Licenciatura em Engenharia Informática
Algoritmos e Estruturas de Dados
Aveiro, 20 de Novembro 2023

Processamento de Imagens PGM

Portable Gray Map

Henrique Teixeira - 114588
Gabriel Silva - 113786

Índice

Introdução

O objetivo deste trabalho foi efetuar o desenvolvimento e teste do TAD (Tipo Abstrato de Dados) "image8bit", com o intuito de instanciar e operar em imagens do tipo PGM.

PGM, ou *Portable Gray Maps*, é um formato de imagem usado para armazenar imagens em níveis de cinzento, com valores de intensidade entre 0 e 255 (8 bits). Os ficheiros PGM são usados principalmente para representar imagens em preto e branco ou em tons de cinza, e são um formato simples e legível para armazenar esse tipo de imagem.

O "image8bit.c" contém duas principais funções de maior desafio:

- A "ImageLocateSubImage", cuja função é determinar, caso exista, a localização de uma subimagem numa imagem dada
- A "ImageBlur", que pretende aplicar um filtro a uma imagem de forma a torná-la mais baça.

Para estas foi analisada a complexidade e a eficiência computacional do seu algoritmo através de uma sequência de testes, com imagens e filtros de diferentes tamanhos, nas quais foram analisadas o número de comparações e a quantidade de tempo que estas demoraram a ser efetuadas. Foi também efetuada uma análise formal da complexidade do algoritmo para o melhor caso e para o pior caso.

ImageLocateSubImage

A função ImageLocateSubImage tem como objetivo verificar se uma sub-imagem se encontra contida noutra imagem, caso isto se verifique, retorna também a posição em que ela está através de dois ponteiros, tendo assim 4 argumentos de chamada.

Funcionamento da Função

A ideia de resolução chegada como a mais otimizada foi de percorrer a Imagem inicial pixel a pixel, usando iterativamente a função ImageMatchSubImage, de forma a que quando esta função retornar 0 a ImageLocateSubImage possa ser concluída sem executar iterações desnecessárias, verificando os pixels de ambas as imagens no mesmo Loop.

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

  for (int i = 0; i < img1->height; i++) {
    for (int j = 0; j < img1->width; j++) {
      if (ImageMatchSubImage(img1, j, i, img2)) {
        *px = j;
        *py = i;
        return 1;
      }
      continue;
    }
  }
  return 0;
}
```

ImageMatchSubImage

Esta função foi desenvolvida de modo a dar *return 0* logo que encontre um pixel que não seja igual na imagem e na sub-imagem, para além disso, a primeira verificação que é feita é se a sub-imagem está de facto contida na outra, através da função ImageValidRect.

Podemos observar que a segunda verificação que é feita é se o primeiro pixel da sub-imagem tem valor semelhante ao da imagem, valor obtido através da função ImageGetPixel, caso isso não aconteça é poupada uma iteração.

```
int ImageMatchSubImage(Image img1, int x, int y, Image img2) { ///
  assert (img1 != NULL);
  assert (img2 != NULL);
  assert (ImageValidPos(img1, x, y));
  // Check if img2 fits inside img1 at position (x, y)
  if (ImageValidRect(img1, x, y, img2->width, img2->height)) {
      for (int i = 0; i < img2->height; i++) {
        for (int j = 0; j < img2->width; j++) {
          // If any pixel of img1 is different from img2 return 0
          if (ImageGetPixel(img1, x+j, y+i) != ImageGetPixel(img2, j, i)) {
            return 0;
          }
        }
      }
      // If all pixels are equal return 1
      return 1;
  }
  return 0;
}
```

Análise Formal da Complexidade

Melhor caso

Tendo agora em conta como foi desenvolvido o código podemos analisar formalmente a complexidade do melhor caso da função ImageLocateSubImage.

Como se pode verificar, a "PIXMEM" é aumentado duas vezes por cada iteração desta função, visto que ele é incrementado em 1 por cada vez que é chamado o ImageGetPixel, função chamada duas vezes por iteração na ImageMatchSubImage, o melhor caso da função vai ocorrer quando a sub-imagem começar nas coordenadas (0,0), sendo proporcional ao seu tamanho, ou seja, a complexidade do melhor caso desta função vai ser:

$$\mathrm{O}\left((largura \times altura)_{sub-imagem} \times 2\right)$$

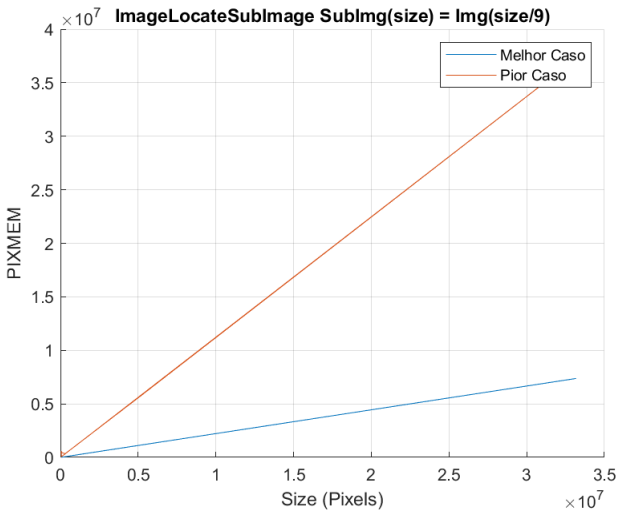
o que pode ser simplificado a

$$\mathrm{O}\left((largura \times altura)_{sub-imagem}\right)$$

Pior caso

O pior caso desta função é quando a sub-imagem se encontra localizada nas últimas posições da imagem, visto que a função teria de percorrer todos os pixels da imagem e da sub-imagem. No entanto, indo ainda mais longe, é expectável que, se a sub-imagem estiver contida várias vezes na imagem, com apenas o último pixel diferente da sub-imagem pretendida, sejam executadas *loops* extras, assim, a complexidade do pior caso desta função é:

$$\mathrm{O}\left[\left((largura \times altura)_{imagem}\right) \times \left((largura \times altura)_{sub-imagem}\right)\right]$$



	Pior Caso		Melhor Caso	
SIZE	**PIXMEM**	**TIME**	**PIXMEM**	**TIME**
48174	570636	0.001021	10656	0.000023
50400	56832	0.000339	11200	0.000020
65536	73592	0.000267	14450	0.000026
90000	440618	0.000875	20000	0.000033
262144	293936	0.001027	57800	0.000087
307200	343576	0.001183	68160	0.000132
883600	985348	0.003420	195938	0.000291
1920000	2148060	0.007571	426400	0.000632
2073600	2315900	0.008481	460800	0.000694
5184000	5768836	0.025690	1152000	0.001759
33177600	37289734	0.144226	7372800	0.015947

ImageBlur

Esta função foi desenvolvida com o intuito de aplicar um filtro numa imagem de modo a torná-la mais baça. O filtro da função é aplicado passando uma imagem, à qual o filtro vai ser aplicado, e dois ponteiros de inteiros dx e dy, que definem o raio do desfoque em ambas as direções da imagem (horizontal e vertical). O desfoque que vai ser aplicado vai ser a média do valor dos *pixels* na área (2dx+1)x(2dy+1).

Primeira Versão Desenvolvida

Esta primeira versão da função teve como principal objetivo ajudar-nos a ter uma melhor e completa noção de como se pareceria o *script* completo e a passar nos testes, assim, foi criada uma versão da ImageBlur com elevado nível de complexidade e extremamente pouco otimizada.

Esta versão estava organizada em duas passagens, a horizontal e a vertical,

- **Passagem Horizontal:**

Para cada pixel da imagem, a função calculava a média dos valores dos pixels numa linha horizontal de tamanho 2dx + 1.

```
// Horizontal pass
for (int y = 0; y < img->height; y++) {
    for (int x = 0; x < img->width; x++) {
        sum = 0;
        count = 0;
        for (int i = -dx; i <= dx; i++) {
            int nx = x + i;
            if (nx >= 0 && nx < img->width) {
                sum += img->pixel[G(img, nx, y)];
                count++;
            }
        }
        temp[G(img, x, y)] = sum / count;
    }
}
```

- **Passagem Vertical:**

Após a passagem horizontal, a função aplica um filtro semelhante na vertical, mas agora usando o *buffer* temporário "temp" que armazena os resultados da passagem horizontal.

A imagem que resultava da filtragem é então criada pixel a pixel.

```
// Vertical pass on the horizontally blurred image (stored in temp)
for (int x = 0; x < img->width; x++) {
    for (int y = 0; y < img->height; y++) {
        sum = 0;
        count = 0;
        for (int i = -dy; i <= dy; i++) {
            int ny = y + i;
            if (ny >= 0 && ny < img->height) {
                sum += temp[G(img, x, ny)];
                count++;
            }
        }
        img->pixel[G(img, x, y)] = (int)((sum / count) + 0.5);
    }
}
```

Podemos então concluir que esta segunda função tinha complexidade:

$$O(N \times (2dy + 1) + N \times (2dx + 1) + 2N) = O(N \times (2dy + 2dx + 4))$$

o que pode ser simplificado a

$$O(N \times (dy + dx))$$

Versão Otimizada

Não satisfeitos com os resultados obtidos com os métodos anteriormente referidos, decidimos mudar completamente a estrutura usada, passando assim a usar *Summed-Area Tables*.

Uma *Summed-Area Table,* ou "tabela de área integral", é uma estrutura de dados bidimensional construída a partir de uma matriz original, com o propósito de permitir o cálculo eficiente das somas acumuladas de elementos em sub-regiões de matrizes. Elas são usadas para melhorar o desempenho na resolução de problemas que envolvem a computação frequente de somas de subconjuntos em uma matriz, o que se encaixa perfeitamente no nosso problema a resolver.

Funcionamento de *Summed-Area Tables*

Estas tabelas guardam em cada ponto a soma de todos os pixels na imagem original que estão acima e à esquerda desse ponto.

Podemos ver o funcionamento delas através da sua fórmula:

$$I(x,y)=i(x,y)+I(x,y-1)+I(x-1,y)-I(x-1,y-1)$$

I é a tabela, i os pontos

Para calcular a soma dos valores de pixel em qualquer área retangular da imagem, basta fazer uma pequena quantidade de operações aritméticas com os valores da tabela, sendo isto muito mais rápido do que somar diretamente os valores dos pixels, especialmente para áreas grandes e para múltiplas iterações sobre a mesma área da imagem.

Funcionamento da Função

1. **Inicialização e preenchimento da tabela**

   Aqui, a matriz é a imagem, imaginando a altura como as linhas e as colunas como a largura desta. Para iniciar a tabela é feito um *malloc* com o tamanho da imagem.

   O primeiro duplo *for loop* preenche a tabela. Para cada pixel (x, y), ele calcula a soma de todos os pixels da origem da imagem até o pixel (x, y), onde as diferentes condições servem para tratar do cálculo de pixels presentes na borda superior e esquerda

```
for (int y = 0; y < img->height; y++) {
  for (int x = 0; x < img->width; x++) {
    if (x == 0 && y == 0) {
      summedTable[G(img, x, y)] = ImageGetPixel(img,x,y);
    } else if (x == 0) {
      summedTable[G(img, x, y)] = ImageGetPixel(img,x,y) + summedTable[G(img, x, y-1)];
    } else if (y == 0) {
      summedTable[G(img, x, y)] = ImageGetPixel(img,x,y) + summedTable[G(img, x-1, y)];
    } else {
      summedTable[G(img, x, y)] = ImageGetPixel(img,x,y) + summedTable[G(img, x, y-1)] + summedTable[G(img, x-1, y)] - summedTable[G(img, x-1, y-1)];
    }
  }
}
```

2. **Cálculo da média e criação da imagem**

   O segundo *loop* duplo percorre novamente a imagem. Para cada pixel vai então ser calculada a média dos pixels no retângulo (2dx + 1) x (2dy + 1) no redor do pixel, usando a tabela para eficiência.

   Após a área do retângulo ser calculada é de seguida encontrada a média usando a soma dos valores dos pixels dentro do retângulo.

   O pixel original na imagem é então substituído pelo valor médio calculado.

Análise Formal da Complexidade

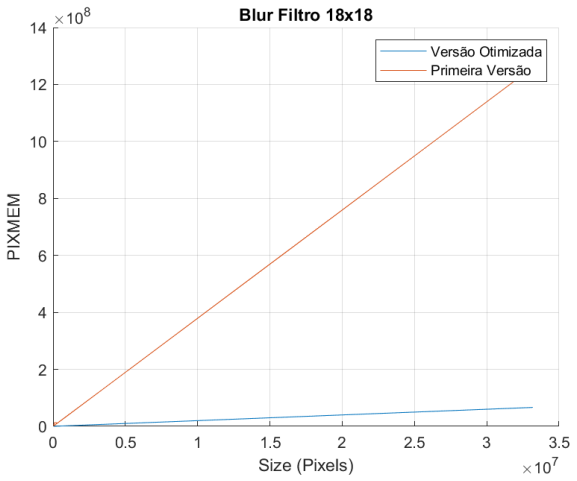
Podemos observar que em cada um dos tópicos cada pixel só é percorrido uma vez, quer na inicialização da tabela, como no cálculo da tabela e na aplicação do filtro, assim, a complexidade seria de:

$$O\ (2\ \times N)$$

o que pode ser simplificado a

$$O\ (N)$$

Comparando então todos os algoritmos desenvolvidos,



	Versão otimizada O(2N)		Primeira versão O(N(dy+dx))	
SIZE	**PIXMEM**	**TIME**	**PIXMEM**	**TIME**
48174	96348	0.000687	1756398	0.005775
50400	100800	0.000586	1857744	0.006270
65536	131072	0.000598	2402816	0.008563
90000	180000	0.000896	3317400	0.012182
262144	524288	0.003982	9786368	0.035392
307200	614400	0.003004	11509440	0.047324
883600	1767200	0.010622	33255320	0.138777
1920000	3840000	0.021837	72412800	0.279412
2073600	4147200	0.023396	78427440	0.280546
5184000	10368000	0.055391	196376400	0.811172
33177600	66355200	0.332577	1260010080	5.036689

Conclusão

Os resultados deste trabalho foram os esperados, uma vez que, segundo os testes feitos, a complexidade das funções correspondem com a nossa análise formal.

Nossa participação neste projeto foi muito benéfica. Ela nos permitiu aprimorar nossas habilidades em C, e também de Matlab, essencial para a criação dos gráficos. Além disso, contribuiu para melhorar nosso trabalho em equipa e fortaleceu a nossa capacidade de solucionar problemas complexos através de algoritmos, um desafio fundamental neste projeto.