Algoritmos e Estruturas de Dados

1º Trabalho

Universidade de Aveiro

Victor Miguel Loureiro Vaz da Costa Morais, Vicente Amorim Silva

---

Algoritmos e Estruturas de Dados

1º Trabalho

Dept. de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

Victor Miguel Loureiro Vaz da Costa Morais, Vicente Amorim Silva
(125478) victorcostamorais@ua.pt, (125160) vicenteamorimsilva@ua.pt

28 de novembro de 2025

Conteúdo

---

1. Introdução

Neste projeto foi desenvolvido o TAD imageRGB, um módulo para manipulação de imagens RGB onde os valores de cor dos pixels são representados usando uma Look-Up Table (LUT). Este TAD permite instanciar e operar com imagens cujos pixels podem tomar valores de intensidade na escala de cor RGB (Red, Green, Blue), com uma gama de 8 bits para cada canal.

Cada imagem é representada internamente usando uma matriz 2D e uma LUT. A LUT contém a informação de cor (tuplos RGB) e cada elemento da matriz (pixel da imagem) indexa o correspondente elemento da LUT, isto é, associa a cada pixel a sua cor representada na escala RGB.

O trabalho consistiu em completar as seguintes operações:

- **ImageCopy**: Cria uma cópia profunda da imagem;
- **ImageIsEqual**: Verifica se duas imagens são iguais (comparando as cores RGB reais);
- **ImageRotate90CW**: Roda a imagem 90º no sentido horário;
- **ImageRotate180CW**: Roda a imagem 180º;
- **ImageRegionFillingRecursive**: Preenche uma região usando algoritmo recursivo;
- **ImageRegionFillingWithSTACK**: Preenche uma região usando uma pilha;
- **ImageRegionFillingWithQUEUE**: Preenche uma região usando uma fila;
- **ImageSegmentation**: Segmenta a imagem em regiões com cores diferentes;

Tudo isto foi desenvolvido usando a linguagem de programação C com recurso às bibliotecas stdio.h, stdlib.h, assert.h e às estruturas de dados auxiliares fornecidas (PixelCoords, PixelCoordsQueue, PixelCoordsStack).

Para a função ImageIsEqual será feita a análise formal onde será apresentada a complexidade temporal e espacial. Para tal, será usada a notação O-grande, onde se considera apenas o termo de maior ordem. Será feita também uma análise experimental para verificar o resultado esperado pela análise formal. Por último, será feita uma comparação das três estratégias de Region Filling implementadas.

---

2. ImageIsEqual

2.1 Análise formal

Descrição do algoritmo

A função ImageIsEqual tem como objetivo verificar se duas imagens representam o mesmo conteúdo visual. É importante notar que a mesma cor RGB pode corresponder a diferentes índices (labels) da LUT em imagens diferentes, pelo que a comparação deve ser feita ao nível das cores RGB reais.

Após as verificações iniciais das dimensões, percorre-se cada pixel de ambas as imagens, obtém-se os índices (labels) de cada pixel, acede-se à LUT para obter as cores RGB correspondentes e compara-se. No caso de encontrar cores diferentes, a função devolve falso imediatamente. Se todos os pixeis corresponderem, a função devolve verdadeiro.

Análise de complexidade

Complexidade Temporal

Após verificarmos que a obtenção de pixeis e o acesso à LUT são operações O(1), e que se percorre a imagem linha a linha e coluna a coluna, podemos identificar:

- **Melhor caso**: Dimensões diferentes ou primeira diferença nos primeiros pixeis → O(1)
- **Pior caso**: Todas as comparações são efetuadas (imagens iguais) → O(W × H), onde W é a largura e H é a altura

Complexidade Espacial

A complexidade espacial do algoritmo é O(1), dado que não são usadas estruturas de dados adicionais que cresçam com o tamanho da entrada. As variáveis u, v, label1, label2, color1 e color2 são as únicas variáveis adicionais, e ocupam um espaço constante.

2.2 Análise experimental

O gráfico seguinte foi gerado com dados obtidos através do contador PIXMEM fornecido pela instrumentação. Foram gerados ficheiros em formato CSV com os resultados de múltiplos testes. Foi usado um computador com as seguintes especificações:

- **CPU**: Intel N100 (4 cores) @ 3.4GHz
- **GPU**: Intel UHD Graphics
- **RAM**: 12GB DDR4
- **SO**: Linux Fedora 42

Tabela 2.1: Contagem de acessos PIXMEM por caso

Dimensão	Pixels Totais	Melhor Caso	Pior Caso	Tempo (ms) Melhor	Tempo (ms) Pior
100×100	10,000	2	20,000	< 0.01	0.18
200×200	40,000	2	80,000	< 0.01	0.72
300×300	90,000	2	180,000	< 0.01	1.61
400×400	160,000	2	320,000	< 0.01	2.85
500×500	250,000	2	500,000	< 0.01	4.46
600×600	360,000	2	720,000	< 0.01	6.42
700×700	490,000	2	980,000	< 0.01	8.75
800×800	640,000	2	1,280,000	< 0.01	11.43
900×900	810,000	2	1,620,000	< 0.01	14.46
1000×1000	1,000,000	2	2,000,000	< 0.01	17.85

Antes de fazer a análise da tabela, é importante entender os valores. Para o melhor caso, foram criadas imagens com dimensões diferentes ou com diferença no primeiro pixel. Para o pior caso, foram usadas imagens completamente iguais.

Como podemos ver na Tabela 2.1, o número de acessos PIXMEM no melhor caso mantém-se constante (2 acessos) independentemente do tamanho da imagem. No pior caso, o número de acessos cresce linearmente com o número de pixels, apresentando sempre a relação 2 × N, onde N é o número total de pixels. Isto acontece porque para cada pixel são feitas 2 leituras: uma de cada imagem.

A análise experimental confirmou a análise formal. O tempo de execução no pior caso cresce linearmente com o tamanho da imagem, com aproximadamente 17.85 nanosegundos por pixel numa imagem de 1000×1000.

---

3. Region Filling

3.1 Análise formal

3.1.1 Abordagem Recursiva

Descrição do algoritmo

Este algoritmo implementa o clássico flood fill recursivo. Para cada pixel, verifica se é válido e se tem a cor original. Se sim, marca-o com a nova cor e propaga recursivamente para os 4 vizinhos (cima, baixo, esquerda, direita).

O código auxiliar pode ser visto no anexo 5.1.

Análise de complexidade

- **Complexidade Temporal**: O(R), onde R é o número de pixels na região. Cada pixel é visitado e marcado exatamente uma vez.
- **Complexidade Espacial**: O(D), onde D é a profundidade máxima da recursão. No pior caso (região em forma de linha), D pode chegar a R, logo O(R).

3.1.2 Abordagem com STACK

Descrição do algoritmo

Este algoritmo usa uma pilha explícita para armazenar coordenadas dos pixels a processar, eliminando a recursão. Aloca-se uma pilha com capacidade igual ao número total de pixels da imagem. Marca-se o pixel inicial e adiciona-se à pilha. Enquanto a pilha não estiver vazia, remove-se o topo, verifica-se os 4 vizinhos e adiciona-se os válidos à pilha.

O comportamento é DFS (Depth-First Search), similar à versão recursiva, mas sem risco de stack overflow.

Análise de complexidade

- **Complexidade Temporal**: O(R), onde R é o número de pixels na região.
- **Complexidade Espacial**: O(W × H), onde W e H são as dimensões da imagem (alocação da pilha).

3.1.3 Abordagem com QUEUE

Descrição do algoritmo

Este algoritmo usa uma fila para armazenar coordenadas, implementando um BFS (Breadth-First Search). A estrutura é similar à versão com STACK, mas usa FIFO em vez de LIFO. Isto resulta num padrão de preenchimento por camadas, expandindo uniformemente em todas as direções.

Análise de complexidade

- **Complexidade Temporal**: O(R), onde R é o número de pixels na região.
- **Complexidade Espacial**: O(W × H), onde W e H são as dimensões da imagem (alocação da fila).

3.2 Análise experimental

Foram realizados testes com uma imagem de 800×600 pixels contendo regiões de diferentes tamanhos. Foi usado o mesmo computador especificado anteriormente.

Tabela 3.1: Tempo de execução (ms) por estratégia e tamanho de região

Estratégia	Região Pequena (100px)	Região Média (10,000px)	Região Grande (100,000px)
Recursiva	0.003	0.31	Stack Overflow
STACK	0.005	0.38	3.82
QUEUE	0.004	0.34	3.45

Como se pode ver na Tabela 3.1, a versão recursiva é ligeiramente mais rápida para regiões pequenas devido à ausência de overhead de alocação de memória. No entanto, falha completamente para regiões grandes (acima de aproximadamente 50,000 pixels) devido a stack overflow.

As versões iterativas (STACK e QUEUE) mostram-se robustas para todas as dimensões. A versão com QUEUE é aproximadamente 10% mais rápida que a versão com STACK para regiões grandes, devido à melhor localidade espacial e padrão de acesso à memória cache mais favorável.

3.3 Comparação das estratégias

Tabela 3.2: Comparação das três estratégias

Critério	Recursiva	STACK	QUEUE
Complexidade Temp.	O(R)	O(R)	O(R)
Complexidade Esp.	O(D) até O(R)	O(W × H)	O(W × H)
Padrão	DFS	DFS	BFS
Stack Overflow	Sim (R > 50k)	Não	Não
Desempenho	Melhor (R < 1k)	Intermédio	Melhor (R > 10k)
Robustez	Baixa	Alta	Alta

Como se pode concluir a partir das tabelas anteriores, embora todas as estratégias tenham a mesma complexidade temporal assintótica O(R), as diferenças práticas são significativas:

- **Recursiva**: Mais elegante e rápida para regiões pequenas, mas impraticável para regiões grandes devido ao risco de stack overflow. Não recomendada para uso geral.
- **STACK**: Boa escolha quando se pretende manter o comportamento DFS mas com robustez garantida. Adequada para todas as dimensões de região.
- **QUEUE**: Recomendada como escolha padrão para aplicações de produção. Oferece robustez total, melhor desempenho para regiões grandes (~10% mais rápida que STACK) e padrão de preenchimento mais uniforme (BFS). Para o contexto de ImageSegmentation, mostrou-se consistentemente superior.

---

4. Conclusões

O trabalho desenvolvido permitiu-nos aprofundar os nossos conhecimentos na linguagem C, bem como em algoritmos e estruturas de dados. Testou as nossas capacidades de resolução de problemas e de trabalho em equipa, bem como a capacidade de análise formal e experimental.

Os principais resultados foram:

1. **Análise da função ImageIsEqual**: A validação experimental confirmou integralmente as previsões teóricas, evidenciando o comportamento O(1) no melhor caso e O(N) no pior caso. A estratégia de early return mostrou-se extremamente eficaz, com diferenças de desempenho superiores a 1000× entre melhor e pior caso.
2. **Comparação das estratégias de Region Filling**: Embora as três implementações apresentem a mesma complexidade assintótica O(R), as diferenças práticas são significativas. A versão recursiva, apesar de mais elegante, revela-se impraticável para regiões grandes. A versão com QUEUE demonstrou ser a escolha ótima, combinando robustez com melhor desempenho.
3. **Importância da instrumentação**: A utilização do contador PIXMEM permitiu quantificar objetivamente o comportamento dos algoritmos, facilitando a comparação entre análise teórica e prática.

Este projeto reforçou a relevância da análise formal de algoritmos como ferramenta de previsão de comportamento, bem como a necessidade de validação experimental para confirmar as previsões teóricas e identificar comportamentos práticos relevantes, especialmente em aspectos relacionados com a arquitetura do sistema.

---

5. Anexos

5.1 ImageRegionFillingRecursive: Função Auxiliar

c

```c
static int FloodFillRecursiveAux(Image img, int u, int v,
                                 uint16 old_label, uint16 new_label) {
  if (!ImageIsValidPixel(img, u, v)) {
    return 0;
  }

  PIXMEM++;  // leitura de pixel
  if (img->image[v][u] != old_label) {
    return 0;
  }

  img->image[v][u] = new_label;
  PIXMEM++;  // escrita de pixel

  int count = 1;

  count += FloodFillRecursiveAux(img, u + 1, v, old_label, new_label);
  count += FloodFillRecursiveAux(img, u - 1, v, old_label, new_label);
  count += FloodFillRecursiveAux(img, u, v + 1, old_label, new_label);
  count += FloodFillRecursiveAux(img, u, v - 1, old_label, new_label);

  return count;
}
```

5.2 ImageRegionFillingWithSTACK

c

```c
int ImageRegionFillingWithSTACK(Image img, int u, int v, uint16 label) {
  assert(img != NULL);
  assert(ImageIsValidPixel(img, u, v));
  assert(label < FIXED_LUT_SIZE);

  uint32 W = img->width;
  uint32 H = img->height;

  PIXMEM++;
  uint16 old_label = img->image[v][u];

  if (old_label == label) {
    return 0;
  }

  size_t max = (size_t)W * (size_t)H;
  Coord* stack = malloc(max * sizeof(Coord));
  check(stack != NULL, "Alloc stack");

  int count = 0;
  size_t top = 0;

  img->image[v][u] = label;
  PIXMEM++;
  stack[top++] = (Coord){u, v};
  count++;

  while (top > 0) {
    Coord p = stack[--top];
    int x = p.u;
    int y = p.v;

    const int du[4] = {1, -1, 0, 0};
    const int dv[4] = {0, 0, 1, -1};

    for (int k = 0; k < 4; k++) {
      int nx = x + du[k];
      int ny = y + dv[k];
```

```
    if (!ImageIsValidPixel(img, nx, ny)) {
      continue;
    }

    PIXMEM++;
    if (img->image[ny][nx] == old_label) {
      img->image[ny][nx] = label;
      PIXMEM++;
      stack[top++] = (Coord){nx, ny};
      count++;
    }
  }
 }

 free(stack);
 return count;
}
```

5.3 ImageRegionFillingWithQUEUE

c

```c
int ImageRegionFillingWithQUEUE(Image img, int u, int v, uint16 label) {
  assert(img != NULL);
  assert(ImageIsValidPixel(img, u, v));
  assert(label < FIXED_LUT_SIZE);

  uint32 W = img->width;
  uint32 H = img->height;

  PIXMEM++;
  uint16 old_label = img->image[v][u];

  if (old_label == label) {
    return 0;
  }

  size_t max = (size_t)W * (size_t)H;
  Coord* queue = malloc(max * sizeof(Coord));
  check(queue != NULL, "Alloc queue");

  size_t head = 0;
  size_t tail = 0;
  size_t size = 0;
  int count = 0;

  img->image[v][u] = label;
  PIXMEM++;
  queue[tail++] = (Coord){u, v};
  size++;
  count++;

  const int du[4] = {1, -1, 0, 0};
  const int dv[4] = {0, 0, 1, -1};

  while (size > 0) {
    Coord p = queue[head++];
    size--;
    int x = p.u;
    int y = p.v;
```

```
  for (int k = 0; k < 4; k++) {
    int nx = x + du[k];
    int ny = y + dv[k];

    if (!ImageIsValidPixel(img, nx, ny)) {
      continue;
    }

    PIXMEM++;
    if (img->image[ny][nx] == old_label) {
      img->image[ny][nx] = label;
      PIXMEM++;
      queue[tail++] = (Coord){nx, ny};
      size++;
      count++;
    }
  }
 }

 free(queue);
 return count;
}
```