Teoria da Informação
Trabalho Prático n.º2

CODEC não destrutivo para Imagens

Alexy de Almeida
2019192123

Sofia Botelho
2019227240

Sofia Neves
2019220082

Abstract*—Este trabalho irá abordar as diversas técnicas de compressão de imagens sem perda de informação (lossless). A utilidade da compressão lossless é reduzir o espaço ocupado por determinado ficheiro sem reduzir informação visível a olho humano. O objetivo é criar um CODEC que permita comprimir quatro imagens monocromáticas e analisar os resultados em cada uma, tendo como base de comparação essas imagens comprimidas no formato png. Como não existe nenhum algoritmo de compressão perfeito para todas as imagens, pretendemos criar aquele que nos pareça adequado para diversas situações e que tenha em conta as características das quatro imagens, sem torná-las o foco principal do trabalho.

Palavras-chave*—compressão, codec, lossy, lossless, transformada, Huffman, aritmético, dicionário, preditivo, redundância, codificação.

I. Introdução

A compressão surgiu com o propósito de reduzir o espaço ocupado por um determinado ficheiro num determinado dispositivo através da junção de um ou vários algoritmos de compressão, obtendo-se um resultado final que denominamos por "codec". Os codecs podem ser de um de dois tipos diferentes: lossless ou lossy.

II. Tipos de compressão

A. *Compressão Lossless*

A compressão lossless é uma compressão sem perda de dados, ou seja, não existe perda de informação. Para tal, é feita uma reconstrução do ficheiro com o menor número de bits possível. Exemplos de compressão lossless são Huffman Encoding, LZ77, BWT, CALIC, entre outros.

B. *Compressão Lossy*

A compressão lossy é uma compressão onde existe perda de informação ao removerem-se bits desnecessários, sem comprometer a fidelidade e autenticidade do ficheiro. Uma vez feita a compressão lossy, o ficheiro não pode ser descomprimido de volta para o original, uma vez que parte da informação original foi perdida. Isto implica que os decoders não consigam reconstruir a imagem original com a mesma informação antes do encoding. Logo, há um maior rácio de compressão que a compressão lossless. Exemplos de compressão lossy são: MP3, JPEG, HEVC, MPEG4, AVC, entre outros.

III. Processo de compressão lossless

A. *Transformada*

É aplicada uma transformada aos dados da imagem, convertendo-os para uma forma que possa depois ser comprimida mais eficientemente, permitindo reduzir a redundância. Exemplos de transformadas: Burrows-Wheeler, Move-To-Front, entre outros.

B. *Data-to-symbol mapping*

A transformada aplicada anteriormente é convertida para um conjunto de símbolos. É um passo importante para otimizar o código, dependendo das suas aplicações e limitações da complexidade do hardware/software.

C. *Lossless symbol coding*

Aos símbolos previamente criados são associados códigos binários. Os mesmos podem ser representados através de esquemas:

(i) Estatísticos: Requerem conhecimento da probabilidade dos símbolos fonte. Os códigos menores são atribuídos aos símbolos de maior probabilidade de ocorrência. Ex.: Huffman, Aritmético, etc..

(ii) Baseados em dicionários: Não necessitam de um conhecimento prévio da probabilidade de cada símbolo. Dinamicamente constroem tabelas de compressão e descompressão de strings de tamanho variável. À medida que é construída, são gerados códigos binários que depois são indexados na tabela de codificação. Ex.: LZ77, LZ78, LZW, etc..

(iii) De codificação universal: Geram códigos binários de tamanho variável. Não é necessário conhecer as probabilidades à priori, no entanto são construídos com a assumpção que as distribuições vão decrescendo, ou seja, valores binários mais baixos são mais prováveis. Ex.: Elias e Exp-Golomb.

Neste trabalho, apenas os esquemas (i) e (ii) serão abordados.

IV. Tipos de redundância

A. *Redundância de codificação*

Baixa-se o número médio de bits por pixel. Quanto maior a frequência de ocorrência do pixel, menos bits serão codificados, reduzindo-se o tamanho do código.

B. *Redundância espacial inter-pixel*

O valor de qualquer pixel pode ser pressuposto através do valor dos pixels adjacentes se estiverem fortemente correlacionados.

C. *Redundância psico-visual*

O olho humano ignora alguns pixels e por isso alguma informação pode ser descartada sem degradar a imagem.

V. Fatores de escolha de um codec

A. *Eficiência de compressão*

Compara o tamanho do input original com o tamanho do output gerado após comprimido. A eficiência é dada pelo rácio de compressão, C_R:

$$C_R = \frac{Tamanho\ total\ de\ bits\ do\ input\ original\ da\ imagem}{Tamanho\ total\ de\ bits\ após\ ter\ comprimido\ a\ imagem}$$

É comum a taxa média de bits ser expressa também em bits por pixel, ou *bbp* resumidamente:

$$B = \frac{Tamanho\ total\ de\ bits\ da\ bitstream\ comprimida}{Número\ total\ de\ pixeis\ da\ imagem\ input\ \ original}$$

B. Tempo de compressão

Tempo que demora a codificar e decodificar determinados dados de entrada. O tempo aumenta de acordo com o número de operações e memória necessários.

C. Complexidade de implementação

A complexidade aumenta de acordo com o total de operações aritméticas necessárias de realizar e com o espaço em memória necessário de ocupar. Para aplicações que necessitam de consumir energia, esse também será um fator importante. No geral, maior compressão é alcançada quanto maior é a complexidade de implementação.

D. Robustez

É necessário que o código seja robusto para não gerar erros de transmissão quando o mesmo esteja a ser comprimido.

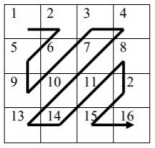
E. Escalabilidade

Os codificadores escaláveis geram uma bitstream de dados em camadas incorporando uma representação dos dados por hierarquia. Assim, os dados de entrada podem ser recuperados em diferentes resoluções.

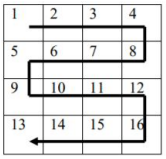
VI. PATH SCANNING

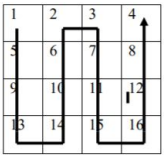
Após terem sido extraídos os dados de entrada, é gerada uma matriz *N*x*N* da imagem. Para ser possível tratar os dados da imagem é necessário converter essa matriz na forma $1 \times N^2$. Para tal, existem várias formas de percorrer os dados de entrada - *scan*:

(i) Zig-Zag Scan

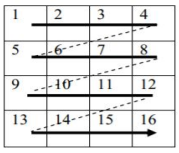


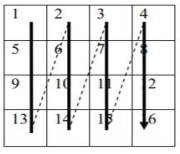
(ii) Snake Scan - Horizontal e Vertical





(iii) Raster Scan - Horizontal e Vertical





VII. TRANSFORMADAS

A. Move to front (MTF)

O método substitui os símbolos da fonte pelo seu índice numa lista constituída pelo respectivo alfabeto ordenado por ordem de símbolos que ocorreram mais recentemente. Se o símbolo que queremos codificar ainda não ocorreu anteriormente, devolvemos o seu índice da lista atual e reordenamos-la, colocando o símbolo no seu início. Se o símbolo ocorrer mais do que uma vez consecutiva, devolvemos o seu índice atual e não reordenamos a lista.

Vantagens: para fontes com muitos símbolos repetidos, são atribuídos índices com valores menos elevados, o que é vantajoso para algoritmos de compressão que possam ser combinados com o MTF.

Desvantagens: para fontes com uma maior diversidade de símbolos, irá haver uma menor correlação, resultando numa maior discrepância entre os índices, sendo mais vantajoso quando usado depois de ser aplicado um algoritmo tal como o BWT, pois nesse contexto os símbolos estarão previamente mais agrupados, resultando numa maior correlação, o que irá ajudar a diminuir o valor dos seus índices.

Iteration	Sequence	List
bananaaa	1	(abcdefghijklmnopqrstuvwxyz)
bananaaa	1,1	(bacdefghijklmnopqrstuvwxyz)
bananaaa	1,1,13	(abcdefghijklmnopqrstuvwxyz)
bananaaa	1,1,13,1	(nabcdefghijklmopqrstuvwxyz)
bananaaa	1,1,13,1,1	(anbcdefghijklmopqrstuvwxyz)
bananaaa	1,1,13,1,1,1	(nabcdefghijklmopqrstuvwxyz)
bananaaa	1,1,13,1,1,1,0	(anbcdefghijklmopqrstuvwxyz)
bananaaa	1,1,13,1,1,1,0,0	(anbcdefghijklmopqrstuvwxyz)
Final	1,1,13,1,1,1,0,0	(anbcdefghijklmopqrstuvwxyz)

B. Burrows-Wheeler Transform (BWT)

Através de um bloco de símbolos de comprimento *n*, o método constrói uma matriz *n* x *n,* onde cada linha deriva da anterior, movendo o primeiro símbolo da linha anterior para a sua última posição (*n* - 1). O processo pára quando o primeiro símbolo do input se encontra no final após rodar a matriz. Reordenamos as linhas da matriz por ordem lexicográfica e devolvemos como output a última coluna e o índice da linha onde ocorre a sequência de símbolos que queríamos inicialmente codificar. Fazendo este procedimento, conseguimos obter uma sequência de símbolos com regiões onde se repetem com mais frequência certos símbolos.

Vantagens: permite aumentar a redundância espacial, facilitando o uso posterior de outras técnicas de compressão como MTF e RLE; bom para textos e está a ser mais usado em imagens;

Desvantagens: a necessidade de reordenar constantemente a matriz poderá vir a aumentar a complexidade computacional para valores de *n* muito elevados; pode induzir latência.

Input	All Rotations	Sorting All Rows into Lex Order	Taking Last Column	Output Last Column
^BANANA|	^BANANA| |^BANANA A|^BANAN NA|^BANA ANA|^BAN NANA|^BA ANANA|^B BANANA|^	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA	BNN^AA|A

VIII. ALGORITMOS DE COMPRESSÃO

A. Algoritmos tradicionais

1) Huffman Encoding

O método usa probabilidades de ocorrência de cada símbolo e organiza-os por ordem decrescente de probabilidade, agrupando sempre os dois últimos símbolos menos prováveis num só nó ao qual é atribuído a soma das suas probabilidades. Esse processo é repetido até todos os

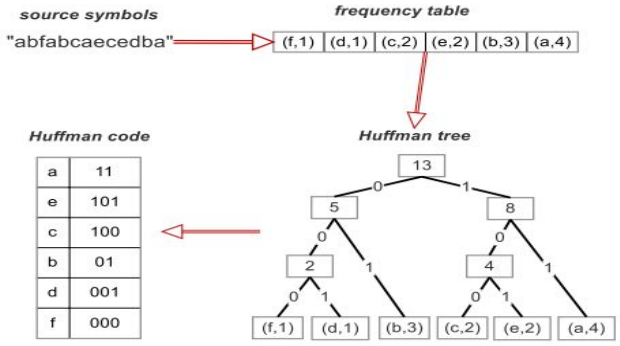
símbolos fazerem parte de um só nó, a raíz, obtendo-se assim uma árvore binária. Assim, a símbolos com maior frequência de ocorrência são atribuídos códigos binários menores e a símbolos com menor frequência de ocorrência são atribuídos códigos binários maiores.

A cada aresta vai corresponder um dígito binário 0 ou 1. Logo, para obter o código de um símbolo é necessário percorrer a árvore até atingir a folha da árvore do símbolo correspondente e ir anotando o dígito de cada aresta à medida que a árvore é percorrida.

Para este método ser utilizado, é necessário pressupor o conhecimento prévio da distribuição estatística da fonte, no entanto, se não for conhecido, é feita uma variação deste método denominada Adaptive Huffman Coding.

Vantagens: poupa espaço de memória; cada código é de prefixo único; é eficiente a comprimir ficheiros de texto; de todos os códigos unicamente decodificáveis (prefixo único) este é o melhor para codificar códigos sem memória, ou seja, cujos símbolos são independentes e identicamente distribuídos.

Desvantagens: algoritmo consome muita memória (lento) e é complexo, uma vez que precisa de construir o modelo estatístico e fazer a codificação; comprimento dos códigos binários são diferentes, tornando a descodificação mais problemática; outros algoritmos são melhores para codificar imagens.



2) Arithmetic

Utiliza esquemas estatísticos em que cada símbolo não necessita de ser codificado com um número inteiro de bits. Logo, é possível alcançar um bit rate fracional (em bits por símbolo).

A ideia é mapear a sequência de símbolos num só símbolo binário. Para tal, cada símbolo tem uma probabilidade associada (p_k) e um sub intervalo entre 0 e 1 com a forma $[L_{sk}, H_{sk})$ onde:

$$L_{sk} = \sum_{i=0}^{k-1} p_k = p_{k-1};\ 0 \le k \le (N-1)$$

$$H_{sk} = \sum_{i=0}^{k} p_k = p_k;\ 0 \le k \le (N-1)$$

A TAG = $[L_c, H_c)$ corresponde ao intervalo da sequência de símbolos que ocorreram até ao momento. No início, $L_c = 0$ e $H_c = 1$. Depois, é calculado o algoritmo da seguinte maneira até os símbolos terem sido todos codificados:

(i) Calcula-se o tamanho do sub intervalo, $tamanho = H_c - L_c$.

(ii) Vai-se buscar o símbolo seguinte.

(iii) Atualiza-se a TAG na forma $[L_c = L_c + tamanho * L_{sk},\ H_c = L_c + tamanho * H_{sk})$.

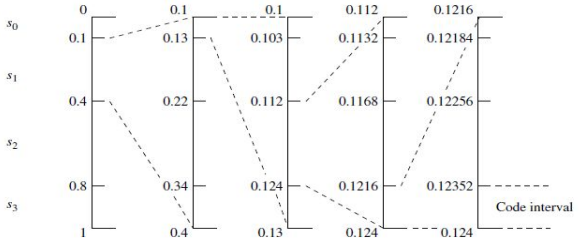
No fim, ter-se-á uma TAG atualizada onde o output será dado por TAG = $(H_c - L_c)/2$.

Também existe o Adaptive Arithmetic Coding que, por ser adaptativa, as frequências dos símbolos vão alterando à medida que é percorrido o input.

Vantagens: tem um grau de compressão maior do que huffman; não necessita de agrupar os símbolos em símbolos compostos.

Desvantagens: precisa de software e hardware avançado; é lento computacionalmente porque tem de fazer cálculos pesados como multiplicações e divisões; é de difícil implementação; não produz códigos com prefixo; pode gerar erros durante a transmissão.

Iteration # I	Encoded symbol s_k	Code subinterval $[L_c, H_c)$
1	s_1	[0.1, 0.4)
2	s_0	[0.1, 0.13)
3	s_2	[0.112, 0.124)
4	s_3	[0.1216, 0.124)
5	s_3	[0.12352, 0.124)

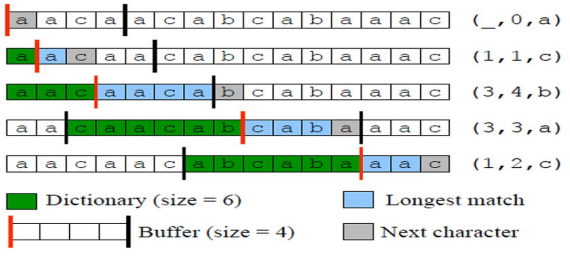


3) LZ77

O método usa o esquema baseado em dicionários e, para tal, utiliza dois buffers. O Search Buffer (SB) serve como dicionário e contém os símbolos da fonte, enquanto o Look-Ahead Buffer (LAB) serve como janela de pesquisa e procura no SB sequências de símbolos a serem codificados. Se um símbolo já existir, é substituído no dicionário pela posição da sua última ocorrência. A codificação é feita da forma {offset, length of match, next symbol}, sendo *offset* o número de posições que temos de recuar para encontrar o início da *match*, *length of match* o número de símbolos a serem copiados correspondentes à *match* e *next symbol* o próximo símbolo do LAB.

Vantagens: não precisa da construção explícita de um dicionário.

Desvantagens: o padrão tem que existir na search window podendo desperdiçar códigos maiores que não estão na mesma; o dicionário pode ter um comprimento muito grande se os dados de entrada forem muito pequenos, neste caso, pode expandir em vez de comprimir os dados.

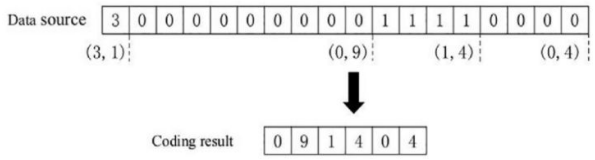


4) Run Length Encoding (RLE)

O método transforma sequências longas de caracteres repetidos num só com a contagem das vezes que se repetem. É vantajoso quando há sequências de 4 ou mais elementos iguais seguidos.

Vantagens: pouca capacidade computacional; simples; ideal para imagens monocromáticas.

Desvantagens: inútil para ficheiros que não contenham muita data repetida pois pode aumentar o tamanho.

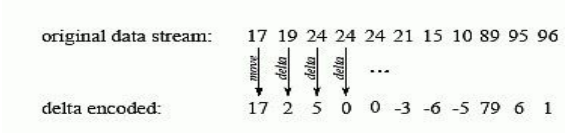


5) Delta Encoding

O método usa uma técnica que calcula a diferença entre o valor de dois símbolos seguidos de uma data e retorna essa diferença.

Vantagens: eficaz se a diferença entre os valores for baixa; torna-se ainda mais eficaz usado juntamente com Huffman e/ou RLE.

Desvantagens: é necessário fazer sempre uma dupla leitura: a do símbolo atual e a do símbolo anterior; se a diferença entre os símbolos não mudar ou for uma mudança proporcional o codec torna-se ineficiente.



B. Algoritmos modernos

1) Deflate

O método é uma junção dos algoritmos Códigos de Huffman e LZ77. Primeiro o input é dividido em vários blocos e é aplicado o método do LZ77. Uma vez os tuplos gerados, aplicamos o método de Huffman.

Vantagens: bom rácio de compressão.

Desvantagens: não comprime eficientemente ficheiros com pouca data repetida.

2) PNG - Portable Network Graphics

Atualmente, é o formato de imagens raster mais utilizado.

É um algoritmo de compressão que tem duas fases: *prediction* e depois compressão. A primeira fase faz uso da codificação Delta que não filtra a informação por pixels mas sim por canal. Na segunda fase, é usado o método de compressão Deflate mencionado anteriormente.

Vantagens: não é patenteado; é dos poucos formatos que permitem ter imagens com transparência; maior gama de profundidade de cores; alta compressão de imagem.

Desvantagens: não é bom para imagens com muitas cores.

3) Prediction by Partial Matching (PPM)

O método codifica cada símbolo baseando-se num contexto de comprimento *n*, que corresponde aos *n* símbolos que antecedem o atual. Através deste contexto, tenta-se encontrar símbolos repetidos com o mesmo contexto. Caso não sejam encontrados, diminui-se o contexto. Durante este processo de busca, cada símbolo e o respetivo contexto possuem um *counter* e um *cum counter* que irão ser usados para calcular o próximo bit a ser enviado no output.

Vantagens: rácio de compressão bastante favorável em comparação com outros algoritmos de compressão (Deflate, LZMA, Bzip2).

Desvantagens: devido à sua complexidade, possui velocidades de compressão/descompressão demorosas.

4) Context-based Adaptive Lossless Image Code (CALIC)

Este método é muito usado para codificar e descodificar informação presente em imagens de tons contínuo. Para tal, o CALIC analisa os pixels na vizinhança do pixel atual observando os valores duas linhas anteriores e das duas linhas que o precedem através de um *buffer*. Uma característica particular deste esquema é que este opera em dois modos, binário e de tom contínuo. Isto permite identificar localmente a forma como serão tratados os valores, sendo que se for constatado a presença de, no máximo, dois valores distintos, o modo binário será automaticamente ativado pelo CALIC.

Vantagens: permite tratar os dados de uma forma mais específica, sem ser de um modo global, o que traduz numa melhor compressão dos dados.

Desvantagens: velocidade de compressão reduzida devido à complexidade algorítmica.

5) JBIG2 Standard

Este método é usado para comprimir imagens binárias e pode ser aplicado tanto para compressão lossless como lossy. Para comprimir a imagem, é feita uma divisão da fonte em três regiões: 1) regiões de texto, 2) regiões com imagem de meio tom e 3) regiões genéricas (que não correspondem a nenhuma das referidas anteriormente). A cada uma destas regiões serão posteriormente aplicados diferentes algoritmos de compressão, dependendo seu tipo.

Vantagens: possui rácios de compressão até 3 vezes mais elevados que restantes *standards* de compressão; sendo excelente para codificar especificamente imagens binárias, parece ser ideal para as imagens que iremos testar.

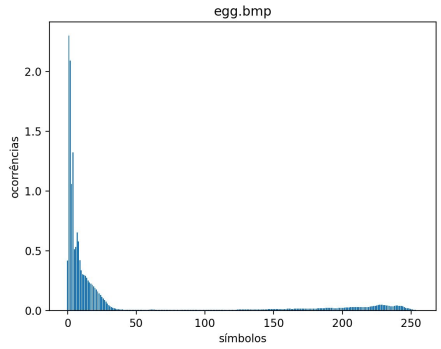
Desvantagens: quando usado para fazer compressão lossy, poderá alterar bastante a informação original, não preservando as características qualitativas da fonte.

IX. ANÁLISE DAS IMAGENS

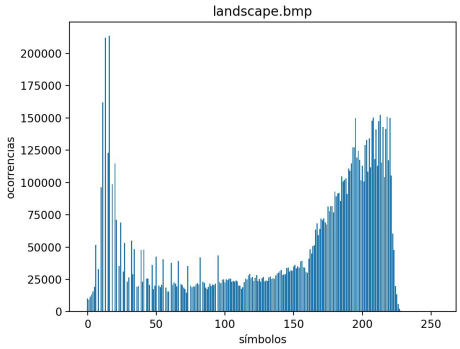
Passando para o lado prático deste trabalho, na tentativa de encontrar um algoritmo adequado, fizemos primeiramente uma análise das quatro imagens.

1) *egg.bmp* - é possível observar que a imagem possui um fundo preto que irá resultar num pico de valores entre 0 e 50, com maior incidência em 0. Este fundo não é totalmente contínuo, apresentando duas zonas ligeiramente mais claras: uma mancha acima dos ovos à direita e o reflexo dos ovos sobre

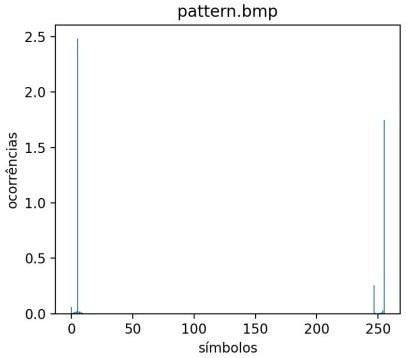
a superfície. Analisando os ovos, ao contrário do fundo, observamos uma superfície com um certo "grão", havendo uma grande irregularidade de valores que irá resultar na existência de valores entre 100 e 250.



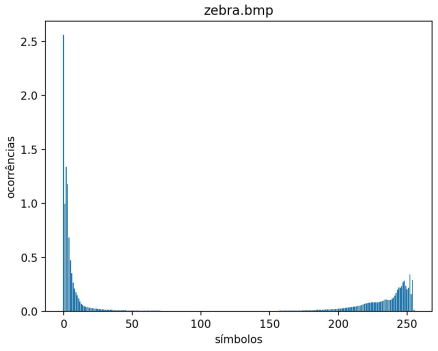
2) *landscape.bmp* - é a imagem que possui uma maior variância de valores. O nevoeiro que cobre as árvores confere à imagem uma espécie de degradê, o que irá resultar numa grande quantidade de valores distintos entre os pixels para produzir este efeito.



3) *pattern.bmp* - a imagem aparenta ser uma imagem binária, no entanto, fazendo um histograma, observamos que esta na realidade possui mais valores concentrados em 255 (branco) e em 5 (preto), sendo a imagem que confere uma menor variação de valores.



4) *zebra.bmp* - existe um grande contraste entre as riscas pretas e brancas, não havendo muitos tons intermédios. No entanto, consegue-se ainda assim reparar em algumas irregularidades no pêlo da zebra e de algumas manchas, não havendo uma persistência assim tão consistente de 2 únicos valores. Assim, a imagem irá possuir dois picos entre os valores 0 e 50, que corresponde às riscas pretas, e 200 e 250, que corresponde às riscas brancas.



X. ESCOLHA DOS ALGORITMOS

Analisadas as imagens, passamos à escolha dos algoritmos. Como refere o *abstract*, o objetivo não é achar o melhor codec para cada imagem em específico, mas sim aquele que aparenta ser um bom codec para diversas imagens, tendo em conta as quatro imagens analisadas.. É natural que, como cada imagem possui características distintas, certos codecs e formas de path scanning irão funcionar melhor numas do que outras, não sendo possível encontrar o codec que funciona igualmente melhor para todas as imagens.

Os codecs escolhidos para a compressão das nossas imagens são:

1. BWT -> MTF -> RLE
2. BWT -> MTF -> RLE -> Deflate
3. BWT -> MTF -> RLE -> Huffman
4. BWT -> MTF -> RLE -> PPM
5. Deflate
6. Delta -> Huffman
7. Delta -> RLE
8. Delta -> RLE -> Huffman
9. CALIC
10. PPM

Para a escolha dos nossos codecs, tivemos em consideração o processo de compressão lossless descrito no capítulo III. As suas implementações foram feitas na linguagem Java, exceto para o CALIC que possui uma particularidade. Este último tem o seu código fonte escrito em Python mas implementado em Java através do uso de Jython. Devido a essa forma de implementação, existem algumas particularidades quanto ao seu uso que estão descritas num README.txt na pasta do projeto.

Assim, numa fase inicial, contemplamos a transformação com uso das transformadas BWT e MTF.

Prevê-se que esta combinação de transformadas será bastante útil, pois ao aplicar-se em primeiro lugar BWT os símbolos idênticos irão ser mais agrupados, o que irá solidificar o desempenho do MTF, que funciona melhor quando há uma grande correlação entre os símbolos. Esta combinação poderá ser mais útil quando usada, por exemplo, na imagem landscape.bmp, pois como esta possui uma grande dispersão de valores, estas transformadas irão ajudar a agrupar os valores idênticos, aumentando a

redundância, que posteriormente será resolvida nos algoritmos de compressão seguintes.

Para realizar a segunda etapa, *data-to-symbol mapping*, fizemos uso do RLE, que se for usado depois de serem aplicadas as transformadas, irá ter em princípio um bom desempenho, pois tendo vários símbolos idênticos alinhados, irá fazer a substituição destes por apenas dois valores (o símbolo e as vezes que este se repete). No entanto, este poderá apresentar resultados menos bons quando usado numa fonte com maior dispersão de valores, podendo até mesmo aumentar a informação, contrariando o propósito da compressão. Assim, o RLE poderá funcionar melhor para imagens como o pattern.bmp devido às grandes sequências de valores de 5 e 255; e pior para imagens como a landscape.bmp, que possui uma grande diversidade de valores.

Por fim, para a etapa final *lossless symbol encoding* usamos os algoritmos Huffman, Delta encoding, Deflate, PPM e CALIC.

A. Combinação BWT -> MTF -> RLE (BMR)

A combinação foi feita com vista a estudar o comportamento de codecs mais tradicionais com transformadas para averiguar se é possível obterem-se valores favoráveis de compressão com algoritmos tradicionais reforçados com a transformação e o RLE.

B. Combinação Delta -> Huffman/RLE

Delta funciona melhor para imagens com valores mais próximos. Por um lado, como este resulta numa menor amplitude entre os valores, significa que a probabilidade de os valores estarem perto de 0 aumenta, sendo conveniente para o uso de Huffman. Por outro lado, se os valores não se alterarem entre si ou se diferenciarem de modo proporcional, o encoding irá resultar numa sequência de valores idênticos, sendo ideal para o uso de RLE. Teoricamente, estas combinações proporcionarão melhor compressão nas imagens egg.bmp, pattern.bmp e landscape.bmp.

C. Deflate, PPM e CALIC

Foi feita a combinação do Deflate após BMR e sozinho; do PPM após BMR e sozinho; e do CALIC sozinho, com o objetivo de averiguar se será efetivamente vantajoso combinar estes algoritmos com transformadas e com o RLE, visto que estes se caracterizam por serem algoritmos de uma grande complexidade e por já possuírem a combinação de outros algoritmos na sua arquitetura (Arithmetic encoding no PPM e LZ77 e Huffman no Delfate).

O Deflate revela tempos de compressão e de descompressão bastante positivos [2]. O PPM tem um desempenho muito positivo registado noutros trabalhos [2], apresentando um maior rácio de compressão comparado com o Deflate mas velocidades de compressão e de descompressão inferiores. O CALIC parece-nos bastante favorável pois possui dois modos, o de tom contínuo e o binário, sendo assim bastante útil para as imagens egg.bmp, pattern.bmp e zebra.bmp.

De todas estas combinações, aquelas que nos pareceram mais promissoras são BMR -> PPM (4) e Delta -> Huffman (6) para todas as imagens.

Assim, devido a todas as características das quais os outros codecs carecem, explicadas neste capítulo e no capítulo VIII, estes três codecs são os que irão obter eventuais melhores resultados.

XI. Análise e Discussão dos Resultados

Os seguintes resultados foram obtidos usando a linguagem Java e códigos de várias fontes. É de salientar que os valores de compressão incluem os ficheiros de texto necessários para a descompressão.

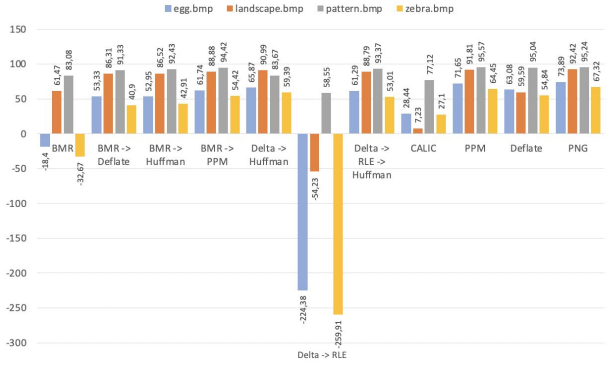
A. Entropia

		BMR	BMR-> Deflate	BMR-> Huffman	BMR-> PPM	Delta -> Huffman	Delta -> RLE	Delta -> RLE -> Huffman
Egg.bmp	Entropia Inicial	5,724	5,724	5,724	5,724	5,724	5,724	5,724
	Entropia Final	3,344	3,344	3,344	3,344	2,706	2,706	2,706
Landscape.bmp	Entropia Inicial	7,421	7,421	7,421	7,421	7,421	7,421	7,421
	Entropia Final	3,425	3,425	3,425	3,425	2,825	2,825	2,825
Pattern.bmp	Entropia Inicial	1,829	1,829	1,829	1,829	1,829	1,829	1,829
	Entropia Final	0,635	0,635	0,635	0,635	0,610	0,610	0,610
Zebra.bmp	Entropia Inicial	5,831	5,831	5,831	5,831	5,831	5,831	5,831
	Entropia Final	3,948	3,948	3,948	3,948	3,224	3,224	3,224

Podemos observar que a entropia inicial para a entropia final tem uma variação grande na imagem landscape.bmp comparativamente com a imagem pattern.bmp, cuja variação é a mais baixa.
A landscape.bmp possui imensa informação diferente e, por isso, uma vez as transformadas e codecs aplicados, a informação vai ser bastante mais redundante. Por outro lado, o pattern.bmp já possui bastante redundância, daí não ser possível haver uma grande variação. Essa lógica aplica-se também para as imagens egg.bmp e pattern.bmp.

B. Compressão (%)

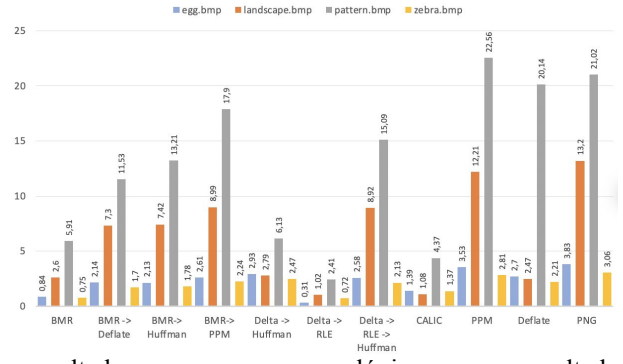
Analisemos, por ordem decrescente de eficiência, os algoritmos para cada imagem.



1) *egg.bmp* - PNG, PPM, Delta -> Huffman, Deflate.
2) *landscape.bmp* - PNG, PPM, Delta -> Huffman, Deflate.
3) *pattern.bmp* - PPM, PNG, Deflate, BMR -> PPM.
4) *zebra.bmp* - PNG, PPM, Delta -> Huffman, BMR -> PPM.

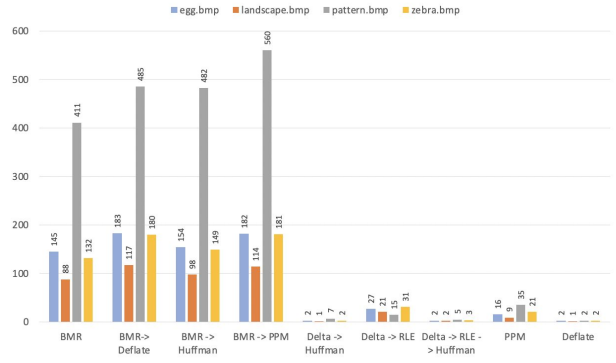
O PNG, codec de comparação para as nossas implementações, apresenta os melhores resultados, exceto para a imagem pattern.bmp, onde os resultados do PPM são ligeiramente mais favoráveis. Comparando com o PNG, o PPM é o formato que apresenta melhores resultados para todas as imagens.
Os resultados do pattern.bmp indicam que uma complexidade algorítmica menor (PPM e Deflate) resulta numa maior eficiência, devendo-se à simplicidade e pouca diversidade de informação da imagem. Já para as outras imagens, PPM e Delta -> Huffman são sempre os que apresentam melhores resultados, devido à forma como a informação está distribuída nas imagens e na forma como estes codecs atuam.
O gráfico também apresenta valores de compressão negativos nos codecs BMR e Delta -> RLE para as imagens, exceto o pattern.bmp. Isso deve-se ao facto do último codec aplicado ser o RLE, uma vez que, dependendo da complexidade e diversidade de informação da imagem, pode ou não gerar mais informação, o que se verifica nestes casos.
Relativamente aos restantes codecs, o CALIC apresenta os piores resultados para todas as imagens, enquanto os outros apresentam resultados razoáveis, sendo as combinações BMR e Delta -> Huffman/RLE boas escolhas como base para implementação de outros algoritmos.

C. Rácio de compressão



Os resultados seguem a mesma lógica que os resultados anteriores, mas é de salientar os elevados valores para o pattern.bmp. Devido à pouca variação de informação e simplicidade da imagem, trata-se de uma imagem muito fácil de comprimir e de fornecer resultados bastantes favoráveis.

D. Tempo de compressão



Analisando o gráfico, podemos verificar que, para qualquer imagem, os codecs com menor tempo de compressão são, por ordem crescente: Deflate, Delta -> Huffman, Delta -> RLE -> Huffman e PPM. No entanto, é importante salientar que estes tempos de compressão podem variar dependendo de vários fatores externos, tais como: o IDE onde o programa é corrido; a memória alocada para o mesmo; ou o número de operações a serem efetuadas no computador enquanto este corre o programa.
Como era de esperar, implementações mais complexas tais como BMR fornecem resultados maiores, demonstrando-se sobretudo na combinação BMR -> PPM, uma vez que se trata da junção de três algoritmos tradicionais com um algoritmo moderno, sendo a implementação mais complexa de todas.
Existe também um pico muito grande para a imagem pattern.bmp, uma vez que estamos a implementar o algoritmo mais complexo para a imagem com menor informação visual e maior tamanho, o que acaba por gerar esse tempo de compressão maior.

XII. Conclusão

Com base nos resultados da taxa de compressão e dos tempos de compressão, averiguamos que os codecs com melhor compressão são o PPM, Delta -> Huffman e Deflate, enquanto os codecs com melhor tempo de compressão são o Deflate, Delta -> Huffman e Delta -> RLE. Isto leva-nos a concluir que, apesar do PPM apresentar os melhores resultados e o Deflate ser o mais rápido, o Delta -> Huffman é codec mais equilibrado em termos de rácio e tempo de compressão.
Das duas hipóteses iniciais, verifica-se que, de facto, o codec Delta -> Huffman é um codec adaptado para diversas imagens que cumpre todos os requisitos essenciais para um bom codec, sendo eles a taxa de compressão, tempo de compressão e complexidade algorítmica.
Sugerimos então, para uma eventual futura implementação, esta combinação como base e ponto de partida de um algoritmo mais desenvolvido, devido à forma como harmoniza bem com outros codecs e devido às suas características favoráveis a uma implementação robusta e eficaz.
Para além disso, uma possível melhoria para o algoritmo RLE seria quando for encontrado um símbolo que se repete uma única vez, em vez de o substituir pelo par (vezes repetidas, símbolo), manter esse mesmo símbolo, não correndo o risco de aumentar o número de elementos presentes nas fontes.

XIII. Referências

[1] G. Apoorv, B. Aman, K. Vidhi, “Modern Lossless Compression Techniques: Review, Comparison and Analysis”, 2017.
[2] I. Shilpa, K. Deepak, “A Review on - Lossless Image Compression Techniques and Algorithms”, 2014.
[3] K. Lina J., **“**Lossless Image Compression”, Capítulo 16, pp. 383 - 419, 2009.
[4] C. Paulo, “Capítulo II – Teoria da Informação e Codificação Entrópica”, Edição 2020-2021
[5] S. Peng, BinLi, PhyuHninThike, D. Lianhong, “A knowledge-embedded lossless image compressing method for high-throughput corrosion experiment”, 2018.

[6] Department of Computer Science and Engineering <https://cse-robotics.engr.tamu.edu/dshell/cs314/sa6/sa6.html>
[7] S. Marco, encode_demo_with_table.py
[8] Wikipedia, <https://en.wi<kipedia.org/wiki/Move-to-front_transform>
[9] Wikipedia, <https://medium.com/algopods/burrows-wheeler-transform-c743a2c23e0a>
[10] Steven W. Smith, Ph.D. - "The Scientist and Engineer's Guide to Digital Signal Processing", capítulo 27 <https://www.dspguide.com/ch27/4.htm>
[11] Github, Deflate, <https://gist.github.com/Crydust/3e8d279c82d5cca26227>
[12] Github, PPM/Arithmetic, <https://github.com/nayuki/Reference-arithmetic-coding>
[13] Github, CALIC, <https://github.com/siddharths2710/CALIC>
[14] Github, Delta, HackerRank/DeltaEncoding.java at master · sebysr/HackerRank · GitHub
[15] Rosetta Code, BWT, https://rosettacode.org/wiki/Burrows%E2%80%93Wheeler_transform#Python
[16] Rosetta Code, RLE, https://rosettacode.org/wiki/Run-length_encoding
[17] Rosetta Code, MTF, https://rosettacode.org/wiki/Move-to-front_algorithm#Java
[18] Rosetta Code, Entropy, Entropy - Rosetta Code
[19] Techie Delight, Huffman, https://www.techiedelight.com/huffman-coding/
[20] Alphabet Java, Alphabet.java, Alphabet.java (princeton.edu)