Avaliação de algoritmos de compressão clássicos aplicados a imagens binárias

André Moleirinho, (61609)
Fábio Macarrão, (57098)
Vitor Costa, (70323)

Orientador: Prof. Miguel Barão

UNIVERSIDADE DE ÉVORA

Janeiro, 2026

Resumo

O presente documento consta de um estudo de desempenho a algoritmos para compressão de imagens binárias sem perdas. Os algoritmos desenvolvidos são submetidos a uma análise de variação de entropia antes e após transformações aplicadas a ficheiros no formato *portable bitmap* (.pbm).

Palavras-chave: algoritmos de compressão de dados clássicos, imagens binárias.

Introdução

No estudo da codificação e compressão de dados apresentado por Claude Shannon[1], o autor apresenta a métrica *entropia,* $H(x)$, como sendo uma medida de incerteza sobre uma variável aleatória, representando o quão pouco sabemos sobre um determinado acontecimento.

No mesmo artigo, C. Shannon expõe a conclusão sobre compressão de informação, no Teorema do código fonte: "O valor médio minimo de bits necessário para codificar símbolos de uma fonte é igual à sua entropia - H(símbolo)", qualquer codificação/compressão que resulte num valor inferior a este, apresenta perdas de informação, não se garantido a sua reversibilidade, ou descompressão.

Neste artigo comparamos a evolução de entropia em ficheiros de imagem binários submetidos a algoritmos de compressão, sem perdas, implementados em python. Começamos por descrever as caracteristicas de imagens convertidas para o formato *portable bitmap* pelo software *ImageMagik*, introduzimos os algoritmos clássicos implementados para comprimir e descomprimir de volta para a imagem original. Interpretamos o impacto de algumas transformações de dados no processo de compressão e terminamos com uma conclusão da performance dos algoritmos desenvolvidos.

A investigação presente neste documento foi construída com recurso à ferramenta de inteligência artificial Gemini para interpretação de resultados provenientes de algoritmos implementados em python e por nós também validados.

1 Sobre o formato portable bitmap

O formato *portable bitmap* (.pbm) foi desenvolvido por Jef Poskanser em 1980, e permitia a introdução de imagens mono-cromáticas em emails a circular pela internet.

Tirando partido do software de código aberto *ImageMagick* que implementa a conversão de imagens de vários formatos como *portable bitmap*, com o seguinte comando (em linux):

> **convert** <image.format> **-compress none** <image_name>.pbm

Conseguimos um ficheiro .pbm com a informação organizada da seguinte forma:

1º linha	Magic Number - resultado da conversão efectuada e atribuida pelo software, para pbm é geralmente P1;
2º linha	Largura e Altura - representa as dimensões originais da imagem;
r linhas	As restantes linhas são o resultado da conversão para o formato branco e preto, zeros e uns respectivamente.

Os algoritmos de compressão desenvolvidos vão atuar nas r linhas do ficheiro convertido, identificando padrões e oportunidades de compressão da informação.

Abordagem ao problema

De forma a simplificar o processo de implementação e validação dos algoritmos desenvolvidos, escolhemos uma imagem simples de tamanho relativamente pequeno para submeter aos processos de compressão e descompressão implementados.

A imagem *pixel_ character*, apresentada na figura 1, tem um tamanho de 48 por 48 pixeis, ao todo são 2304 pixeis, no formato original (.png) ocupa um total de 4665 *bytes*.

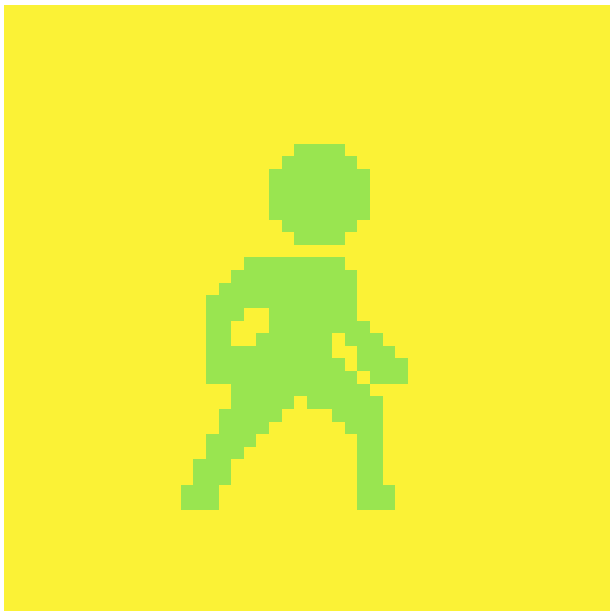


Figura 1: Ficheiro pixel_character.png a comprimir.

Ao aplicar a conversão para o formato *portable bitmap* descrita anteriormente, resulta um ficheiro com o valor 'P1' na primeira linha, '48 48' na segunda, e as restantes linhas tem um total de 2066 zeros e 238 uns, evidenciando um desequilibrio entre a quantidade de branco e preto na imagem convertida, ou, no alfabeto a ser submetido ao processo de compressão.

Os algoritmos de compressão implementados para este estudo atuam apenas nas '[r linhas]' do ficheiro .pbm, o que significa que o alfabeto a cumprimir tem apenas dois caracteres e pode ser definido por $X = \{0, 1\}, x \in X$. Podemos assim determinar a probabilidade de ocorrencia

para cada simbolo da imagem: $P(0) = 0.8967$ e $P(1) = 0.1033$. Com a informação anterior e de acordo com a fórmula da entropia apresentada por Shannon[1], podemos determinar o valor das entropias marginais e somar para cada simbolo do alfabeto a comprimir, os resultados são apresentados na tabela 1.

O cálculo da entropia condicional depende da forma como o contexto de dependencia (ou condição) é formulado. No algoritmo implementado decidimos definir o contexto de transições de caractéres de dimensão 2, resultando o dominio $C = \{00, 01, 10, 11\}, c \in C$. Para o ficheiro pixel_character.pbm o número de ocorrencias para cada par é: 0 = 2021, 01 = 44, 10 = 44, 11 = 194, o que nos permite determinar a probabilidade de ocorrencia de um caracter (zero ou um), dado que houve uma transição de caractéres, por exemplo P(1|01).

A tabela seguinte demonstra os resultados obtidos dos métodos de cálculo da entropia H(X) e entropia condicional H(X|C) implementados:

entropia	**bits**
H(0)	0.1411
H(1)	0.3383
H(X)	0.4794
H(X|C)	0.2047

Tabela 1: entropia e entropia condicional da imagem original.

Devemos notar que o valor da entropia condicional (H(X|C)) é menor do que o valor da entropia marginal H(1) e H(X). Segundo Shannon[1], $H(X|Y) < H(X)$, que se valida pelos nossos algoritmos. Já o valor alto de H(1), superior a H(0), é caracteristico de probabilidades de ocorrencia desequilibradas, neste caso, a imagem tem bem mais zeros do que uns, pelo que estariamos mais confiantes de que um caracter gerado por esta fonte seja '0' do que '1', ou estamos mais incertos de que será '1'.

A entropia será máxima nos casos em que temos o mesmo número de pixeis brancos e pretos. Para situações em que só temos dois casos possíveis o valor máximo da entropia é 1 bit, como demonstra a seguinte equação:

$$H(X) = -[0.5.log_2(0.5) + 0.5.log_2(0.5)] = 1 \text{ bit.}$$

Apresentamos de seguida os algoritmos de compressão implementados e avaliamos a sua taxa de compressão de acordo com a redução de tamanho do ficheiro e variação de entropia pós compressão.

2 Algoritmos de compressão

Decidimos implementar os seguintes algoritmos de codificação clássicos:

- Aritmética;
- Huffman;
- LZ78;

Os dois primeiros são algoritmos lentos que necessitam da probabilidade de ocorrência dos simbolos existentes para conseguir elaborar *códigos instantâneos*, enquanto o último consegue processar um ficheiro à medida que o interpreta.

Os *códigos instantâneos* tem de validar a desigualdade de Kraft, que nos permite escrever o comprimento médio de um código em função do comprimento das palavras do alfabeto e das respectivas probabilidades de ocorrencia.

O comprimento médio da operação de codificação é dado pela expressão:

$$C(L) = \sum p(x).l(x) = H(x).$$

2.1 Aritmética

Seguimos a abordagem estática na codificação aritmética, o que significa que as probabilidades de ocorrência dos símbolos 0 e 1 são calculadas no início e são utilizadas no processo de descompressão.

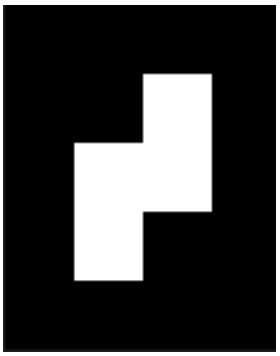
O output gerado pelo algoritmo de compressão é então um ficheiro txt com informação separada por espaços. Para a nossa imagem de exemplo, *pixel_ character.pbm*, obtemos o seguinte output:

2066 238 48 48 0.9999999999999996

Os valores significam: quantidade de zeros, quantidade de uns, largura da imagem, altura da imagem e o resultado da codificação aritmética respectivamente.

O valor extremamente alto resultado da codificação da imagem sugere que um dos dígitos do alfabeto dominou o processo de codificação e corrompeu o mesmo, inviabilizando a aplicação deste algoritmo à imagem. O Python 3.13.19 utiliza variáveis de ponto flutuante com uma precisão de 16 casas decimais, o que limita então a precisão do algoritmo, talvez após a aplicação de uma das transformações, descritas mais a diante neste documento, torne a compressão possível com o algoritmo desenvolvido.

Por agora demonstramos que o algoritmo funciona para ficheiros mais pequenos como a figura seguinte:



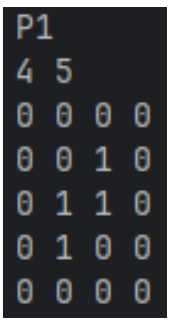


Figura 2: Exemplo com peça de tetris. Ficheiro *tetris.pbm* antes de compressão (esquerda) e após descompressão (direita).

A entropia do ficheiro *tetris.pbm*,de tamanho 57 bytes, original é:

$$H(X) = -[0.2.log_2(0.2) + 0.8.log_2(0.8)] = 0.72193 \text{ bits} .$$

O algoritmo de codificação aritmética implementado gera o seguinte output para a imagem da esquerda na figura 2:

16 4 4 5 0.7573628325145032

O ficheiro *txt* com a informação resultante do algoritmo de compressão tem 27 bytes, apresenta uma taxa de compressão de 45%.

O método de descodificação implementado determina as probabilidades de ocorrência dos símbolos com os primeiros dois valores do ficheiro, constrói uma tabela de probabilidades acomulada com o símbolo mais comum no topo e reconstroi a stream de dígitos 0 e 1, com os valores da altura e comprimento da imagem para atingir o formato *pbm* da imagem original convertida com o software ImageMagick.

2.2 Huffman

Na aplicação do algoritmo de Huffman todos os símbolos são folhas de uma árvore, que vão sendo adicionados dos menos para os mais prováveis, cujas palavras são formadas pelo caminho da raíz até à folha. Segundo a teoria, o comprimento médio do código de huffman será um valor no intervalo:

$$H(X) < L(C) < H(X) + 1 \text{ bits.}$$

As desvantagens são que tanto o compressor como o descompressor precisam de conhecer a árvore, que as probabilidades iniciais não podem ser editadas e que é assumido que os símbolos são independentes.

Nos métodos implementados em python, ao utilizar o comando:

$$> \textbf{python huffman c 'pixel\_character.pbm'},$$

é criado um ficheiro comprimido chamado 'huffman_output.bin', 15 vezes mais pequeno que o original ($4665/306 = 15,25$).

Quando aplicado ao exemplo da figura 2 obtemos um ficheiro com 25 bytes de tamanho, revelando o poder deste método de compressão.

O mesmo script pode ser utilizado para descomprimir o comando com o argumento 'd':

$$> \textbf{python huffman d 'pixel\_character.pbm.bin'},$$

É criado um novo ficheiro 'huffman_output.pbm' igual ao covertido anteriormente. Como huffman não tem perdas, a imagem gerada é igual à imagem/ficheiro original:

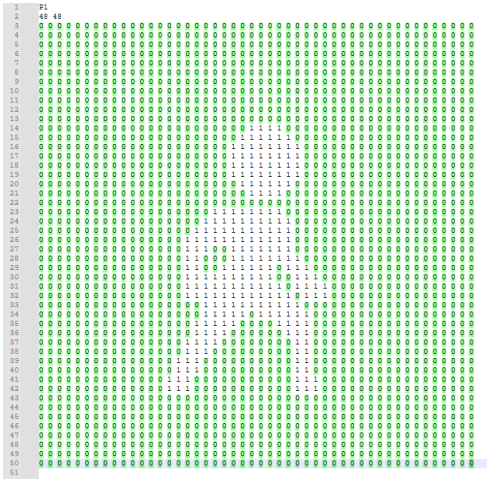


Figura 3: Ficheiro resultado da descompressão huffman, *pixel_character.pbm*.

2.3 LZ78

O LZ78 é um algoritmo de compressão sem perdas que constrói um dicionário dinâmico de "frases"(sequências de símbolos) à medida que lê os dados, cada nova sequência encontrada é guardada no dicionário com um índice único.

A saída do compressor consiste em pares (índice, símbolo), onde o índice refere o prefixo já conhecido e o símbolo indica o novo dado que completa a frase. Este método é particularmente eficaz em imagens *pbm*, pois substitui padrões repetitivos de pixels por índices numéricos curtos, maximizando a eficiência quando aplicado após transformações como o Run-Length Encoding sugeridas pelo professor.

Transformações

Com a informação anterior em mente foram abordadas 4 maneiras diferentes de comprimir o ficheiro:

- apenas codificar o ficheiro diretamente com LZ78;
- **Transformação A:** aplicar uma transformação do alfabeto da imagem em que os símbolos são blocos 2x2 antes da codificação;
- **Transformação B:** aplicar um XOR entre cada pixel original e o pixel original na posição acima antes da codificação;
- **Transformação C:** aplicar o algoritmo Run-Length Encoding antes da codificação.

Para todas as abordagens os ficheiros comprimidos foram guardados de 2 maneiras diferentes, uma com texto, sendo mais human-readable, e os ficheiros binários sendo uma compressão mais "a sério".

- **[A] - Blocos 2x2:** Esta transformação captura a correlação espacial entre os pixels vizinhos o que permite ao LZ78 encontrar padrões mais depressa;
- **[B] - XOR:** A transformação XOR limpou as redundâncias verticais o que fez aparecerem muitos 0s ou 1s seguidos, criando um cenário onde o LZ78 brilha na compressão;
- **[C] - Run-Length Encoding:** Com RLE houve uma drástica redução de símbolos, o que facilitou ao LZ78 a compressão, além de que o LZ78 passou a comprimir padrões de comprimento e a representar centenas que pixeis com menos bytes;

Apresentamos de seguida os resultados obtidos para as transformações aplicadas ao ficheiro pbm antes da operação de compressão:

Transformação *opção de compressão*	**LZ78 apenas**		**[A]**		**[B]**		**[C]**	
	Texto	**Binária**	**Texto**	**Binária**	**Texto**	**Binária**	**Texto**	**Binária**
Tamanho Comprimido (bytes)	1447	401	950	223	1201	338	491	170
Redução (%)	68.98	91.40	91.40	95.22	74.26	92.75	89.47	96.36
Rácio (bits/pixel)	5.0243	1.3924	3.2986	0.7743	4.1701	1.1736	1.7049	0.5903

Tabela 2: Resultados obtidos com algoritmo de compressão LZ78 aplicado ao ficheiro *pixel_ character.pbm.*

Como é possível observar em todos os casos a compressão com texto é pior que a compressão binária, pois a compressão com texto necessita de ocupar 1 byte com a codificação ASCII de cada caractere, logo apesar de ter menos caracteres, o facto de ser uma string e não binário faz com que a compressão em texto ocupe mais espaço. Além disso, entre as 4 abordagens a que obteve piores resultados foi a codificação sem transformações e o melhor resultado foi aplicando

o RLE. Ao longo das imagens destaca-se que a compressão vai melhorando, concluindo que a aplicação das transformações ajuda a obter uma maior compressão com a codificação LZ78.

Impacto das transformações na entropia do ficheiro

De seguida mostramos o impacto de cada transformação na entropia do ficheiro. A reorganização das sequencias de zeros e uns cria diferenças nas probabilidades de cada símbolo.

transformação	H(X)	H(X|C)
A	0.00000	0.17952
B	0.20062	0.17191
C	4.40	1.79

Tabela 3: Entropia H(X) e entropia condicional H(X|C) por transformação aplicada ao ficheiro *pixel_ character.pbm*.

O valor 0.0 de entropia resultado da transformação A indica que há uma probabilidade desequilibrada dos simbolos para esta imagem (o valor não é realmente 0, note-se o valor da entropia condicional), o alfabeto transformado pela construção de blocos 2x2 é dominado por um dos simbolos.

As variações de entropia com a introdução das transformações revelam a redundância na ordem dos números. Em C, grande diferença entre H(X) e H(C|C) mostra que ainda existe muita redundância "escondida"na ordem dos números, a aplicação do rle irá reduzir isso mesmo. O valor de entropia é superior a 1 neste caso porque o alfabeto deixa de ser $X = \{0, 1\}$, passa a ser constituido pelos simbolos do dicionário.

Impacto das transformações na compressão aritmética

Aplicá-mos as transformações A e B ao ficheiro antes da compressão aritmética com o objectivo de obter um resultado possível de codificar pelo nosso limitado algoritmo.

Foram obtidos os seguintes valores de número a codificar e comprimento médio do código, L(image):

transformação	resultado	L(image) bits
solo	0.9999999999999996	1073.0
A	0.9999994317311537	87.0
B	0.9999999999999996	588.0

Tabela 4: Impacto de transformações A e B na codificação aritmética.

Com a tabela anterior confirmamos que a aplicação da transformação [A], por blocos 2x2, reduz bastante o número de pixeis da imagem, permitindo ao algoritmo de codificação aritmética implementado encontrar uma melhor definição do valor de virgula flutuante. No entanto, não validamos a descompressão e reconstrução da imagem, mas acreditamos que o valor resultado da transformação A do nosso algoritmo ainda não tem a definição suficiente para o processo de reversibilidade.

Conclusão

O presente estudo permitiu avaliar e comparar o desempenho de algoritmos clássicos de compressão de dados — Aritmética, Huffman e LZ78 — aplicados a imagens binárias no formato

PBM. A análise baseou-se não apenas nas taxas de compressão finais, mas também na variação da entropia, validando experimentalmente os teoremas fundamentais da Teoria da Informação de Shannon.

Os resultados obtidos demonstram que a eficácia da compressão está intrinsecamente ligada ao pré-processamento dos dados. Enquanto a aplicação direta dos algoritmos obteve resultados modestos, a introdução de transformações revelou-se decisiva. Destaca-se a combinação do algoritmo LZ78 com a transformação Run-Length Encoding (RLE) e codificação binária, que atingiu a melhor performance global, com uma redução de tamanho de ficheiro de 96.36% e um rácio de 0.5903 bits/pixel. Este resultado corrobora a análise de entropia da Transformação C, que indicava uma redução drástica na redundância espacial da imagem através da alteração do alfabeto de símbolos.

Em suma, o trabalho evidencia que não existe um "melhor algoritmo"universal, mas sim uma melhor estratégia combinatória. Para imagens binárias com grandes áreas contíguas de cor (como a analisada), a estratégia de dicionário (LZ78) auxiliada por RLE supera as abordagens puramente estatísticas (Huffman/Aritmética), explorando de forma ótima a baixa entropia condicional da fonte.

Referências

[1] Shannon C. E. (1948). *A Mathematical Theory of Communication*, The Bell System Technical Journal.