deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Trabalho 1 – Algoritmos e Estruturas de Dados

Universidade de Aveiro

Dep. de Eletrónica, Telecomunicações e Informática

Rodrigo Gonçalves 124750

Rodrigo Silva 125171

Índice

Introdução

Este relatório tem como objetivo apresentar a análise e os resultados obtidos durante a implementação e testes do **TAD *imageRGB***, desenvolvido no âmbito da unidade curricular Algoritmos e Estruturas de Dados. Este TAD permite representar e manipular imagens baseadas no modelo de cor RGB, recorrendo a uma matriz 2D e uma LUT (LookUp Table). Este trabalho vai permitir compreender o funcionamento do TAD e avaliar o desempenho das funções fornecidas, através da implementação das operações básicas, pela análise experimental e formal da função ***ImageIsEqual***, e pela comparação de diferentes abordagens de segmentação baseadas no algoritmo ***Region Growing by Flood Filling***.

Especificações do pc usado para os testes

CPU: AMD RYZEN 5 7520U | GPU: AMD Radeon™ Graphics | Memoria: 16GB

1 Desenvolvimento do TAD imageRGB

1.1 **ImageCopy:**

Realiza uma cópia profunda (deep copy) da imagem original, criando uma nova estrutura totalmente independente na memória. Ela aloca um novo cabeçalho com as mesmas dimensões, duplica a tabela de cores (LUT) e copia os dados dos pixels linha por linha. O resultado é um clone exato que pode ser modificado sem afetar a imagem de origem.

1.2 **ImageIsEqual:**

Verifica se duas imagens são visualmente idênticas, validando primeiro se possuem as mesmas dimensões. Em seguida, ela percorre todos os pixels e compara as suas cores reais (resolvidas através da tabela LUT), garantindo que o resultado visual seja igual, mesmo que os índices internos ou a organização da memória sejam diferentes.

1.3 **ImageRotate90CW:**

Cria uma nova imagem rodada 90 graus no sentido horário, trocando as dimensões de largura e altura da imagem original. Ela copia a paleta de cores e remapeia cada pixel, transformando as coordenadas de modo que a linha original i e coluna j passem para a nova posição (j, altura - 1 - i).

1.4 **ImageRotate180CW:**

Cria uma nova imagem rodada 180 graus, mantendo as mesmas dimensões de largura e altura da original. Ela copia a paleta de cores e inverte a posição de cada pixel tanto na vertical quanto na horizontal, mapeando a coordenada original (i, j) para a posição oposta (altura - 1 - i, largura - 1 - j).

2 Analise da função ImageIsEqual(img1, img2)

2.1 Estrutura e fluxo do algoritmo

Validações Iniciais: Garante que os ponteiros são válidos e verifica se apontam para o mesmo endereço de memória.

Verificação de Dimensões: Confirma se a largura e altura coincidem. Se forem diferentes, rejeita imediatamente

Otimização de Acesso: Guarda os ponteiros para as LUTs e para as linhas atuais fora do ciclo interno para evitar acessos repetidos à estrutura principal.

Comparação Pixel a Pixel: Percorre as colunas, contabiliza os acessos para a análise experimental (PIXMEM) e compara as cores reais na LUT. Retorna 0 à primeira diferença encontrada.

```c
int ImageIsEqual(const Image img1, const Image img2) {
  assert(img1 != NULL);
  assert(img2 != NULL);

  //Se forem a mesma imagem em memoria
  if (img1 == img2) {
    return 1;
  }

  //Verifica se têm o mesmo tamanho
  if (img1->width != img2->width || img1->height != img2->height) {
    return 0;
  }

  rgb_t* lut1 = img1->LUT;
  rgb_t* lut2 = img2->LUT;

  //percorrer todos os pixels
  for (uint32 i = 0; i < img1->height; i++) {

    //ir buscar o índice do pixel nas duas imagens
    uint16* row1 = img1->image[i];
    uint16* row2 = img2->image[i];

    for (uint32 j = 0; j < img1->width; j++) {

      //contar acessos
      PIXMEM += 2;

      //obter a cor verdadeira no LUT
      uint16 label1 = row1[j];
      uint16 label2 = row2[j];

      //Se as cores forem diferentes as imagens são diferentes
      if (lut1[label1] != lut2[label2]) {
        return 0; // Encontrou uma diferença
      }
    }
  }
  return 1; // Imagens são iguais
}
```

Figura 1: Código anotado da função ImageIsEqual e respetiva lógica.

2.2 Análise Formal

A operação central do algoritmo, que determina se os pixels das duas imagens são iguais, é

$$\text{if } (\text{lut1}[\text{label1}] \; != \; \text{lut2}[\text{label2}])$$

Pior caso:

Se as imagens forem iguais, o algoritmo compara todos os pixels:

$$T_{\text{pior}}(n) = w \times h = n^2$$

Melhor caso:

Se as dimensões das imagens forem diferentes o a função não chega a fazer comparações:

$$T_{melhor}(0) = 0$$

Assim, a complexidade temporal do algoritmo é proporcional ao número total de pixels da imagem:

$$O(w \times h) = O(n^2)$$

2.3 Análise Experimental

Foram realizados 20 testes computacionais utilizando imagens com diferentes resoluções, e em cada teste foram registados o tempo de execução da função e o número de comparações feitas para o melhor e pior caso.

2.3.1 Análise dos resultados

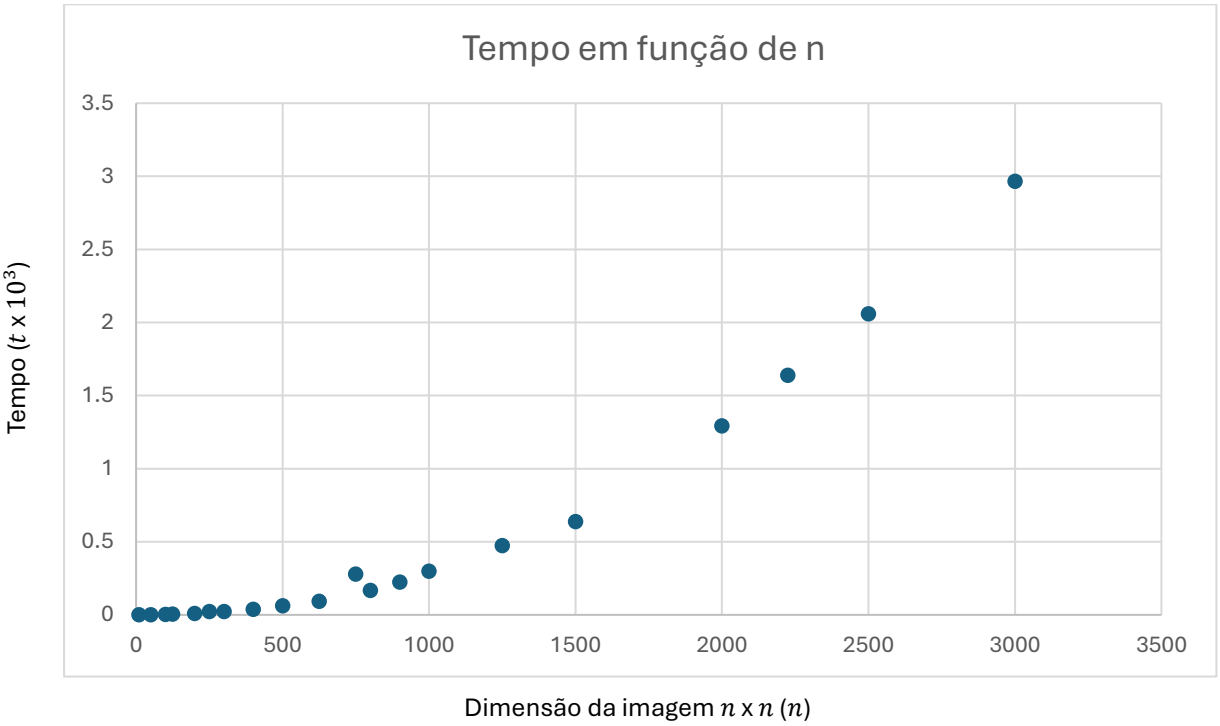

Figura 2: Gráfico de dispersão que relaciona a dimensão linear da imagem (n) com o tempo de execução (caltime) no pior caso.

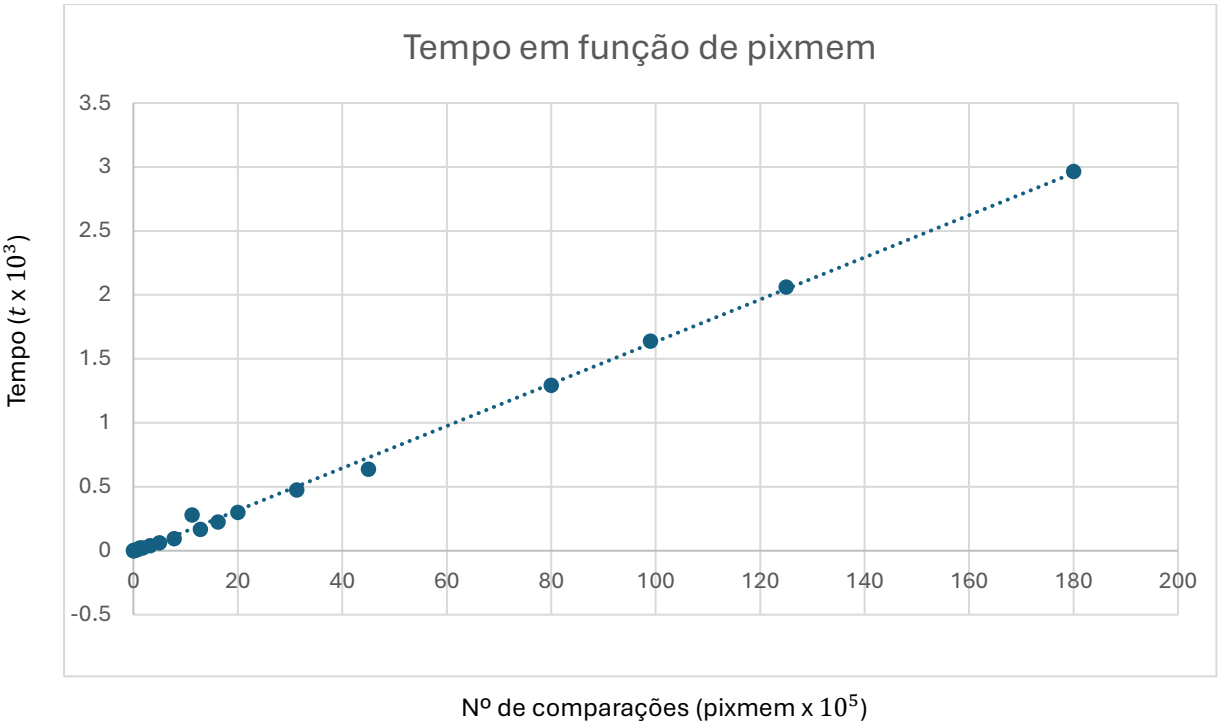

Figura 3: Relação entre o número de comparações e o tempo de execução, evidenciando um comportamento linear.

Os resultados experimentais demonstram que, no melhor caso, não chega a ser feita nenhuma comparação e o tempo de execução é sempre praticamente nulo. Já no pior caso, o tempo de execução cresce exponencialmente em relação a n e aumenta de forma aproximadamente linear com o crescimento do número de comparações feitas ($n\ x\ n$).

2.4 **Conclusão final**

A análise experimental comprovou a análise formal, que o desempenho da função cresce linearmente com o número de pixels da imagem, validando que o algoritmo possui ordem de complexidade linear.

3 Region Growing by Flood Filling

Neste trabalho foram usados os seguintes três métodos do algoritmo de *Region Growing by Flood Filing*:

- ImageRegionFillingRecursive(img, u, v, color)
- ImageRegionFillingWithSTACK(img, u, v, color)
- ImageRegionFillingWithQUEUE(img, u, v, color)

Todos estes métodos têm o objetivo de, a partir de um pixel semente, preencher todos os pixels vizinhos que possuem a mesma cor original, substituindo-a por uma nova cor, porém com abordagens diferentes.

3.1 **ImageRegionFillingRecursive**

Esta é uma abordagem recursiva do algoritmo que explora os pixels vizinhos chamando a própria função para cada pixel adjacente.

A função chama-se si própria, tornando o código simples e intuitivo e guardando cada chamada da função automaticamente. Isto torna o código mais curto e fácil de compreender e excelente para imagens pequenas; porém há um risco de stack overflow se houver regiões demasiado grandes.

3.2 **ImageRegionFillingWhithSTACK**

Esta versão simula o comportamento da recursão utilizando essencialmente uma estrutura de dados do tipo STACK. As suas características implementam a estrutura STACK para armazenar pixels por processar e envia chamadas recursivas.

Quanto às vantagens, esta abordagem não corre o risco de stack overflow da recursão do sistema, permitindo um maior controlo de memoria utilizada, daí ser mais robusta para imagens grandes; mas também apresenta desvantagens, como um código mais extenso e complexo que a versão recursiva, também exige uma criação e destruição manual de pilha.

3.3 **ImageRegionFillingWithQUEUE**

Esta versão usa uma Fila que organiza os pixels da mesma cor numa ordem de proximidade ao pixel inicial, sendo então processados nessa ordem.

Como vantagens, esta abordagem é mais previsível em termos do uso da memória, mais útil caso se pretenda um crescimento equilibrado da região, e previne stack overflow. No entanto torna-se desvantajosa na eficiência de memória (comparativamente com as outras funções) e por ser mais complexa.

3.4 **Conclusão**

Embora as três opções preencham corretamente as regiões, a versão recursiva é adequada exclusivamente para imagens mais pequenas devido ao risco de stack overflow. Já a versão Stack e Queue são mais seguras e escaláveis.
A versão com Stack é eficiente e tem um comportamento parecido á recursiva, enquanto a versão com Queue oferece um preenchimento mais homogéneo e controlado da região.

Para aplicações reais e imagens de grandes dimensões, as abordagens iterativas são as que apresentam melhores resultados.

4 Conclusão

Com a realização deste trabalho foi possível compreender, de forma prática, o funcionamento de um Tipo Abstrato de Dados (TAD) para imagens RGB, bem como a importância das estruturas de dados e da análise de algoritmos no desempenho de operações em imagens.

A implementação das funções ImageCopy, ImageIsEqual, ImageRotate90CW e ImageRotate180CW permitiu consolidar conceitos fundamentais relacionados com manipulação de matrizes, acesso eficiente à memória e utilização de tabelas de cores (LUT). Já a função ImageIsEqual mostrou-se essencial para o estudo da eficiência algorítmica, possibilitando a realização de uma análise formal e experimental. Os resultados obtidos na análise experimental confirmaram o que foi previsto teoricamente: o tempo de execução no pior caso cresce de forma proporcional ao número total de pixels da imagem, o que valida a complexidade temporal O(n^2).

Relativamente ao algoritmo de Region Growing by Flood Filling, foi possível comparar e compreender as vantagens e limitações de diferentes abordagens. Concluiu-se que, embora a versão recursiva seja mais simples e intuitiva, as versões iterativas são mais adequadas para imagens de grandes dimensões, por evitarem problemas como *stack overflow* e permitirem maior controlo da memória.

Concluindo, este trabalho permitiu não só desenvolver competências na implementação e análise de algoritmos, mas também reforçar a compreensão da relação entre escolha de estruturas de dados, eficiência computacional e aplicabilidade prática em problemas reais de processamento de imagem.

5 Dados

Dimensão	Time	Caltime	Pixmem
10x10	0.000001	0.000000	200
50x50	0.000002	0.000001	5000
100x100	0.000006	0.000002	20000
125x125	0.000009	0.000004	31250
200x200	0.000020	0.000009	80000
250x250	0.000052	0.000022	125000
300x300	0.000051	0.000022	180000
400x400	0.000089	0.000038	320000
500x500	0.000144	0.000061	500000
625x625	0.000221	0.000093	781250
750x750	0.000660	0.000279	1125000
800x800	0.000395	0.000167	1280000
900x900	0.000529	0.000224	1620000
1000x1000	0.000707	0.000299	2000000
1250x1250	0.001121	0.000474	3125000
1500x1500	0.001511	0.000638	4500000
2000x2000	0.003061	0.001293	8000000
2225x2225	0.003876	0.001638	9901250
2500x2500	0.004876	0.002060	12500000
3000x3000	0.007017	0.002965	18000000

Figura 4: Resultados experimentais de tempo e memória para diferentes dimensões de imagem (Pior Caso).