Codecs lossless de imagem

1st Duarte Meneses
Dept. Engenharia Informática
(Faculdade de Ciências e Tecnologias)
Universidade de Coimbra
Coimbra, Portugal
duartemeneses@student.dei.uc.pt

2nd Inês Marçal
Dept. Engenharia Informática
(Faculdade de Ciências e Tecnologias)
Universidade de Coimbra
Sertã, Portugal
inesmarcal@student.dei.uc.pt

3rd Patrícia Costa
Dept. Engenharia Informática
(Faculdade de Ciências e Tecnologias)
Universidade de Coimbra
Ovar, Portugal
patriciacosta@student.dei.uc.pt

***Abstract*—Neste artigo abordamos o tema da compressão sem perdas (*lossless*) de imagens. Explicamos alguns *codecs*, identificando os principais passos e códigos usados. Explanamos também os principais módulos dos *codecs*, transformadas aplicáveis no domínio e principais códigos utilizados no mesmo, bem como as suas combinações. No final da explicação de cada algoritmo dizemos o motivo para o termos escolhido. Falamos das principais etapas da compressão, comparamos os algoritmos tendo em conta critérios por nós pré-definidos e analisamos que algoritmos serão melhores para comprimir o *dataset* que nos foi apresentado. Posteriormente, testamos vários algoritmos (quer isolados quer combinados entre si) para encontrar uma solução ótima que comprima as imagens presentes no *dataset*, tendo sempre como referência os tamanhos das imagens PNG das originais BMP. No final, analisamos os resultados obtidos em cada teste e delineamos trabalho futuro.**

Palavras-chave—compressão, codec, compressão lossless, redundância, transformada, entropia, algoritmo, preditor, rácio de compressão, tempo de compressão, velocidade de compressão, Codificação de Huffman, Códigos Aritméticos, Delta encoding, LZW, BWT

I. INTRODUÇÃO

Atualmente, deparamo-nos no nosso quotidiano com grandes quantidades de informação. Estas podem ser de vários tipos: imagem, som, vídeo, texto, entre outras. Deste modo, como existe uma largura limitada de banda [2], é necessário comprimir estas fontes de informação, isto é, reduzir o espaço que estas ocupam, diminuindo a quantidade de bits necessários à sua representação. Posteriormente é necessário descomprimir essas fontes de informação para se poderem utilizar. Ao processo de codificar/descodificar deu-se o nome de *codec* [1].

Existem dois tipos de *codecs*: com perdas (*lossy*) e sem perdas (*lossless*) [2]. No caso de ser *lossless*, ao efetuar-se a descompressão, a fonte de informação estará exatamente igual a como estava antes da compressão. Já no caso de ser *lossy*, a fonte estará apenas muito semelhante (não igual) ao que estava anteriormente pois ocorrem perdas de informação. Neste estado da arte vamos apenas debruçar-nos sob o tipo *lossless* em imagens.

A compressão caracteriza-se por receber uma imagem (neste caso) como *input* e transformá-la numa imagem comprimida. A descompressão é o processo inverso [3].

II. ETAPAS DA COMPRESSÃO

A compressão subdivide-se em várias etapas, sendo as principais [1][3]:

1) Transformação: O objetivo desta etapa é tornar a fonte de informação a comprimir numa que se comprima mais eficaz e facilmente. Nesta etapa tanto pode ocorrer aumento da redundância da fonte, alteração da distribuição estatística dos símbolos e/ou empacotamento de grandes quantidades de informação em poucas amostras de dados ou regiões sub-banda.

2) Mapeamento de dados para símbolos: Esta etapa converte a fonte de informação em elementos que se chamam símbolos. Estes símbolos serão mais facilmente codificados na última etapa. Agrupando vários símbolos pode-se reduzir a correlação que eventualmente exista. Resumidamente, nesta etapa cria-se uma sequência de pares *(i, v)* (os chamados símbolos), em que *v* representa o valor da informação na fonte e *i* o número de vezes que o *v* é repetido. Este processo é super importante para otimizar o código. Mediante o tipo de codificador, este método pode sofrer algumas alterações (como no caso do JPEG) mas no geral, a ideia é esta que explicitamos.

3) Codificação de símbolos lossless: Nesta etapa é onde a codificação se realiza. Dependendo do *codec* utilizado, esta etapa comporta-se de diferentes maneiras.

Após esta explicação, fica claro que as primeiras 2 etapas podem ser consideradas de pré-processamento, uma vez que se destinam a trabalhar a fonte de informação de modo a que esta chegue à etapa 3 o melhor possível para ser comprimida.

Em suma, as etapas resumem-se em [3]:

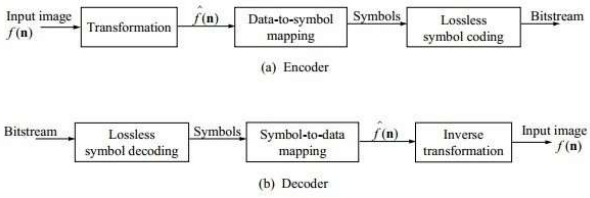


Figura 1 - Etapas de um *codec*

III. MÓDULOS DE COMPRESSÃO

Dependendo do seu objetivo e da maneira como o fazem, os métodos de compressão são divididos em categorias (módulos) [4]:

- **Métodos baseados na distribuição estatística:** Por exemplo, RLE e os algoritmos desenvolvidos por Lempel e Ziv. Ambos servem para eliminar a repetição de símbolos [5].
- **Métodos baseados na codificação entrópica:** Por exemplo, os Códigos de Huffman e os Códigos Aritméticos. Ambos os algoritmos acima mencionados, através da probabilidade de ocorrência de símbolos, tentam reduzir o número de bits necessários para codificar cada símbolo [5].
- **Métodos baseados na redundância:** Muitas vezes, é preciso tratar da informação antes de ela ser realmente comprimida para que, na fase da compressão, os dados sejam mais fáceis de comprimir. Para isto, usualmente, aumenta-se a redundância para diminuir a entropia. Isto é feito, por exemplo, por preditores e pela Transformada Burrow-Wheeler [6].

IV. CRITÉRIOS DE COMPRESSÃO

Não existem *codecs* melhores que outros. Cada situação deve ser avaliada isoladamente, isto é, para cada imagem existe 1 *codec* que se destaca dos outros por ser mais ótimo. Porém, esse mesmo *codec* pode não ser o aconselhável para outras imagens. Existem inúmeros critérios para avaliar a eficácia de um *codec*. Destes, destacam-se 3: velocidade de compressão, o rácio de compressão e velocidade de descompressão [1].
O rácio de compressão é dado por:

$$Rácio\ de\ compressão = \frac{Tamanho\ do\ ficheiro\ sem\ compressão}{Tamanho\ do\ ficheiro\ com\ compressão}$$

Fica assim evidente pela fórmula que quanto maior for a redução do tamanho do ficheiro, maior será o rácio de compressão.

A velocidade de compressão é calculada através de:

$$Velocidade\ de\ compressão\ (MB/s) = \frac{Tamanho\ do\ ficheiro\ sem\ compressão\ (MB)}{Tempo\ de\ compressão\ (s)}$$

E a velocidade de descompressão por:

$$Velocidade\ de\ descompressão\ (MB/s) = \frac{Tamanho\ do\ ficheiro\ sem\ compressão\ (MB)}{Tempo\ de\ descompressão\ (s)}$$

V. TRANSFORMADAS APLICÁVEIS

Transformadas são métodos que, por si só, não comprimem dados, mas que são capazes de os transformar em elementos que posteriormente serão mais facilmente comprimidos.

Tanto a transformada de Burrows-Wheeler (BWT) como a de Move-to-front, servem para codificar sequências grandes de caracteres iguais. Mais especificamente:

- **Burrows-Wheeler** [5]-[7]**:** realiza a rotação de um caracter de uma *string*, ordenando lexicograficamente a última coluna. Após todas as rotações, é possível verificar que os caracteres semelhantes se encontram agrupados. Deste modo, a compressão é facilitada.
- **Move-to-Front** [5][8]**:**

1.	Dado um alfabeto $\alpha = \{\alpha_1, \ldots, \alpha_n\}$ colocamos a posição de cada símbolo α_i numa sequência;
2.	O símbolo a codificar é movido para o topo do nosso alfabeto;
3.	Repete até ao fim da *string*.

Tabela 1 - Algoritmo da Transformada Move-to-Front

Assim, se um símbolo se repetir, serão colocados zeros a partir do início da repetição, substituindo os símbolos (uma vez que o caracter se irá encontrar sempre no início do alfabeto). Com isto, a entropia irá reduzir drasticamente, visto que a probabilidade de ocorrências de zeros aumentou. No entanto, para este algoritmo ser implementado tem de ser precedido de BWT para que os símbolos iguais sejam agrupados em grandes sequências. Posteriormente ainda é aplicado RLE (definido na secção *Principais Algoritmos*).

- **Preditores - "Partial Matching" (PPM)** [2][5][12]**:** Este algoritmo é basicamente um modelo de Markov de *n*-ésima ordem. Para cada símbolo de *A* numa dada posição, os *n* símbolos anteriores (contexto) irão ser utilizados para atualizar a sequência *B* de probabilidades até ao símbolo em questão. Caso o símbolo já exista, as estruturas necessárias são atualizadas. Se algum dos símbolos em *B* não estiver presente nesse contexto, voltamos a reduzi-lo, continuando à procura.
- **Delta encoding:** Este algoritmo é um dos mais utilizados visto ser um dos mais básicos. Baseia-se em codificar os símbolos da fonte com base na diferença do número de ocorrências em relação ao símbolo anterior. Isto faz com que se aumente a redundância da fonte, diminuindo a entropia.

VI. PRINCIPAIS ALGORITMOS

As seguintes técnicas de compressão de imagem são as que consideramos mais adequadas a utilizar nas fontes de informação que tínhamos para analisar.

A. *Codificação Run Length (RLE)*

Este é um dos métodos mais simples e práticos de codificar uma imagem [9]. Baseia-se em codificar pixéis de igual tom, isto é, de uma sequência de pixéis iguais, resulta apenas um símbolo $\{v, r\}$, onde *v* simboliza o tom dos pixéis e *r* o número de pixéis idênticos em sequência.

Na imagem abaixo [9], aparece exemplificado o processo.

80	80	80	56	56	56	56	56	78

{80,3}	{56,5}	{78,1}

Figura 2 - Exemplo do processo do RLE

Decidimos que devíamos testar este algoritmo, uma vez que é um dos mais simples e utilizados para codificar uma imagem.

B. *Codificação de Huffman*

Para utilizar este método com segurança de que será eficaz, convém conhecer a distribuição estatística e probabilística dos símbolos da fonte de informação a priori [3]. Com este método conseguimos atribuir um menor número de bits aos símbolos que têm maior probabilidade de ocorrer [10]. Afirmamos isto uma vez que este algoritmo se resume em [2][5]:

1.	Ordena os símbolos por ordem crescente de probabilidade de ocorrência;
2.	Constrói árvore binária: Combina os dois símbolos menos frequentes num único e acrescenta o símbolo resultante à lista em que a frequência de ocorrência é a soma das frequências individuais;
3.	Itera até encontrar novo símbolo.

Tabela 2 - Algoritmo da Codificação de Huffman

Consideramos que este algoritmo seja uma boa forma de comprimir imagens pois consegue reduzir bastante o número de bits necessários para codificar os símbolos de maior ocorrência. Isto leva a uma diminuição significativa do tamanho de um ficheiro.

C. *Códigos Aritméticos*

À semelhança da Codificação de Huffman, estes códigos também se baseiam na codificação entrópica [3]. A diferença entre eles é que, enquanto o de Huffman codifica pelo menos 1 bit por símbolo (a menos que se agrupem símbolos, o que leva a um crescimento exponencial do dicionário), nos códigos aritméticos pode existir um número fracionário de bits por símbolo. No entanto, este código requer mais software e hardware, uma vez que obtém um rácio de compressão maior que o de Huffman. Resumidamente, os códigos aritméticos representam a sequência de símbolos com base numa TAG [5]. Existem infinitas TAGs uma vez que são retiradas de um intervalo [0, 1[.

Escolhemos este algoritmo para comprimir as nossas fontes de informação pela mesma razão que escolhemos a Codificação de Huffman.

D. *Codificação Lempel-Ziv (LZ)*

Como nem sempre é possível saber de antemão a distribuição estatística de uma fonte de informação, existem outros tipos de algoritmos que não necessitam dessa informação *a priori* [3]. É o caso dos algoritmos desenvolvidos por Lempel e Ziv. Estes desenvolveram um tipo de codificação que envolve dicionários. Como existem padrões que ocorrem com maior frequência que outros, a utilização de dicionários nestas circunstâncias permite poupar muitos bits na transmissão [5].

Existem inúmeras variações da Codificação LZ (LZ77, LZ78 e LZW) diferindo, por exemplo, na forma como os dicionários são implementados [3]. Resumidamente:

1.	Envolve uma *search window* que vai guardando a informação já lida;
2.	É passada a referência da informação já existente, à nova (que é igual);
3.	Itera até ao fim.

Tabela 3 – Algoritmo geral da Codificação LZ

Isto leva a que se consiga fazer uma redução da redundância. No entanto, estes algoritmos baseados em dicionários adaptativos são ineficazes quando se pretende codificar uma fonte de informação pequena [3].

Decidimos que devíamos testar as variantes deste algoritmo, uma vez que nem sempre se consegue ter acesso à distribuição estatística da fonte *a priori*. Por esta razão, e como este método, geralmente, consegue reduzir em muito o número de bits da fonte, consideramos este algoritmo uma boa solução para o nosso problema.

D.1. LZ77

A primeira variante a ser criada foi o LZ77 [5]. Este tem como objetivo encontrar a maior sequência comum que ocorre na *search window* para poder transmitir uma referência dessa sequência. Assim que o codificador verifica qual o maior padrão na *search window*, codifica o seguinte: {*offset*, *comprimento do padrão*, *código*}, sendo *offset* o número de elementos a recuar na *search window*, o *comprimento do padrão* o número de elementos a copiar e o *código* o próximo caracter a transmitir. No entanto, apesar de o LZ77 ter a vantagem de não necessitar da construção de um dicionário (construção implícita), o padrão que procuramos tem de se encontrar na *search window*. Como a *search window* tem um tamanho limitado, mesmo que esse padrão já exista anteriormente (mais atrás do que a *window* abrange), já não se pode utilizar essa referência.

Deste modo, como este algoritmo apenas procura o padrão na vizinhança do local em que se encontra (daí o LZ77 ser um algoritmo de procura local), torna-se ineficiente.

D.2. LZ78

Para contrariar a ineficiência do LZ77, o LZ78 [5] é já um algoritmo de procura global e constrói explicitamente um dicionário composto por tuplos do tipo <*i*, *c*>, que apresentam um índice (*i*) e o código do próximo caracter (*c*), além da entrada no dicionário. Resumidamente, a cada iteração, o algoritmo procura o índice da maior *substring* comum já existente no dicionário, assim como o primeiro caracter fora do padrão. No entanto, à semelhança do que acontece com o LZ77, este algoritmo também apresenta desvantagens. Além de enviar o índice da maior *substring* para codificar a fonte de informação em questão, envia também o primeiro caracter que estava fora do padrão, atualizando o dicionário. Como este próximo caracter vai criar um *overhead,* torna-se ineficiente. O ideal seria apenas enviar os índices.

D.3. LZW

A codificação que veio resolver o problema do LZ78 foi o LZW [5]. Este inicia já com o dicionário pré-preenchido com todo o alfabeto existente na mensagem. Outra diferença é que já não utiliza o próximo caracter da *string*: usa apenas os índices. Deste modo, caso a *substring* ainda não se encontre no dicionário será transmitido o índice da já presente e cria-se uma nova entrada no dicionário para a *substring* total.

E. *Deflate*

Este algoritmo [5][11] utiliza 2 já abordados: LZ77 e Codificação de Huffman [2]. O Deflate vai começar por dividir os dados de entrada em blocos. Posteriormente, vai utilizar o LZ77 para encontrar os dados repetidos. De seguida, os símbolos são substituídos por outros com base no seu número de ocorrências (após Codificação de Huffman).

Escolhemos este algoritmo uma vez que, analisando alguns resultados, percebemos que apresentava ótimas velocidades de compressão e descompressão.

F. *Bzip2*

Este é um algoritmo [2][5] que surgiu em 1996 e que utiliza Códigos de Huffman, RLE e BWT. O Bzip2 divide os dados em blocos entre 100KB e 900KB e posteriormente comprime cada um utilizando BWT. Basicamente, o seu processo é o seguinte [2]:

1.	Lê 1 caracter de cada vez;
2.	Aplica RLE nos dados lidos;
3.	BWT vai organizar os dados, juntando os que são iguais;
4.	Cada caracter do bloco é substituído pelo índice da primeira ocorrência, sendo colocado de seguida no topo do bloco;
5.	Por fim, aplica a Codificação de Huffman a cada caracter.

Tabela 4 - Algoritmo do Bzip2

Escolhemos este algoritmo uma vez que, analisando alguns resultados, percebemos que apresentava bons rácios de compressão e ótimas velocidades de compressão.

VII. COMPARAÇÃO DE ALGORITMOS

Utilizando o *Silesia Corpus*, testaram-se alguns algoritmos. Destes testes [2], conclui-se que:

- Deflate apresenta um rácio de compressão baixo, mas é o que evidencia velocidades de compressão e descompressão mais rápidas;
- PPM, por outro lado, apresenta um elevado rácio de compressão, mas velocidades de compressão e descompressão baixas.
- Bzip2 apresenta um rácio de compressão interessante e uma velocidade de compressão elevada.

VIII. PRINCIPAIS *CODECS LOSSLESS*

Codec, tal como já foi explicado na introdução, é o nome dado ao processo de codificar/descodificar uma fonte de informação. Para isso, cada *codec* utiliza 1 ou mais algoritmos.

A. GIF

GIF [14] foi apresentado ao mundo em 1987 como algo que permitia ter uma imagem a cores na área de transferência de arquivos. Este veio substituir um formato RLE que era apenas a preto e branco. Como o GIF utiliza codificação LZW, na altura tornou-se muito popular pois era possível transferir imagens relativamente grandes a uma velocidade rápida. Resumidamente, as imagens GIF são compactadas usando a técnica de compactação de dados sem perdas LZW para reduzir o tamanho do arquivo. Porém, cada pixel, em vez de possuir o seu valor RGB (24 bits por pixel), possui os índices da "paleta de cores" [5].

Escolhemos o GIF, uma vez que é dos mais utilizados.

B. PNG

PNG [13][15] apareceu em 1996 e foi um *codec* que veio melhorar e substituir o GIF, uma vez que cada pixel possui o seu valor RGB (mudança de um índice da paleta de cores para um valor de 24 bits). Este *codec* permite retirar o fundo de imagens com uso do canal alfa, uma vez que tal canal consegue definir a opacidade de cada pixel. No seu funcionamento, PNG utiliza Deflate, isto é, uma mistura de LZ77 com Codificação de Huffman.

Escolhemos este *codec*, uma vez que apresenta vantagens em comparação ao GIF, sendo também muito utilizado.

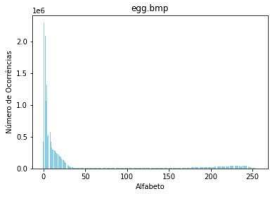
C. JBIG2

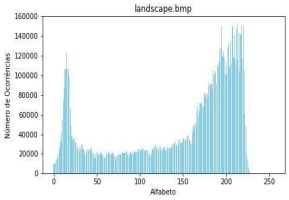
Em 2000 apareceu o JBIG2 [16], um melhoramento do JBIG. Este apresenta rácios de compressão 2 a 4 vezes maiores que os do seu antecessor. Basicamente, o JBIG2 divide a informação em zonas de texto, de imagens de tons médios e em zonas de outras coisas. Estas últimas são comprimidas utilizando códigos aritméticos.

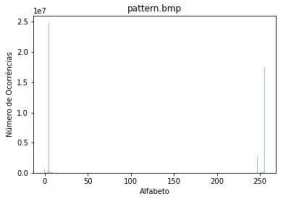
Escolhemos JBIG2 por ser um dos mais recentes a ter aparecido e apresentar rácios de compressão interessantes.

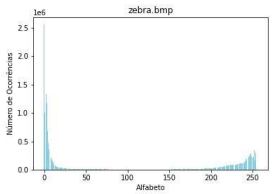
IX. *ANÁLISE DO DATASET*

Das 4 imagens que tínhamos para analisar, estes foram os resultados da distribuição estatística, entropia e potencial de compressão entrópica:









Figuras 3 a 7 – Distribuições estatísticas do *dataset*

Imagem	Entropia	Potencial de compressão entrópica
egg.bmp	5.7242	28.45%
landscape.bmp	7.4205	7.24%
pattern.bmp	1.8292	77.13%
zebra.bmp	5.8312	27.11%

Tabela 5 - Valores da entropia e potencial de compressão entrópica do *dataset*

Pela análise dos histogramas de ocorrência dos símbolos de cada uma das imagens do *dataset* e dos valores da entropia e potencial de compressão entrópica, retiramos as seguintes conclusões:

- pattern.bmp, tendo um elevado potencial de compressão entrópica, deve comprimir bem com algoritmos que se baseiam na entropia (Codificação de *Huffman* e Códigos Aritméticos);
- Pelo contrário, landscape.bmp apresenta uma entropia elevada. Por isso, entendemos que os métodos baseados na redução desta são os mais aconselháveis, por exemplo, utilizar *Delta encoding*, para aumentar a redundância, mas diminuir a entropia;
- Já zebra.bmp e egg.bmp, tendo histogramas, entropias e potenciais de compressão semelhantes, devem utilizar os mesmos algoritmos na sua melhor maneira de compressão. Como ambas as imagens apresentam um "pico" de concentração nos histogramas, ao utilizar algoritmos que diminuem a repetição de símbolos deve ajudar a reduzir a redundância. Isto levará a que os ficheiros sejam melhor comprimidos.

Decidimos começar por aplicar *Delta encoding* [18] em todas as imagens para aumentar a redundância. Isto levou a uma natural diminuição da entropia, tal como consta na tabela seguinte:

Imagem	Entropia antes	Entropia após Delta encoding
egg.bmp	5.7242	2.6994
landscape.bmp	7.4205	2.8242
pattern.bmp	1.8292	0.5957
zebra.bmp	5.8312	3.2219

Tabela 6 - Valores da entropia antes e depois de se aplicar *Delta encoding* no *dataset*

Tendo a informação codificada pelo *Delta encoding*, decidimos aplicar um codec que envolve Códigos de *Huffman* [17]. O resultado da compressão foi notório. Embora tenha demorado algum tempo a comprimir, conseguiram-se bons rácios de compressão:

Imagem	Rácio de compressão
egg.bmp	16.9 MB / 5.74 MB = 2.944
landscape.bmp	10.4 MB / 3.76 MB = 2.766
pattern.bmp	45.7 MB / 7.37 MB = 6.201
zebra.bmp	15.9 MB / 6.46 MB = 2.461

Tabela 7 – Rácios de compressão do dataset após se implementar a combinação de Delta encoding com Codificação de Huffman

Este foi um processo que demorou sensivelmente 2 minutos a comprimir, o que, podemos considerar tempos bastante aceitáveis e interessantes. Aliando isso aos rácios de compressão que obtivemos, podemos concluir que este é um bom método de compressão de imagens *lossless*.

Sendo que o *Delta encoding* é um processo que não comprime dados e que apenas os reescreve tendo em conta a diferença do número de ocorrências entre um símbolo e o seu antecessor, decidimos ver como se comportaria a Codificação de *Huffman* sem que existisse este tratamento prévio dos dados.

Imagem	Rácio de compressão
egg.bmp	16.9 MB / 12.1 MB = 1.397
landscape.bmp	10.4 MB / 9.77 MB = 1.064
pattern.bmp	45.7 MB / 11.0 MB = 4.155
zebra.bmp	15.9 MB / 11.7 MB = 1.359

Tabela 8 – Rácios de compressão do *dataset* após se implementar Codificação de *Huffman*

Fica assim evidente que, embora os tempos de compressão e descompressão sejam idênticos, os rácios de compressão são menores. Com isto, concluímos que a implementação do *Delta encoding* antes da utilização da Codificação de *Huffman* é vantajosa. Isto justifica-se pelo facto de não alterar o tempo de compressão, não requerer muito mais esforço de *hardware* e *software* e os rácios de compressão serem mais elevados.

Sendo os códigos aritméticos um algoritmo muito semelhante à Codificação de Huffman, com a vantagem de estes pouparem alguns bits na codificação, decidimos testá-los no nosso *dataset*. Utilizando apenas códigos aritméticos [20], os resultados são os seguintes:

Imagem	Rácio de compressão
egg.bmp	16.9 MB / 12.1 MB = 1.397
landscape.bmp	10.4 MB / 9.73 MB = 1.069
pattern.bmp	45.7 MB / 10.4 MB = 4.394
zebra.bmp	15.9 MB / 11.6 MB = 1.371

Tabela 9 – Rácios de compressão do *dataset* após se implementar códigos aritméticos

Com isto, concluímos que para o nosso *dataset*, não é vantajoso utilizar códigos aritméticos em detrimento de Codificação de Huffman. Verificamos isto pois para além de o algoritmo agora testado exigir mais de *hardware*, de *software* e de demorar mais tempo a comprimir, os rácios de compressão não apresentam uma melhoria significativa em relação aos apresentados pela Codificação de *Huffman*.

Como o algoritmo de *Huffman* melhorou o seu desempenho aplicando previamente Delta encoding no *dataset*, decidimos testar essa técnica neste caso também. Deste modo, os resultados da combinação de Delta encoding com códigos aritméticos encontram-se abaixo:

Imagem	Rácio de compressão
egg.bmp	16.9 MB / 5.70 MB = 2.964
landscape.bmp	10.4 MB / 3.70 MB = 2.811
pattern.bmp	45.7 MB / 3.4 MB = 13.441
zebra.bmp	15.9 MB / 6.42 MB = 2.477

Tabela 10 – Rácios de compressão do *dataset* após se implementar a combinação de *Delta encoding* com Códigos Aritméticos

Estes resultados vieram comprovar o que já tínhamos concluído no teste anterior. O acréscimo de tempo de compressão, de esforço de *hardware* e a similaridade com os rácios de compressão obtidos na codificação de Huffman não compensam a utilização dos códigos aritméticos. Apenas *pattern.bmp* apresenta diferenças significativas e, por essa razão, esta é uma combinação mais ótima para essa fonte.

Como vimos anteriormente, os algoritmos desenvolvidos por Lempel e Ziv conseguem reduzir eficazmente a redundância de uma fonte. Como o algoritmo LZW [19] é muito utilizado para comprimir fontes de informação e, como das variantes LZ abordadas neste artigo é o que apresenta mais vantagens, decidimos testá-lo.

Apenas comprimindo utilizando LZW, os resultados são os seguintes:

Imagem	Rácio de compressão
egg.bmp	16.9 MB / 11.0 MB = 1.536
landscape.bmp	10.4 MB / 10.7 MB = 0.972
pattern.bmp	45.7 MB / 4.61 MB = 9.913
zebra.bmp	15.9 MB / 9.83 MB = 1.617

Tabela 11 – Rácios de compressão do *dataset* após se implementar LZW

Apesar de o processo ser rápido (tanto o de compressão como o de descompressão), nem sempre se obtém um rácio de compressão interessante ao aplicar diretamente o algoritmo LZW. Como podemos verificar nos resultados apresentados no quadro acima, existe inclusive uma fonte de informação (*landscape.bmp*) em que o ficheiro resultante da suposta compressão tem um tamanho superior ao do que se pretende comprimir. No entanto, para a fonte *pattern.bmp* o rácio obtido foi muito interessante. Isto pode-se explicar uma vez que esta fonte tem uma baixa entropia que pode estar associada ao facto de possuir bastante redundância nos seus dados. Com isto, torna-se mais comprimível pelo LZW que é uma excelente forma de reduzir a redundância de uma fonte.

Tendo este aspeto como referência, sendo o *Delta encoding* uma excelente forma de aumentar a redundância de uma fonte, pensamos que o LZW seria mais eficaz se utilizado após a implementação do Delta encoding.

Com isto, os resultados dos testes foram os seguintes:

Imagem	Rácio de compressão
egg.bmp	16.9 MB / 5.72 MB = 2.955
landscape.bmp	10.4 MB / 3.66 MB = 2.842
pattern.bmp	45.7 MB / 2.75 MB = 16.618
zebra.bmp	15.9 MB / 6.41 MB = 2.480

Tabela 12 – Rácios de compressão do *dataset* após se implementar a combinação de *Delta encoding* com LZW

Tal como era de esperar, os rácios de compressão aumentaram. Os tempos de compressão mantiveram-se baixos, portanto esta é uma alternativa melhor que a anterior.

Ficou por demais evidente nestes 2 últimos testes que o LZW é tão mais eficaz quanto mais redundante for a fonte de informação que se pretende comprimir.

Outro algoritmo que gera redundância da fonte e por isso aconselhável a utilizar com LZW é a Transformada de *Burrows-Wheeler*. Uma vez que as fontes de informação que pretendemos comprimir têm um tamanho assinalável, tivemos de proceder à implementação da BWT por partes, isto é, dividir os dados em blocos de tamanho x e aplicar a referida transformada bloco a bloco.

Decidimos testar esta combinação no nosso *dataset*.

Fizemo-lo em 2 versões. Uma em que o tamanho do bloco era de 512 e outra de 1024.

No final dos testes, concluímos que quanto maior o tamanho dos blocos, maior é o tempo de compressão. Por outro lado, quanto maior for esse tamanho, melhor depois é a compressão por parte do algoritmo LZW.

Os resultados desta combinação são os seguintes:

Tamanho = 512	Imagem	Rácio de compressão
	egg.bmp	16.9 MB / 10.3 MB = 1.641
	landscape.bmp	10.4 MB / 11.1 MB = 0.937
	pattern.bmp	45.7 MB / 6.87 MB = 6.652
	zebra.bmp	15.9 MB / 10.1 MB = 1.574

Tamanho = 1024	Imagem	Rácio de compressão
	egg.bmp	16.9 MB / 10.2 MB = 1.657
	landscape.bmp	10.4 MB / 11.1 MB = 0.937
	pattern.bmp	45.7 MB / 6.94 MB = 6.585
	zebra.bmp	15.9 MB / 9.94 MB = 1.600

Tabelas 13 e 14 – Rácios de compressão do *dataset* após se implementar a combinação de BWT (aplicada em blocos de tamanho referido nas tabelas) com LZW

Tal como podemos constatar, a diferença entre implementar a Transformada Burrows-Wheeler em blocos de tamanho 512 e de tamanho 1024 não é significativa ao nível dos rácios de compressão. Porém, o processo que envolve BWT, no caso de os blocos terem 1024 elementos, demora aproximadamente o dobro do tempo do que quando os blocos têm tamanho 512. Isto pode-se explicar devido ao facto de quantos mais elementos tiver o bloco, mais combinações este terá de gerar e, por conseguinte, mais ordenações tem de efetuar (resumo simples do processo da BWT).

Com isto, fica evidente que utilizar uma diferença tão pequena entre os tamanhos dos blocos não revela nenhuma melhoria significativa no rácio de compressão. Sendo o ideal conseguir aplicar BWT na informação toda de uma vez (não possível com os meios de *hardware* e *software* que dispomos), quanto maior for o tamanho do bloco, mais eficaz se torna o uso da transformada.

Por tudo o que enunciamos acima, entendemos que, não compensou aplicar BWT em blocos de 1024 elementos. Além de o tempo de execução do código ter passado para o dobro, não houve melhorias nos rácios de compressão.

Em suma, os resultados dos testes obtidos para cada imagem foram os seguintes:

Algoritmo(s)	egg.bmp	landscape.bmp	pattern.bmp	zebra.bmp
Huffman	1.397	1.064	4.155	1.359
Delta + Huffman	2.944	2.766	6.201	2.461
Aritméticos	1.397	1.069	4.394	1.371
Delta + Aritméticos	2.964	2.811	13.441	2.477
LZW	1.536	0.972	9.913	1.617
Delta + LZW	2.955	2.842	16.618	2.480
BWT (512) + LZW	1.641	0.937	6.652	1.574
BWT (1024) + LZW	1.657	0.937	6.585	1.600

Tabela 15 – Resumo dos rácios de compressão do *dataset* obtidos em cada teste

Sendo os valores destacados a amarelo os rácios de compressão mais elevados que obtivemos nos nossos testes, fica claro que a combinação de Delta encoding com LZW foi a mais ótima em termos de rácios de compressão na maior parte dos ficheiros. Apenas *egg.bmp* comprimiu melhor com a combinação de Delta encoding com códigos aritméticos. Porém, como o algoritmo dos códigos aritméticos tem um tempo de execução maior que a combinação Delta encoding com LZW (que apresentou o 2.º maior rácio de compressão para esta fonte), podemos afirmar que esta seria uma forma de compressão mais adequada que a que revelou um rácio de compressão ligeiramente superior (na ordem das centésimas).

Para *pattern.bmp*, ficou claro que das experiências que fizemos, a que melhor se adequa a si é a combinação Delta encoding com LZW. Além de serem algoritmos que têm um baixo tempo de execução, foram os que comprimiram mais esta imagem.

Já para *landscape.bmp* e *zebra.bmp* existiram várias experiências em que os rácios de compressão verificados foram muito semelhantes entre si. Com isto, para determinarmos qual a melhor combinação, tivemos de ter em conta o fator tempo de execução. Assim, excluíram-se logo as combinações que envolviam códigos aritméticos, ficando em análise as combinações entre Delta encoding e Codificação de Huffman e entre Delta encoding e LZW. Em ambos os casos, o tempo de execução dos algoritmos foi muito parecido, pelo que prevaleceu a combinação que apresentava maior rácio de compressão (mesmo que a diferença fosse na ordem das centésimas), ou seja Delta encoding com LZW.

X. *COMPARAÇÃO COM VALORES DE REFERÊNCIA*

À partida para este projeto, tínhamos valores de referência para o tamanho das imagens comprimidas (imagens PNG das originais BMP). Tendo em conta os maiores rácios de compressão que obtivemos nos nossos testes e não as soluções mais ótimas (ou seja, não contando com o fator tempo de execução do código), chegamos a estes tamanhos:

Imagem	PNG	Nossas
egg.bmp	4.41 MB	5.70 MB
landscape.bmp	3.17 MB	3.66 MB
pattern.bmp	2.17 MB	2.75 MB
zebra.bmp	5.21 MB	6.41 MB

Tabela 16 – Comparação dos tamanhos das imagens em formato PNG fornecidas com o menor tamanho atingido pelas nossas compressões

Analisando a tabela acima, rapidamente percebemos que quase atingimos o rácio de compressão ideal em duas das imagens do *dataset* (*landscape.bmp* e *pattern.bmp*). Isto deve-se ao facto de termos focado mais em soluções que aumentavam a redundância dos dados, o que facilitou a compressão dessas duas imagens.

Já *zebra.bmp* e *egg.bmp*, como já referido anteriormente, pela similaridade dos seus histogramas de ocorrência de símbolos, devem obter a sua melhor compressão com os mesmos algoritmos. Posto isto, ao não encontrar a melhor combinação para comprimir uma das fontes, obrigatoriamente também não se encontra o algoritmo ideal para a outra. Pensamos que a utilização de mecanismos que reduzem a redundância dos dados (como o LZW) resultaria, uma vez que estas apresentam um "pico" de concentração nos seus histogramas, mas os resultados vieram a revelar-se pouco eficazes.

XI. *CONCLUSÃO E TRABALHO FUTURO*

Tínhamos como objetivo à partida para este trabalho encontrar a melhor forma de comprimir o *dataset* que nos era proposto (*egg.bmp, landscape.bmp, pattern.bmp, zebra.bmp*). Na nossa opinião, conseguimos ser bem sucedidos em duas das imagens (*landscape.bmp* e *pattern.bmp*). Porém, nas restantes não conseguimos atingir os valores de referência que tínhamos das imagens em formato PNG. Posto isto, entendemos que futuramente devíamos testar outro tipo de algoritmos nestas duas fontes de informação, como o Bzip2. Pensamos que assim se consiga obter valores próximos dos de referência (próximos dos das imagens PNG).

Outra situação que gostávamos de testar no futuro era aplicar a Transformada Burrows-Wheeler em blocos de tamanho superior ao que aplicamos. Pensamos que, apesar de o tempo de execução do código aumentar, os valores dos rácios de compressão passariam a ser mais interessantes.

Em suma, tendo este trabalho de investigação o objetivo de explorar os conceitos de teoria da informação, mais concretamente, os que dizem respeito à teoria da compressão, tínhamos de encontrar uma solução para comprimir de forma eficiente e não destrutiva imagens monocromáticas. Após apresentarmos os conceitos inerentes a este assunto, tentamos encontrar a melhor solução para comprimir o *dataset*. Nesta fase, encontramos uma combinação de algoritmos quase ótima para duas das imagens que nos foram propostas comprimir e ainda soluções para as outras duas que, apesar de não ótimas, revelaram valores interessantes (de acordo com os critérios por nós pré definidos).

Por tudo isto, pensamos que fomos bem sucedidos. O tema da compressão de dados é algo que envolve vários conceitos que depois interligados entre si dão aso a inúmeras formas de comprimir fontes de informação. Deste modo, sendo nós alunos de Engenharia Informática, ficamos motivados para no futuro explorar mais algoritmos, não só para comprimir imagens como também outro tipo de ficheiros.

REFERÊNCIAS

[1] "Codec" in *pt – Wikipedia*. https://pt.wikipedia.org/wiki/Codec [26 de novembro de 2020]

[2] Gupta, A. Bansal, V. Khanduja, Modern Lossless Compression Techniques: Review, Comparison and Analysis, Second International Conference on Electrical, Computer and Communication Technologies (ICECCT 2017)

[3] Lina J., K. (2009). Lossless Image Compression. In The Essential Guide to Image Processing (1st ed.). Elsevier. https://doi.org/10.1016/B978-0-12-374457-9.00016-0

[4] "Compressão de dados" in *pt – Wikipedia*. https://pt.wikipedia.org/wiki/Compress%C3%A3o_de_dados [25 de novembro de 2020]

[5] Carvalho P., (2020). Capítulo II – Teoria da Informação e Codificação Entrópica

[6] Martins, André; Laranjeira, Filipe; Cunha, José; Silva, Luís. *Transformada de Burrows-Wheeler.*http://multimedia.ufp.pt/codecs/compressao-sem-perdas/supressao-de-sequencias-repetitivas/metodo-de-burrows-wheeler/ [25 de novembro de 2020]

[7] "Método de Burrows-Wheeler" in *pt – Wikipedia*. https://pt.wikipedia.org/wiki/M%C3%A9todo_de_Burrows-Wheeler [28 de novembro de 2020]

[8] "Move-to-front transform" in *en – Wikipedia*. https://en.wikipedia.org/wiki/Move-to-front_transform [28 de novembro de 2020]

[9] Ms. Ijmulwar, D. Kapgate, A Review on - Lossless Image Compression Techniques and Algorithms, International Journal of Computing and Technology, Volume 1, Issue 9, October 2014

[10] Enes, Diogo; Domingues, Filipe; Alão, Tiago Mota. *Algoritmo de Huffman.*http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/algoritmo-de-huffman/ [28 de novembro de 2020]

[11] "DEFLATE" in *pt – Wikipedia*. https://pt.wikipedia.org/wiki/DEFLATE [25 de novembro de 2020]

[12] Santos, Ricardo Caló; Ferreira, Nelson. *Prediction by Partial Matching (PPM).*http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/prediction-by-partial-matching-ppm/ [26 de novembro de 2020]

[13] Kaur, Rajandeep. *A review of Image Compression Techniques*. https://www.researchgate.net/publication/303325238_A_Review_of_Image_Compression_Techniques [28 de novembro de 2020]

[14] "GIF" in *en – Wikipedia*. https://en.wikipedia.org/wiki/GIF [28 de novembro de 2020]

[15] "PNG" in *pt – Wikipedia*. https://pt.wikipedia.org/wiki/PNG [26 de novembro de 2020]

[16] "JBIG2" in *en – Wikipedia.* https://en.wikipedia.org/wiki/JBIG2 [28 de novembro de 2020]

[17] Código fornecido pelos docentes da cadeira

[18] "Delta encoding" in *en – Wikipedia.* https://en.wikipedia.org/wiki/Delta_encoding [07 de dezembro de 2020]

[19] "Image-Compression-using-LZW" in *GitHub*. https://github.com/Showndarya/Image-Compression-using-LZW [02 de dezembro de 2020]

[20] "Reference-arithmetic-coding" in *GitHub*. https://github.com/nayuki/Reference-arithmetic-coding/tree/master/python [21 de dezembro de 2020]