Codecs não-destrutivos (*lossless*) de imagem

Carlos Jordão
Departamento de Engenharia Informática)
Faculdade de Ciências e Tecnologias da Universidade de Coimbra
Coimbra, Portugal
cejordao@student.dei.uc.pt

Francisco Carreira
Departamento de Engenharia Informática)
Faculdade de Ciências e Tecnologias da Universidade de Coimbra
Coimbra, Portugal
mcarreira@student.dei.uc.pt

Rodrigo Machado
Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologias da Universidade de Coimbra
Coimbra, Portugal
ramachado@student.dei.uc.pt

***Abstract* — A compressão de imagens de forma não destrutiva é um tema já bastante estudado e trabalhado ao longo das últimas décadas, como resultado da popularização da internet. A necessidade de acesso rápido e de armazenamento fácil impulsionou a criação de vários algoritmos e codecs de imagem, uns melhores em comprimir e descomprimir rapidamente, outros a valorizar mais o rácio de compressão, enquanto alguns tentam encontrar um equilíbrio. Para perceber o estado atual da compressão de imagens de forma não-destrutiva, são estudadas as diferentes componentes algorítmicas e os codecs existentes para este efeito, descrevendo o método e os resultados obtidos.**

Keywords— compressão não-destrutiva, codecs de imagem, lossless, velocidade de compressão, algoritmos.

I. Introdução

No mundo atual, toda e qualquer informação, útil ou não, está prontamente disponível com a proliferação massiva de dispositivos móveis como os *smartphones*, os *laptops*, e com o desenvolvimento de redes móveis de alta velocidade – 4G ou até 5G. Toda essa informação é o resultado de quantidades massivas de dados que têm de ser armazenadas e facilmente acedidas.

Em 2020, o vírus SARS-CoV-2 obrigou as pessoas a permanecer em casa, o que levou a que muitas das atividades, como o trabalho, o ensino ou o entretenimento, passassem a ser feitas com o recurso da internet. Mesmo com o avanço tecnológico das redes, este uso massivo seria um teste exigente das redes e das soluções de armazenamento de dados caso as transmissões não sofressem nenhum tipo de compressão. Neste trabalho estudamos a compressão não-destrutiva de imagens.

Geralmente uma imagem de uso pessoal não necessita de muitos dos dados originais, pois existem codecs de compressão *lossy* que reduzem bastante o tamanho destas, reconstruindo uma imagem com fidelidade suficiente. No entanto, existem campos em que a reprodução fiel de uma imagem é necessária, como na medicina – por exemplo, um raio-x ou uma ressonância magnética, em que todo o detalhe pode ser importante. O armazenamento de imagens de alta fidelidade torna-se mais eficiente e barato quando se consegue uma compressão que não destrua os dados originais, assim como o envio destas pela internet se torna mais rápido, o que poderá ser crítico para o exercício da medicina remota.

Para estudar métodos atuais de compressão *lossless* iremos primeiramente rever algoritmos de compressão de dados; posteriormente analisaremos codecs populares, que transformações aplicam e algoritmos usam.

II. Objetivos e Plano de Trabalho

Pretendemos criar um codec de compressão não-destrutiva de imagens capaz de atingir rácios de compressão melhores do que PNG, e que ao mesmo tempo seja de baixa complexidade. Para tal, decidiu-se usar Python 3.x pela simplicidade sintáxica, e pela facilidade de importação de código através de módulos.

Partimos de um dataset de 4 imagens monocromáticas, *landscape*, *zebra*, *egg* e *pattern*, em formato bitmap, estudando a sua entropia e histogramas, e verificou-se que algumas das imagens, nomeadamente a *pattern*, apresentam uma entropia baixa e um tamanho de ficheiro elevado, pelo que existe um potencial de compressão a explorar.

Explorou-se a aplicação de diversas transformadas – sozinhas, juntas, com diferentes ordens e repetidas – com vista à redução da entropia das fontes, ou seja, ao aumento no potencial de compressão. Escolhida a(s) transformada(s) em função da entropia, foram explorados os algoritmos de compressão, principalmente BZip2 e LZMA que já estão implementados em módulos incluídos em Python.

Fig. 1 landscape

Fig. 2 egg

Fig. 3 zebra

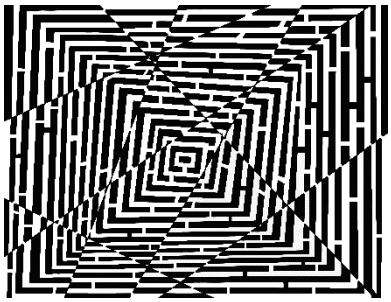

Fig. 4 pattern

III. Transformadas de Dados

As transformações de dados são úteis para agrupar símbolos semelhantes e/ou descorrelacionar dados, de forma a diminuir a entropia e a conseguir um rácio de compressão mais elevado. Para ser útil, a transformada tem de admitir uma inversa, para recuperar os dados originais.

A. *Transformada de Burrows-Wheeler (BWT)*

A transformada cria um N-1 sequências a partir de shifts cíclicos de uma sequência original de tamanho N, resultando em N-1 novas sequências. Estas e a original são então ordenadas lexicograficamente, sendo o resultado da transformada a última coluna das sequências ordenadas.

O processo inverso é um pouco mais moroso, mas concatenando o resultado e uma versão organizada lexicalmente N vezes recria a lista de shifts cíclicos criada. A sequência original pode ser então identificada usando um símbolo de escape, por exemplo um EOF no final da sequência.

Esta transformada tende a agrupar símbolos iguais, criando sequências de repetições que podem ser aproveitadas por outras transformadas ou algoritmos de compressão.

B. *Move-To-Front (MTF)*

Geralmente é precedido por uma transformada de Burrows-Wheeler e procedido por um algoritmo de compressão *Run-Length Encoding* (RLE) ou outro codificador entrópico.

Com o auxílio de um alfabeto, substitui cada símbolo pela sua posição no alfabeto um a um, mas sempre que a substituição ocorre esse símbolo é movido para o início do alfabeto.

Em sequências de dados com repetições, a transformada irá criar sequências de 0s , das quais um algoritmo como o RLE beneficia.

Sendo uma transformada de baixa complexidade computacional e rapidez de execução, o seu uso em processos de compressão é popular.

C. *Delta Encoding*

A transformada mais simples que permite descorrelacionar dados é a codificação por diferenças ou deltas. Assim, um símbolo é a diferença do símbolo original com o símbolo original anterior.

IV. Algoritmos de Compressão

A. *Run-Length Encoding (RLE)*

O RLE é um algoritmo muito simples que procura reduzir o tamanho de repetições de símbolos, criando pares <*valor*, *tamanho*> em repetições compridas, sendo *valor* o símbolo repetido e tamanho o número de ocorrências consecutivas do símbolo. Como o par tem um tamanho mínimo, nem todas as repetições são substituídas por um par.

B. *Codificação de Huffman*

Criado por David A. Huffman em 1952, é um algoritmo que permite a criação de códigos de prefixo de tamanho variável por símbolo. Aos símbolos de maior probabilidade de ocorrência faz corresponder os menores códigos, e vice-versa. Os códigos podem ser visualizados como uma árvore binária

C. *Algoritmo Lempel-Ziv* – 1977 (*LZ-77*)

Publicado em 1977 por Abraham Lempel e Jacob Ziv, este algoritmo usa uma janela deslizante (*search buffer*), de tamanho fixo. O conhecimento dos dados vistos na stream previamente é então usado para substituir padrões encontrados durante a compressão, criando o trio<offset, comprimento, símbolo>, sendo offset a distância a recuar, comprimento o número de símbolos a copiar, e símbolo o próximo símbolo a enviar.

D. *Algoritmo Lempel-Ziv* – 1978 *(LZ-78)*

Em 1978, Abraham Lempel e Jacob Ziv publicaram um novo algoritmo de compressão, este usando um dicionário construído durante a compressão. Cria pares <*índice*, *símbolo*>, sendo *índice* o índice do dicionário a enviar e *símbolo* o próximo símbolo. Procura-se no dicionário a maior entrada igual ao que se pretende codificar. Se existir, codifica-se o par e cria-se uma entrada no dicionário igual a maior encontrada concatenada com o próximo a enviar. Caso não se encontre uma entrada, envia-se um par <0, *símbolo*> e cria-se uma entrada igual ao símbolo.

E. *Algoritmo Lempel-Ziv-Welch (LZW)*

Publicado em 1984 por Terry Welch, é uma melhoria do LZ78. Em vez de iniciar a compressão com um dicionário vazio, inicializa-se o dicionário com todos os símbolos do alfabeto.

F. *Algoritmo Lempel-Ziv-Markov (LZMA)*

Entre 1996 e 1998, Igor Pavlov (criador do programa de arquivação *7-Zip*) desenvolveu um algoritmo de compressão com um dicionário de compressão parecido com o LZ77 com uma taxa de compressão elevada e uma velocidade similar aos mais utilizados algoritmos de compressão. A sequência a codificar é transformada em deltas (diferença entre um símbolo e o seu antecedente). Essa transformada é então codificada com um algoritmo de dicionário parecido ao LZ77. Finalmente, codifica-se o resultado anterior com um *range encoder* (um codificador aritmético que usa uma base de dígitos qualquer – por exemplo, um byte).

G. *Prediction by Partial Matching (PPM)*

Desenvolvido em 1984 por John Cleary e Ian Witten, o algoritmo tenta usar o contexto passado no processo de compressão para prever os símbolos seguintes da sequência a comprimir.

O maior contexto tem um tamanho predefinido. Se o símbolo a codificar não for encontrado no maior contexto, um símbolo especial é enviado para iniciar a procura num contexto menor. Caso o símbolo não tenha ocorrido na sequência, assume-se um contexto 0 em que os símbolos do alfabeto são equiprováveis. Cada contexto mantêm uma contagem das ocorrências do símbolo, e a codificação pode ser feita com codificação aritmética inteira, por exemplo.

Este algoritmo permite rácios de compressão elevados, mas é mais lento do que outros algoritmos como os LZ. Variantes mais eficientes têm vindo a ser desenvolvidos, como o PPMd.

H. *Deflate*

O algoritmo Deflate, desenvolvido por Philip Katz, é uma evolução do LZ77, juntando-lhe codificação de Huffman.

Os dados são repartidos em blocos, os quais, após um codificador semelhante ao LZ77, podem ser alvo de compressão com Huffman fixo ou adaptativo, ou até não ser comprimido.

I. *zlib*

Lançado ao público em 1995, a biblioteca *zlib* tinha sido criada com a intenção de uso na biblioteca *libpng*, uma biblioteca multi-plataforma escrita em C para uso de imagens PNG. Atualmente, *zlib* é sinónimo de *Deflate*.

J. *BZip2*

Criado por Julian Seward em 1996, o Bzip2 é um algoritmo de compressão *open-source* de várias camadas, que oferece rácios de compressão mais elevados que os algoritmos LZW e Deflate, no entanto é significativamente mais lento.

O algoritmo divide os dados em blocos de tamanho entre 100KB e 900KB, e aplica-lhes as seguintes operações: i) RLE, ii) BWT, iii) MTF, iv) RLE, e v) Huffman.

K. *CALIC (Context Adaptive Lossless Image Coding)*

O CALIC, criado por Xiaolin Wu e Nasir Memon, tem como objetivo principal a obtenção de rácios de compressão extremamente elevados, mesmo que seja com o uso de técnicas complexas e/ou lentas para sua implementação.

O algoritmo opera em dois modos: i) binário e ii) continuous-tone, e suporta a mudança on-the-fly entre estes, dependendo do contexto do pixel a ser codificado no momento.

Caso a região do pixel conste de apenas dois valores de intensidade distintos, o codificador usa o modo binário. Neste modo, um codificador aritmético adaptativo ao contexto é usado para codificar três símbolos, incluído um símbolo de escape para sair do modo binário.

No segundo modo, o processo de codificação usa uma vizinhança de pixéis das duas linhas anteriores para fazer uma previsão não linear do pixel a codificar. Posteriormente, os erros de previsão são modelados para redução de redundância e codificados por um qualquer algoritmo de codificação entrópica.

L. *FELICS (Fast, Efficient & Lossless Image Compression System)*

O FELICS utiliza os dois pixels vizinhos mais próximos de um pixel para o prever. O algoritmo é cerca de 5x mais rápido que o JPEG Lossless ('93), mas com uma compressão equiparável.

É feita uma aproximação da distribuição de probabilidade de cada pixel usando os seu vizinho diretos -superior e esquerdo. Se o pixel estiver dentro do intervalo de intensidades dos seus vizinhos, é codificado com um código binário ajustado simples, caso contrário recorre-se aos códigos de Golomb Rice devido à natureza exponencial da distribuição.

M. *LOCO-I (Low Complexity Image Compression)*

O algoritmo LOCO-I foi criado por Marcelo Weinberger, Gadiel Seroussi e Guillermo Sapiro com o objetivo de atingir rácios de compressão semelhantes a algoritmos mais complexos baseados em códigos aritméticos, mantendo a complexidade algorítmica do *FELICS*.

O sistema preditivo tem 3 preditores que fazem uso de 3 pixéis vizinhos – superior, esquerdo, e superior esquerdo. Os erros resultantes são depois codificados com códigos de Golomb-Rice.

V. CODECS EXISTENTES

A. *Lossless JPEG (1993)*

O antigo modo lossless do JPEG é uma adição ao *standard* JPEG que permite compressão não-destrutiva de imagens.

Este codec usa um esquema de previsão baseado nos três pixéis mais próximos (superior, esquerdo, e superior esquerdo). Os erros da previsão são depois comprimidos usando um codificador entrópico como Huffman ou códigos aritméticos. Dispõe de 7 esquemas preditivos, e de um modo sem esquema preditivo. O esquema utilizado é decidido pelo utilizador e serve de modelo para a imagem inteira.

É mais ou menos obsoleto, tendo sido substituído pelo JPEG-LS, que oferece uma maior eficiência na compressão que o seu antecessor.

B. *JPEG-LS*

Introduzido em 1994, o novo standard lossless consegue uma compressão mais rápida e obtém rácios de compressão maiores do que o antigo modo lossless JPEG.

Ainda em 2003 foram introduzidas extensões ao algoritmo, como por exemplo codificação aritmética.

O codec usa o algoritmo LOCO-I em dois modos: i) normal, e ii) *run mode*. O *run mode* permite o uso de símbolos compostos, útil para a codificação de regiões na imagem sem variação, dado que estas podem ser codificadas com menos de 1 bit por símbolo.

C. *GIF (Graphics Interchange Format)*

O formato PNG, criado em 1996 por uma colaboração num fórum online entre vários entusiastas e experts, como forma de substituir o GIF pois este i) usava um algoritmo de compressão patenteado e ii) dada a popularização de monitores mais avançados, o limite a 256 cores do formato tornava-se obsoleto.

Pré-compressão, o codec usa um sistema preditor com 5 métodos distintos (sendo um deles sem previsão) que permite o uso de um método de previsão diferente em cada linha da imagem. A diferença entre a previsão e o valor real em módulo de 256 é então codificada com o algoritmo Deflate.

D. *TIFF (Tag Image File Format)*

O formato TIFF, criado pela Aldus Corporation (agora Adobe Systems), foi criado como uma tentativa para os fabricantes de scanners usarem um formato standard de ficheiro para as digitalizações. No início, era apenas um formato para imagens binárias, visto que era o que os scanners eram capazes de capturar. À medida que as máquinas foram evoluindo, o formato também evoluiu e agora, juntamente com JPEG e o PNG, é considerado um formato popular para imagens de cor.

A implementação baseline de TIFF dispõe 3 modos: i) sem compressão, ii) uma versão modificada de Huffman, ou iii) PackBits – uma modificação simples de RLE. Existem ainda extensões que permitem a compressão com LZW.

E. PNG (Portable Network Graphics)

Criado pela CompuServe, uma equipa americana liderada pelo cientista Steve Wilhite em 1987, pretendia ser um método de compressão/descompressão mais rápido do que os utilizados na altura, para que imagens com tamanhos maiores pudessem ser descarregados mais rapidamente, mesmo com modems mais lentos.

Cada imagem dispõe de uma palete de 256 cores, sendo que cada pixel é representado como um índice dessa mesma palete. A imagem era depois comprimida com recurso a LZW.

A Unisys patenteou o algoritmo LZW em 1985 e a controvérsia resultante do acordo entre a CompuServe e a Unisys, que obrigava ao pagamento de licenças para uso comercial, culminou na criação do standard PNG.

VI. Transformadas e Algoritmos de Compressão Testados

Não querendo criar um script muito complexo, procuramos transformadas e algoritmos de compressão já existentes. Foram então alvo de teste os seguintes algoritmos:

- Move-to-Front (implementada pelo grupo)
- Delta Encoding (implementada pelo grupo)
- RLE (módulo de Python – função de encoding já existente; a função de decoding foi implementada pelo grupo pois a existente não funciona)
- zlib (módulo de Python)
- BZip2 (módulo de Python)
- LZMA (módulo de Python)

As implementações da transformada de Burrows-Wheeler apenas usam strings, e a sua modificação para uso de *bytes* não era simples. Além disso, o algoritmo *BZip2* implementa esta e outras transformadas na compressão.

VII. Testes preliminares

Antes de decidir que transformadas e algoritmos de compressão seriam as mais interessantes de usar, testamos a influência de cada transformada no dataset fornecido, nomeadamente a alteração da entropia destes. É de esperar que as transformadas aumentem a redundância estatística dos dados, ou seja uma produzam uma redução da entropia e , por sua vez, um aumento no potencial de compressão da fonte.

Todos os testes apresentados neste relatório foram executados num sistema com um processador Intel© Core i7-7500U, 8GB de memória e com o sistema operativo Windows 10 (versão 2004).

TABELA I. Ficheiros Originais

Ficheiro	Entropia
landspace.bmp	7.420784
zebra.bmp	5.831376
egg.bmp	5.724301
pattern.bmp	1.829457

TABELA II. Delta Encoding 1x

Ficheiro	Entropia
landspace.bmp	2.824984
zebra.bmp	3.222192
egg.bmp	2.700172
pattern.bmp	0.595832

TABELA III. Delta Encoding 2x

Ficheiro	Entropia
landspace.bmp	3.090073
zebra.bmp	3.272709
egg.bmp	2.893933
pattern.bmp	0.714130

TABELA IV. Move-to-Front

Ficheiro	Entropia
landspace.bmp	3.465445
zebra.bmp	4.330849
egg.bmp	3.417975
pattern.bmp	0.617801

Após testar as transformadas, optamos por usar Delta Encoding, pois aumenta mais a redundância dos dados. Esta transformada e a sua inversa são facilmente implementadas com o módulo *NumPy* – já usado no decorrer da cadeira. Todos os algoritmos de compressão foram aplicados após esta transformada.

Os algoritmos de compressão têm um argumento – nível de compressão. Para todos eles, os valores máximo e mínimo são 9 e 0, respetivamente. Por *default* o nível de compressão para LZMA e zlib é 6, enquanto que para o BZip2 é 9, o que coincide com o valor máximo.

TABELA V. Run Length Encoding

Ficheiro	Rácio de Compressão
landspace.bmp	0.38
zebra.bmp	0.43
egg.bmp	0.47
pattern.bmp	3.90

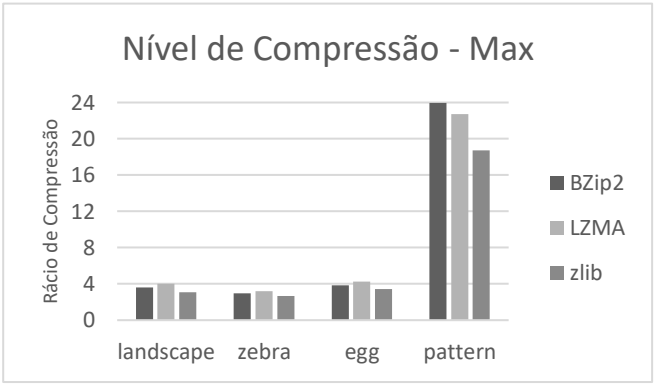

Fig. 5 Comparação dos rácios de compressãoentre BZip2, LZMA e zlib, com nível de compressão máximo.

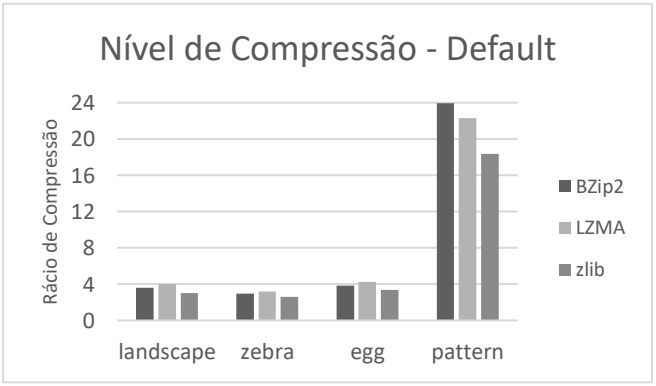

Fig. 6 Comparação dos rácios de compressãoentre BZip2, LZMA e zlib, com nível de compressão default.

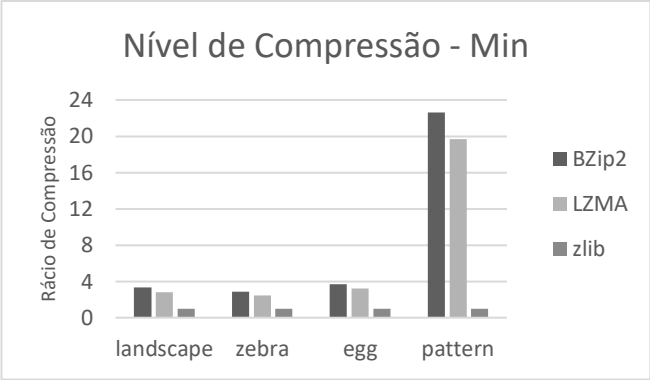

Fig. 7 Comparação dos rácios de compressãoentre BZip2, LZMA e zlib, com nível de compressão minímo.

O algoritmo *RLE* apenas comprimiu a imagem *pattern*, criando ficheiros maiores para os restantes, pelo que não será considerado para este trabalho. Dos restantes, o *zlib* é o que retorna os menores rácios, com o *LZMA* a retornar os maiores rácios no geral, sendo também o mais lento.

De notar que embora o *zlib* seja uma implementação do *Deflate*, o algoritmo principal do *PNG*, os rácios de compressão máximos obtidos com este algoritmo são inferiores aos obtidos com o *PNG*. Isto deve-se que este algoritmo utiliza um sistema de previsão mais complexo do que a transformada *Delta Encodin*g usada para a compressão.

Se o único interesse for o rácio de compressão obtido, o algoritmo *LZMA* é a melhor escolha, com o *BZip2* a ser o mais balanceado.

VIII. Descrição do Código Proposto

O codec criado utiliza apenas a transformada de *Delta Encoding* e o algoritmo de compressão *LZMA*, sendo que o script resultante é curto, de fácil compreensão e reversível.

O ficheiro a comprimir é aberto em modo binário e o seu conteúdo é lido para um *bytearray*. Este é então passado pela função *ediff1d* do módulo *NumPy*, com o argumento *to_begin*, retornando um array das diferenças entre bytes consecutivos, prefixado com o primeiro byte. Estas diferenças são então escritas para um objeto *bytes*, com recurso a um *bytearray*, e comprimidas com a função *compress* do módulo *lzma* de Python. Finalmente, o script cria um ficheiro binário e guarda o resultado.

Para reverter o processo, o ficheiro resultante da compressão é aberto em modo binário, e o seu conteúdo é passado pela função *decompress* de *lzma*. Os bytes resultantes são convertidos num *bytearray*, que é passado como argumento à função *cumsum* de *NumPy*, que calcula a soma cumulativa de todos os *bytes*. Para restaurar os *bytes* originais, basta calcular o resto da divisão inteira de cada soma por 256 com a função *mod*, também esta do módulo *NumPy*. O array resultante é escrito para bytes e guardado num ficheiro *bmp*.

É importante realçar que a passagem de um array de *NumPy* a bytes não deve ser feita com a função *to_bytes* deste módulo, pois escreve bytes vazios que não fazem sentido. Para realizar esta operação, cria-se um *bytearray* vazio e adiciona-se lhe os bytes iterando pelo array com recurso a ***append***. Para obter *bytes* a partir de um *bytearray* basta fazer cast do objeto.

IX. Análise dos Resultados

TABELA VI. Resultados do Codec PNG

Ficheiro	Tamanho original (MB)	Rácio de Compressão
landscape	10.4	3.30
zebra	15.9	3.05
egg	16.9	3.83
pattern	45.7	21.02

TABELA VII. Resultados do Codec Proposto

Ficheiro	Tamanho original (MB)	Tamanho Codec (MB)	Rácio de Compressão
landscape	10.4	2.59	4.04
zebra	15.9	4.95	3.22
egg	16.9	3.98	4.25
pattern	45.7	2.01	22.71

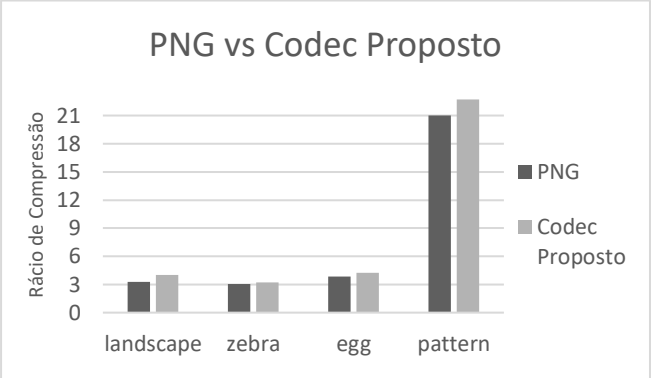

Fig. 8 Comparação dos rácios de compressão entre PNG e o codec proposto.

O codec proposto conseguiu atingir rácios de compressão ligeiramente superiores aos obtidos com o *PNG* no dataset fornecido para este trabalho, ainda que este seja mais complexo do que o desenvolvido.

De notar que o dataset usado é diminuto, e composto por imagens em tons de cinza apenas, pelo que seria erróneo generalizar estas conclusões.

X. Conclusões

XI. Bibliografia

[1] 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.

[2] S. Ijmulwar, D. Kapgate (2014): "A Review on – Lossless Image Compression Techniques and Algorithms", *International Journal of Computing and Technology*, vol. 1 issue 9, pp. 1-4.

[3] "Transformada de Burrows-Wheeler" May 23, 2016. [Online]. Disponível em: http://multimedia.ufp.pt/codecs/compressao-sem-perdas/supressao-de-sequencias-repetitivas/metodo-de-burrows-wheeler/. [Lido: Nov. 23, 2020].

[4] Sayood, Khalid (2017): "The Burrows-Wheeler Transform", in *Introduction to Data Compression*. Cambridge: Morgan Kauffman, 175-178

[5] Sayood, Khalid (2017): "Move-to-Front Coding", in *Introduction to Data Compression*. Cambridge: Morgan Kauffman, 178-179

[6] "Algoritmo de Huffman" Oct. 13, 2020. [Online]. Disponível em: http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/algoritmo-de-huffman/ [Lido: Nov. 23, 2020]

[7] "Codificação de Huffman" Apr. 02, 2016. [Online]. Disponível em: https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman [Lido: Nov. 23, 2020]

[8] "Huffman Coding" Apr. 10, 2001. [Online]. Disponível em: https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node210.html/ [Lido: Nov. 23, 2020].

[9] Sayood, Khalid (2017): "The LZ77 approach", in *Introduction to Data Compression*. Cambridge: Morgan Kauffman, 135-139

[10] Sayood, Khalid (2017): "The LZ78 Approach", in *Introduction to Data Compression*. Cambridge: Morgan Kauffman, 139-140

[11] Sayood, Khalid (2017): "Variations on the LZ78 Theme—The LZW Algorithm", in *Introduction to Data Compression*. Cambridge: Morgan Kauffman, 140-146

[12] "How LZMA decompression algorithm works?" Oct. 12, 2020. [Online]. Disponível em: https://stackoverflow.com/questions/64314593/how-lzma-decompression-algorithm-works [Lido: Nov. 23, 2020]

[13] "Lempel–Ziv–Markov chain algorithm" Oct. 13, 2020. [Online]. Disponível em: https://en.wikipedia.org/wiki/Lempel-Ziv-Markov_chain_algorithm [Lido: Nov. 23, 2020]

[14] Sayood, Khalid (2017): "Prediction with Partial Match (ppm)", in *Introduction to Data Compression*. Cambridge: Morgan Kauffman, 167-175

[15] "Prediction by partial matching" Sep. 12, 2020. [Online]. Disponível em: https://en.wikipedia.org/wiki/Prediction_by_partial_matching/ [Lido: Nov. 23, 2020]

[16] "Deflate" Oct. 13, 2020. [Online]. Disponível em: http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/codificador-qm/deflate/ [Lido: Nov. 23, 2020]

[17] Tsai, Joe: "BZip2: Format Specification" March 17, 2016 [Online]. Disponível em https://github.com/dsnet/compress/blob/master/doc/bzip2-format.pdf [Lido: Nov 23, 2020]

[18] "Bzip2" Jul. 31, 2002. [Online]. Disponível em: https://en.wikipedia.org/wiki/Bzip2/ [Lido: Nov. 23, 2020]

[19] Sayood, Khalid (2017): "CALIC", in *Introduction to Data Compression*. Cambridge: Morgan Kauffman, 190-194

[20] X. Wu and N. Memon, "Context-based, adaptive, lossless image coding," in IEEE Transactions on Communications, vol. 45, no. 4, pp. 437-444, April 1997, doi: 10.1109/26.585919.

[21] P. Howard and J. Vitter, "Fast and efficient lossless image compression," in 1993 Data Compression Conference, Snowbird, UT, USA, 1993 pp. 351,352,353,354,355,356,357,358,359,360. url: https://doi.ieeecomputersociety.org/10.1109/DCC.1993.253114

[22] "FELICS" Nov. 14, 2020. [Online]. Disponível em: https://pt.qaz.wiki/wiki/FELICS [Lido: Nov. 23, 2020]

[23] M. J. Weinberger, G. Seroussi and G. Sapiro, "The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS," in IEEE Transactions on Image Processing, vol. 9, no. 8, pp. 1309-1324, Aug. 2000, doi: 10.1109/83.855427.

[24] "LOCO-I algorithm" Jan. 17, 2006. [Online]. Disponível em: https://en.wikipedia.org/wiki/Lossless_JPEG#LOCO-I_algorithm/ [Lido: Nov. 23, 2020]

[25] "Lossless JPEG" Jan. 17, 2006. [Online]. Disponível em: https://en.wikipedia.org/wiki/Lossless_JPEG/ [Lido: Nov. 23, 2020]

[26] Sayood, Khalid (2017): "JPEG-LS", in *Introduction to Data Compression*. Cambridge: Morgan Kauffman, 194-196

[27] Sayood, Khalid (2017): "Lossless Image Compression Formats", in *Introduction to Data Compression*. Cambridge: Morgan Kauffman, 202-204

[28] W3C®, "Portable Network Graphics (PNG) Specification (Second Edition)" Nov. 10, 2003. [Online]. Disponível em: https://www.w3.org/TR/PNG/#9Filters/ [Lido: Nov. 25, 2020]

[29] Colt McAnlis, "How PNG Works" Apr. 6, 2006. [Online]. Disponível em: https://medium.com/@duhroach/how-png-works-f1174e3cc7b7/ [Lido: Nov. 25, 2020]

[30] Adobe, "TIFF Revision 6.0" June 03, 1992 [Online]. Disponível em: https://www.adobe.io/content/dam/udp/en/open/standards/tiff/TIFF6.pdf [Lido: Nov 25, 2020]

[31] Python Documentation Team, "lzma — Compression using the LZMA algorithm" Sept. 29, 2012 [Online]. Disponível em: https://docs.python.org/3/library/lzma.html [Lido: Dec 22, 2020]

[32] Python Documentation Team, "bz2 — Support for bzip2 compression" Sept. 29, 2012 [Online]. Disponível em: https://docs.python.org/3/library/bzip2.html [Lido: Dec 22, 2020]

[33] Python Documentation Team, "zlib — Compression compatible with gzip" Sept. 29, 2012 [Online]. Disponível em: https://docs.python.org/3/library/zlib.html [Lido: Dec 22, 2020]

[34] Jean-loup Gailly, Mark Adler, "zlib general purpose compression library" Jan, 15, 2017 [Online]. Disponível em: http://www.zlib.net/manual.html [Lido: Dec 22, 2020]