Projeto 1 - AED

Análise da complexidade: `ImageLocateSubImage()` e `ImageBlur()`

Jorge Domingues nº113278

Guilherme Santos nº113893

---

1. Introdução

Este relatório consiste na realização de uma análise aprofundada de duas funções essenciais: `ImageLocateSubImage()` e `ImageBlur()`. Ambas as funções estão incorporadas no ficheiro *image8bit.c* e desempenham papéis cruciais no processamento de imagens. A função `ImageLocateSubImage()` tem como finalidade a verificação da presença de uma subimagem dentro de outra imagem maior. A segunda função em destaque, `ImageBlur()`, concentra-se na aplicação do efeito de desfoque em imagens. A análise abordará a implementação destas funções assim como as suas respetivas complexidades computacionais.

2. ImageLocateSubImage

2.1. Pré-condições

```
int ImageMatchSubImage(Image img1, int x, int y,
     Image img2)
{
  assert(img1 != NULL);
  assert(img2 != NULL);
  assert(ImageValidPos(img1, x, y));
  ...
}
```

1. `img1` não pode ser `NULL`.
2. `img2` não pode ser `NULL`.
3. As coordenadas `x` e `y` têm de ser válidas, ou seja devem pertencer a um pixel da `img1`.

```
int ImageLocateSubImage(Image img1, int *px, int
    *py, Image img2)
{
  assert(img1 != NULL);
  assert(img2 != NULL);
  ...
}
```

1. `img1` não pode ser `NULL`.
2. `img2` não pode ser `NULL`.

2.2. Algoritmos e complexidade

Nesta etapa, abordaremos o funcionamento do algoritmo e avaliaremos a sua complexidade computacional. A função funciona à base do uso da `ImageMatchSubImage()`, a qual verifica se uma subimagem coincide com uma região específica da imagem principal.

```
int ImageMatchSubImage(Image img1, int x, int y
    , Image img2)
{
  ...
  for (int i = 0; i < img2->width; i++){
    for (int j = 0; j < img2->height; j++){
      ITERATIONS++;
      if (ImageGetPixel(img1, x + i, y + j) !=
     (ImageGetPixel(img2, i, j))){
        return 0;
      }
    }
  }
  return 1;
}
```

```
int ImageLocateSubImage(Image img1, int *px,
    int *py, Image img2)
{
  ...
  for (int x = 0; x <= img1->width - img2->
    width; x++){
    for (int y = 0; y <= img1->height - img2
    ->height; y++)
    {
      if (ImageMatchSubImage(img1, x, y, img2))
      {
        *px = x;
        *py = y;
        return 1;
      }
    }
  }
  return 0;
}
```

O cenário do *Best Case* possível acontece quando a subimagem consiste em apenas um pixel e está localizada no canto superior esquerdo da imagem principal. Assim, o pixel será logo encontrado na primeira iteração fazendo com que $B(\mathrm{n})$, sendo n o número de pixeis da imagem principal, seja 1. No entanto, se considerarmos que a subimagem tem um número de pixeis variável m, o $B(\mathrm{n})$ deste cenário dependerá unicamente de m. Ou seja,

$$B(n) = m,\ se\ m = 1\ ,\ B(n) = 1$$

No que toca ao *Worst Case* possível, é quando a imagem principal tem só uma cor e a subimagem tem a mesma cor mas difere no último pixel no canto inferior direito, no entanto, podemos dividir em duas perspetivas, na primeira, o *Worst Case* para uma determinada imagem de n pixeis é quando a subimagem tem de tamanho $\frac{n}{4}$ pixeis pois, sendo $f(\mathrm{m})$ a função que traduz o número de iterações que o algoritmo faz ao analisar a imagem principal com a subimagem de m pixeis, $f'(\mathrm{m}) = 0 \Leftrightarrow \mathrm{m} = \frac{n}{4}$. Sendo *W1*, *H1*, *W2*, *H2*, o *width* e *height* da principal, o *width* e *height* da secundária, respetivamente, para facilitar os cálculos, sabe-se que se $W2 = \frac{W1}{2}$ e $H2 = \frac{H1}{2}$, $W2\,H2 = \frac{n}{4}$. Assim,

$$\sum_{x=0}^{W1-W2}\sum_{y=0}^{H1-H2}\sum_{i=0}^{W2-1}\sum_{j=0}^{H2-1} 1\,, como\, W2 = \frac{W1}{2}\, e\, H2 = \frac{H1}{2}\,, \sum_{x=0}^{W1-\frac{W1}{2}}\sum_{y=0}^{H1-\frac{H1}{2}}\sum_{i=0}^{\frac{W1}{2}-1}\sum_{j=0}^{\frac{H1}{2}-1} 1 =$$

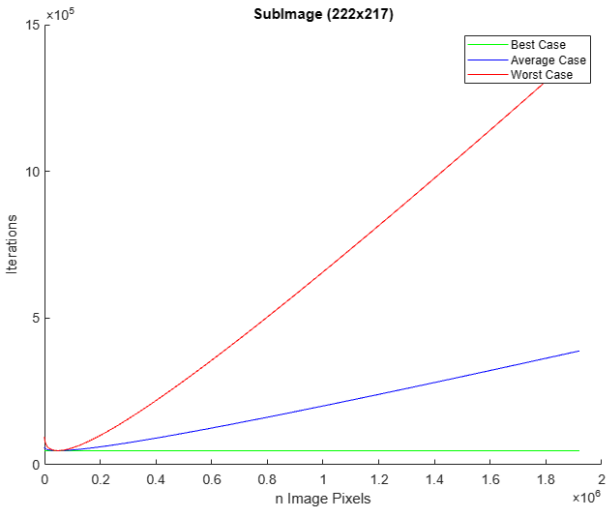
$$= \sum_{x=0}^{\frac{W1}{2}}\sum_{y=0}^{\frac{H1}{2}}\sum_{i=0}^{\frac{W1}{2}-1}\sum_{j=0}^{\frac{H1}{2}-1} 1 = (\frac{W1}{2}+1)\times(\frac{H1}{2}+1)\times(\frac{W1}{2})\times(\frac{H1}{2}) =$$

$$= \frac{(W1\times H1)^2}{16} + \frac{W1^2\times H1}{8} + \frac{W1\times H1^2}{8} + \frac{W1\times H1}{4} \approx \frac{(W1\times H1)^2}{16},\ como\, W1\times H1 = n,\ \frac{n^2}{16} \in O(n^2)$$

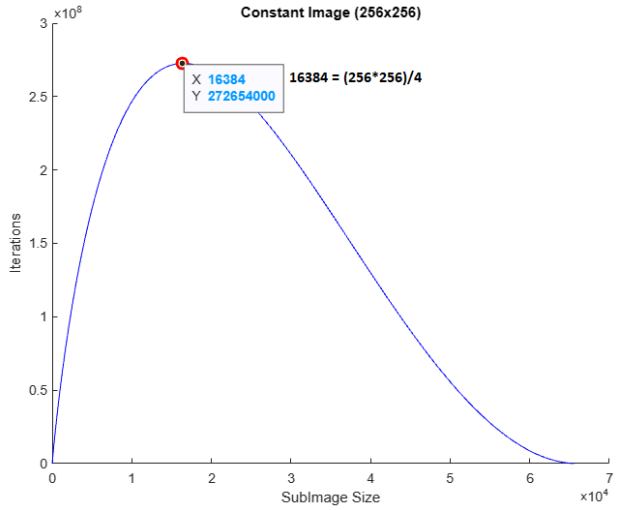
No entanto, olhando pela segunda perspetiva, se virmos o *Worst Case* como o pior caso para a imagem principal dada uma subimagem com m pixeis variáveis, a Ordem de Complexidade vai depender tanto da tamanho da maior imagem ($W1$x$H1$) como da subimagem ($W2$x$H2$), (Img-1). Assim, no caso geral seria,

$$\sum_{x=0}^{W1-W2}\sum_{y=0}^{H1-H2}\sum_{i=0}^{W2-1}\sum_{j=0}^{H2-1} 1 \in O((W1-W2+1)*(H1-H2+1)*W2*H2)$$

2.3. `ImageLocateSubImage` - Análise Experimental




```
--------------------SubImage Size (222x217)--------------------
--------------------LocateImage Best Case--------------------
Small image:
Image found at (0, 0)
#          time          caltime          pixmem       iterations
       0.000221        0.000370           96348            48174
Medium image:
Image found at (0, 0)
#          time          caltime          pixmem       iterations
       0.000219        0.000366           96348            48174
Big image:
Image found at (0, 0)
#          time          caltime          pixmem       iterations
       0.000220        0.000368           96348            48174
------------------------------------------------------------
------------------LocateImage Average Case------------------
Small image:
Image found at (17, 19)
#          time          caltime          pixmem       iterations
       0.000198        0.000331           97746            48873
Medium image:
Image found at (209, 131)
#          time          caltime          pixmem       iterations
       0.000535        0.000895          206986           103493
Big image:
Image found at (689, 491)
#          time          caltime          pixmem       iterations
       0.004155        0.006944         1453526           726763
------------------------------------------------------------
--------------------LocateImage Worst Case--------------------
Small image:
Image found at (34, 39)
#          time          caltime          pixmem       iterations
       0.000203        0.000343           99146            49573
Medium image:
Image found at (418, 263)
#          time          caltime          pixmem       iterations
       0.000853        0.001442          317706           158853
Big image:
Image found at (1378, 983)
#          time          caltime          pixmem       iterations
       0.008062        0.013621         2813692          1406846
------------------------------------------------------------
```




```
-------------------The Worst Case (256x256)------------------
Image (1x1):
#          time         caltime          pixmem      iterations
       0.001144        0.000639          131072           65536
Image (50x50):
#          time         caltime          pixmem      iterations
       1.042297        0.581864       214245000       107122500
Image (128x128):
#          time         caltime          pixmem      iterations
       1.998062        1.115421       545292288       272646144
Image (206x206):
#          time         caltime          pixmem      iterations
       0.474967        0.265151       220752072       110376036
Image (256x256):
#          time         caltime          pixmem      iterations
       0.000253        0.000141          131072           65536
-------------------------------------------------------------
```

3. ImageBlur

3.1. Pré-condições

```
void ImageBlur(Image img, int dx, int
    dy) {
  assert(img != NULL);
  assert(dx >= 0);
  assert(dy >= 0);
  assert(2*dx+1 <= img->width);
  assert(2*dy+1 <= img->height);
  ...
```

1. `img` não pode ser NULL.
2. `dx` e `dy` devem ser inteiros não-negativos.
3. O retângulo definido por $2 \times dx + 1$ e $2 \times dy + 1$ deve estar totalmente contido nas dimensões da imagem de entrada.

3.2. Versão 1 `ImageBlur`

A ordem de complexidade desta versão da `ImageBlur` vai ser determinada pelo número de vezes que é verificada a `ImageValidPos` dentro do último `for`. Neste caso, a gestão dos dados é sempre feita em $O(1)$ complexidade de tempo, logo não influencia o número de operações, (Versão 1)

```
Image blurredImg = ImageCreate(...);
if (blurredImg == NULL) ...
for (int x = 0; x < img->width; x++){
    for (int y = 0; y < img->height; y++){
        ...
        for (i = -dx; i <= dx; i++){
            for (j = -dy; j <= dy; j++){
                ITERATIONS++;
                if (ImageValidPos(img, x + i, y + j))
                ...
            }
        }
        ...
    }
}
free(img->pixel);
img->pixel = blurredImg->pixel;
free(blurredImg);
```

Assim, a ordem de complexidade dá-se pelo pior caso onde $(2dx + 1) \times (2dy + 1) =$ `img->width`$(W) \times$`img->height`$(H) = n$ pixeis:

$$\sum_{x=0}^{W-1}\sum_{y=0}^{H-1}\sum_{i=-dx}^{dx}\sum_{j=-dx}^{dy} 1 =$$

$$\sum_{x=1}^{W}\sum_{y=1}^{H}\sum_{i=1}^{2dx+1}\sum_{j=1}^{2dy+1} 1 = \sum_{x=1}^{W}\sum_{y=1}^{H}\sum_{i=1}^{W}\sum_{j=1}^{H} 1 =$$

$$\sum_{x=1}^{W}\sum_{y=1}^{H} WH = WHWH = n \times n \in O(n^2)$$

3.3. Versão 2 `ImageBlur`

O nº de operações desta versão, para além de ser determinado pelo nº de vezes que é atribuída a média a cada pixel, também é influenciado pela gestão de dados pois, como é usada a array de 2 dimensões `cumulativeSums` para guardar os valores da soma cumulativa dos pixeis da imagem inicial, o nº de operações também tem de ter em conta, para além do cálculo desses valores, os loops usados para alocar e libertar memória, (Versão 2)

$$\sum_{x=0}^{W-1} 1 + \sum_{x=0}^{W-1} 1 + \sum_{y=1}^{H-1} 1 + \sum_{x=1}^{W-1}\sum_{y=1}^{H-1} 1 + \sum_{x=0}^{W-1}\sum_{y=0}^{H-1} 1 + \sum_{x=0}^{W-1} 1 = W + W + H - 1 + (W-1)(H-1) + WH + W$$

$$= 3W + H - 1 + WH - W - H + 1 + WH = 2WH + 2W + 1, \; como \; W \times H = n, \; 2n + \frac{2}{H}n + 1 = (2 + \frac{2}{H})n + 1 \in O(n)$$

```
**calculeCumulativeSums(Image img){
  unsigned long **cumulativeSums =
  *alloc memory*
  for (x = 0; x < img->width; x++){
    *alloc memory*
  }
  for (x = 0; x < img->width; x++){
    ...
  }

  for (y = 1; y < img->height; y++){
    ...
  }
  for (x = 1; x < img->width; x++){
    for (y = 1; y < img->height; y++){
      ...
    }
  }
}
```

```
void ImageBlur(Image img, int dx, int dy){
  ...
  **cumulativeSums = calculeCumulativeSums(img);
  double sum, mean;
  ...
  for (x = 0; x < img->width; x++){
    for (y = 0; y < img->height; y++){
      ITERATIONS++;
      ...
      mean = sum / window + 0.5;
      ImageSetPixel(img, x, y, (uint8)mean);
    }
  }
  for (x = 0; x < img->width; x++){
      ITERATIONS++;
      free(cumulativeSums[x]);
  }
  free(cumulativeSums);
  ...
```

3.4. `ImageBlur` - Análise Experimental

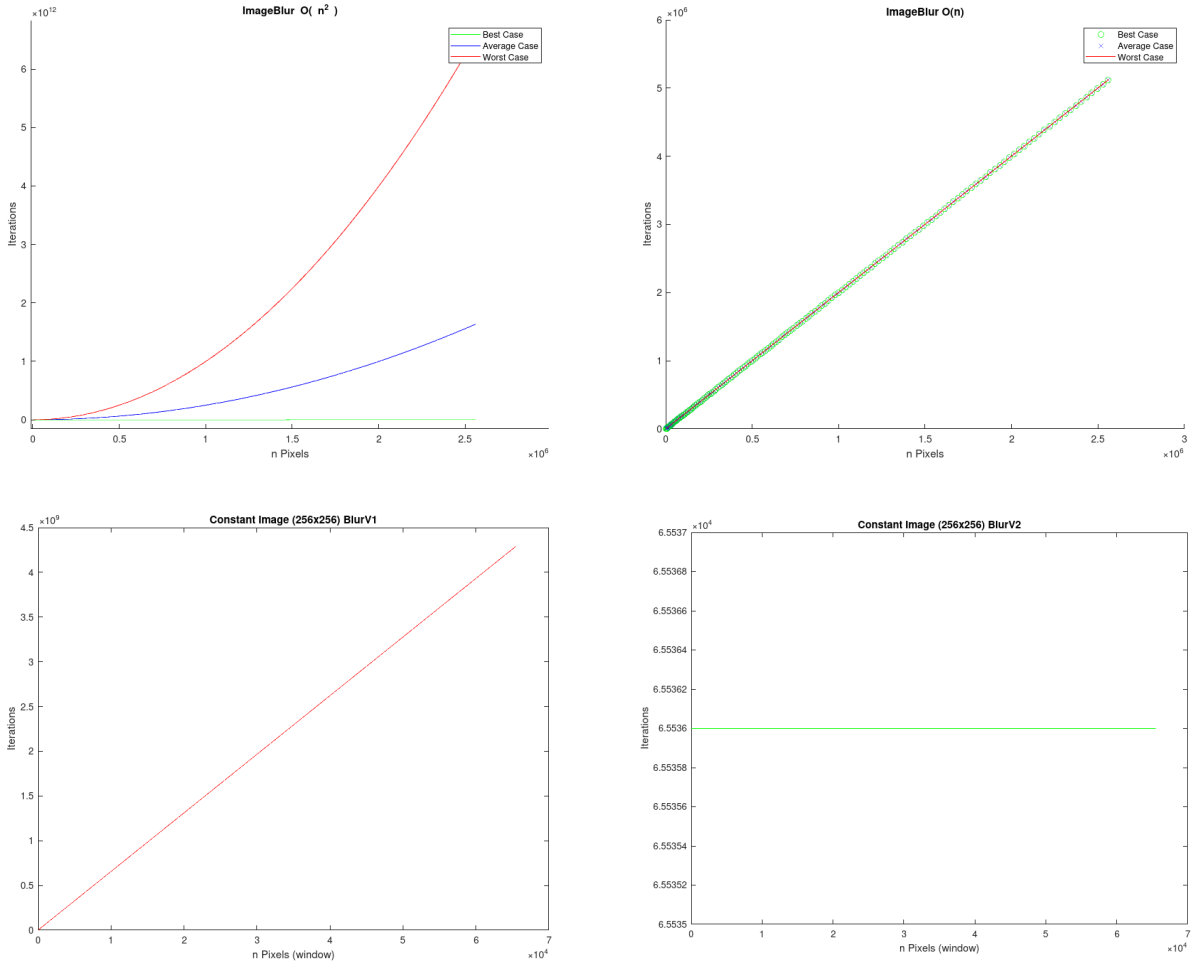
Versão 1

```
----------------------Blur Best Case----------------------
Blur Small Image (256x256):
#         time         caltime          pixmem      iterations
      0.000648        0.000712          131072           65536
Blur Medium Image (640x480):
#         time         caltime          pixmem      iterations
      0.003131        0.003439          614400          307200
Blur Big Image (1600x1200):
#         time         caltime          pixmem      iterations
      0.018944        0.020811         3840000         1920000
----------------------------------------------------------
----------------------Blur Average Case----------------------
Blur Small Image (256x256):
#         time         caltime          pixmem      iterations
      3.734032        4.146383       811175936      1057030144
Blur Medium Image (640x480):
#         time         caltime          pixmem      iterations
     55.684594       61.833857     17950944000     23421235200
Blur Big Image (1600x1200):
```

Dada a complexidade temporal de O(n^2) desta versão, a partir do teste da Imagem Maior do Average Case, o tempo e as iterações crescem demasiado para serem captudaras pelo computador onde, no Worst Case da Imagem Grande, demoraria aproximadamente 2,4horas com 3681025920000 iterações

Versão 2

```
----------------------Blur Best Case----------------------
Blur Small Image (256x256):
#         time         caltime          pixmem      iterations
      0.000701        0.001182          131072          131584
Blur Medium Image (640x480):
#         time         caltime          pixmem      iterations
      0.003139        0.005290          614400          615680
Blur Big Image (1600x1200):
#         time         caltime          pixmem      iterations
      0.020438        0.034446         3840000         3843200
----------------------------------------------------------
----------------------Blur Average Case----------------------
Blur Small Image (256x256):
#         time         caltime          pixmem      iterations
      0.000683        0.001149          131072          131584
Blur Medium Image (640x480):
#         time         caltime          pixmem      iterations
      0.002912        0.004900          614400          615680
Blur Big Image (1600x1200):
#         time         caltime          pixmem      iterations
      0.019345        0.032550         3840000         3843200
----------------------------------------------------------
----------------------Blur Worst Case----------------------
Blur Small Image (256x256):
#         time         caltime          pixmem      iterations
      0.000624        0.001052          131072          131584
Blur Medium Image (640x480):
#         time         caltime          pixmem      iterations
      0.002874        0.004847          614400          615680
Blur Big Image (1600x1200):
#         time         caltime          pixmem      iterations
      0.018569        0.031308         3840000         3843200
----------------------------------------------------------
```



3.5. Análise Comparativa BlurV1/BlurV2

No que toca ás duas versões da função `ImageBLur`, é possível reparar que apresentam ordens de complexidade muito diferentes, a Versão 1 a crescer quadráticamente $O(n^2)$ e a Versão 2 a crescer linearmente O(n). Esta diferença existe pois, na Versão 1, a média de cada pixel é calculada iterando pela janela de input $(2dx+1)$x$(2dy+1)$, fazendo com que o programa dependa, não só do tamanho da imagem como também dessa janela. Assim, mesmo com uma imagem constante, se o filtro aumenta, o número de operações também. A Versão 2 depende só dos pixeis da imagem pois, é usada uma array para guardar as suas somas cumulativas e, assim, a operação de cálculo da média é feita em $O(1)$ tempo tornando-se independente da janela de filtro.

4. Conclusão

Em suma, o nosso programa *image8bit.c* foi implementado, no geral, de forma otimizada e eficiente no que toca, tanto á Ordem de Complexidade de Tempo, como á Ordem de Complexidade de Espaço, nomeadamente nas funções `ImageLocateSubImage`, `ImageMatchSubImage` e `ImageBlur`.

Neste relatório, é possível reter a linha de raciocínio e adversidades encontradas ao longo da otimização do código destas funções, principalmente na extrema mudança de complexidade da função `ImageBlur` e na procura do *WorstCase* da função `ImageLocateSubImage`, no entanto, também é importante ressaltar a capacidade que nos deu de melhorar a nossa análise, no geral, de vários tipos de algoritmos e de implementar a arquitetura mais adequada a cada um deles.