**Projeto 1 AED**

O TAD *imageBW*

**Marta Cruz nº 119572 Catarina Ribeiro nº 119467**

**Introdução**

O objetivo deste trabalho foi desenvolver um tipo abstrato de dados (*imageBW*) que permite instânciar e operar com imagens binárias (*black and white*) em que cada pixel pode tomar a intensidade de 0 (*white*) ou 1 (*black*). Para avaliar o desenvolvimento utilizamos a ferramenta *imageBWTool* fornecida pelos docentes, de forma a aplicar testes específicos para cada função. Contudo, para analisar a complexidade computacional das funções *imageCreateChessboard()* e *imageAND()* optou-se por desenvolver 2 simples *scripts* de teste, os módulos *imageChessboardTest.c* e *imageANDTest.c*. Para testar estas funções basta executar os *script* Matlab *main.m* localizados nas diretorias *imageChessboardTest* e *imageANDTest*, respectivamente. O *script* começa por correr um outro *script shell* bash (*execute\_imageChessboardTest.sh* ou *execute\_imageANDTest.sh*), que por sua vez compilará e correrá os módulos de teste desenvolvidos na linguagem C. Estes módulos criam diversas instâncias de imagens *black-and-white* com diferentes valores números de linhas (m), colunas (n) e comprimento de aresta (s) de cada quadrado no padrão de xadrez, guardando os valores pretendidos para a análise num ficheiro de texto (*data\_imageChessboardTest.txt* ou *data\_imageANDTest.txt*). Estes valores são depois lidos pelo *script* Matlab, fornecendo gráficos para a interpretação de resultados. Estes diferentes módulos de teste permitem uma análise completa e variada dos casos solicitados de análise e desempenho do TAD *imageBW*.

*main.m*

*execute\_imageChessboardtest.sh*

*imageChessboardtest.c*

*imageBW.c*

*data\_imageChessboardTest.txt*

1

2

3

4

5

Figura 1- módulo de teste da função imageCreateChessboard()

### **Análise da complexidade da função *imageCreateChessboard()***

Desenvolvemos a função *imageCreateChessboard()* que, dada uma largura (*width*), uma altura (*height*), um comprimento de aresta (*square\_edge*) e um primeiro valor para o primeiro quadrado (*first\_value*) permite a criação de um padrão de xadrez perfeito. Considerando m como o número de linhas da imagem resultante, n como o número de colunas da imagem resultante e s o comprimento da aresta dos quadrados que compõem o padrão podemos fazer uma análise dos casos possíveis. Apresentamos um exemplo em baixo, em que *rslt* (imagem resultante)tem *m* = 6 e *n* = 12, onde iremos analisar o número total de *runs* e espaço em memória ocupado pela imagem.

A black and white rectangle

Description automatically generatedA black and white checkered pattern

Description automatically generated

Figura 2b – Pior caso

Figura 2a – Melhor caso

### **Número de *runs***

* 1) Melhor caso ( *s* = min(*m*,*n*)) = . No exemplo acima teríamos apenas 12 *runs*.
* 2) Pior caso (*s* = 1) = . No exemplo acima teríamos, então, 72 *runs*.

A dedução das expressões utilizadas em cima encontra-se nas equações abaixo. Durante a dedução foi utilizada a expressão base .

(1)

Podemos, a partir desta equação, explicitar as expressões particulares para o melhor (2) e pior (3) caso. O melhor caso ocorre quando *s* é maximizado, ou seja, *s* = min(*n*,*m*), portanto, substituindo na equação (1) temos:

(2)

O pior caso ocorre ao minimizar *s*, ou seja, *s* = 1. Substituindo na equação (1) temos então:

(3)

1. **Melhor caso (*s* = min(*m*,*n*))** = . Neste caso em particular o número de *runs* é dado exclusivamente pelo número de linhas e colunas da imagem resultante, não dependendo do comprimento da aresta (será sempre igual ao valor mínimo entre *m* e *n*). De forma a analisarmos este caso, geramos diversas imagens com tamanhos distintos, com *m* variante entre 50 e 100 e *n* variante entre 25 e 70, todas com o valor do pixel inicial igual a 1 e o comprimento da aresta igual ao valor mínimo entre *m* e *n*. Em seguida analisámos o número de *runs* em cada imagem resultante. As figuras seguintes mostram os resultados obtidos ao correr o *script Matlab* desenvolvido.

**A graph with a line

Description automatically generatedA graph with a line

Description automatically generated**

*Figura 3a - Número de runs em função de m (Melhor caso)*

Figura 3b - Número de runs em função de n (Pior caso)

Podemos então verificar empiricamente as expressões obtidas formalmente. A figura 4a ilustra o número de *runs* em função do valor de *m* com *n* constante e igual a 70. Enquanto *m* < *n* verificamos que o número de *runs* cresce linearmente até ao seu valor máximo, 138 (), iguala-se a 70 quando *m* toma o valor de *n* e cresce, novamente, linearmente até 100 (valor máximo de *m*). A figura 4b ilustra o número de *runs* em função do valor de *n* com *m* constante e igual a 50. O número de *runs* é igual ao valor de *m* até *n* > *m* e passa depois para o dobro, confirmando assim as deduções teóricas. É importante referir que os valores de *m* e *n* foram escolhidos propositadamente para demostrar os 2 ramos da equação (2).

1. **Pior Caso (s = 1)** = . Neste caso, o número de *runs* continua apenas a depender do número de linhas e colunas da imagem resultante, não dependendo do valor do comprimento da aresta, que será mínimo e igual a 1. Para proceder à análise da complexidade geramos as mesmas imagens utilizadas no cenário descrito acima, mas com o valor do comprimento da aresta a 1. As figuras seguintes mostram os resultados obtidos ao correr o *script Matlab* desenvolvido.

A graph with a line

Description automatically generatedA graph with a line

Description automatically generated

Figura 4b – Número de runs em função de n (Pior caso)

Figura 4a – Número de runs em função de m (Pior caso)

Podemos então confirmar a nossa dedução formal. A figura 5a ilustra o número de *runs* em função do valor de *m* com *n* constante e igual a 70. Para os diversos pontos no gráfico verifica-se a relação: . O que comprova a expressão obtida (). É portanto, apresentada uma regressão linear de declive igual ao valor de *n* (70). A figura 5b ilustra o número de runs em função do número de *n* com *m* constante e igual a 50. Portanto, para os diversos pontos verifica-se a relação: , o que comprova, mais uma vez a expressão obtida. É, portanto, apresentada uma regressão linear de declive igual ao valor de *m* (50).

Figura 4b – Número de runs em função de n (Pior caso)

### **Espaço de memória ocupado (*memspace*)**

* 1) Melhor caso ( *s* = min(*m*,*n*)) = . No exemplo apresentado acima teríamos .
* Pior caso (*s* = 1) = . No exemplo apresentado acime teríamos .

A dedução das expressões utilizadas em cima encontra-se nas equações abaixo. Durante a dedução foi utilizada a expressão base e admite-se *sizeof(int)* = 4.

(4)

Podemos, a partir desta equação derivar as expressões para o melhor (5) e pior (6) caso. O melhor caso ocorre quando *s* é maximizado, ou seja, *s* = min(*n*,*m*), portanto, substituindo na equação (4) temos:

(5)

O pior caso ocorre ao minimizar *s*, ou seja, *s* = 1. Substituindo na equação (4) temos então:

(6)

1. **Melhor caso (*s* = min(*m*,*n*))** = .Neste caso em particular o espaço ocupado em memória é dado exclusivamente pelo número de linhas e colunas da imagem resultante, não dependendo do comprimento da aresta (será sempre igual ao valor mínimo entre *m* e *n*). De forma a analisarmos este caso, geramos, novamente, diversas imagens com tamanhos distintos, com *m* variante entre 50 e 100 e *n* variante entre 25 e 70, todas com o valor do pixel inicial igual a 1 e o comprimento da aresta igual ao valor mínimo entre *m* e *n*. Em seguida analisámos o espaço em memóriaocupado por cada imagem resultante. As figuras seguintes mostram os resultados obtidos ao correr o *script Matlab* desenvolvido.

A graph with a line

Description automatically generatedA graph with a line

Description automatically generated

Figura 5b – Espaço em memória em função de n (Melhor caso)

Figura 5a – Espaço em memória em função de m (Melhor caso)

Podemos então verificar, mais uma vez, as expressões obtidas analiticamente. A figura 6a ilustra o espaço ocupado em memória (*memspace*) em função do valor de *m* com *n* constante e igual a 70. Enquanto *m* < *n* verificamos que o espaço ocupado cresce linearmente até ao seu valor máximo, tendo depois uma quebra *m* toma o valor de *n* (quando *n = m* todas as linhas terão apenas 1 *run*) e cresce, novamente, linearmente .A figura 6b ilustra o espaço ocupado em memória em função do valor de *n* com *m* constante e igual a 50. O espaço ocupado em memória toma o valor constante de 600 ( quando *m < n* e passa ao valor de 800 () , confirmando assim as deduções teóricas. É importante referir que os valores de *m* e *n* foram escolhidos propositadamente para demostrar os 2 ramos da equação (5).

1. **Pior Caso (s = 1)** = . Neste caso, o espaço ocupado em memória continua apenas a depender do número de linhas e colunas da imagem resultante, não dependendo do valor do comprimento da aresta, que será mínimo e igual a 1. Para proceder à análise da complexidade geramos as mesmas imagens utilizadas no cenário descrito acima, mas com o valor do comprimento da aresta a 1. As figuras seguintes mostram os resultados obtidos ao correr o *script Matlab* desenvolvido.

A graph with a line

Description automatically generatedA graph of a line

Description automatically generated

Figura 6a – Espaço em memória em função de m (Pior caso)

Figura 6b – Espaço em memória em função de n (Pior caso)

Verifica-se então a dedução formal. A figura 7a ilustra o espaço ocupado em memória em função do valor de *m* com *n* constante e igual a 70. Para os diversos pontos no gráfico verifica-se a relação: . O que comprova a expressão obtida (). É, portanto, apresentada uma regressão linear () de declive igual ao valor de (288). A figura 7b ilustra o espaço ocupado em memória em função do número de *n* com *m* constante e igual a 50. Portanto, para os diversos pontos verifica-se a relação: , o que, comprova, mais uma vez, a expressão obtida. É, portanto, apresentada uma regressão linear de declive igual ao valor de (200) e ordenada na origem (400).

De forma a concluir a nossa análise, vimos também o comportamento do número de *runs* e do espaço ocupado em memória em função do valor de *s*, que, até ao momento temos sempre considerado constante. Geramos, portanto, diversas imagens com o valor de *m* = 50, o valor de *n =* 70 e o valor de *s* a variar entre 1 e 100. Em seguida, analisámos o número de *runs* e o espaço ocupado em memória em cada imagem resultante. As figuras seguintes mostram os resultados obtidos ao correr o *script Matlab* desenvolvido.

A graph of a number of objects

Description automatically generatedA graph of a number of runs

Description automatically generated

Figura 7a – Número de runs em função de s

Figura 7b – Espaço em memória em função de s

Verificamos então que tanto o número de *runs* como o espaço ocupado em memória diminuem exponencialmente com o aumento do valor de *s*. Podemos ver uma estabilização dos valores quando *s* se iguala a 70 (*numruns* e *memspace* são mínimos).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Exemples** | **Width (*n*)** | **Height (m)** | **Square\_edge (*s*)** | **First Pixel** | **Runs** | **Memspace** |
| Melhor Caso | 2 | 1 | 1 | 1 | 2 | 16 |
| 12 | 6 | 6 | 1 | 12 | 96 |
| 70 | 70 | 70 | 0 | 70 | 840 |
| 36 | 72 | 36 | 0 | 72 | 864 |
| Pior Caso | 12 | 6 | 1 | 1 | 72 | 336 |
| 36 | 72 | 1 | 1 | 2592 | 10944 |
| 50 | 85 | 1 | 0 | 4250 | 17680 |
| 70 | 70 | 1 | 0 | 4900 | 20160 |

Figura 8 – Tabela com casos aleatórios da função ImageCreateChessBoard()

### **Análise da complexidade da função *imageAND()***

Img2

Img1

Img2

Img1

Img2

Img1

Uma imagem com quadrado, padrão, Retângulo, Simetria

Descrição gerada automaticamenteUma imagem com quadrado, padrão, Simetria, Retângulo

Descrição gerada automaticamenteUma imagem com preto, captura de ecrã

Descrição gerada automaticamenteUma imagem com preto, captura de ecrã

Descrição gerada automaticamenteUma imagem com captura de ecrã, branco, design

Descrição gerada automaticamenteUma imagem com quadrado, padrão, Simetria, Retângulo

Descrição gerada automaticamente

Figura 9c – Pior Caso

Figura 9b – Caso Médio

Figura 9a – Melhor Caso

Desenvolvemos duas versões da função *imageAND()*, sendo a primeira uma versão otimizada. A 2ª versão da função aceita duas imagens codificadas em RLE, percorre as imagens linha a linha, descodifica-as e realiza a operação AND entre os valores dos pixéis (pixel a pixel) no formato *raw*. Finalmente volta a codificar as linhas em RLE, originando, como resultado, uma nova imagem. Esta função funciona, mas não é ótima, uma vez que percorre sempre todos os pixéis 1 a 1, independentemente das *runs.* Percebemos esta falta de relação através da expressão deduzida abaixo. Considere-se *h* e *w* como sendo a altura e largura das imagens passadas como argumentos, respectivamente. Utilizamos, novamente, a expressão: .

(1)

De uma forma sumária, temos de contabilizar o número de operações realizadas pela função *UncompressRow()* (termos :) e *CompressRow()* (termo: ). Adicionalmente contabilizamos o número de operações lógicas *and* que se realizam (termo: ). É importante referir que, por cada linha, a função *CompressRow()* não realiza 1 incremento da variável que contabiliza o número de operações, uma vez que a função não necessita de outro *“toggle”* para fechar a última sequência. Isto leva a uma poupança de 1 operação por linha. Deste modo, temos de retirar *h* operações para contabilizar esses casos.

Procurámos desenvolver logo a partir do início uma versão otimizada desta função, fazendo as operações diretamente nas linhas comprimidas. Deste modo, a nossa primeira versão da função percorre as linhas codificadas, *run* a *run* e concretiza a operação lógica AND entre a run de menor tamanho e os pixéis pertencentes à run de maior tamanho que se sobreponham. Deste modo conseguimos diminuir significativamente o número de operações quando as imagens passadas como argumentos se encontram muito comprimidas. Apresentamos abaixo as deduções para o melhor (2) e pior (3) caso da função ótima. Durante a dedução foi utilizada novamente a expressão base .

(2)

No melhor caso ambas as imagens estão completamente comprimidas, ou seja, cada *row* é representada por 1 *array* com 3 elementos, apenas 1 deles representando a *run*. Deste modo apenas contabilizamos 2 operações por cada *row* (os 2 *“toggles”* não ocorrem uma vez que o *array* só tem 1 *run*, sendo o elemento seguinte = EOR).

(3)

No pior caso, as duas imagens têm todas as *rows* codificadas em RLE em que cada *run* representa 1 pixel, deste modo o *while loop* correrá *w* vezes e incrementará o contador 4 vezes. Tal como no caso da função acima, temos de contabilizar que na útlima *run* da *row* não serão realizados 2 *“toggles”*, o que se traduz numa poupança de 2 operações por cada linha, resultando assim numa subtração de operações.

Concluindo, a função ótima salva operações significativamente quando estamos perante o melhor caso, e consegue ser melhor, mesmo que ligeiramente, quando estamos perante o pior caso. Abaixo apresentamos os gráficos gerados pelos *script Matlab* que permitem uma análise computacional dos cenários descritos formalmente em cima.

A graph with red and blue lines

Description automatically generatedA graph of a number of rows

Description automatically generatedA graph with a number of rows

Description automatically generated

Figura 10a – Número de operações em função de m (Melhor caso)

Figura 10c – Número de operações em função de m (Pior caso)

Figura 10b – Número de operações em função de m (Caso médio)

Figura 11 – Tabela com casos aleatórios da função ImageAND()

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Exemples** | **Width (*n*)** | **Height (*h*)** | **Square\_edge (*s*)** | **First Pixel \*** | **Operations V1** | **Operations V2** |
| Melhor Caso | 6 | 6 | 1 | 1/1 | 138 | 12 |
| 24 | 12 | 1 | 1/0 | 1140 | 24 |
| 30 | 45 | 1 | 0/1 | 5355 | 90 |
| Caso Médio | 6 | 6 | 1 | 0/1 | 138 | 102 |
| 24 | 12 | 1 | 1/0 | 1140 | 852 |
| 30 | 45 | 1 | 0/0 | 5355 | 4005 |
| Pior Caso | 6 | 6 | 1 | 1/0 | 138 | 132 |
| 24 | 12 | 1 | 0/1 | 1140 | 1128 |
| 30 | 45 | 1 | 1/0 | 5355 | 5310 |

\*Nota: 1/0 significa que o primeiro pixel da primeira imagem tem valor 1 e o primeiro pixel da segunda imagem tem valor 0

1. **Conclusão**

O desenvolvimento do TAD imageBW permitiu implementar e testar operações eficientes sobre imagens binárias compactadas através do método de codificação Run-Length Encoding (RLE). Este trabalho abordou desde a criação de imagens com padrões específicos até operações lógicas entre imagens, demonstrando as vantagens e desafios associados à compactação e manipulação de dados em imagens binárias.

Na função *ImageCreateChessBoard(),* analisamos como o padrão de xadrez influencia o número de runs e o espaço em memória, confirmando que padrões mais regulares ocupam menos memória. Já na função *ImageAND(),*os resultados experimentais e a análise formal revelaram uma complexidade que varia linearmente no melhor caso e quadraticamente no pior caso, mostrando a influência da densidade de transições nos dados.

Os testes realizados demonstraram a eficácia das implementações e confirmaram as análises formais. Este trabalho destacou a importância de equilibrar eficiência e simplicidade na manipulação de dados compactados, mostrando como algoritmos bem projetados podem otimizar o desempenho em diferentes cenários.

O TAD *imageBW* é uma solução prática e eficiente para operar com imagens binárias, oferecendo compressão, operações geométricas e lógicas de forma integrada e funcional.