TAD imageRGB

Grupo (P8B)

Miguel Soares Sousa - 125624

Margarida Simões Pinheiro - 125011

Introdução

Ao longo deste trabalho, foram analisados e implementados diversos algoritmos associados ao Tipo Abstrato de Dados (TAD) imageRGB, cujo objetivo é representar e manipular imagens a cores através de uma Look-Up Table (LUT) e de uma matriz de índice de rótulos

O TAD permite representar imagensRGB de forma compacta e eficiente, onde cada píxel é identificado por um índice que referencia uma cor na tabela de cores.

O foco principal deste relatório é a análise e a implementação das seguintes funções: ImageIsEqual, ImageRegionFillingRecursive, ImageRegionFillingWithSTACK e ImageRegionFillingWithQUEUE, sendo que a função ImageIsEqual tem o objetivo de comparar duas imagens independentemente das LUTs associadas e as outras 3 funções têm o objetivo de preencher regiões de formas diferentes.

Funções Implementadas

a. ImageIsEqual()

Esta função verifica se duas imagens (img1, img2) são iguais visualmente.

Como uma das particularidades do TAD imageRGB é o facto de a mesma cor RGB poder corresponder a rótulos distintos em imagens diferentes a comparação não pode ser feita diretamente sobre os índices da matriz, e sim sobre os valores RGB armazenados na LUT.

Portanto, esta função percorre todos os píxels das imagens, comparando os valores RGB reais de cada píxel obtidos na LUT para verificarmos se são equivalentes. Caso seja encontrada uma diferença, é retornado 0, senão retorna 1.

Sobre a complexidade, tem complexidade linear visto que o número de operações realizadas depende do número total de píxels, *N*, ou seja a função tem de percorrer todos os píxels no pior caso. No melhor caso, a função retornaria logo 1 pois as imagens teriam dimensões diferentes.

$$O(height \times width) = O(N)$$

b. ImageRegionFillingRecursive()

O objetivo desta função, assim como de todas as funções filling é preencher uma região contígua da imagem a partir de um píxel base (u, v), alterando o seu rótulo e o de todos os

píxeis vizinhos que tenham o mesmo valor original recorrendo à abordagem *Flood Fill*, que garante que cada píxel não seja preenchido mais que uma única vez.

A implementação recorre à função auxiliar FloodFillRecursive, que por si recorre a um método recursivo de depth-first search (DFS). Cada chamada verifica se o píxel é válido e se possui o mesmo label que o píxel original e então a função é chamada para os vizinhos

A complexidade é linear, pois o tempo demorado a percorrer a imagem depende do número de píxels. Também é importante saber que se a imagem tiver um grande número de píxels, pode ocorrer um caso de stack overflow.

c. ImageRegionFillingWithSTACK()

Esta função implementa o mesmo conceito de preencher uma região mas substitui as chamadas recursivas por um stack, simulando o comportamento recursivo de forma iterativa.

Utilizando a TAD píxelCoordsStack, que empilha inicialmente a coordenada dada (u, v), a cada iteração o topo do stack é removido, atualizado e os seus vizinhos válidos são adicionados, repetindo até que a pilha fique vazia.

A complexidade é idêntica à versão recursiva e tem ainda a vantagem de alocar memória dinamicamente usando “malloc”, o que evita o stack overflow.

d. ImageRegionFillingWithQUEUE()

Esta função usa o algoritmo “flood filling” com uma queue, com o objetivo de preencher uma região contígua da imagem que possui o mesmo rótulo (cor original) a partir de um píxelel semente, substituido-a por um novo rótu.

É criada uma queue com a capacidade de armazenar o número total de píxels da imagem (width * height), a primeira coordenada é enfileirada na queue (se não for igual ao novo rótulo), e a função vai enfileirando coordenadas adjacentes e incrementando um contador para cada píxel modificado até a queue ficar vazia.

No final a função retorna o número de píxels que foram rotulados (contador).

Esta função tem complexidade linear (O(N)), pois no pior caso possível a função percorre todos os píxels da imagem.

$$O(height \times width) = O(N)$$

Análise da função ImageIsEqual()

A função ImageIsEqual() verifica se 2 imagens são iguais (como referido anteriormente), retornando 1 se este for o caso e 0 se forem diferentes. Para analisar a sua complexidade em função do número de comparações efetuadas envolvendo os píxels das imagens usamos dois métodos diferentes: análise experimental vs análise formal, e no fim comparamos os resultados para ver se coincidem.

- **Análise experimental**

Para a análise experimental foram realizados testes computacionais com imagens de diferentes tamanhos, usando um contador que é incrementado todas as vezes que os píxels das imagens são comparados. Só foram analisados casos em que as duas imagens têm tamanhos iguais, pois quando isto não acontece a função retorna 0 imediatamente e nenhuma comparação é efetuada.

Para o 1º caso observamos, para diferentes tamanhos, quantas comparações são efetuadas quando as imagens comparadas são iguais:

Nº de Pixels	Nº de Comparações	Retorna
1	1	1
4	4	1
9	9	1
16	16	1
25	25	1
36	36	1
49	49	1
64	64	1
81	81	1
100	100	1
121	121	1
144	144	1
169	169	1
196	196	1
225	225	1

Como é possível observar, o número de comparações cresce proporcionalmente ao número de píxels (nº de píxels = nº de comparações). Assim, concluímos que a complexidade neste caso é linear, O(N).

Para o 2º caso observamos quantas comparações são efetuadas quando o primeiro píxel de ambas as imagens difere:

Nº de Pixels	Nº de Comparações	Retorna
1	1	0
4	1	0
9	1	0
16	1	0
25	1	0
36	1	0
49	1	0
64	1	0
81	1	0
100	1	0
121	1	0
144	1	0
169	1	0
196	1	0
225	1	0

Como podemos ver o nº de comparações mantém-se constante à medida que o nº de píxels aumenta, logo a complexidade neste caso é constante, O(1).

Pode se concluir que o pior caso é quando as imagens são iguais (1º caso) pois todos os píxels são comparados, enquanto o melhor caso é aquele em que as imagens são diferentes logo no primeiro píxel (2ºcaso) pois só é feita é uma comparação e a função retorna 0 imediatamente.

- **Análise Formal**

  Tendo em conta os mesmo 2 casos que foram usados na análise experimental, foi feita a análise formal onde se obteve uma expressão matemática para o número de comparações efetuadas em função do tamanho das imagens. Nota: N = número total de píxels da imagem (height x width = nº píxels).

  1º caso (imagens iguais):

$$\sum_{i=0}^{height-1} \sum_{j=0}^{width-1} 1 = \sum_{i=0}^{height-1} width = height \times width -> O(height \times width) -> O(N)$$

  Como se pode observar a complexidade é linear (O(N)).

  2ºcaso (imagens diferentes logo no primeiro píxel):

$$\sum_{i=0}^{0}\sum_{j=0}^{0} 1 = 1 -> O(1)$$

Como se pode verificar a complexidade é constante (O(1)).

- **Comparação: análise experimental vs formal**

  Os resultados obtidos na análise experimental coincidem com os obtidos na análise formal.

  Em ambos os métodos verificou-se que no pior caso (imagens iguais) o número de comparações cresce proporcionalmente ao número total de píxels, resultando numa complexidade linear(O(N)) e no melhor caso (imagens diferentes logo no primeiro píxel), apenas é feita uma comparação, pelo que a complexidade é constante O(1).

  Assim, a análise experimental confirmou na prática o comportamento previsto teoricamente, validando a correção da análise formal e demonstrando que a função ImageIsEqual() tem um comportamento previsível e consistente em termos de complexidade.

Comparação entre Estratégias de funções que efetuam a segmentação

Para avaliar as diferentes abordagens que as funções utilizam, foram analisados os seus comportamentos em termos de complexidade, memória e robustez. As seguintes tabelas resumem as principais diferenças entre o método recursivo, usando stack e queue.

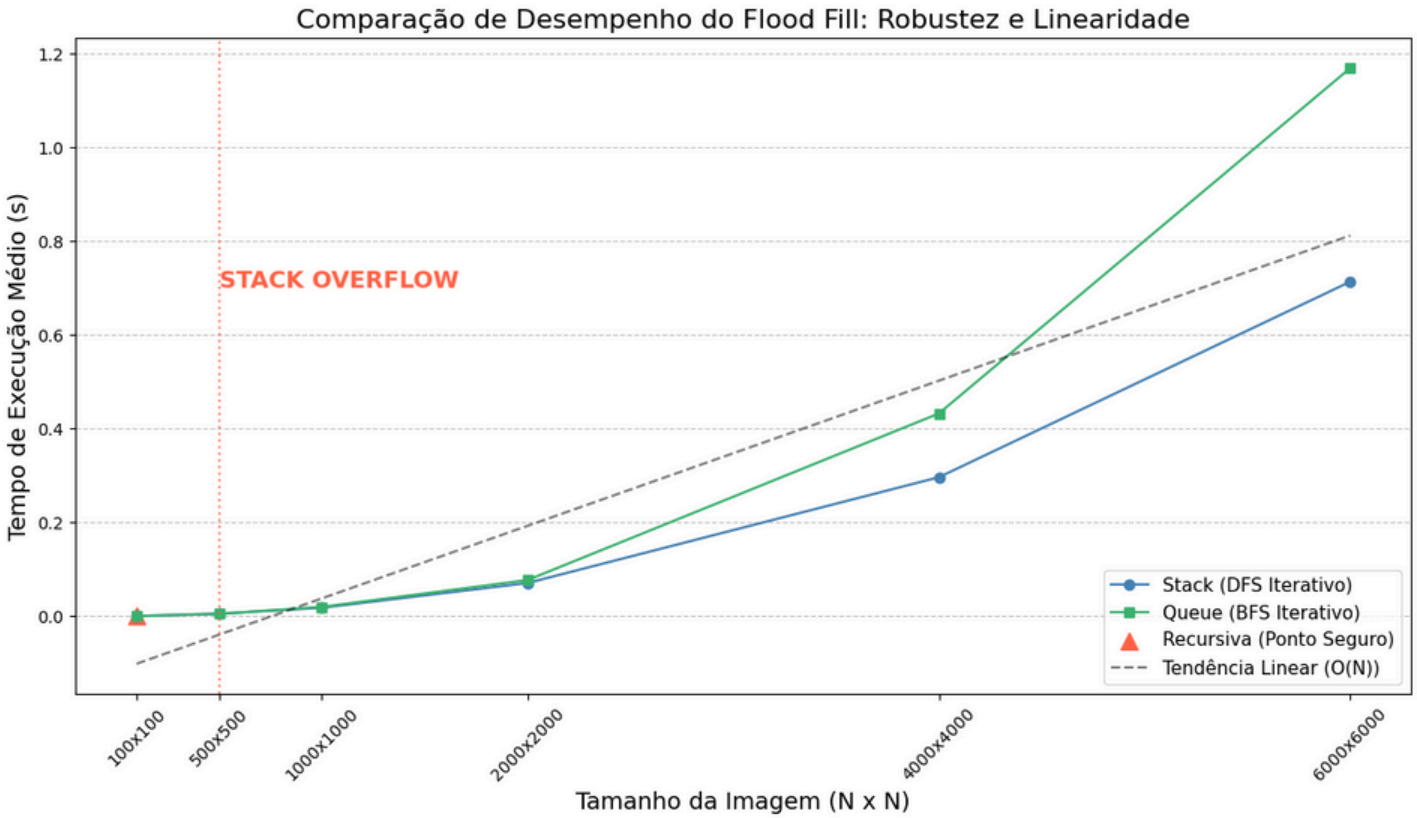
Estrategia	Complexidade	Estrutura de memória	Risco de Stack Overflow
Recursiva	O(N)	call stack	alto
Stack	O(N)	heap	nulo
Queue	O(N)	heap	nulo

Tabela 1: Comparação em termos de complexidade e gestão de memória

Estrategia	Tipo de Busca	Vantagem Principal	Desvantagem Principal
Recursiva	DFS	implementação Simples e intuitiva	Limitada pela profundidade da pilha
Stack	DFS	Maior controlo de memória	Ordem de preenchimento menos intuitiva
Queue	BFS	Ordem de preenchimento mais previsível e uniforme	Requer estrutura de dados adicional

Tabela 2: Comparação em termos de comportamento algorítmico

Para ajudar na análise dos 3 métodos diferentes, foram feitos testes computacionais e dos seus valores foi feito o gráfico visto a baixo.



A abordagem recursiva é a mais simples de implementar, porém torna-se inviável para imagens de grande dimensão assim como se pode analisar, visto que dá stack overflow para imagens de 500x500.

As versões com estruturas de stack e queue eliminam o problema da recursiva, com complexidade idêntica mas usando um diferente padrão de preenchimento. Embora ambas as abordagens iterativas sejam viáveis, os testes demonstram um desempenho melhor na implementação com stack.

Conclusão

O desenvolvimento e análise do TAD imageRGB confirmaram a sua eficácia na representação e manipulação de imagens a cores. A função ImageIsEqual() demonstrou a complexidade prevista: O(N) no pior caso e O(1) no melhor caso.

As três abordagens filling , apresentam complexidade temporal equivalente, O(N). Contudo, a análise prática revelou que a versão recursiva tem risco de stack overflow para imagens maiores. Concluímos, então, que as versões iterativas são a melhor opção em contextos práticos.

Referências

- Guião do 1º projeto de AED - 2025/2026
- Documentação fornecida nos ficheiros já feitos
- Slides das aulas teóricas de AED
- GeeksforGeeks. (n.d.). *Flood Fill Algorithm – Explained with Examples.* Disponível em: https://www.geeksforgeeks.org/flood-fill-algorithm/