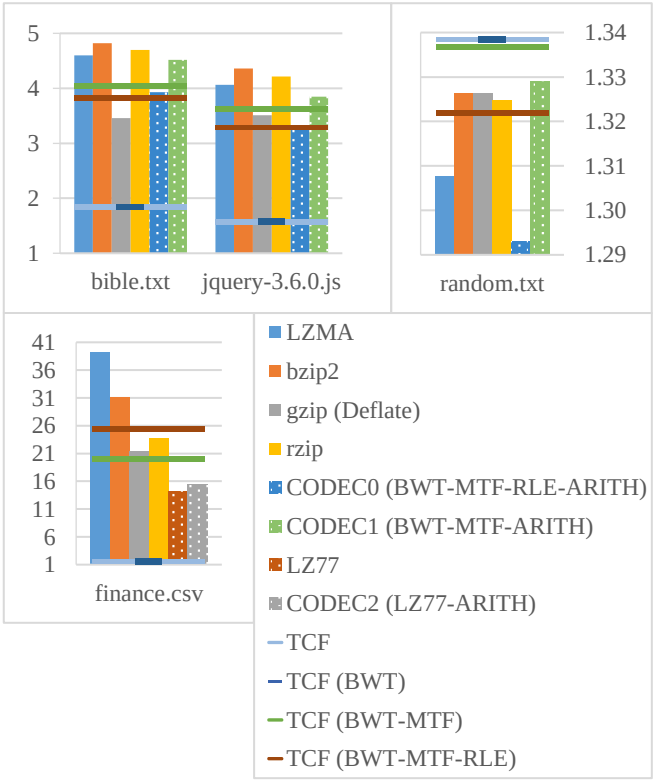
Na A apresentam-se as taxas de compressão obtidas pelos *codecs* desenvolvidos em comparação com as *baselines* e com os sistemas do Ubuntu.

Para comparações de velocidade foram realizadas 20 compressões e descompressões com os *codecs*, usando um computador com CPU Intel i7-9750H a 4.00GHz e armazenamento *solid-state drive*. Os tempos médios são apresentados na TABELA V. e na TABELA VI. acompanhados dos valores obtidos pelos sistemas do Ubuntu.

A. *Codec genérico*

Confirmou-se a previsão de que a implementação de RLE antes de codificação aritmética obteria taxas de compressão mais baixas. O CODEC0 apresentou também um tempo de compressão demasiado elevado. Esta implementação foi consequentemente descartada em favor do CODEC1.

TABELA IV. BENCHMARKING – TAXAS DE COMPRESSÃO DOS CODECS



O CODEC1 atinge taxas de compressão comparáveis aos sistemas de estado da arte, superando-os no caso de *random.txt*, e com valores superiores ao esperado (linha verde) para *bible.txt* e *finance.csv*. Assume-se que o modelo efetivamente utilizado para o cálculo de probabilidades na implementação de codificação aritmética poderá ser mais preciso ou representativo dos dados em comparação com o que é utilizado no cálculo da entropia de 1ª ordem, o que permite alcançar uma taxa de compressão superior à definida pelo TCF.

Durante testes isolados de cada componente verificou-se que o módulo de codificação aritmética atinge uma taxa de compressão marginalmente superior a CODEC1 para *random.txt*. Isto deve-se à ausência de redundância nos dados deste ficheiro: o grande número de caracteres não permite o seu agrupamento usando BWT, e a transformada MTF apenas pode explorar grandes correlações locais que não ocorrem neste ficheiro. No entanto, os resultados para o resto do *dataset* são taxas de compressão próximas de 1, pelo que o uso direto de codificação aritmética não é viável para uso geral.

A velocidade do CODEC1 foi relativamente baixa, especialmente para a compressão (0). Isto era esperado, visto que o *output* de cada fase do algoritmo é armazenado no disco antes de ser processado pelo módulo seguinte, introduzindo um *overhead* na forma de leitura e escrita.

B. *Codec específico para o ficheiro finance.csv*

Comparado com LZ77 isolado (A), o CODEC2 indica uma pequena vantagem em aplicar codificação aritmética ao resultado da codificação por dicionário: os blocos codificadores de referências a entradas do dicionário construído poderão continuar a apresentar alguma redundância que permite codificação entrópica eficiente.

TABELA V. BENCHMARKING – TEMPO DE COMPRESSÃO DOS CODECS

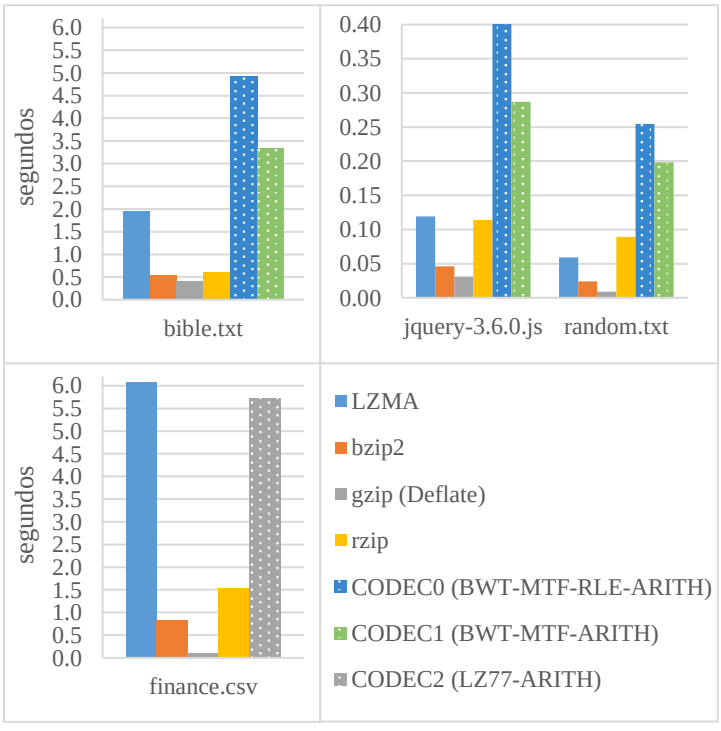
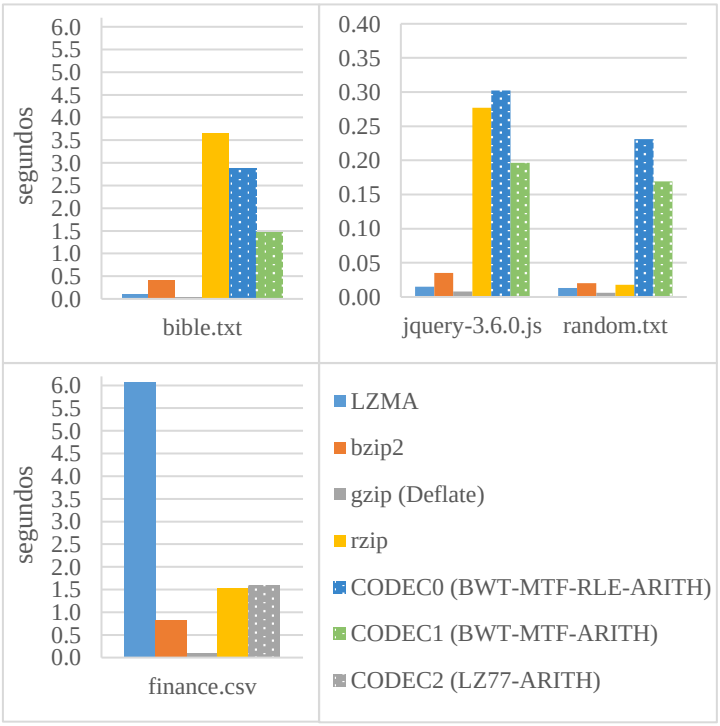


TABELA VI. BENCHMARKING – TEMPO DE DESCOMPRESSÃO DOS CODECS



Os tempos de compressão e descompressão do CODEC2 são semelhantes aos do LZMA e rzip respetivamente, no entanto a taxa de compressão é muito inferior às obtidas pelas ferramentas do Ubuntu. A baixa velocidade é novamente justificável pelo *overhead* das leituras e escritas sucessivas dos dados manipulados, e a implementação de LZ77 poderá também ter um custo computacional demasiado elevado nas configurações utilizadas.

V. Conclusão

Neste trabalho foram exploradas técnicas de compressão *lossless* de texto e concebidos dois algoritmos que as aplicam com sucesso a quatro ficheiros. O programa criado para implementar os algoritmos obtém boas taxas de compressão para três dos ficheiros, mas apresenta velocidades relativamente baixas de compressão e descompressão. Face aos resultados obtidos, o objetivo seguinte seria a alteração do programa para manipular os dados em memória, eliminando o *overhead* da leitura e escrita no armazenamento em disco.

A implementação de LZ77 poderá ser alvo de melhorias, nomeadamente a procura do conjunto de configurações que obtêm a melhor relação entre velocidade e taxa de compressão. Propõe-se também uma análise aprofundada da implementação de BWT utilizada, de forma a identificar o problema na transformação do ficheiro *finance.csv* e desenvolver um *codec* verdadeiramente genérico para o *dataset*.

Referências

[1] N. Sharma, J. Kaur e K. Kaur, "A Review on various Lossless Text Data Compression Techniques," *Research Cell: An International Journal of Engineering Sciences,* vol. 2, pp. 58-63, December 2014.

[2] S. Shanmugasundaram e L. Robert, "A Comparative Study Of Text Compression Algorithms," *ICTACT Journal on Communication Technology,* vol. 02, nº 04, pp. 444-451, 2011.

[3] T. Bell, I. Witten e J. Cleary, "Modeling for text compression," *ACM Computing Surveys,* vol. 21, nº 4, pp. 557-591, 1989.

[4] T. B. Terriberry, "On the Overhead of Range Coders," 7 3 2008. [Online]. Available: https://people.xiph.org/~tterribe/notes/range.html#T1. [Acedido em 24 Nov 2021].

[5] A. Jain e K. I. Lakhtaria, "Comparative Study of Dictionary based Compression Algorithms," *IJCSNS International Journal of Computer Science and Network Security,* vol. 16, nº 2, pp. 88-92, 2016.

[6] R. Rădescu, "Transform Methods Used in Lossless Compression of Text Files," *Romanian Journal of Information Science and Technology,* vol. 12, nº 1, pp. 101-115, 2009.

[7] "bzip2 and libbzip2, version 1.0.8," Sourceware.org, [Online]. Available: https://sourceware.org/bzip2/manual/manual.html. [Acedido em 21 Nov 2021].

[8] "bzip2," Bzip.org, [Online]. Available: http://www.bzip.org/. [Acedido em 21 Nov 2021].

[9] "rzip," Rzip.samba.org, [Online]. Available: https://rzip.samba.org/. [Acedido em 21 Nov 2021].

[10] L. P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," 03 1996. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1951. [Acedido em 25 Nov 2021].

[11] A. Feldspar, "An Explanation of the Deflate Algorithm," 23 08 1997. [Online]. Available: https://zlib.net/feldspar.html. [Acedido em 25 Nov 2021].

[12] "7z Format," 7-zip.org, [Online]. Available: https://7-zip.org/7z.html. [Acedido em 24 Nov 2021].

[13] I. Pavlov, *LZMA specification (DRAFT version),* 2015.

[14] "xz, unxz, xzcat, lzma, unlzma, lzcat - Compress or decompress .xz and .lzma files," [Online]. Available: https://linux.die.net/man/1/lzma.

[15] J. Seward, "bzip2, bunzip2 - a block-sorting file compressor," [Online]. Available: https://linux.die.net/man/1/bzip2.

[16] Free Software Foundation, Inc., "gzip, gunzip, zcat - compress or expand files," [Online]. Available: https://linux.die.net/man/1/gzip.

[17] A. Tridgell, "rzip - a large-file compression program," [Online]. Available: https://linux.die.net/man/1/rzip.

[18] D. Kravchenko, "Arithmetic Coding for Data Compression," 09 Março 2018. [Online]. Available: https://github.com/dmitrykravchenko2018/arithmetic_coding.

[19] izflare, "BWT," 23 Abril 2019. [Online]. Available: https://github.com/izflare/BWT.

[20] fujiawu, "Burrows-Wheeler Data Compression," 28 Maio 2015. [Online]. Available: https://github.com/fujiawu/burrows-wheeler-compression.

[21] A. Soursou, "RLE compression implementation in C," 15 Março 2020. [Online]. Available: https://github.com/ChuOkupai/rle-compression.

[22] D. Costa e drsptr, "LZ77 compressor and decompressor," 26 Outubro 2015. [Online]. Available: https://github.com/cstdvd/lz77.

[23] M. Zbili e S. Rama, "A Quick and Easy Way to Estimate Entropy and Mutual Information for Neuroscience," *Front. Neuroinform.,* 15 Junho 2021.

[24] J. Walker, *ent - A Pseudorandom Number Sequence Test Program,* 2008.