Uma análise sobre – Compressão de Imagem sem Perda de Informação

Edgar Duarte
Dept. de Engenharia Informática
Universidade de Coimbra
Miranda do Corvo, Portugal
edgarduarte@student.dei.uc.pt

Miguel Dinis
Dept. de Engenharia Informática
Universidade de Coimbra
Cantanhede, Portugal
miguelbarroso@student.dei.uc.pt

Rodrigo Ferreira
Dept. de Engenharia Informática
Universidade de Coimbra
Leiria, Portugal
rferreira@student.dei.uc.pt

***Abstract*— Este artigo apresenta um estudo simples acerca de compressão de informação – em específico, imagem – sem qualquer perda de informação. Existem diversas situações na computação em que há uma extrema necessidade de comprimir dados – de qualquer natureza e tipo – para os poder armazenar ou enviar através de uma rede. Pretende-se, matemática e estatisticamente, criar uma transformação reversível, de forma a evitar a perda de informação, em que se obtenha uma representação da informação que ocupe menos bits e que remova redundâncias. É feita uma análise aos métodos, algoritmos e *codecs* existentes versando diversos parâmetros: rácio de compressão, eficiência e velocidade de compressão/descompressão. As imagens BMP (*bitmap format*), que são usadas como base nesta investigação, são monocromáticas – apresentando, na sua maioria, cores binárias (preto e branco) e alguns tons de cinza. Como tal, os *codecs* que foram trabalhados são os mais adequados ao tipo de informação que se pretende comprimir.**

Keywords*— *compressão, sem perdas, redundância, imagens monocromáticas, codificação, entropia

I. Introdução

Desde muito cedo na computação se sentiu a necessidade de comprimir informação. Um vídeo de 5 minutos, Full HD (1920x1080), a 30fps (*frames per second*) e com esquema de cores RGB (Red-Green-Blue) de 24bits, sem qualquer compressão e com todos os seus píxeis representados, ocuparia 52.14 GB (*GigaBytes*). Percebe-se facilmente que é impossível armazenar ou transmitir qualquer conteúdo multimédia de forma útil sem recorrer a um método de compressão. Deste modo, utilizaram-se quatro imagens monocromáticas – na sua maioria a preto e branco, mas também com escala de cinza – como *dataset* de base de estudo que estão representadas na Tabela 1.

Existem dois tipos de compressão: com perda (*lossy*) e sem perda (*lossless*). Uma compressão sem perda é dada por uma função do tipo:

$$T(i) = c$$

$$T^{-1}(c) = i$$

Seja i a informação a comprimir, c a informação comprimida e seja a função T bijetiva e invertível. Como é invertível, é possível fazer o processo inverso da codificação e reaver toda a informação codificada para o estado original. Este tipo de compressão é muito usada em informações cuja integridade é crucial – não se pode perder um único bit – tais como texto, alguns pacotes TCP (protocolo de transmissão de informação pela internet), entre outros.

A compressão com perda, ao contrário do exemplo anterior, não é invertível. Usa, de forma essencial, os mesmos mecanismos que uma compressão sem perdas, mas também remove alguma informação inútil que não seria detetada (ou detetada muito dificilmente) pelo olho humano. A compressão com perdas é muito usada em imagens, vídeos e áudio (multimédia em geral), visto ser mais eficiente que uma compressão sem perdas, apresentando melhores resultados, e visto não ser muito relevante a perda de algumas informações que não seriam detetadas pelos sentidos humanos.

A compressão estudada neste artigo, bem como os seus principais algoritmos e *codecs*, é a compressão sem perdas. Na Tabela 1 estão descritas as quatro imagens que foram usadas na elaboração do mesmo: o (1) *egg* apresenta-se como uma imagem monocromática numa escala de cinzentos embora apresente algum detalhe; o (2) *landscape* apresenta cerca de 75% da sua dimensão a cinzento, o que poderia indicar uma maior possível compressão, porém, após uma análise detalhada, observa-se que existem vários tons de cinzento presentes, o que aumenta bastante a entropia; o (3) *pattern* é uma imagem totalmente binária (a preto e branco) o que potencia uma maior compressão; e a (4) *zebra* é uma imagem monocromática mas apresenta algum detalhe no relevo, o que pode ter implicações adicionais.

Também pela Tabela 1, é observável a entropia de cada imagem e a sua percentagem de compressão entrópica (tendo por base uma entropia simples). Não são considerados pares de símbolos nem agrupamentos de ordem superior. De uma forma geral observa-se o seguinte: em (1) a percentagem considera-se adequada visto que a imagem possui algum relevo e detalhe mesmo nos tons de preto e branco, o que só poderia ser compensado numa compressão com perdas; em (2) percebe-se que mesmo no cinzento existem vários tons – o que se reflete entropicamente – e na base da imagem existe muito detalhe nas árvores, logo, atinge-se uma percentagem de compressão baixa; em (3) a imagem é praticamente binária, o que facilita a compressão por parte de alguns algoritmos; e em (4) observa-se uma imagem com o mesmo nível de detalhe que em (1), logo, obtendo-se uma entropia e compressão possível semelhante.

Ficheiro	**Tamanho (Bytes)**	**Entropia**	**Percentagem de Compressão**
egg.bmp	17.744.598	5.724	28.47%
landscape.bmp	11.006.422	7.420	07.25%
pattern.bmp	48.005.942	1.829	77.14%
zebra.bmp	16.738.210	5.831	27.11%

Tabela 1 – apresentação do dataset e compressão entrópica

II. Estado da Arte - Alguns Algoritmos Existentes

A. *RLE - Run Length Encoding*

O RLE é dos métodos de compressão mais simples. Tem como objetivo principal representar sequências de valores iguais de forma eficiente. O algoritmo junta valores contíguos iguais, agrupando-os na forma (r, v), onde v representa um valor e r um inteiro que representa a quantidade de símbolos seguidos com o valor v. Exemplificando:

255 255 255 255 0 0 0 0 0 12 23 23 255 255 255 255 255

(4,255) (5, 0) (1,12) (2,2) (5,255)

Como o *dataset* é composto por imagens monocromáticas e, na sua maioria, têm valores muito próximos uns dos outros, será expectável que este método de compressão tenha bons resultados. No caso do "*pattern*" existem linhas com a mesma cor em vários píxeis sucessivos, o que, por consequência, se traduz numa grande eficiência de compressão por RLE.

B. *Huffman Codes*

A codificação de *Huffman* é um método de compressão cuja função é remover a redundância da fonte de informação. O método cria uma árvore baseada na probabilidade de ocorrência de cada símbolo. O algoritmo para criar uma árvore de *Huffman*, por passos, é o seguinte:

1. Ordena-se os símbolos por ordem crescente;
2. Constrói-se uma árvore binária em que se combina, ciclicamente, os dois símbolos menos frequentes num só símbolo (os símbolos são folhas). Esse novo símbolo é adicionado à lista de valores e fica com o número de ocorrências associado igual à soma das ocorrências dos símbolos que o compõem;
3. Repete-se o passo anterior até sobrar 1 elemento na lista.

Após criada a árvore, sabe-se que os símbolos com maior probabilidade (têm mais ocorrências) ficam mais perto do topo da árvore, enquanto que os símbolos com menor probabilidade ficam no fundo da árvore. Isto é beneficial, pois os símbolos que terão de ser mais codificados, são os que têm um comprimento menor enquanto que os símbolos que terão de ser menos codificados, podem até ocupar mais bits do que ocupavam vindos da mensagem fonte, mas, como são mais raros, não gastam tanta memória como a que foi ganha com os símbolos de maior frequência.

Assim, é presumível que em imagens em que o alfabeto é muito extenso, mas que exista picos grandes de frequência em alguns símbolos, este método de compressão seja muito eficiente – que é o caso da maior parte do *dataset*.

C. *LZ-77*

O LZ77 é um método desenvolvido por Abraham Lempel e Jacob Ziv em 1977.

O método tem dois buffers:

- *Search buffer* – consiste num número N de símbolos que o método pode pesquisar por sequências repetidas de símbolos;
- *Look-ahead buffer* – *buffer* com os símbolos que irão ser enviados;

O método procura no search buffer pelo maior padrão que ocorre no look-ahead buffer começando pelo primeiro símbolo a enviar. De seguida, codifica-a da seguinte forma:

{*Offset*, *Length*, Código do próximo símbolo}

O *offset* é o número de bits a recuar e o *length* é o número de bits a adicionar à informação. Após isso, envia o código do símbolo seguinte. Caso não se encontre o símbolo a enviar no *search buffer*, será enviado:

{0,0, código do símbolo}

Este processo repete-se até ser enviada toda a informação.

O número de bits usando LZ77 pode ser calculado pela seguinte fórmula:

$$\lceil \log_2 N_S \rceil + \lceil \log_2 N_S + N_L \rceil + \lceil \log_2 A \rceil$$

Este método terá maior eficiência para padrões de informação repetidos e menores que o *search buffer*.

D. *LZ-78*

O LZ78 é um método desenvolvido por Abraham Lempel e Jacob Ziv em 1978. É composto por um dicionário de tamanho máximo de bits definido anteriormente.

Os símbolos são codificados no seguinte formato:

<índice do dicionário, código próximo símbolo>

O método procura pelo maior padrão de informação disponível no dicionário e envia o duplo constituído pelo índice correspondente juntamente com o código do próximo símbolo. Caso não exista referência no dicionário, o índice enviado será 0 juntamente com o código.

O duplo formado será uma nova entrada no dicionário (caso exista espaço disponível para novas combinações), sendo que os índices do dicionário começam em 1 pois 0 está reservado para novos símbolos.

Este processo será repetido até ser enviada toda a informação.

Este método, em teoria, tem maior eficiência para padrões de informação repetidos que estejam no dicionário.

E. *LZW*

O algoritmo LZW é uma variante do LZ78 em que se evita o envio de um caractere, apenas são enviados índices do dicionário.

O dicionário inicia-se com todos os símbolos do alfabeto.

O método corre a informação e procura padrões na informação.

Quando ocorre um novo padrão K = P|A, sendo que P é o símbolo ou conjunto de símbolos que já existe no dicionário e A o símbolo ainda sem a sequência anterior P no dicionário. O método envia o símbolo P e de seguida cria uma nova entrada PA no dicionário.

Continua a codificação no símbolo A e repete o processo anterior para o novo símbolo e assim sucessivamente até ser enviada toda a informação.

Este método, bem como o LZ77 e 78, são mais eficientes em padrões de informação que se repitam periodicamente.

F. DEFLATE

O algoritmo DEFLATE é uma combinação do algoritmo LZSS (derivação do LZ77) com códigos de Huffman. Sendo, portanto, uma combinação, consegue-se o melhor de vários mundos, maximizando a sua eficiência.

Este algoritmo é muito usado em compressões ZIP e PNG.

G. Predictive Transformer

Um transformador preditivo, como qualquer transformador, tem como principal função reduzir a entropia de uma determinada fonte.

Existem vários tipos de transformadores preditivos, mas todos se baseiam na premissa de que o valor de um certo píxel pode ser deduzido de forma derivada dos píxeis à sua volta. Exemplificando: existem os (1) transfomadores preditivos horizontais, que fazem a sua predição a partir da subtração do píxel atual com o píxel anterior e o (2) transformador preditivo vertical que faz a sua predição a partir da subtração do píxel atual com o píxel acima. O modo vertical ou horizontal deverá ser escolhido tendo em conta a fonte, de forma a obter um melhor resultado entrópico.

Por exemplo, nas imagens do dataset existem tanto imagens binárias como gradientes. Nas imagens binárias, se for aplicado, por exemplo, um transformador preditivo horizontal, espera-se ficar com uma entropia muito menor, visto que a informação obtida após a transformação conterá muitos zeros. Nos gradientes, se se usar o mesmo transformador, esperamos ter uma redução significante da entropia, visto que a nova informação ficará com valores constantes (isto se as diferenças entre valores dos píxeis forem constantes).

H. PNG

O algoritmo PNG comprime dados sem perdas. É composto por duas etapas:

1. Pré-compressão: filtro;
2. Compressão: DEFLATE.

Na etapa da pré-compressão, é aplicado um filtro (dentro de 5 tipos possíveis). Os valores são alterados consoante o anterior, sendo deixada apenas a diferença. Exemplificando:

100 102 104 106 108

100 2 2 2 2

Neste caso, cada pixel em vez de ter o seu valor descrito, tem a sua diferença perante o pixel anterior. Desta forma, é reduzido o tamanho em *bits* necessário para representar cada pedaço de informação. Para além disso, é possível construir uma função aproximada (por aproximação linear) que preveja os valores, sem a necessidade de os armazenar. Neste caso, seria algo como:

$$y = 2x + 100$$

Existem vários métodos:

1. *None* – nenhum filtro é aplicado;
2. *Sub* – cada byte é previsto consoante o da esquerda;
3. *Up* – cada byte é previsto consoante o de cima;
4. *Average* – cada byte é previsto conforme a média do bloco;
5. *Paeth* – baseado num algoritmo feito por Alan. W. Paeth.

É possível escolher o filtro que mais se adapta à imagem em questão e inclusive aplicar um filtro diferente em cada linha de píxeis.

Na imagem "*zebra*" é possível aplicar o filtro *Up* com melhor eficiência, visto que as listras pretas estão dispostas verticalmente.

Resumidamente, este algoritmo apresenta várias tecnologias. Desde códigos de Huffman, filtros inteligentes, algoritmo LZSS, entre outros. Parece ser dos melhores e mais eficientes algoritmos de compressão sem perdas. Mais à frente neste artigo, são comparados resultados usando este algoritmo e outros.

III. Algoritmos Desenvolvidos

Primeiramente foi construído um transformador preditivo que efetua as suas predições tanto na horizontal como na vertical. O processo do algoritmo já foi descrito no capítulo II – G – *Predictive Transformer*. Estes transformadores diminuem a entropia do *dataset*, potenciando uma melhor performance da compressão. Por esta razão, este algoritmo é imprescindível na compressão. Foi escolhido este transformador visto que o *dataset* possui imagens exclusivamente binárias (*pattern.bmp*) e imagens com gradientes (*landscape.bmp* e *egg.bmp*). Infelizmente a zebra.bmp, embora a olho nu possa parecer uma imagem binária, tem demasiadas imperfeições nas tonalidades das cores, o que faz com que este transformador, tanto na horizontal como na vertical, não diminua a entropia tanto como nas outras imagens do *dataset.*

Seguidamente, foram escolhidos dois codificadores para serem desenvolvidos no âmbito deste artigo. Foram programados os seguintes: (1) RLE+Huffman – que foram conjugados no mesmo algoritmo e que necessita de guardar um dicionário com informações extra e (2) LZW – que possui maior complexidade computacional devido a ter de criar/recriar um dicionário.

No algoritmo (1), primeiramente, realiza-se um RLE (descrita no capítulo II - A – *Run Length Encoding*) sobre a informação. Após essa operação, passa-se um *Huffman* por cima da nova informação com objetivo de fazer com que os grupos de elementos mais comuns custem menos bits a aceder. No fim, guarda-se a informação codificada num ficheiro binário e guarda-se a árvore de *Huffman* num ficheiro JSON (que pode causar *overhead* em fontes excessivamente pequenas).

O algoritmo (1) foi escolhido devido à sua grande eficiência, provada matematicamente por *Huffman*. É um algoritmo estatisticamente dependente do *dataset*, adaptando-se a ele. Para além disso, recorre a um RLE que permite encurtar grandes quantidades de informação seguidas que

sejam iguais. Desta forma, este algoritmo, em conjunção com o transformador preditivo, prova ser uma grande aposta.

No algoritmo (2) cria-se um dicionário que, inicialmente, tem as suas primeiras 512 entradas preenchidas de 0-512 (valores possíveis de aparecer, devido ao transformador preditivo poder gerar números negativos, logo, somando-se 255 a todos os valores garantindo-se assim que não exista nenhum número negativo). Depois, utilizando o algoritmo descrito no capítulo II - E – LZW, preenche-se o dicionário com ocorrências. Foi definido um dicionário de 16 bits e foi escolhido um tamanho de blocos para cada imagem do *dataset*, que subdivide a informação em pedaços menores e aplica o *codec* separadamente como forma de comprimir a informação em tempo útil.

Quanto ao algoritmo (2), foi escolhido devido à sua grande eficiência com imagens cujos valores se repetem ciclicamente e de forma agrupada, adaptando-se, mais uma vez, à fonte, que é o caso do *dataset* após a utilização do transformador preditivo. Por exemplo, no *pattern.bmp*, que inicialmente é uma imagem binária, após aplicado a transformação preditiva, ficará com sequências muito longas de valores 0. Assim, a cada iteração do LZW, geralmente, será adicionada no dicionário uma nova entrada com mais um 0 do que a entrada anterior, o que provoca que, após algumas iterações, uma sequência muito grande de 0s possa ser codificada a partir de apenas um índice do dicionário. Teoricamente, este comportamento também se deve verificar em gradientes, transformados com um preditor horizontal, em que as diferenças de valores sejam relativamente constantes (*pattern.bmp* e *egg.bmp*).

É de notar que também foram desenvolvidos os descodificadores de todos os algoritmos acima referidos, de forma a se poder garantir que os códigos desenvolvidos funcionam. Os descodificadores criam um ficheiro de *output* no formato BMP.

O repositório com a solução desenvolvida está disponível no GitHub, neste link:

https://github.com/MigDinny/prototype-compressor

IV. Simulações e Análise de Resultados

A. *Entropia e Pré-Simulação*

Foram feitas simulações de compressão e descompressão de cada algoritmo. Ambos os algoritmos usam o transformador na fase de pré-compressão. Posto isto, ambos os algoritmos, antes de serem analisados e postos à prova, estão nas mesmas condições de partida – com a mesma transformação, o mesmo *dataset* e a mesma máquina.

As especificações da máquina são as seguintes:

- HP Pavillon Power Laptop 15-cb003np
- Intel core i5-7300HQ @ 2.50GHz
- 8GB RAM @ 2400MHz
- Windows 10 20H2 Home 64bit
- Python 3.9.1 64bit

Primeiramente foram calculados os valores de entropia após a transformação preditiva, de forma a se conseguir perceber qual a compressão máxima atingível. Os valores obtidos foram os seguintes:

Ficheiro	Entropia pré-T / pós-T	Compressão máx. pré-T / pós-T
egg.bmp	5.724 / 2.701	28.47% / 66.24%
landscape.bmp	7.420 / 2.765	07.25% / 65.44%
pattern.bmp	1.829 / 0.611	77.14% / 92.36%
zebra.bmp	5.831 / 3.181	27.11% / 60.24%

Tabela 2 – entropias pré e pós transformação (T) entrópica

Nota: o valor escolhido para a entropia após a transformação é o valor de entropia mais baixo entre o transformador preditivo horizontal e vertical.

A partir da tabela é evidente que o transformador preditivo reduz a entropia significativamente em todas as imagens do *dataset*. É de notar que, embora a redução percentual da entropia seja maior no *pattern.bmp*, é no *landscape.bmp* onde ocorre uma maior perda pontual de entropia (4.655 bits). Assim, é nesta imagem onde o ganho de compressão é maior (por um fator de 8).

Estes valores são muito satisfatórios pois, através de um algoritmo muito simples e de alta performance de execução (pouco tempo de execução devido à utilização no numpy), foi possível reduzir significativamente a entropia do *dataset*.

B. *Compressões Obtidas na Simulação*

Os ficheiros do *dataset* foram comprimidos usando os dois algoritmos desenvolvidos e apresentam-se os resultados obtidos.

Nota: ter uma compressão de, por exemplo, 60%, significa que o ficheiro final vai ter menos 60% do tamanho inicial. Ou seja, se a fonte tiver 100MB, vai ter menos 60%, ou seja, menos 60MB, acabando por ficar com 40MB. Uma compressão de 0% seria uma compressão onde o ficheiro não é comprimido, e uma compressão de 100% seria uma utopia porque não ficaria nada guardado.

Ficheiro	Compressão máx. teórica	RLEHuff	LZW
egg.bmp	66.24%	50.72%	69.29%
landscape.bmp	65.44%	47.75%	59.23%
pattern.bmp	92.36%	92.11%	92.37%
zebra.bmp	60.24%	49.12%	49.34%

Tabela 3 – performance da compressão atingida em ambos os algoritmos desenvolvidos comparada com a compressão entrópica máxima possível (prevista)

Nota: as compressões atingidas pelo LZW poderão ser diferentes se os parâmetros (como o tamanho do bloco) forem alterados. Logo, estes valores não são únicos para este algoritmo. Neste caso, optou-se por parâmetros que levassem a uma compressão razoável e que fosse útil computacional e temporalmente – isto é, que fosse executada em cerca de 15 minutos. Ou seja, é possível definir parâmetros de compressão mais eficiente, mas o tempo de execução ascenderia exponencialmente para a várias horas. Mais à frente este fenómeno é explicado.

Quanto ao algoritmo (1), as percentagens da compressão atingida, são próximas do máximo, visto que os códigos de *Huffman* são bastante eficientes. Nada de surpreendente, portanto.

Já o algoritmo (2) apresenta compressões bastante boas. Mas, porque é que em alguns casos, a compressão é superior ao máximo atingível? Isso deve-se ao facto de que o LZW também é um algoritmo que aplica uma transformação – ao juntar cadeias de símbolos num só símbolo no dicionário.

C. *Performance do Tempo de Execução*

Embora neste artigo seja feita uma análise exclusivamente à eficiência da compressão, é sempre sensato analisar também a eficiência computacional.

O algoritmo (2), LZW, é, à partida, mais lento que o (1). Isso deve-se ao facto de possuir um dicionário a ser constantemente criado, recriado e atualizado que, a cada iteração, interage com o dicionário que, ao longo do tempo, vai aumentando de tamanho. A vantagem é que este algoritmo se adapta ao *dataset* porque deteta vários agrupamentos de informação, otimizando a compressão.

Nota: o transformador preditivo diminui o tempo de execução do algoritmo (2) pois, como ele provoca sequências grandes que serão ciclicamente repetidas, o que levará a menos entradas no dicionário, consequentemente menos comparações o que aumentará a eficiência.

O algoritmo (1) é consideravelmente mais rápido, como irá ser demonstrado mais à frente. Posto isto, na maior parte das situações, compensa, de uma forma prática, aplicar este algoritmo de compressão, mesmo que não se atinjam valores de compressão tão altos – visto que a velocidade é muito mais rápida.

Um caso isolado para exemplificar:

egg.bmp 530sec 69.29% (LZW)

66sec 50.62% (RLE+Huff)

Como se pode observar, existe uma grande redução de tempo de execução e uma redução significativa, mas aceitável, de compressão. Se o tempo de execução for crucial, a escolha mais sensata é a do RLE+HUFF. Se o importante for conseguir uma compressão muito boa, independentemente do tempo de execução, a melhor escolha é o LZW.

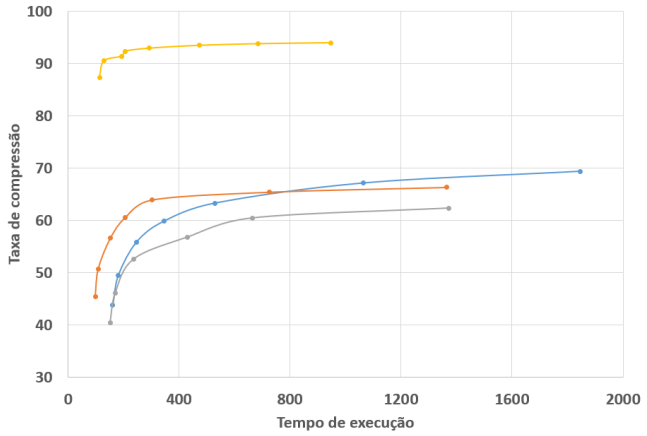

Figura 1 - Evolução da compressão à medida que se aumenta o chunksize (diretamente proporcional ao tempo de execução)

Nota: o eixo horizontal diz respeito ao tempo de execução de uma dada compressão por LZW – foram feitas várias observações empíricas com parâmetros diferentes. Observa-se que o tempo de execução cresce à medida que o parâmetro do tamanho do dicionário cresce. O eixo vertical diz respeito à compressão percentual atingida – quanto maior o número, melhor.

Amarelo – pattern.bmp

Laranja – landscape.bmp

Azul – egg.bmp

Cinzento – zebra.bmp

Observa-se uma tendência logarítmica na eficiência do LZW – que se achata à medida que se aproxima do limite entrópico. Por mais que o parâmetro dado seja maior, a entropia é impossível de vencer – sem recorrer a transformações –, que se pode verificar nesse achatamento do gráfico. Isto significa que, à medida que se aumenta o tamanho do dicionário (que potencia maiores compressões), a perda na performance computacional é exponencial, sendo, portanto, desproporcional. Desta forma, é sensato escolher um tamanho de dicionário correspondente ao "ponto de inflexão" desse mesmo gráfico – ou seja, onde o crescimento da eficiência de compressão ponderada pelo tempo de execução diminui.

Tendo isto em conta, conclui-se que o LZW poderá ter diferentes performances dependendo da disponibilidade computacional. Para efeitos práticos, o algoritmo (1) revela-se mais útil, mas para efeitos em que o que realmente importa é a compressão atingida, o (2) revela ser melhor. Cada caso deve ser analisado de forma a ser feita uma escolha inteligente.

Como forma de perceber o quão bons são os nossos *codecs* em relação ao PNG, apresentam-se as seguintes percentagens de compressão para comparação:

Ficheiro	RLEHuff	LZW	PNG
egg.bmp	50.72%	69.29%	72.26%
landscape.bmp	47.75%	59.23%	69.51%
pattern.bmp	92.11%	92.37%	95.25%
zebra.bmp	49.12%	49.34%	67.23%

Tabela 4 – comparação das compressões atingidas pelos algoritmos desenvolvidos e o algoritmo standard PNG

É evidente que os *codecs* desenvolvidos ficam muito aquém do PNG. Além dos tempos de execução serem muito mais elevados, a compressão é significativamente menor. É de notar que quando os *codecs* desenvolvidos têm um pior desempenho, curiosamente, também PNG tem um pior desempenho. Isto pode significar que o PNG usa transformadores e algoritmos de compressão idênticos ou parecidos aos desenvolvidos.

Nota: com o LZW obtem-se uma compressão muito próxima da do PNG, quando é parametrizada uma *chunksize* muito alta (na ordem dos milhões) mas, em contrapartida, o tempo de execução fica extremamente elevado (cerca de várias horas).

V. Conclusão e Referências Bibliográficas

A. Conclusão

Concluindo este artigo, o ponto principal a reter é que, empiricamente, comprova-se que o algoritmo (1) é melhor num contexto prático enquanto que o algoritmo (2) é melhor se se desejar uma melhor compressão.

Também se sabe que é possível maximizar a eficiência computacional dos algoritmos se eles forem programados para usar vários núcleos do processador – tecnologia *multi-threading* – e se forem otimizados, o que não acontece.

Existem algoritmos muito melhores que estes que foram desenvolvidos visto que possuem técnicas mais aprofundadas, estudadas e melhoradas para compressão. Um deles é o PNG, que é rápido e eficiente – não é por mero acaso que é um algoritmo *standard*.

B. Trabalho Futuro

Futuramente, é possível continuar esta investigação de forma a obter melhores resultados. Apresentam-se algumas ideias.

O transformador preditivo possui dois modos, horizontal e vertical. Isto pode ser expandido para outros modos, como vizinhanças ou predição baseada numa função linear. Para além disso, o próprio *software* de compressão pode selecionar o transformador mais indicado (aquele que reduz mais a entropia) de forma inteligente.

O tamanho dos blocos (*chunksize*) também poderá ser escolhido de forma inteligente de forma a que o LZW tenha uma melhor performance.

Por último, estes algoritmos poderão ser aperfeiçoados e otimizados através de técnicas de *multi-threading* e através da introdução da linguagem C e Assembly para melhores resultados – e não exclusivamente Python.

C. Referências Bibliográficas

[1] Aditya Gupta, "LZW-Compressor-in-Python", 20 Dezembro 2020. Obtido de https://github.com/adityagupta3006/LZW-Compressor-in-Python

[2] Stefaan Lippens, "dahuffman 0.4.1", 20 Dezembro 2020, https://pypi.org/project/dahuffman/ , https://github.com/soxofaan/dahuffman/blob/master/setup.py

[3] Wikipédia, "Portable Network Graphics: Revision history", 20 Dezembro 2020. https://en.wikipedia.org/wiki/Portable_Network_Graphics

[4] Wikipédia, "Run-length encoding", 20 Dezembro 2020. https://en.wikipedia.org/wiki/Run-length_encoding

[5] Wikipédia, "LZW",20 Dezembro 2020. https://pt.wikipedia.org/wiki/LZW

[6] Apoorv Gupta, Aman Bansal, Vidhi Khanduja, "Modern Lossless Compression Techniques: Review, Comparison and Analysis" (2017), pp. 8.

[7] Ms. Shilpa Ijmulwar, Deepak Kapgate, "A Review on - Lossless Image Compression Techniques and Algorithms", October 2014. International Journal of Computing and Technology, Volume 1, Issue 9. ISSN: 2348 – 6090

[8] Lina J. Karam , "Lossless ImageCompression", Arizona State University, chapter 16

[9] PowerPoints disponiblizados em aula