CODEC, não destrutivo, para imagens monocromáticas

Tomás B. Mendes
2019232272

Joel P. Oliveira
2019227468

João C. V. Silva
2019217672

***Abstract*—No seguimento do trabalho prático 1, neste artigo serão explorados alguns algoritmos de compressão não destrutivos (lossless) para imagens monocromáticas (escalas de cinza). Assim, com o objetivo de determinar os algoritmos mais eficientes e eficazes, serão testados os seguintes algoritmos de compressão e transformação no domínio: *LZW*, *RLE*, *Delta Filtering, Burrows Wheeler, Move-to-front, Huffman Codes*. Os algoritmos serão comparados em termos de velocidade de compressão e rácio de compressão.**

***Keywords*—compressão, *lossless*, imagens monocromáticas, transformação, velocidade compressão, rácio de compressão**

I. Introdução

Atualmente, o acesso à internet é fundamental, seja por lazer, por pesquisa para trabalho ou para estudo, ou até mesmo para ações mais avançadas, como operações médicas, e ataques aéreos etc. Assim, é essencial que a transferência de informação emissor-recetor seja feita o mais rapidamente possível, de modo a evitar perdas de tempo que removam eficiência ao trabalho do utilizador ou que causem problemas catastróficos, utilizando o exemplo acima, errar o alvo do míssil balístico. Se os ficheiros ou sites a que estamos a tentar aceder forem pequenos, estes carregam mais depressa, pois, logicamente, tendo uma transferência de dados constante, quanto menor o tamanho do ficheiro de envio, menor o tempo que esse processo demora. É aqui que entram os algoritmos de compressão.

Estes podem conseguir reduzir significativamente o tamanho que um ficheiro ocupa, tendo como custo operações computacionais adicionais. Existem dois tipos de compressão, *lossless* e *lossy*. Tal como o nome sugere, num algoritmo do tipo *lossless* (sem perda)não se perde informação aquando da compressão da mesma. A compressão é feita de modo à informação poder ser representada numa quantidade de bits inferior à original, mas mantendo a sua integridade. A compressão não destrutiva é usada em situações em que é essencial a informação manter-se intacta, tal como um executável. Por outro lado, os algoritmos de compressão destrutivos, reduzem e/ou removem bits desnecessários, como frequências superiores às capacidades de audição normal de uma pessoa. Assim, é impossível reverter completamente a conversão, sendo o resultado da descompressão uma aproximação do original. Como uma grande quantidade de informação desnecessária é eliminada, este tipo de algoritmo consegue alcançar um rácio de compressão mais alto do que os algoritmos não destrutivos.

II. Estado da Arte

A. Algortimos de compressão básicos

Existem já vários algoritmos de compressão não destrutiva. Alguns destes fazem uso prévio dos modelos estatísticos da informação presente, para obterem uma melhor eficiência na compressão, pois esta aproveita o conhecimento na probabilidade concreta de ocorrência de cada símbolo. Por este mesmo motivo, estes algoritmos têm de fazer uma pré-análise da informação a ser comprimida, o que aumenta a complexidade da compressão, e têm também de guardar o modelo probabilístico da ocorrência dos símbolos, para o momento da descompressão. Alguns exemplos clássicos deste tipo de método são os códigos de Huffman (Huffman Codes) [4], a codificação Aritmética [4], Golomb coding [4].

Estes são também algoritmos de compressão entrópicos, ou seja, têm como base aproximar o comprimento médio dos símbolos existentes na informação à sua entropia (comprimento mínimo teórico para a representação média de um símbolo)

Outros não necessitam do conhecimento a *priori* do modelo de probabilidades dos símbolos, pois constroem um dicionário dinâmico que contem referências para símbolos ou para um seguimento concreto de símbolos, enquanto decorre a compressão/descompressão. Este é o caso de algoritmos como o *LZ-77* (Lempel-Ziv 1977) [4] ou *LZ-78* (Lempel-Ziv 1978) [4], ou até mesmo uma evolução deste, como o *LZW* (Lempel-Ziv-Welch) [2].

Assim, para este tipo de compressão, é expectável um menor espaço para o ficheiro comprimido, mas é, consequentemente, necessária mais memória aquando da compressão, pois as dimensões do dicionário são, no caso dos algoritmos *LZ-78* e *LZW* bastante grandes. Para evitar isto, por vezes, limita-se o tamanho do dicionário, e inicia-se um dicionário novo quando o anterior atinge o limite definido. Em contrapartida, caso exista relação na imagem entre a informação já codificada e a informação por codificar, esta perde-se, reduzindo a eficiência de compressão destes algoritmos.

No caso do *LZ-77* este não guarda um dicionário, mas sim utiliza uma janela deslizante para codificar com base na ocorrência de símbolos anteriormente, dentro do espaço limitado da janela. Este codifica do pressuposto que os símbolos se repetem maioritariamente dentro do espaço limitado da janela, o que pode não acontecer.

Um caso de outro algoritmo que não faz uso de um processo a priori é a codificação *Run-Lenght Encoding* (*RLE*). Este é um algoritmo bastante simples, nem sempre eficiente, que modifica uma sequência de valores contíguos repetidos, por um novo símbolo valor-quantidade. Ou seja, quantos mais valores repetidos contíguos existirem, maior a compressão por este alcançada. Isto só é feito quando o símbolo ocorre mais do que duas vezes seguidas, ou o ficheiro teria o seu tamanho aumentado em vez de reduzido.

B. Algoritmos Transformação do domínio da fonte

De modo a auxiliar a compressão, existem alguns métodos que aumentam a redundância dos valores contidos na informação, sem estes perderem o seu significado, ou seja, de modo a estas alterações poderem ser revertidas. Quanto maior for a redundância de uma fonte, maior será o número de símbolos repetidos nela, portanto maior a compressão da fonte.

1) Delta Filters: Muitas imagens têm dependências lineares entre pixéis [5], pelo que bons algoritmos para transformar imagens monocromáticas, como por exemplo o *Delta Filters*[5], que representa a informação por "*deltas*", ou seja, pela distância entre a intensidade anterior e a própria.

Este método pode ser executado a nível de relação entre pixéis nas linhas, ou nas colunas, ou até uma média entre ambos. Pode ter níveis de aumento de redundância elevados, quando a informação apresenta uma relação linear entre as intensidades dos pixéis, e é facilmente invertível.

2) Burrows-Wheeler Transform (BWT): Cria todas as ordenações possíveis entre os símbolos, com *shifts* lógicos entre eles, ordenando-os, seguidamente, por ordem lexicográfica. A última coluna da matriz obtida por este processo é, então a transformada de *BWT*, que tem tendência a agrupar símbolos iguais. Devido a serem necessárias todas as sequências possíveis para codificar a informação, este método induz latência quando a fonte a ser transformada é grande.

3) Move-To-Front Transform (MTF): É também uma transformada aplicável ao domínio de informação, para compressão não destrutiva. Esta transformada tem por base alterar os valores existentes, com base na sua posição no alfabeto. Sempre que um valor é utilizado, a posição no alfabeto é alterada para a primeira, mantendo-se os restantes pela mesma ordem.

C. Algoritmos de compressão avançados

Existem vários algoritmos mais avançados que os clássicos, os chamados algoritmos modernos.

Estes, por norma consistem de algoritmos formados por combinações de algoritmos básicos e de algoritmos de transformações de domínio, pela ordem correta, obtendo resultados com uma eficiência muito superior

Um algoritmo bastante utilizado neste domínio é o *Deflate*[1]. Este é implementado com recurso a dois outros algoritmos, *LZ-77* e *Huffman Codes*. Inicialmente, é feita a compressão através do algoritmo LZ77, sendo o ficheiro dividido em blocos de tamanho arbitrário e posteriormente codificado com recurso a árvores de Huffman

Este algoritmo carece, no entanto, tanto das vantagens do *LZ-77*, como as suas desvantagens.

Bzip2 é um algoritmo com uma compressão, por norma superior à do *Deflate*, mas com uma velocidade de compressão bastante inferior, devido a complexidade do seu procedimento.

Este faz uso a uma combinação de vários algoritmos, tais como o *RLE*, o *BWT*, o *MTFT*, os *Huffman Codes*, e ainda o *Delta Encoding*.

De acordo com [4] o *LOCO-I*, utilizado como *standart* do *JPEG-LS*, é um dos recentes algoritmos com melhor rácio compressão/complexidade, o que num bom algoritmo de compressão.

Este, primeiramente faz uso de um algoritmo de previsão de esquinas, de modo a descorrelacionar a informação. De seguida usa um algoritmo de modelação de contexto, de modo a verificar relações entre pixéis em determinadas zonas, através do gradiente médio da zona, calculado através da média de vários "contextos". Por fim é utilizada a codificação de *Golomb-Rice* e, seguidamente, procede-se a um *RLE* em zonas lisas da imagem.

Existem muitos outros algoritmos de compressão modernos. Alguns exemplos são o *PPM (Prediction by Partial Matching)*, o *LZMA (Lempel-Ziv-Markov Chain Algorithm)*, o *FELICS (Fast Efficient and Lossless Image Compression System)*, etc. Uns com maiores níveis de compressão, outros com uma velocidade de compressão melhor.

III. Descrição de Algoritmos Simples

- LZ-77 - *Lempel-Ziv 77*
- LZ-78 - *Lempel-Ziv 78*
- LZW - *Lempel-Ziv Welsh*
- HF - *Huffman Codes*
- RLE - *Run Length Encoding*
- BWT - *Burrows-Wheaeler Transform*
- MTF - *Move to Front*

Run Length Encoding
1) O Algoritmo procura repetições de símbolos numa fonte. 2) As repetições de um símbolo são substituídas pelo símbolo e pelo número de vezes que se repete.

TABELA I
ALGORITMO RLE

Huffman Code
1) O algoritmo cria uma tabela com as repetições dos carateres. 2) Os 2 valores menos frequentes são removidos e agrupados. Esses carateres agrupados são repostos na tabela como uma entrada e a sua frequência é a soma das frequências dos carateres removidos. 3) É repetido o passo 2 até só haver uma entrada na tabela. **Nota:** O resultado é então uma árvore (*Huffman tree*). Percorrendo a arvore, é possível gerar o *Huffman code* de cada símbolo.

TABELA II
ALGORITMO *Huffman Codes*

LZ-77
1) O Algoritmo percorre a fonte, com uso de uma janela deslizante, de modo a pesquisar ocorrências anteriores de sequências de símbolos. 2) Se padrão já existir no dicionário, procurar padrões que comecem com o próximo símbolo, caso contrário adicionar o padrão ao dicionário.

TABELA III
ALGORITMO *LZ-77*

LZ-78
1) Neste neste caso é utilizado um dicionário explícito. 2) A codificação é feita com recurso ao índice no dicionário e ao código do próximo caracter. 3) À medida que se percorre a fonte vão sendo adicionados e codificados no dicionário os padrões únicos encontrados. 4) Se um símbolo já se encontrar no dicionário lê-se o próximo símbolo. Se a sequência destes dois símbolos já se encontrar no dicionário, lê-se o próximo símbolo. Caso contrário, adiciona-se a sequência ao dicionário.

TABELA IV
ALGORITMO *LZ-78*

LZW
1) Variante do algoritmo LZ-78 em que são apenas enviados os índices no dicionário, em vez de ser também enviado o código do próximo caracter. 2) O dicionário contém à partida entradas para todos os símbolos do alfabeto.

TABELA V
ALGORITMO *LZW*

Burrows-Wheeler Transform
1) Cria uma tabela NxN (onde N é o tamanho da frase) com todas as diferentes possibilidades da frase. O Primeiro elemento é a frase e os seguintes correspondem à frase anterior respetiva, sofrida de um shift left. 2) Ordena essa tabela de forma lexicográfica. 3) A transformação é a ultima coluna desta tabela ordenada, sendo que se guarda a posição da linha que continha a frase original.

TABELA VI
TRANSFORMAÇÃO *BWT*

Delta filtering
1) Substitui-se cada valor pela diferença entre o próprio e o anterior. 2) Por norma é aplicado a linha, coluna, ou média entre ambos. 3) No caso da linha, a primeira mantém-se, no caso da coluna, a primeira mantém-se, no caso da média, tanto a primeira linha como a primeira coluna se mantêm.

TABELA VII
TRANSFORMAÇÃO *Delta Filter*

Move to Front
1) Para esta transformação é utilizado o alfabeto da fonte. Mais precisamente os index do alfabeto. 2) Cada símbolo encontrado é reposto pelo index respetivo no momento. De cada vez que um símbolo e utilizado, a sua posição na lista do alfabeto altera para a inicial, aumentando assim a redundância.

TABELA VIII
TRANSFORMAÇÃO *MTF*

IV. ESCOLHA DOS ALGORITMOS

A. Algoritmos de Transformação da Fonte

Como referenciado em II-B, a literatura refere que em grande parte das imagens os pixeis têm relações lineares com os pixeis das suas proximidades.

Inicialmente foram comparadas três transformações não destrutivas com potencialidade de aumentar a redundância estatística das fontes. Todas estas se baseavam em distâncias entre pixeis próximos. Distância entre o pixel e o pixel anterior na linha. Distância entre o pixel e o pixel anterior na coluna ou distância média entre o pixel anterior na coluna e o pixel anterior na linha. Em média, dentro destas três transformações, a que obteve menor entropia foi a transformação aplicada à diferença entre as colunas. Assim, foi escolhida esta transformação, pois a redução da entropia implica um aumento na redundância estatística da fonte, e uma maior redundância estatística na fonte é uma ajuda à compressão da mesma.

De facto, a transformação por deltas executado ao nível das colunas foi a que obteve melhores compressões, como é verificável na tabela XI).

Transformação	*DeltaRows*	*DeltaColumns*	*DeltaAverage*
Entropia (bits/simb)	2.335	2.329	2.987

TABELA IX
ENTROPIA MÉDIA DOS FICHEIROS APÓS TRANSFORMAÇÃO

Foram testados ainda outros algoritmos de transformação da informação da fonte, mas com resultados insatisfatórios para o *dataset* em questão, como é o caso dos *Gray Codes*[2]. Estes foram utilizados para outra tentativa de compressão, que tratava a fonte como oito ficheiros binários diferentes. Mas os resultados não foram positivos, devido ao *overhead* do algoritmo, no caso do proposto por nós, do dicionário com os códgios de *Huffman*.

1) Delta Filter: Na tabela X, abaixo, está representado o valor da entropia do *dataset* em questão, com a entropia da imagem transformada com o *Delta Filter* ao longo da coluna.

Ficheiro	egg.bmp	landscape.bmp	pattern.bmp	zebra.bmp
s/ *Delta*	5.72	7.42	1.83	5.83
Coluna	2.75	2.77	0.64	3.18

TABELA X
ENTROPIA DO *dataset* (*bits/simb*.)

Pode-se notar uma redução grande na entropia entre o *dataset* original e o transformado, que vai de aproximadamente 45% a 63%, respetivamente.

Visto que este algoritmo não altera a estrutura dos dados, nem a quantidade de informação, presentes na fonte, esta redução entrópica implica que a redundância estatística aumentou significativamente, como era esperado.

2) Gray Codes: Estes fazem uma transformam cada número inteiro, de modo a diferi-lo apenas um *bit* do número transformado anterior.

A ideia de tentar utilizar estas transformação surgiu do facto de o metodo inicial não estar a obter os resultados esperados. Assim, visto que a compressão de uma imagem binária é bastante superior à de uma imagem monocromática, decidimos dividir, aquando da compressão, a imagem em 8 imagens binárias, cada uma constituída pelo *bit* 'i' de cada *byte*.

Os *Gray Codes* mostraram-se úteis para este cenário, pois a quantidade de *bits* seguidos iguais aumenta consideravelmente conforme a significância do *bit*.

Tendo em conta que desta forma estamos a trabalhar com imagens binárias, sabemos que cada bloco 2x2 terá reduzidas possibilidades. Mais concretamente existem 2^4 possibilidades. A cada parte binária da imagem realizámos uma transformação por blocos, onde cada bloco seria transformado no inteiro correspondente.

Foi testado aplicar o algoritmo *LZMA*[13] após estas transformações, no entanto os resultados não foram positivos. O *overhead* torna-se demasiado grande, pois é necessária informação extra para a descompressão de não só para uma fonte, mas sim para oito, pelo que este método não se tornou viável. Apesar de este não ser proposto, está presente nas comparações entre algoritmos.

B. Algoritmos de Compressão da Fonte

Devido ao tempo reduzido para a execução deste projeto decidimos utilizar algoritmos mais simples de implementar. Assim, o principal algoritmo escolhido, que é baseado em dicionário, foi o *LZW*. A compressão deste aumenta consoante a repetição de séries de valores na fonte.

De forma a evitar consumo excessivo de memória pelo dicionário, este foi limitado a um valor fixo. Uma forma de melhorar a eficiência deste seria não elimina-lo totalmente cada vez que fosse necessário espaço, mas sim ir eliminando consoante os valores já tinham sido utilizados ou não, conforme demonstrado em [16]. Não conseguimos executar esse dicionário adaptativo.

Assim, perde-se a dependência entre valores distantes na fonte, pois certos padrões podem repetir-se apenas fora do espaço da janela, o que reduz a compressão deste algoritmo.

O algoritmo entrópico escolhido para comprimir a fonte já comprimida por dicionário, acabou por ser a codificação de *Huffman*. Apesar dos códigos aritméticos terem um nível de compressão superior, para o tamanho de uma fonte como as imagens em causa, é necessária uma precisão elevadíssima, o que em *Python*, não é fácil de alcançar.

Os códigos de *Huffman* aproveitam-se do facto dos valores da fonte se repetirem, colocando símbolos mais frequentes referenciados por códigos mais pequenos. Portanto a limitação do dicionário do *LZW* é um aspeto positivo para gerar estes códigos. Deste modo o tamanho do dicionário não pode ser demasiado pequeno, de modo ao algoritmo *LZW* funcionar, mas também não deve ser grande de modo ao overhead da árvore de *Huffman* ser demasiado grande.

Assim, foram testados dicionários limitados a várias grandezas, nomeadamente de 2^{10} até 2^{16}, sendo que a melhor compressão no ficheiro final foi com um dicionário limitado a 2^{13}. Como foi dito, isto deve-se ao facto do *overhead* da árvore de *Huffman* ser mais pequena.

V. IMPLEMENTAÇÃO

Tal como no TP1, a leitura da imagem é feita com a função *imread()* do módulo *image* do *matplotlib*[10] e é convertida num *array* de *numpy*[9] com *data type signed int*.

Após a leitura, como foi referido, o algoritmo proposto começa por transformar a informação através de um *Delta filter* aplicado às colunas.

Seguidamente comprime a informação com uso do algoritmo *LZW*, com limite de 2^{13} elementos no dicionário, como referido.

O algoritmo *LZW* foi baseado no agoritmo descrito em [11].

Visto que o *Delta filter* é aplicado, os valores possíveis na imagem passam de [0,255] a [-255,255], pelo que o dicionário do *LZW* tem que conter sempre referências para esses valores. Assim o dicionário nunca tem menos menos de 511 elementos.

Esta compressão devolve um *array* unidimensional. Isto deve-se ao facto de as linhas poderem passar a ter um número diferente de elementos. Os *arrays* de *numpy*, bidimensionais, não permitem que isto aconteça, daí decidirmos guardar a informação de forma unidimensional, em vez de alterar-mos a estrutura de dados onde a guardarmos, pois trabalhar a interação com *arrays* de *numpy* é bastante mais prática.

Para não se perder informação, as dimensões, da imagem, têm que ser mantidas. Assim aquando da compressão desta, as dimensões são guardadas numa variável e posteriormente escritas no ficheiro de modo a ser possível reverter a compressão.

Após esta compressão, é criada uma tabela de referências entre os símbolos da fonte e a sua representação em códigos de *Huffman*, com recurso à função *from_data()* da classe *HuffmanCodec* do módulo disponível no *GitHub*[1], *dahuffman*[8].

Após a tabela ser criada, a fonte é então comprimida em códigos de *Huffman* com recurso à função *encode()* da classe com que foi criada a tabela.

Por fim a tabela com a codificação de *Huffman* é guardada num ficheiro com a extensão *.dat*, seguidamente das dimensões da imagem e, por fim, da fonte comprimida.

Este ficheiro é escrito com a função *dump()* do módulo *Pickle*[12] do *Python*, que é chamada três vezes de modo a escrever os 3 objetos diferentes.

VI. Comparação dos algoritmos

Quando comparamos algoritmos, podemos compará-los de várias maneiras, a sua necessidade. Podemos comparar algoritmos em termos de velocidade de compressão, de rácio de compressão ou num equilíbrio entre ambos. O rácio de compressão dá-se por:

$$Compretion\_Ratio = \frac{Uncompressed\_Size}{Compressed\_Size}$$

Por norma a velocidade de compressão tende a diminuir com o aumento da complexidade do algoritmo de compressão. No entanto, isto não implica que a compressão melhore. De forma a verificarmos a eficiência do nosso algoritmo, foi feita a comparação da sua compressão com alguns algoritmos de compressão do já implementados nos módulos do *Python*, nomeadamente [13] [14] e [15]

Deste modo, para comparar-mos algoritmos em termos de velocidade de compressão, é necessário que eles estejam igualmente otimizados, o que não é o caso.

Alguns dos algoritmos podem até estar desenvolvidos noutras linguagens, com C[17] ou C++[18], mas com compatibilidade para *Python*,

Assim a comparação realizada será apenas ao nível do rácio de compressão e não da velocidade de execução.

Primeiramente vamos comparar as compressões conforme a transformada aplicada. A maior compressão foi sempre obtida com a transformação delta aplicada às colunas, como se pode verificar na tabela XI.

Ficheiro	egg.bmp	landscape.bmp	pattern.bmp	zebra.bmp	Média
Linha	3.09	2,67	16.10	2.46	6.08
Coluna	3.55	3.05	17.36	2.88	6.71
Media	3.44	2.85	15.00	2.78	6.02

TABELA XI
Compression Ration do LZW com os vários *Delta filters*

Os algoritmos com os quais vamos comparar o nosso são o algoritmo da compressão pelo modulo *zLib*[15] do *python*, a compressão pelo modulo *LZMA*[13] do *python*, a compressão pelo modulo *bz2*[14] do *python* e ainda com o *PNG*. Os resultados podem ser vistos na Tabela XII

Ficheiro	egg.bmp	landscape.bmp	pattern.bmp	zebra.bmp
Proposto	3.55	3.05	17.36	2.88
Bzip2	3.74	3.15	26.60	2.98
Lzma	3.56	3.39	24.86	2.96
Zlib	2.66	2.46	19.30	2.18
PNG	3.83	3.30	21.02	3.06
2º Testado	3.07	2.69	13.15	2.40

TABELA XII
Compression Ration do algoritmo proposto e de outros populares

O algoritmo proposto obteve pior compressão que o do *PNG* em todas as imagens. Dentro das comparações que foram feitas, o único algoritmo superado foi o algoritmo de compressão do módulo *zLib*, no geral.

Apesar disto, a compressão do algoritmo proposto tem um nível de compressão aceitável.

VII. Conclusão

Este projeto tinha como principal objetivo explorar algoritmos de compressão não destrutivos para imagens em escalas de cinza. Para tal, foram exploradas algumas transformações estatísticas que melhoravam, em alguns casos, o desempenho dos algoritmos de compressão testados e diminuindo assim, o tamanho total ocupado pela imagem.

Concluímos assim, com um resultado positivo, tendo o algoritmo proposto obtido uma compressão superior a 50% no *dataset* fornecido, sendo o *PNG* cerca de 30% melhor do que o algoritmo proposto.

Referências

[1] Knuth: Computers and Typesetting, http://www-cs-faculty.stanford.edu/~uno/abcde.html

[2] A. Gupta, A. Bansal and V. Khanduja, "Modern lossless compression techniques: Review, comparison and analysis," 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, 2017, pp. 1-8, doi: 10.1109/ICECCT.2017.8117850.J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] Tnwei, "python-rle", https://github.com/tnwei/python-rle

[4] Lina J., K. (2009). Chapter 16 - Lossless Image Compression. In A. Bovik (Ed.), The Essential Guide to Image Processing (pp. 385–419). Academic Press, https://doi.org/https://doi.org/10.1016/B978-0-12-374457-9.00016-0

[5] (McAnlis & amp; Haecky, Understanding compression: data compression for modern developers 2016)
https://books.google.pt/books?id=Ii6rDAAAQBAJ
[6] Salomon, D. (2007). Data Compression: The Complete Reference,
https://doi.org/10.1007/978-1-84628-603-2
[7] Python Software Foundation. Python Language Reference, version 3.8,
http://www.python.org
[8] dahuffman - Python Module for Huffman Encoding and Decoding, visto a 23/12/2020
https://github.com/soxofaan/dahuffman
[9] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 0.1038/s41586-020-2649-2. (Publisher link).
https://numpy.org
[10] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.
https://matplotlib.org
[11] Wikipédia, sbore o LZW, visitado pela ultima vez a 23/12/2020
https://pt.wikipedia.org/w/index.php?title=LZW&oldid=58054185
[12] Módulo Pickle Python,
https://docs.python.org/3/library/pickle.html
[13] Módulo LZMA do Python,
https://docs.python.org/3/library/lzma.html
[14] Módulo bz2 do Python,
https://docs.python.org/3/library/bz2.html
[15] Módulo zlib do Pythhon,
https://docs.python.org/3/library/zlib.html
[16] K. Ouaissa, M. Abdat and P. Plume, "Adaptive limitation of the dictionary size in LZW data compression," Proceedings of 1995 IEEE International Symposium on Information Theory, Whistler, BC, Canada, 1995, pp. 18-, doi: 10.1109/ISIT.1995.531120.
https://ieeexplore.ieee.org/abstract/document/531120
[17] Linguagem de programação C++,
https://docs.microsoft.com/en-us/cpp/cpp/cpp-language-reference?view=msvc-160
[18] Linguagem de Programação C,
https://docs.microsoft.com/en-us/cpp/c-language/c-language-reference?view=msvc-160