Algoritmos e Estruturas de Dados

1 º Projeto

Universidade de Aveiro

Dept. de Eletrónica, Telecomunicações e Informática

Rodrigo Nunes (119527), Paulo Lacerda(120202)

Índice

Capítulo 1

Introdução

No contexto da disciplina de Algoritmos e Estrutura de Dados foi-nos proposto um primeiro projeto que tem como base manipular e operar as imagens binárias (BW – black-and-white). Para isso cada pixel da imagem pode tomar um valor de intensidade 0 (white) ou 1 (black).

O objetivo principal deste projeto é criar e avaliar o TAD imageBW, concebido para representar e manipular imagens binárias comprimidas utilizando a técnica RLE (Run-Length Encoding). Este TAD deverá oferecer uma implementação eficaz, funcional e otimizada, permitindo o processamento de imagens onde cada pixel pode assumir um valor binário (0 ou 1). Será assegurada uma representação compacta das linhas através de sequências comprimidas. Adicionalmente, serão conduzidos testes exaustivos para verificar a precisão das suas funções e a eficiência da implementação com recurso aos próprios computadores dos alunos.

Computadores utilizados (especificações)			
Nº Mec / Nº PC	Memória RAM	CPU	GPU
119527 / PC1	16 GB	AMD Ryzen™ 7 5800H with Radeon Graphics	NVIDIA GeForce RTX 3050 Ti
120202 / PC2	16 GB	Intel® Core ™ i7-7500 U	Intel® HD Graphics 650

Capítulo 2

Análise da função ImageCreateChessboard ()

2.1 Dados experimentais

O seguinte gráfico foi realizado no PC2.

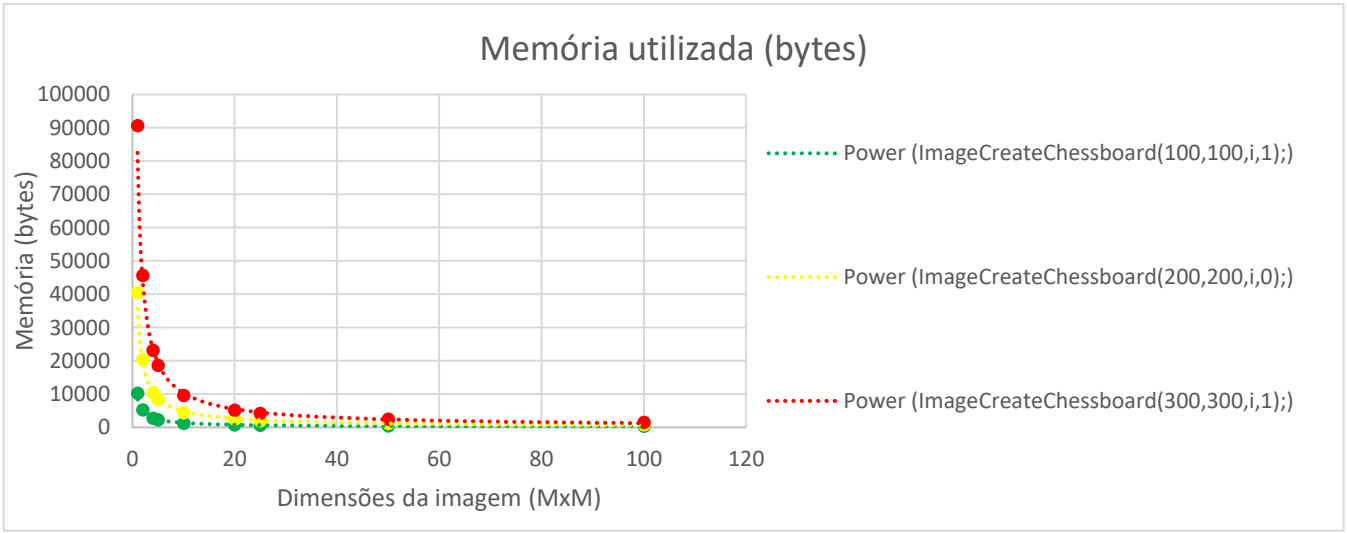

Gráfico 1- Memória em função do tamanho da imagem

A ordem de complexidade da função ImageCreateChessboard() é **log N**, pois T(2N) / T(N) ≈ 1, o que pode ser confirmado pela visualização de um dos gráficos da função. Também se pode concluir que a memória utilizada depende do *edge_size* (Memória = edge_size $^{-2}$), sabendo que o *edge_size* depende diretamente das dimensões da imagem.

ImageCreateChessboard(300,300,i,1);	
Tamanho do quadrado (MxM)	Memória utilizada (bytes)
1	90600
2	45600
4	23100
5	18600
10	9600
20	5100
25	4200
50	2400
100	1500

① 45600 / 90600 = 0,5033 ≈ 1

② 23100 / 45600 = 0,5079 ≈ 1

③ 18600 / 23100 = 0,8043 ≈ 1

④ 9600 / 18600 = 0,5161 ≈ 1

⑤ 5100 / 9600 = 0,5313 ≈ 1

⑥ 4200 / 5100 = 0,8235 ≈ 1

⑦ 2400 / 4200 = 0,5714 ≈ 1

⑧ 1500 / 2400 = 0,625 ≈ 1

O motivo para a redução da utilização da memória é porque o algoritmo **RLE** (Run-Length Encoding) trabalha com sequências consecutivas (VALOR, QUANTIDADE,QUANTIDADE…, -1) ou seja, em vez de armazenar cada pixel individualmente, ele agrupa-os em sequências de quantidades iguais , reduzindo assim o número total de elementos a serem armazenados.

2.2 Análise do espaço de memória ocupado

Em seguida analisou-se a execução da função com os parâmetros:

- **Largura (width) =** 12 pixels
- **Altura (height) =** 6 pixels
- **Tamanho do quadrado (square_edge) =** 3 pixels
- **Valor inicial (first_value) =** 0 (preto)

```
ImageCreateChessBoard(12, 6, 3, 0) -> I0
memoria utlizada : 36   bytes
ImageRAWPrint(I0)
width = 12 height = 6
RAW image:
000111000111
000111000111
000111000111
111000111000
111000111000
111000111000

ImageRLEPrint(I0)
width = 12 height = 6
RLE encoding:
0 3 3 3 3 -1
0 3 3 3 3 -1
0 3 3 3 3 -1
1 3 3 3 3 -1
1 3 3 3 3 -1
1 3 3 3 3 -1
```

Fig. 1 Output do comando *make test2*

1. Linha RAW

Cada linha da imagem é representada por um vetor de **12 pixels**, e como cada pixel ocupa 1 byte (varia entre 0 e 1) a memória necessária para armazenar uma linha da imagem sem compressão é:

$$Memoria\ por\ linha\ RAW = 12 * 1 = 12\ bytes$$

Como existem **6 linhas** de altura, a memória total necessária é:

$$Memória\ total\ de\ linhas = 6 * 12 = 72\ bytes$$

2. Compressão da Linha (RLE - Run-Length Encoding)

Como o padrão de tabuleiro de xadrez alterna as cores de forma regular (com quadrados de 3x3 pixéis), o padrão da primeira linha da imagem é:

[PRETO, PRETO, PRETO, BRANCO, BRANCO, BRANCO, PRETO, PRETO, PRETO, BRANCO, BRANCO, BRANCO]

[0 , 0 , 0 , 1 , 1 , 1 , 0 , 0 , 0 , 1 , 1 , 1]

1º fragmento 2º fragmento 3º fragmento 4º fragmento

Para cada fragmento o algoritmo RLE pode armazenar **2 valores**: a largura (3) e o valor do pixel (0 ou 1).

$$Memória\ por\ linha\ comprimida\ (M) = 4\ (segmentos) * 2(valores\ do\ pixel) * 4\ (bytes\ por\ inteiro) = 32\ bytes$$

A memória total necessária para armazenar todas as 6 linhas comprimidas será:

$$Memória\ total\ das\ linhas\ comprimidas = 6 * M = 192\ bytes$$

4. Cálculo da Memória Total

Para o cálculo total da memória utilizada é necessário somar o *Header, p*ois a função tem 2 parâmetros cada um deles com 4 bytes (**x**, **y**, i, 0/1) e o Array tem um tamanho de 8 bytes.

$$Header = \ 4 + 4 + 8 = 16\ bytes$$

$$Memória\ total = 192 + 16 = 208\ bytes$$

Capítulo 3

Análise da função ImageAnd ()

3.1 Dados experimentais

O gráfico seguinte representa a relação entre a evolução do tempo de execução, ou seja, o tempo que o algoritmo demora a fazer a função AND, em função do tamanho da imagem. Foi utilizado o PC2 na compilação e execução do código usando 200 imagens.

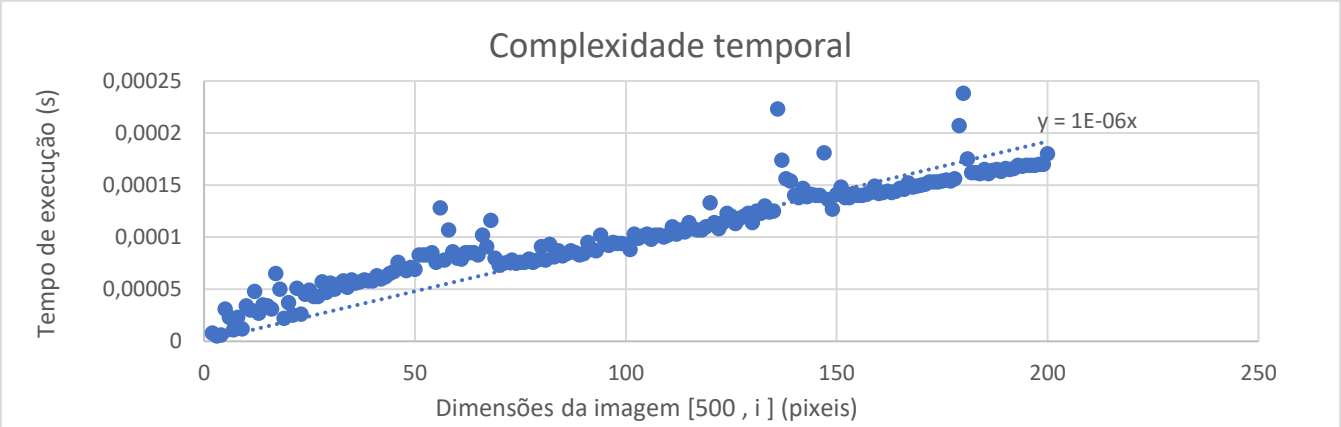

Gráfico 2- Evolução da complexidade computacional da função ImageAND()

3.2 Análise Formal

A função ImageAND () tem como objetivo combinar duas imagens (img1 e img2) aplicando a operação lógica AND bit a bit entre os seus dados. A função suporta dois formatos de representação de imagem: **RAW** (dados não comprimidos) e **RLE** (dados comprimidos com Run-Length Encoding). No caso de as imagens estarem no formato RLE, estas são descomprimidas, processadas, e depois comprimidas

novamente. O resultado é uma nova imagem no mesmo formato das imagens de entrada. Em seguida analisou-se para os dois formatos:

1 Formato RLE:

Descompressão compressão:

- A função **UncompressRow** (width, row) transforma a linha de formato RLE em formato RAW. O custo depende da largura da imagem, w, e é proporcional a O(w). A função CompressRow(width, result_raw) comprime o vetor resultante de volta para o formato RLE. O custo depende da compressibilidade da linha, mas no pior caso é proporcional a O(w). O custo para um linha é:

$$\boldsymbol{O}(w)+\boldsymbol{O}(w)+\boldsymbol{O}(w)=\boldsymbol{O}(w)$$

Operação AND:

- A operação lógica AND sobre duas linhas RAW (vetores de *w* pixels) requer O(w).

2 Formato RAW:

A operação lógica AND sobre duas linhas RAW é diretamente $\boldsymbol{O}(w)$. Não há necessidade de compressão

Complexidade Total do algoritmo:

- Dado que o algoritmo itera sobre *h* linhas e realiza *w* operações por linha, a complexidade no pior caso (onde cada linha é processada como RAW ou RLE) é: $\boldsymbol{O}\ (h \times w)$

Uso de Memória

- No total, o uso de memória é proporcional a $\boldsymbol{O}(w+h)$, pois o algoritmo processa uma linha de cada vez

Para todas as linhas (Total):

$$\sum_{i=1}^{h}\sum_{j=1}^{w} C = h * wC = \boldsymbol{O}(h * w)$$

h -> altura da imagem

w -> largura da imagem

C -> tempo/custo de uma operação

Capítulo 4

Conclusão

A função ChessBoard() tem o seu melhor e pior caso. Para ser o melhor caso é necessário que todos os pixéis da imagem sejam 1 ou 0, não havendo alternância de valores. Já o pior caso é se a sequência dos valores do pixel alternar entre 1 e 0 (1 0 1 0 1 0 …).

Na função imageAND() o gráfico poderá ser diferente do que o esperado, pois para os valores ao longo da função pode arredondar para o inferior nos valores maiores , e consequentemente para o superior para os valores mais baixos, devido ao método de verificação de memória. Logo o no seu gráfico a variável dependente vai ser superior/inferior à esperada.

Em suma, este trabalho para além de desenvolvermos as nossas capacidades de escrita na linguagem C, também mostrou a importância da procura de várias soluções para perceber as vantagens e desvantagens de cada uma delas.