Relatório Técnico Avançado — TAD `imageRGB`

Implementação, Verificação e Análise Experimental

Aluno: _____
Docente: _____

November 22, 2025

Abstract

Este relatório apresenta a implementação final, a verificação de memória e a análise experimental do TAD `imageRGB` — um módulo para manipulação de imagens indexadas por uma Look-Up Table (LUT). O documento detalha as operações completadas (cópia profunda, comparação por cor, rotações, preenchimento de regiões e segmentação), descreve a análise formal da complexidade da função `ImageIsEqual`, propõe um protocolo experimental para validar as propriedades teóricas e compara três estratégias de *region growing*. Inclui também um apêndice técnico com instruções para execução de testes dinâmicos com `Valgrind` e `ASAN`.

Contents

1 Introdução

1.1 Contexto

O TAD `imageRGB` representa imagens em formato indexado: cada pixel armazena um índice que referencia uma LUT com tuplos RGB de 24 bits. Esta representação é adequada para aplicações onde o número de cores distintas é significativamente menor que o número de pixels, facilitando operações como segmentação, transformação e compressão básica.

1.2 Objectivos do trabalho

Os objectivos principais são:

1. Completar as funções do ficheiro `imageRGB.c` marcadas como "TO BE COMPLETED".
2. Garantir que não existem fugas de memória em rotas principais de execução.
3. Analisar formal e experimentalmente a função `ImageIsEqual`.
4. Implementar e comparar três estratégias de *region growing* (recursiva, STACK, QUEUE).
5. Produzir um relatório de nível universitário com metodologia, resultados e conclusões.

2 Descrição do TAD e Estrutura de Dados

2.1 Representação interna

O TAD usa:

- `struct image` com campos: `width, height, uint16** image, uint16 num_colors, rgb_t* LUT`.
- LUT de tamanho fixo (`FIXED_LUT_SIZE`).
- Cada pixel guarda um índice (`uint16`) na tabela `image[row][col]`.

2.2 Invariantes e pré-condições

- `width > 0`, `height > 0`.
- `image` é um array de `height` ponteiros válidos.
- `num_colors <= FIXED_LUT_SIZE`.
- Todos os acessos a pixels usam `ImageIsValidPixel` como verificação.

3 Implementações completadas

3.1 ImageCopy

Implementação: aloca um novo header via `AllocateImageHeader`, copia os `num_colors`, copia a LUT (somente as entradas usadas) e duplica linha-a-linha os arrays de índices. Em caso de falha intermédia, deve libertar o que foi alocado.

3.2 ImageIsEqual

Definição: compara dimensões; se iguais, compara o valor RGB real para cada pixel:

$$\text{c1} = \text{img1->LUT[img1->image[i][j]]}, \quad \text{c2} = \text{img2->LUT[img2->image[i][j]]}.$$

Retorna 1 se $\forall$ pixels c1 == c2, senão 0. Esta abordagem garante igualdade sem depender da correspondência de índices de LUT.

3.3 ImageRotate90CW e ImageRotate180CW

Implementações que alocam uma nova imagem com dimensões apropriadas, copiam a LUT usada e preencham a matriz de índices com mapeamento correto:

$$(i, j) \mapsto (j, height - 1 - i) \quad (90° \text{ CW})$$

$$(i, j) \mapsto (height - 1 - i, width - 1 - j) \quad (180° \text{ CW})$$

3.4 Region Filling: Recursive / STACK / QUEUE

As três versões implementadas seguem a mesma política: identificar o *target* (valor de índice do pixel seed), marcar imediatamente os pixels aceites com o novo rótulo para evitar duplicação, e propagar para os 4-vizinhos. Notas importantes:

- Recursiva: elegante, mas risco de `stack overflow` para regiões grandes.
- STACK (DFS): usa TAD `PixelCoordsStack` para evitar limites de recursão.
- QUEUE (BFS): usa `PixelCoordsQueue` para visita em largura; ideal para preenchimento por camadas.

3.5 ImageSegmentation

Percorre a imagem sequencialmente do pixel (0,0) a (width-1,height-1). Quando encontra um pixel branco (rótulo `WHITE`), gera nova cor via `GenerateNextColor`, aloca na LUT com `LUTAllocColor` e chama a função de *filling* escolhida para rotular a região. Conta regiões.

4 Análise Formal: `ImageIsEqual`

4.1 Complexidade temporal

Sejam W (largura), H (altura), $N = W \cdot H$.

- Melhor caso: dimensões diferentes $\Rightarrow O(1)$.
- Caso de curto-circuito: primeira diferença nos pixels $\Rightarrow O(1)$.
- Pior caso: todas as dimensões e pixels iguais $\Rightarrow \Theta(N)$.

4.2 Complexidade espacial

Uso de memória adicional: $O(1)$. Apenas variáveis temporárias para ler LUT e indices.

4.3 Número de comparações

Definimos uma *comparação* como a verificação $c1 != c2$ por pixel. - Melhor caso: 0 comparações (dimensões diferentes). - Pior caso: N comparações.

5 Análise Experimental: metodologia e protocolo

5.1 Objetivos experimentais

Confirmar: comparações $\propto N$ no pior caso; identificar constantes empíricas; medir impacto de diferentes padrões de imagem (aleatória vs homogénea vs palete).

5.2 Gerador de instâncias

Para cada dimensão $W \times H$:

- **Same**: b = ImageCopy(a) (pior caso)
- **Diff_Front**: b difere em pixel (0,0) (curto-circuito)
- **Diff_Back**: b difere no último pixel (próximo ao pior caso)
- **Diff_Dims**: b com dimensões diferentes (melhor caso)

5.3 Métrica

Usar contador atômico ou instr. global `PIXMEM` (ou `cmpcounter`) incrementado na comparação por pixel. Medir também tempo CPU via `cpu_time()` para ver correlação tempo vs comparações.

5.4 Tamanhos testados

10×10, 100×100, 250×250, 500×500, 1000×1000 (dependendo de memória).

5.5 Protocolos de execução

Para cada par:

- Repetir 10 vezes e tirar média.
- Repor memória entre execuções.
- Guardar: (W,H), caso, média comparações, desvio padrão, tempo médio.

6 Comparação das estratégias de Region Growing

6.1 Critérios

Avaliar: tempo de execução, memória máxima adicional, comportamento na forma da região (nomeadamente, fronteira), estabilidade (sem crashs).

6.2 Hipóteses

- Tempo: todas $\Theta(R)$, com pequenas variações constantes.
- Espaço: recursiva arriscada ($O(D)$ stack), STACK e QUEUE $O(R)$ no pior caso.
- Robustez: STACK/QUEUE preferíveis para imagens grandes.

6.3 Testes sugeridos

Gerar imagens com:

- Região conectada de grande dimensão (ex.: um grande rectângulo branco).
- Região "labiríntica" para maximizar profundidade.
- Muitas pequenas regiões dispersas.

Registar tempo e memória (usar `/usr/bin/time -v` para pico de memória).

7 Auditoria de Memória: inspeção estática e práticas

7.1 Pontos de risco identificados

- Falhas intermédias em múltiplas alocações: recomenda-se libertar recursos antes de `exit`.
- Versão recursiva: risco de stack overflow.
- Operações com TADs auxiliares: garantir que `StackDestroy` e `QueueDestroy` são sempre chamados nos caminhos de erro.

7.2 Boas práticas aplicadas

- Usar `assert` para pré-condições e `check()` para erros de I/O ou alocação.
- Documentar claramente a propriedade de cada alocação.
- Em `ImageDestroy`, garantir `if (*imgp == NULL) return;` depois do assert.

8 Resultados (espaço reservado para dados experimentais)

Nota: Nesta versão do relatório incluí as tabelas vazias — assim que forem gerados os resultados experimentais concretos (corridas com Valgrind e medições), substituirei os campos.

Table 1: Exemplo de formatação — Contagens de comparações por caso (média)

W	H	Caso	Comparações (média)	Tempo CPU (ms)
10	10	Same	100	-
100	100	Same	10000	-
250	250	Same	62500	-
500	500	Same	250000	-

9 Plano de Verificação Dinâmica (Valgrind / ASAN)

9.1 Compilar

```
# compilação normal
make clean
gcc -g -O0 -Wall -Wextra *.c -o image_test

# compilação com ASAN
gcc -fsanitize=address -g -O1 *.c -o image_asan
```

9.2 Valgrind

```
valgrind --leak-check=full --show-leak-kinds=all \
         --track-origins=yes ./image_test
```

9.3 Interpretação chave

- `definitely lost > 0` ⇒ fuga real.
- `still reachable` ⇒ analisar stack trace e decidir se é cache ou leak.

10 Apêndices

10.1 Apêndice A — Ficheiros usados (ambiente do relatório)

Os ficheiros que serviram de referência neste relatório encontram-se no ambiente de trabalho do sistema onde o relatório foi gerado:

- `/mnt/data/trabalho.c`
- `/mnt/data/imageRGB.c`
- `/mnt/data/AED_2526_TRAB_1_FICHEIROS_ALUNOS.zip`

Nota: podes abrir o ficheiro `trabalho.c` directamente aqui: sandbox:/mnt/data/trabalho.c.

10.2 Apêndice B — Script de teste sugerido

```
// test_mem.c
#include "imageRGB.h"
#include <stdio.h>

int main(void) {
  ImageInit();
  // Cria imagem
  Image a = ImageCreate(300,200);
  // testa copy
  Image b = ImageCopy(a);
  ImageDestroy(&b);
  // testa rotate
  Image r90 = ImageRotate90CW(a);
  Image r180 = ImageRotate180CW(a);
  // cleanup
  ImageDestroy(&r90);
  ImageDestroy(&r180);
  ImageDestroy(&a);
  return 0;
}
```

10.3 Apêndice C — Exemplo de saída do Valgrind (formato)

```
==12345== HEAP SUMMARY:
==12345==     definitely lost: 0 bytes in 0 blocks
==12345==     indirectly lost: 0 bytes in 0 blocks
==12345==       possibly lost: 0 bytes in 0 blocks
==12345==     still reachable: 72,704 bytes in 14 blocks
==12345==         suppressed: 0 bytes in 0 blocks
==12345== Rerun with --leak-check=full to see details.
```

(Nota: a saída real será colada aqui depois de executar Valgrind).

11 Conclusões

- A arquitetura do TAD é adequada e modular; a abordagem LUT+matrix é correta para tarefas de segmentação e manipulação.
- A função `ImageIsEqual` é linear no pior caso; o protocolo experimental previsto deve confirmar a linearidade e fornecer constantes empíricas.
- As versões iterativas de *region growing* são preferíveis para imagens grandes por robustez.
- Estão definidas práticas para assegurar ausência de fugas de memória, e um plano claro para validação dinâmica que pode ser executado assim que o projeto completo for disponibilizado.