Algoritmos e Estruturas de Dados
1º Projeto

Universidade de Aveiro

Diogo Carvalho, Tiago Garcia

Algoritmos e Estruturas de Dados
1º Projeto

Dept. de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

Diogo Carvalho, Tiago Garcia
(113221) diogo.tav.carvalho@ua.pt, (114184) tiago.rgarcia@ua.pt

26 de novembro de 2023

Conteúdo

1 Introdução **2**

2 ImageBlur **3**
2.1 Análise formal . . . 3
2.1.1 Primeira abordagem . . . 3
2.1.2 Segunda abordagem . . . 4
2.2 Análise experimental . . . 5
2.2.1 Primeira abordagem . . . 5
2.2.2 Segunda abordagem . . . 6
2.2.3 Comparação . . . 6
2.3 Algoritmo optimizado da ImageBlur . . . 7
2.3.1 Imagem Integral . . . 7
2.3.2 Imagem desfocada . . . 8

3 ImageMatchSubImage e ImageLocateSubImage **10**
3.1 Análise formal . . . 10
3.1.1 ImageMatchSubImage . . . 10
3.1.2 ImageLocateSubImage . . . 10
3.2 Análise experimental . . . 11

4 Conclusões **12**

5 Anexos **13**
5.1 ImageBlur: Primeira Abordagem . . . 13
5.2 ImageBlur: Segunda Abordagem . . . 14

Capítulo 1

Introdução

Neste projeto será desenvolvida uma ferramenta de manipulação de imagens monocromáticas (apenas um canal de cor, neste caso apenas escala de cinzentos), que permitirá efetuar diversas operações:

- **ImageStats:** Mostra os valores mínimo e máximo dos pixeis da imagem;
- **ImageNegative:** Inverte os pixeis da imagem ($new\_value = max\_value - old\_value$);
- **ImageThreshold:** Aplica um limiar à imagem, de forma a que os pixeis com valor inferior ao limiar sejam convertidos para 0 e os restantes para o valor máximo;
- **ImageBrighten:** Aumenta ou diminui o brilho da imagem, multiplicando todos os pixeis por um fator ($fator > 1$ aumenta o brilho, $fator < 1$ diminui o brilho);
- **ImageMirror:** Espelha a imagem, horizontalmente;
- **ImageRotate:** Roda a imagem, 90º no sentido contrário ao dos ponteiros do relógio;
- **ImageCrop:** Corta a imagem, de acordo com as coordenadas do canto superior esquerdo e o tamanho do retângulo a cortar;
- **ImagePaste:** Cola uma imagem sobre outra a partir das coordenadas do canto superior esquerdo;
- **ImageBlend:** Cola uma imagem sobre outra, com um fator de transparência (entre 0 e 1), a partir das coordenadas do canto superior esquerdo;
- **ImageMatchSubImage:** Verifica se uma imagem está contida dentro de outra maior, a partir das coordenadas do canto superior esquerdo;
- **ImageLocateSubImage:** Procura uma imagem dentro de outra maior e devolve as coordenadas do canto superior esquerdo da primeira ocorrência;
- **ImageBlur:** Aplica um filtro de desfoque à imagem;

Tudo isto será desenvolvido usando a linguagem de programação C com recurso às bibliotecas stdio.h, stdlib.h, assert.h, ctype.h e errno.h.

Para as funções ImageLocateSubImage e ImageBlur será ainda feita a análise formal onde será apresentada a complexidade temporal e espacial de cada uma das funções. Para tal, será usada a notação O-grande, onde se considera apenas o termo de maior ordem. Será feita também uma análise experimental destas duas funções para verificar o resultado expectado pela análise formal. Por último, será descrito todo o algoritmo otimizado da função ImageBlur.

Capítulo 2

ImageBlur

2.1 Análise formal

2.1.1 Primeira abordagem

Descrição do algoritmo

Este algoritmo percorre todos os pixeis da imagem original e para cada pixel calcula a média dos pixeis num retângulo de desfocagem centrado no pixel atual. Para tal, vai iterar por todos os pixeis da imagem e para cada pixel vai iterar de novo por todos os pixeis que se encontram no retângulo de dimensões $2dx + 1 \times 2dy + 1$, faz a soma dos valores desses pixeis, divide pelo número de pixeis desse retângulo e atribui esse resultado ao pixel central.

Este algoritmo, por mais que seja simples, fica bastante ineficiente quando o retângulo de desfocagem é grande pois temos de calcular a soma de todos os pixeis do retângulo para cada pixel da imagem. Por mais que para imagens pequenas não seja uma diferença notável, à medida que os valores das variáveis aumenta, também o tempo necessário para o algoritmo aumenta imenso.

O código completo deste algoritmo pode ser visto no anexo 5.1.

Análise de complexidade

- **Complexidade Temporal**

  1. **Desfocagem:** Esta parte da função itera sobre todas as linhas ($img{\rightarrow}height$) e para cada linha itera sobre todos os pixeis da linha ($img{\rightarrow}width$). Para cada pixel, itera sobre todas as linhas $(2 * dy + 1)$ e para cada linha itera sobre todos os pixeis da linha $(2 * dx + 1)$. Assim, a complexidade temporal desta parte é $O(img{\rightarrow}height * img{\rightarrow}width * (2 * dy + 1) * (2 * dx + 1))$ ou $O(A_{img} * A_{blur})$.

  Assim, a complexidade temporal total da função é $O(img{\rightarrow}height * img{\rightarrow}width * (2 * dy + 1) * (2 * dx + 1))$ ou $O(A_{img} * A_{blur})$ que significa que a complexidade temporal escala linearmente tanto em função da àrea da imagem quanto em função da área do retângulo de desfocagem (independentes uma da outra).

- **Complexidade Espacial**

  1. **Criação da Nova Imagem:** Esta parte da função aloca memória para a nova imagem onde será guardada a imagem desfocada. Uma vez que a nova imagem tem as mesmas dimensões que a imagem original, a complexidade espacial desta parte é $O(img{\rightarrow}height * img{\rightarrow}width)$.
  2. **Desfocagem da Image:** Esta parte modifica a imagem criada, por isso não terá complexidade espacial associada.
  3. **Transferência da Nova Imagem:** Esta parte da função apenas transfere o ponteiro da nova imagem para o ponteiro da imagem original, por isso não terá complexidade espacial associada.

  Assim, a complexidade espacial total da função é $O(img{\rightarrow}height * img{\rightarrow}width)$ que significa que a complexidade espacial escala linearmente em função da área da imagem.

2.1.2 Segunda abordagem

Descrição do algoritmo

Este algoritmo passa por calcular as somas de todos os pixeis num retângulo entre $(0, 0)$ e (x, y) antes de começar a criar a imagem desfocada. Uma vez que teremos as somas todas calculadas, podemos calcular o valor de cada pixel da imagem desfocada com base nas somas calculadas anteriormente. Este algoritmo é mais eficiente que o anterior pois não temos de calcular as somas de cada pixel do retângulo de desfocagem, mas sim apenas uma vez para cada pixel da imagem original. Visto que este altoritmo é mais complexo, está explicado mais detalhadamente na secção 2.3.

O código completo deste algoritmo pode ser visto no anexo 5.2.

Análise de complexidade

- **Complexidade Temporal**

  1. **Criação da Imagem Integral:** Esta parte da função itera sobre todas as linhas ($img{\rightarrow}height$) e para cada linha itera sobre todos os pixeis da linha ($img{\rightarrow}width$). Assim, a complexidade temporal desta parte é $O(img{\rightarrow}height * img{\rightarrow}width)$.
  2. **Desfocagem da Image:** Tal como a anterior, esta parte também itera sobre todas as linhas ($img{\rightarrow}height$) e para cada linha itera sobre todos os pixeis da linha ($img{\rightarrow}width$). Uma vez que todas as operações dentro dos 'for loops' têm complexidade temporal constante, a complexidade temporal desta parte é $O(img{\rightarrow}height * img{\rightarrow}width)$.
  3. **Limpeza da Imagem Integral:** Esta parte da função tem sempre complexidade temporal constante, por isso a complexidade temporal desta parte é $O(1)$.

  Assim, a complexidade temporal total da função é $O(img{\rightarrow}height * img{\rightarrow}width) + O(img{\rightarrow}height * img{\rightarrow}width) + O(1)$ que simplifica para $O(img{\rightarrow}height * img{\rightarrow}width)$ ou $O(A_{img})$. Isto significa que a complexidade temporal escala linearmente em função da área da imagem, com esta abordagem, a área do retângulo de desfocagem não irá alterar a complexidade do algoritmo.

- **Complexidade Espacial**

  1. **Criação da Imagem Integral:** Esta parte aloca memória para a imagem integral que tem mais uma linha e uma coluna que a imagem original. Assim, a complexidade espacial desta parte é $O((img{\rightarrow}height + 1) * (img{\rightarrow}width + 1))$.
  2. **Desfocagem da Image:** Esta parte modifica a imagem original, por isso não terá complexidade espacial associada.

  Assim, a complexidade espacial total da função é $O((img{\rightarrow}height + 1) * (img{\rightarrow}width + 1))$ que simplifica para $O(img{\rightarrow}height * img{\rightarrow}width)$. Isto significa que a complexidade espacial escala linearmente em função da área da imagem.

2.2 Análise experimental

Os gráficos seguintes foram gerados usando a biblioteca *matplotlib* do *python*. Para gerar os gráficos foram primeiro gerados ficheiros com os dados obtidos através da biblioteca *time* da linguagem C. Depois foram gerados os gráficos a partir desses ficheiros. Para cada gráfico foram usados 12100 resultados. Foi usado um computador com as seguintes especificações:

- **CPU:** Intel Core i7-1165G7 (8) @ 4.7GHz
- **GPU:** Intel TigerLake-LP GT2 [Iris Xe Graphics]
- **RAM:** 16GB

2.2.1 Primeira abordagem

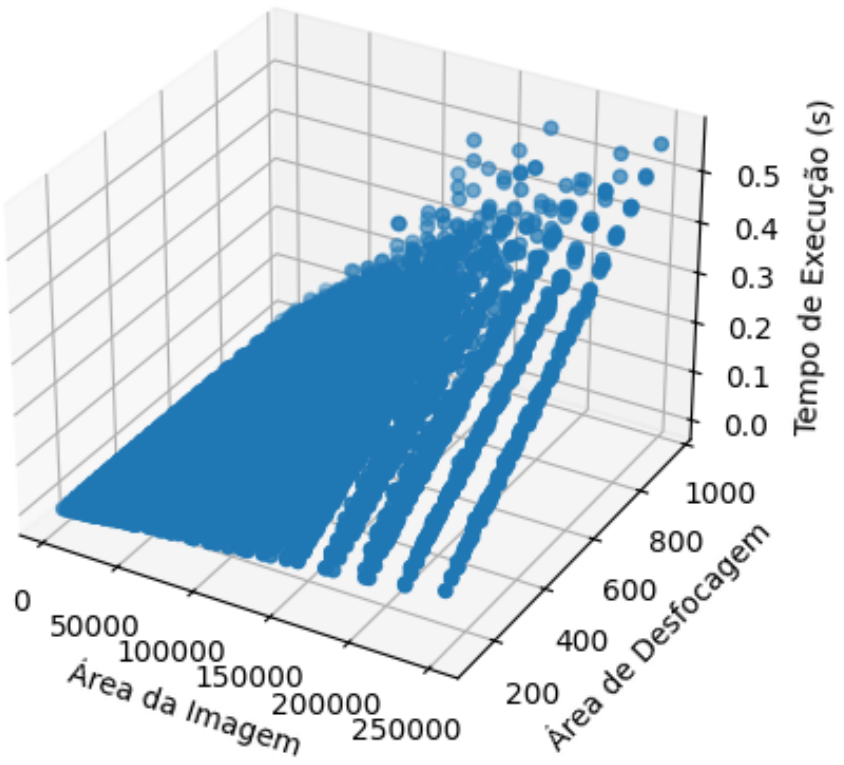

Figura 2.1: Gráfico 3D do tempo de execução da primeira abordagem.

Como podemos ver na figura 2.1, o tempo de execução da primeira abordagem é linear em função do tamanho da imagem e linear em função da área de desfocagem o que implica que o total seja exponencial em função de ambos. Isto para valores pequenos pode não se notar muito mas à medida que queremos trabalhar com imagems maiores ou filtros maiores, este algoritmo torna-se inviável.

A equação que descreve o tempo de execução deste algoritmo é a seguinte:

$$T(A_{img}, A_{blur}) = 2.403 \cdot 10^{-9} \cdot A_{img} \cdot A_{blur} + 9.011 \cdot 10^{-8} \cdot A_{img} - 1.05 \cdot 10^{-7} \cdot A_{blur} \tag{2.1}$$

2.2.2 Segunda abordagem

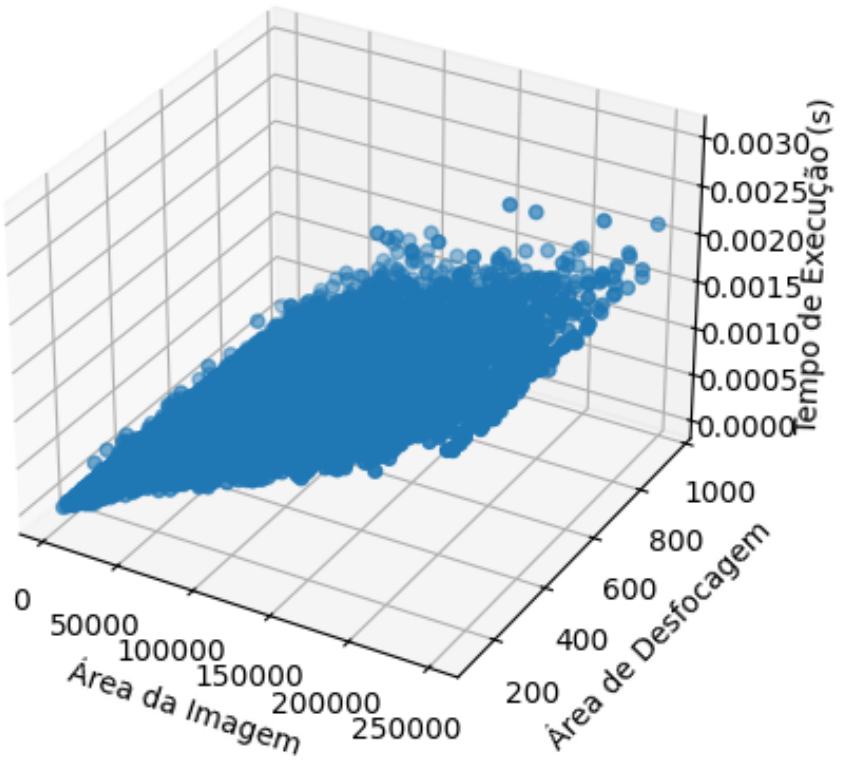

Figura 2.2: Gráfico 3D do tempo de execução da segunda abordagem.

Como podemos ver na figura 2.2, o tempo de execução da segunda abordagem é linear em função do tamanho da imagem. Podemos também ver que a área de desfocagem não tem influência no tempo de execução. Isto permite conseguir trabalhar com imagens maiores sem ter problemas com ineficiência do programa bem como usar qualquer tamanho de filtro sem alterar de forma alguma o desempenho do programa.

A equação que descreve o tempo de execução deste algoritmo é a seguinte:

$$T(A_{img}) = 8.296 \cdot 10^{-9} \cdot A_{img} \tag{2.2}$$

2.2.3 Comparação

Como se pode concluir a partir dos gráficos anteriores, a segunda abordagem é muito mais eficiente que a primeira. O pricipal motivo para isso é que, embora ambas as abordagens usem os mesmos ciclos principais, na primeira abordagem esse ciclo contém ainda um segundo dentro dele, o que vai fazer com que o tempo execução para o mesmo tamanho de imagem seja sempre superior ao da segunda abordagem visto que o tempo da primeira é sempre incrementado pelo tamanho do filtro coisa que não acontece com o segundo algoritmo.

2.3 Algoritmo optimizado da ImageBlur

2.3.1 Imagem Integral

Considerando img como uma matriz $m \times n$, onde $m = img \rightarrow height$ e $n = img \rightarrow width$, vamos gerar uma matriz $m+1 \times n+1$ onde a primeira linha e a primeira coluna vão ser zeros, a partir daí, o elemento (y, x) desta matriz irá corresponder ao elemento $(y-1, x-1)$ da matriz da imagem com $x \in [1, n]$ e $y \in [1, m]$. Esta nova imagem será armazenada como um array de $m+1 * n+1$ elementos, com 32 bits cada (visto que será um array de inteiros).

Para inicializar a matriz, usamos as linhas de código:

```
int integral_width = img->width + 1;
int integral_height = img->height + 1;
int* integral = (int*)calloc(integral_width * integral_height, sizeof(int));
```

Agora vamos ter de completar o resto das células. Cada célula da nova matriz será igual à soma de todos os pixeis que estão acima e à esquerda do mesmo.

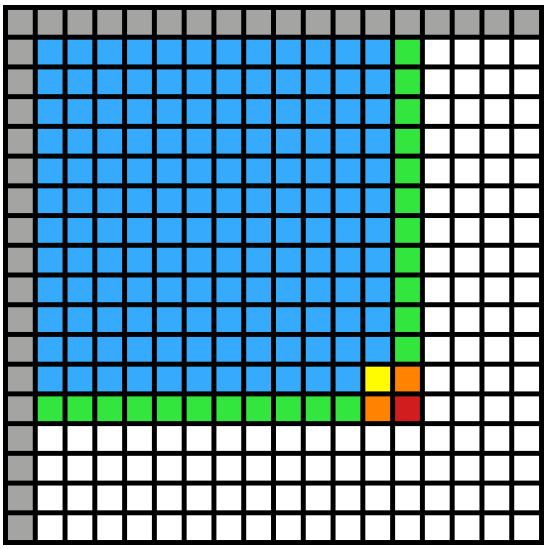

Figura 2.3: Representação da matriz integral

Como podemos ver na figura acima, para calcular o pixel representado a **vermelho** ($I_{(y,x)}$), teremos de adicionar ao pixel original da imagem, depois, o elemento representado acima a **laranja** ($I_{(y-1,x)}$) que já contém todos os pixeis acima e à esquerda do mesmo (representados a **verde** acima do pixel mencionado), e por fim, teremos de adicionar o elemento representado à esquerda também a **laranja** ($I_{(y,x-1)}$) que contém todos os pixeis na linha horizontal à esquerda do mesmo (representados a **verde** à esquerda do pixel mencionado). Uma vez que o pixel $I_{(y,x-1)}$, também contém a área acima dele (parte já incluída pelo pixel $I_{(y-1,x)}$), então teremos de remover a parte duplicada (área representada a **azul**) que corresponde ao valor do pixel representado a **amarelo** ($I_{(y-1,x-1)}$). Com isto, podemos concluir que a expressão para cálcular a área a atribuir a um determinado pixel será:

$$I_{(y,x)} = O_{(y-1,x-1)} + I_{(y-1,x)} + I_{(y,x-1)} - I_{(y-1,x-1)} \tag{2.3}$$

De notar que as células representadas a **cinzento** serão as células cujo valor do elemento será sempre zero.

Para calcular todas estas células da matriz, usamos as linhas de código:

```
for (int y = 1; y < integral_height; y++) {
    for (int x = 1; x < integral_width; x++) {
        integral[y * integral_width + x] = ImageGetPixel(img, x - 1, y - 1)
            + integral[(y - 1) * integral_width + x]
            + integral[y * integral_width + (x - 1)]
            - integral[(y - 1) * integral_width + (x - 1)];
    }
}
```

2.3.2 Imagem desfocada

Uma vez que já temos a imagem integral, podemos agora calcular a imagem desfocada. Para isso, vamos percorrer a imagem integral e para cada pixel, teremos de efetuar o seguinte procedimento:

1. **Calcular os cantos do retângulo de blur**

   Vamos calcular a soma dos pixeis que estão dentro do retângulo de dimensões $2dx+1$ e $2dy+1$ para o comprimento e largura, respetivamente, centrado no pixel atual. Poderemos obter esse retângulo usando o canto superior esquerdo bem como o canto inferior direito.

   (a) **Canto superior esquerdo**

   Para obter cada uma das coordenadas deste ponto, teremos de subtrair a cada coordenada do pixel, os valores do dy e dx, dy para a borda superior e dx para a borda da esquerda. No caso do resultado de uma dessas coordenadas ultrapassar um dos limites mínimos do retângulo, então esta coordenada deverá ser o limite mínimo. Este limite normalmente seria 0 mas visto que na imagem integral temos uma linha e uma coluna extra em cima e à esquerda, então temos de somar 1 a esses limites para compensar pelos pixeis extra e redefinir o início da parte útil da matriz. Isto fará que o limite mínimo seja 1 à esquerda e em cima. Para calcular as coordenadas, usamos as linhas de código:

```
1 int x1 = x - dx < 1 ? 1 : x - dx;
2 int y1 = y - dy < 1 ? 1 : y - dy;
```

   (b) **Canto inferior direito**

   Para obter cada uma das coordenadas deste ponto, teremos de somar a cada coordenada do pixel, os valores do dy e dx, dy para a borda inferior e dx para a borda da direita. No caso do resultado de uma dessas coordenadas ultrapassar um dos limites máximos do retângulo, então esta coordenada deverá ser o limite máximo. Uma vez que deste lado não temos nenhuma linha ou coluna extra, o limite máximo será o último indice, ou seja, $integral\_width - 1$. Isto fará que o limite máximo seja a largura menos 1 à direita e a altura menos 1 em baixo. Aos resultados, teremos de adicionar o valor 1 visto que não é contabilizada a linha/coluna do pixel central. Para calcular as coordenadas, usamos as linhas de código:

```
1 int x2 = x + dx + 1 > integral_width - 1 ? integral_width - 1 : x + dx + 1;
2 int y2 = y + dy + 1 > integral_height - 1 ? integral_height - 1 : y + dy + 1;
```

2. **Cálculo da quantidade de pixeis**

   Para calcular a quantidade de pixeis que estão dentro do retângulo, basta calcular a área do retângulo em pixeis, ou seja, a multiplicação do comprimento pela largura. Para tal, usamos a linha de código:

```
1     int count = (x2 - x1) * (y2 - y1);
```

3. **Cálculo da soma**

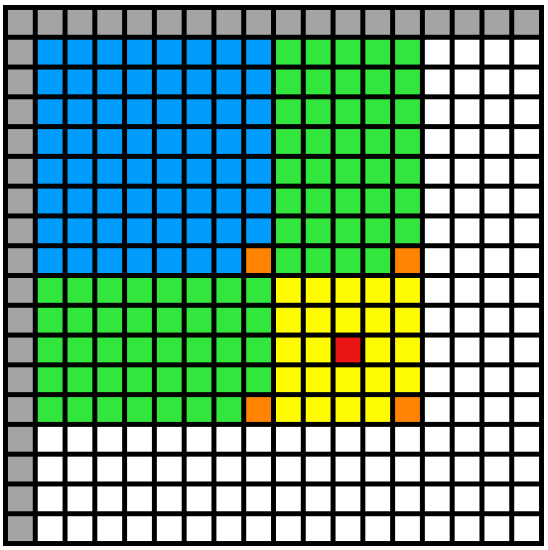

Figura 2.4: Representação do retângulo de blur

Como podemos ver pela figura anterior, para conseguir a área pretendidada (representada a **amarelo**), temos de usar como base a área associada ao pixel do canto inferior direito e subtrair a área associada ao pixel do canto superior direito (área acima do retângulo) bem como subtrair a área associada ao pixel do canto inferior esquerdo (área à esquerda do retângulo). Estas áreas exteriores ao retângulo estão representadas a **verde**. Visto que uma parte destas áreas coincide, temos de tirar a parte duplicada que será a área associada ao pixel do canto superior esquerdo e representada a **azul** na imagem. Importante notar que enquanto que o canto inferior direito será incluído na área, os outros 3 cantos não serão. A partir disto, podemos obter a soma do retângulo de blur a partir da seguinte expressão:

$$S_{(y,x)} = I_{(y2,x2)} - I_{(y1,x2)} - I_{(y2,x1)} + I_{(y1,x1)} \tag{2.4}$$

Para calcular o resultado desta expressão, usamos as linhas de código:

```
1     int sum = integral[y2 * integral_width + x2]
2             - integral[y1 * integral_width + x2]
3             - integral[y2 * integral_width + x1]
4             + integral[y1 * integral_width + x1];
```

4. **Definir o novo valor do pixel**

Para definir o novo valor do pixel, basta dividir a soma pelo número de pixeis que estão dentro do retângulo. Para tal, usamos a linha de código:

```
1     ImageSetPixel(img, x, y, (sum + count / 2) / count);
```

De notar que somamos $count/2$ à soma para que o resultado da divisão seja arredondado corretamente.

Após repetir este procedimento para todos os pixeis da imagem, obtemos a imagem desfocada.

Capítulo 3

ImageMatchSubImage e ImageLocateSubImage

3.1 Análise formal

3.1.1 ImageMatchSubImage

Descrição do algoritmo

É importante analisar a função ImageMatchSubImage, que é chamada dentro do loop interior da função ImageLocate-SubImage. Esta função compara uma imagem (img2) com uma subimagem de uma imagem maior (img1). A subimagem é definida pela posição inicial (x, y) na imagem maior. Após as verificações iniciais, percorre-se cada pixel da imagem, a partir da posição inicial, comparando-o com o da subimagem. No caso de encontrar um pixel que não corresponda, a função devolve falso. Se todos os pixeis corresponderem, a função devolve verdadeiro.

Análise de complexidade

- **Complexidade Temporal**

  Após verificarmos que a obtenção de pixeis, presente dentro do loop interior, é uma operação $O(1)$, e que se percorre a imagem linha a linha e coluna a coluna, podemos afirmar que a complexidade temporal do algoritmo é $O(n * m)$, onde n e m são a altura e largura de img2, respetivamente.

- **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 row e col são as únicas variáveis adicionais, e ocupam um espaço constante.

3.1.2 ImageLocateSubImage

Descrição do algoritmo

A função ImageLocateSubImage compara uma imagem (img2) com uma imagem maior (img1). Após as verificações iniciais, percorre-se cada pixel da imagem, comparando-o com o da subimagem. Se for detetado um pixel igual, chama-se a função ImageMatchSubImage para verificar se a subimagem está presente na imagem maior. No caso de encontrar um pixel que não corresponda, a função continua até encontrar, ou até acabar a imagem. Caso a ImageMatchSubImage devolva verdadeiro, a função devolve verdadeiro.

Análise de complexidade

- **Complexidade Temporal**

  Sabemos que ImageMatchSubImage, presente dentro do loop interior, é uma operação $O(p * o)$, e que se percorre a imagem linha a linha e coluna a coluna, logo é possível afirmar que a complexidade temporal do algoritmo é $O((n * m) * (p * o))$, onde n e m são a altura e largura de img2 e p e o são a altura e largura da subimagem verificada dentro do loop, respetivamente.

- **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 row e col são as únicas variáveis adicionais, e ocupam um espaço constante.

3.2 Análise experimental

O gráfico seguinte foi gerado com a ajuda do *LibreOffice Calc*. Para tal, foram primeiro gerados ficheiros, em formato csv, com os dados obtidos através da biblioteca *time* da linguagem C. Depois foi gerado o gráfico a partir desse ficheiro, com resultados de 20 testes diferentes. Foi usado um computador com as seguintes especificações:

- **CPU:** AMD Ryzen 7 5700U (16) @ 4.3GHz
- **GPU:** AMD Radeon RX Vega 8 [Lucienne]
- **RAM:** 16GB

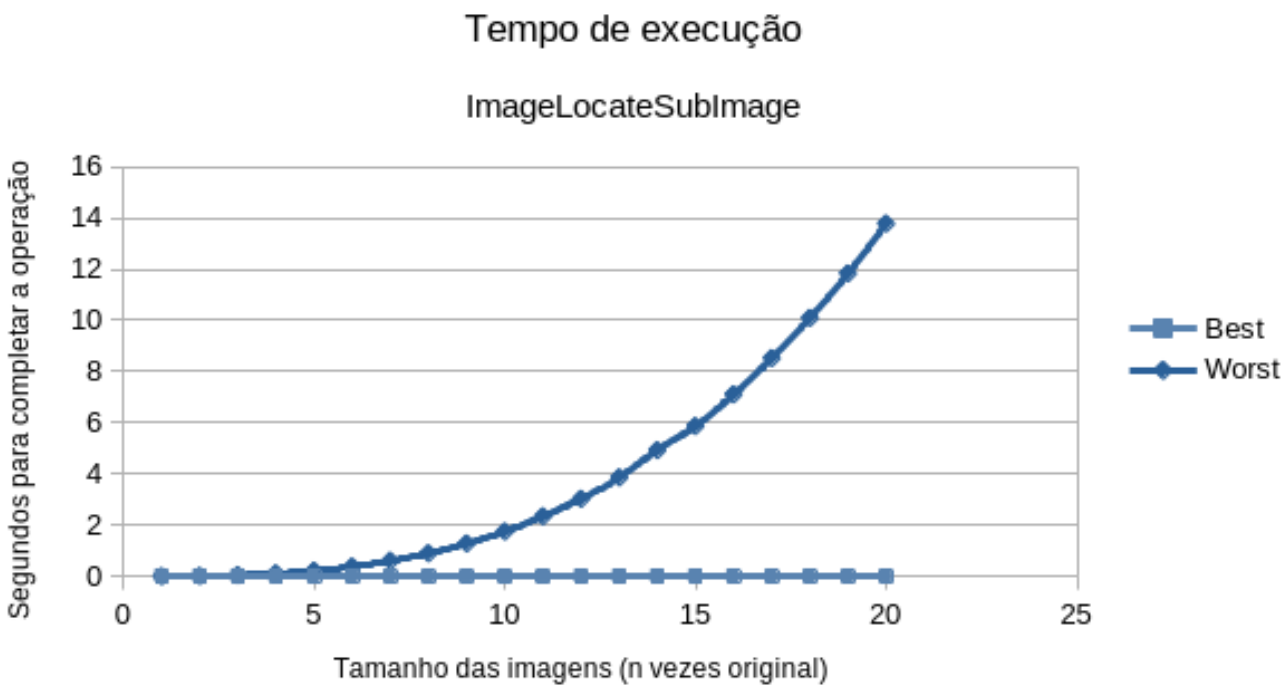

Figura 3.1: Gráfico do tempo de execução em função do tamanho das imagens fonte.

Antes de fazer a análise do gráfico, é importante entender os valores. No eixo das abcissas está representado o tamanho das imagens fonte, ou seja, a imagem original, uma subimagem, e uma outra imagem que não pertence à imagem original. Para começar, a imagem original tem um tamanho de 300x300, a subimagem um tamanho de 50x50 (retirada do canto superior esquerdo da imagem original, de forma a obter o *best case scenario*), e a outra imagem tem um tamanho de 100x100 (de forma a obter o *worst case scenario*). Multiplica-se, então, o tamanho das imagens por cada um dos números de 2 a 20. No final, obtemos os tamanhos máximos de 6000x6000, 1000x1000 e 2000x2000, respetivamente. No eixo das ordenadas está representado o tempo de execução, em segundos, para os melhores e piores casos.

Como podemos ver na figura 3.1, o tempo de execução aumenta exponencialmente. Isto acontece porque o tempo de execução é proporcional ao tamanho das imagens fonte, logo se duplicarmos o tamanho de ambas as imagens, o tempo de execução passa a ser 8 vezes superior.

Capítulo 4

Conclusões

Capítulo 5

Anexos

5.1 ImageBlur: Primeira Abordagem

```
void ImageBlur(Image img, int dx, int dy) {
    assert(img != NULL);
    assert(dx >= 0 && dy >= 0);

    if (dx == 0 && dy == 0) return;

    Image img_new = ImageCreate(img->width, img->height, img->maxval);
    if (img_new == NULL) return;

    for (int x = 0; x < img->width; x++) {
        for (int y = 0; y < img->height; y++) {
            int sum = 0;
            int count = 0;

            for (int ix = x - dx; ix <= x + dx; ix++) {
                for (int iy = y - dy; iy <= y + dy; iy++) {
                    if (!ImageValidPos(img, ix, iy)) continue;
                    sum += ImageGetPixel(img, ix, iy);
                    count++;
                }
            }

            ImageSetPixel(img_new, x, y, (sum + count / 2) / count);
        }
    }

    uint8 *tmp = img->pixel;
    img->pixel = img_new->pixel;
    img_new->pixel = tmp;

    ImageDestroy(&img_new);
}
```

5.2 ImageBlur: Segunda Abordagem

```
void ImageBlur(Image img, int dx, int dy) {
    assert(img != NULL);
    assert(dx >= 0 && dy >= 0);

    if (dx == 0 && dy == 0) return;

    int integral_width = img->width + 1;
    int integral_height = img->height + 1;
    int* integral = (int*)calloc(integral_width * integral_height, sizeof(int));
    for (int y = 1; y < integral_height; y++) {
        for (int x = 1; x < integral_width; x++) {
            integral[y * integral_width + x] = ImageGetPixel(img, x - 1, y - 1)
                + integral[(y - 1) * integral_width + x]
                + integral[y * integral_width + (x - 1)]
                - integral[(y - 1) * integral_width + (x - 1)];
        }
    }

    for (int y = 0; y < img->height; y++) {
        for (int x = 0; x < img->width; x++) {
            int x1 = x - dx < 1 ? 1 : x - dx;
            int y1 = y - dy < 1 ? 1 : y - dy;

            int x2 = x + dx + 1 > integral_width - 1 ? integral_width - 1 : x + dx + 1;
            int y2 = y + dy + 1 > integral_height - 1 ? integral_height - 1 : y + dy + 1;

            int count = (x2 - x1) * (y2 - y1);

            int sum = integral[y2 * integral_width + x2]
                - integral[y1 * integral_width + x2]
                - integral[y2 * integral_width + x1]
                + integral[y1 * integral_width + x1];

            ImageSetPixel(img, x, y, (sum + count / 2) / count);
        }
    }

    free(integral);
}
```