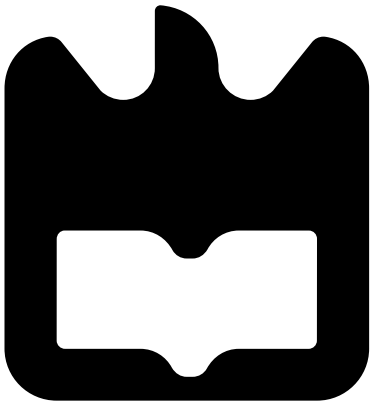


AED: 1º Trabalho

Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática

Catarina Rabaça (Nº 119582)
Francisco Ribeiro (Nº 118993)

2 de Dezembro de 2024

Conteúdo

1 Aspetos Gerais

O presente trabalho tem como objetivo o desenvolvimento e análise do TAD (Tipo Abstrato de Dados) `imageBW`, que manipula imagens binárias (preto e branco). As imagens são comprimidas utilizando a técnica *Run-Length Encoding* (RLE), para otimização do armazenamento, que consiste em substituir sequências repetidas de um mesmo valor por um único valor que é a contagem dessas repetições.

Este relatório abrange uma análise detalhada das funções `ImageCreateChessBoard()` e `ImageAND()`, incluindo dados experimentais, análise de memória alocada e uma análise formal e comparativa entre os algoritmos básico e melhorado da função `ImageAND()`.

1.1 Funções Desenvolvidas

As funções implementadas neste trabalho são:

- **ImageCreateChessBoard:** Cria uma imagem de um tabuleiro de xadrez.
- **ImageIsEqual:** Verifica se duas imagens são iguais.
- **ImageAND:** Faz um AND bit a bit entre os píxeis de duas imagens (que devem ter as mesmas dimensões).
- **ImageOR:** Faz um OR bit a bit entre os píxeis de duas imagens (que devem ter as mesmas dimensões).
- **ImageXOR:** Faz um XOR bit a bit entre os píxeis de duas imagens (que devem ter as mesmas dimensões).
- **ImageHorizontalMirror:** Espelha a imagem segundo um eixo horizontal.
- **ImageVerticalMirror:** Espelha a imagem segundo um eixo vertical.
- **ImageReplicateAtBottom:** Acopla uma imagem por baixo de outra.
- **ImageReplicateAtRight:** Acopla uma imagem ao lado direito de outra.

Tudo isto foi desenvolvido utilizando a linguagem C, estando o código fonte de todas as funções no ficheiro `imageBW.c` e os devidos testes no ficheiro `imageBWTest.c`. As funções foram implementadas e testadas, seguindo as pré-condições e pós-condições descritas no ficheiro `imageBW.h`. Os testes verificaram a correta manipulação de imagens, garantindo que o formato binário comprimido funcione conforme esperado.

2 Análise da função `ImageCreateChessboard`

2.1 Memória Alocada

A memória alocada em função do número de linhas , do número de colunas e do lado dos quadrados é dada por:

$$\text{header} + m\left(\frac{n}{s} + 2\right)$$

Onde:

- **header** - Representa a memória alocada com a função `AllocateImageHeader()`
- **m** - Número de linhas
- **n** - Número de colunas
- **s** - Lado do quadrado (`Square_edge`)

Nos testes realizados, duas variáveis serão mantidas fixas para analisar a memória total da imagem em função da outra variável.

2.2 Resultados Experimentais

Foram realizados testes para gerar padrões de xadrez com a mesma altura (`m`) e largura (`n`) (40×40) e fez se variar o lado (`s`) dos quadrados para observar isoladamente a sua influência na memória alocada. A tabela abaixo resume os resultados:

SquareEdge (s)	Memory (bytes)
1	7056
2	3856
4	2256
5	1936
8	1456
10	1296
20	976
40	816

Tabela 2.1: Memória alocada para diferentes valores do lado (s) dos quadrados.

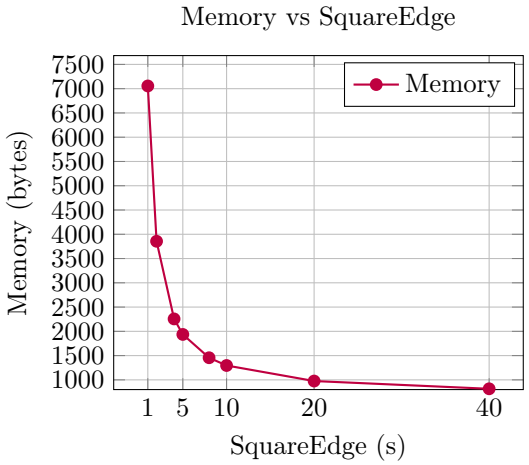


Figura 2.1: Memória alocada em função do lado (s) dos quadrados.

Foram realizados testes para gerar padrões de xadrez com a mesma altura (`m`) e lado (`s`) dos quadrados (40×5) e fez-se variar a largura (`n`) da imagem para observar isoladamente a sua influência na memória alocada. A tabela abaixo resume os resultados:

Width	Memory (bytes)
5	816
15	1136
25	1456
35	1776
45	2096
55	2416
65	2736
75	3056
85	3376
95	3696

Tabela 2.2: Memória alocada para diferentes valores da largura (n) da imagem.

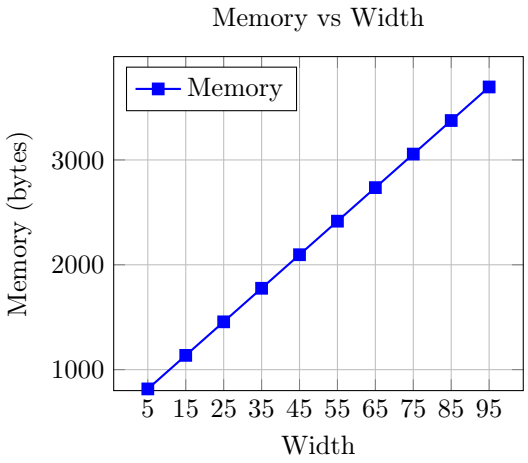


Figura 2.2: Memória alocada em função da largura (n) da imagem.

Foram realizados testes para gerar padrões de xadrez com a mesma largura (`n`) e lado (`s`) dos quadrados (40×5) e fez se variar a altura (`m`) da imagem para observar isoladamente a sua influência na memória alocada. A tabela abaixo resume os resultados:

Height	Memory (bytes)
5	256
15	736
25	1216
35	1696
45	2176
55	2656
65	3136
75	3616
85	4096
95	4576

Tabela 2.3: Memória alocada para diferentes valores da altura (m) da imagem.

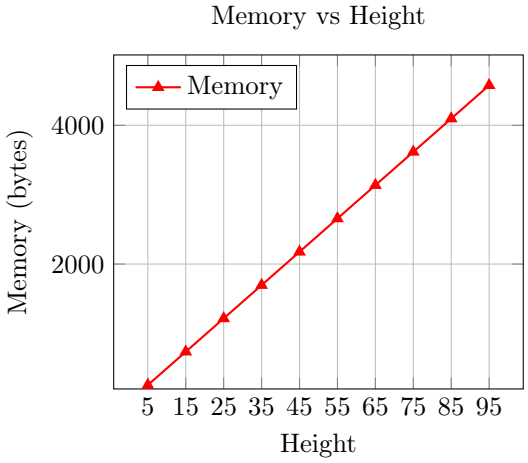


Figura 2.3: Memória alocada em função da altura (m) da imagem.

2.3 Análise

Para procedermos à análise dos dados experimentais, podemos começar inicialmente por obter as expressões que melhor representam os gráficos apresentados e, a partir daí, determinar a ordem de complexidade $\mathcal{O}(\cdot)$ de cada um deles.

Para o gráfico da memória alocada em função do lado (s) dos quadrados, a função que mais se adequa é a seguinte:

$$M(s) = \frac{6400}{s} + 656$$

Logo podemos concluir que a memória alocada tem uma relação inversamente proporcional ao lado (s)

dos quadrados.

Para o gráfico da memória alocada em função da largura (n) da imagem, a função que mais se adequa é a seguinte:

$$M(n) = 32 \cdot n + 656$$

Logo, podemos concluir que a memória alocada tem uma relação diretamente proporcional à largura (n) da imagem, resultando numa ordem de complexidade de $\mathcal{O}(n)$. Para o gráfico da memória alocada em função da altura (m) da imagem, a função que mais se adequa é a seguinte:

$$M(n) = 48 \cdot n + 16$$

Logo, podemos concluir que a memória alocada tem uma relação diretamente proporcional à altura (m) da imagem, resultando numa ordem de complexidade de $\mathcal{O}(n)$.

Em síntese, a relação inversamente proporcional entre a memória alocada e o lado (s) dos quadrados da imagem deve-se ao formato *Run-Length Encoding (RLE)*, que reduz a memória necessária ao aumentar as áreas homogêneas. Em contrapartida, o crescimento linear da memória em função da largura (n) e da altura (m) da imagem reflete a necessidade de armazenar mais dados com o aumento dessas dimensões, resultando em uma complexidade de $\mathcal{O}(n)$ para ambas.

Pior caso

O pior caso ocorre quando a cor dos quadrados alterna constantemente. Neste cenário, a função precisa de processar cada píxel individualmente, sem conseguir aplicar uma compressão significativa. Esse comportamento resulta numa maior alocação de memória, uma vez que o *Run-Length Encoding (RLE)* não consegue agrupar eficientemente os píxeis repetidos. Como consequência, a memória alocada é maximizada, e a complexidade temporal é elevada.

Melhor caso

O melhor caso acontece quando o lado do quadrado (s) é igual à largura da imagem (width). Neste caso, a imagem apresenta uma distribuição homogénea, em que todos os píxeis são iguais. Assim, o *Run-Length Encoding (RLE)* consegue comprimir a imagem de forma ideal, resultando numa alocação mínima de memória.

Caso médio

O caso médio refere-se a uma situação em que o lado (s) dos quadrados da imagem está entre o pior caso e o melhor caso, ou seja, o número de runs seria algo a meio entre 1 e *width*.

Aqui, a compressão da imagem será menos eficiente que o melhor caso, mas mais eficiente que o pior caso, resultando numa alocação de memória e complexidade temporal intermediárias entre eles.

3 Análise da função `ImageAND`

3.1 Descrição

A função `ImageAND` foi implementada com sucesso, utilizando duas abordagens metodologicamente distintas para realizar a operação lógica `AND` entre imagens binárias representadas no formato `RLE` (Run-Length Encoding).

A primeira abordagem recorre às funções auxiliares `UncompressRow` e `CompressRow`. Essa estratégia permite que a operação lógica seja aplicada diretamente sobre os dados descomprimidos, proporcionando clareza e simplicidade na manipulação das informações. Contudo, tal método implica um custo computacional adicional significativo, devido às etapas de descompressão e subsequente recompressão das linhas.

A segunda abordagem opera diretamente sobre as linhas comprimidas em RLE, evitando a descompressão total. O algoritmo compara os runs das duas imagens iterativamente, identificando o menor comprimento entre os runs atuais e realizando a operação AND para essa quantidade de pixels. Após o processamento, o índice do run menor é incrementado, enquanto o maior permanece até ser completamente processado. O processo continua até que todos os runs sejam analisados.

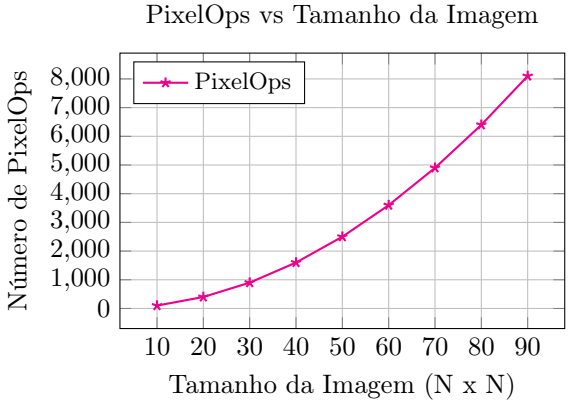
3.2 Resultados Experimentais

Para analisar a eficiência das duas abordagens, foram realizados testes sobre o número de operações de `AND` entre píxeis (`PixelOps`), para diferentes tamanhos de imagens.

3.2.1 Primeira abordagem: Função `ImageAND` Não Otimizada

Tamanho da Imagem	PixelOps
10 x 10	100
20 x 20	400
30 x 30	900
40 x 40	1600
50 x 50	2500
60 x 60	3600
70 x 70	4900
80 x 80	6400
90 x 90	8100

Tabela 3.1: Função `ImageAND` Não Otimizada.

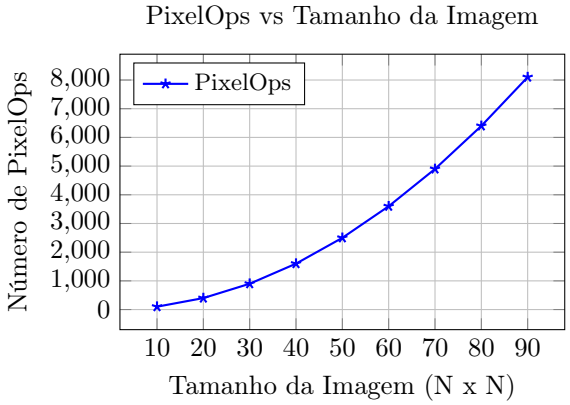


3.2.2 Segunda abordagem: Função `ImageAND` Otimizada

Pior Caso

Tamanho da Imagem	PixelOps
10 x 10	100
20 x 20	400
30 x 30	900
40 x 40	1600
50 x 50	2500
60 x 60	3600
70 x 70	4900
80 x 80	6400
90 x 90	8100

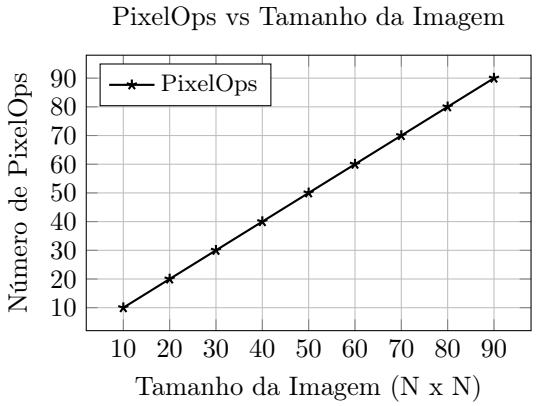
Tabela 3.2: Função `ImageAND` - Pior caso.



Melhor Caso

Tamanho da Imagem	PixelOps
10 x 10	10
20 x 20	20
30 x 30	30
40 x 40	40
50 x 50	50
60 x 60	60
70 x 70	70
80 x 80	80
90 x 90	90

Tabela 3.3: Função `ImageAND` - Melhor caso.



3.3 Análise Formal

Na primeira abordagem, o facto de termos que realizar a operação `AND` bit a bit entre as imagens implica que, para uma imagem com tamanho $n \times n$, a ordem de complexidade seja:

$$\mathcal{O}(n^2)$$

Este comportamento é constante independentemente da estrutura das imagens, dado que cada pixel é descomprimido e processado individualmente. Isto pode ser mais facilmente observado através da Tabela 3.1 e do gráfico correspondente.

Por outro lado, na segunda abordagem, o algoritmo opera diretamente sobre as linhas codificadas em *Run-Length Encoding (RLE)*, o que faz com que a ordem de complexidade dependa diretamente da compressibilidade da imagem. No melhor caso, ambas as imagens apresentam uma única *run* por linha, o que se traduz numa complexidade:

$$\mathcal{O}(n)$$

Este caso reflete imagens com regiões uniformes extensas, podendo ser observado na Tabela 3.3 e no gráfico correspondente.

No pior caso, as imagens apresentam linhas com valores alternados pixel a pixel, resultando no número máximo de *runs*, igual ao número de pixels da linha. Neste cenário, a complexidade seria:

$$\mathcal{O}(n^2)$$

Este comportamento pode ser observado na Tabela 3.2 e no gráfico correspondente.

Por fim, no caso médio, o número de *runs* por linha situar-se-ia entre os extremos dos casos analisados, refletindo imagens com alguma compressibilidade, mas sem uniformidade total. Este comportamento resulta numa complexidade proporcional ao número médio de *runs*, tornando o algoritmo significativamente mais eficiente do que na abordagem que descomprime completamente as imagens.

3.4 Comparação

Com base na análise apresentada, a segunda abordagem é superior à primeira em termos de eficiência computacional e escalabilidade, especialmente em cenários onde o melhor caso é predominante (como imagens altamente comprimidas ou uniformes). Embora na primeira abordagem e no pior caso da segunda ambas tenham complexidade de $\mathcal{O}(n^2)$, a segunda abordagem pode alcançar $\mathcal{O}(n)$ no melhor caso, o que representa uma melhoria substancial.

Assim, a segunda abordagem é a melhor escolha para aplicações onde a manipulação direta do formato comprimido seja viável, justificando a maior complexidade de implementação pelo ganho significativo em desempenho. No entanto, para aplicações onde simplicidade e compatibilidade sejam mais importantes, a primeira abordagem pode ainda ser uma opção válida.

4 Conclusões

O desenvolvimento do TAD `imageBW` e a subsequente análise das funções implementadas possibilitaram a aplicação de conceitos avançados de manipulação de dados binários e otimização. A adoção da técnica de codificação por execução repetida (RLE) revelou-se eficaz, proporcionando uma economia substancial de espaço em memória. As avaliações de complexidade computacional, aliadas aos testes experimentais, corroboraram a eficiência das funções concebidas.