universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

1º Trabalho - O TAD image8bit

Algoritmos e Estruturas de Dados

Ano Letivo: 2023/2024

Prof. Joaquim João Estrela Ribeiro Silvestre Madeira

Prof. Mario Antunes

Trabalho realizado por:

Rodrigo Abreu, Nº113626

João Neto, Nº113482

Índice

Introdução

Este projeto visa desenvolver e testar o Tipo Abstrato de Dados (TAD) *image8bit*, que permite a criação e manipulação de imagens em tons de cinzento variando de 0 a 255 (8 bits). O foco deste trabalho está na análise da complexidade computacional das funções ***ImageLocateSubImage*** e ***ImageBlur***, que buscam localizar uma sub-imagem em uma imagem e aplicar um efeito de desfoque numa imagem, respectivamente.

Durante o desenvolvimento, identificou-se ineficiência do programa em imagens maiores, levando a diversas abordagens e estratégias de otimização. Este documento concentra-se especialmente nessa otimização, comparando versões otimizadas e não otimizadas das funções para uma análise detalhada e precisa.

Análise da complexidade computacional das funções

Função ImageLocateSubImage - 1ª Versão

A primeira abordagem que tivemos no desenvolvimento desta função foi criar 2 loops "for" que iterassem pelos seguintes intervalos: [0, w1-w2+1[, [0, h1-h2+1[, respetivamente, de modo a passar por todos os píxeis que interessam para o problema.

Por cada pixel, é invocada a função *ImageMatchSubImage,* que a partir desse pixel, também através de 2 loops "for", itera por todos os píxeis da sub-imagem e compara com os píxeis da janela correspondente com as mesmas dimensões, pertencente à imagem original.

Quando encontra píxeis diferentes, quebra o loop, retorna 0, e o processo repete-se para o pixel seguinte. Caso isto não se verifique, conclui-se que as imagens são compatíveis.

Complexidade temporal

Procedendo-se à análise da complexidade temporal desta função, deparamo-nos com um cenário de complexidade do tipo *Best Case* e *Worst Case*.

O *Best Case,* verifica-se quando para o pixel (0,0), a condição é satisfeita, ou seja, é encontrada a compatibilidade na função *ImageMacthSubImage,* na primeira iteração da função *ImageLocateSubImage,* logo a complexidade é calculada da seguinte forma:

$$\sum_{y=1}^{h2} \sum_{x=1}^{w2} 1 = h2*w2$$

Assumindo que n é o número total de píxeis da sub-imagem, n=w2h2, a complexidade é *O(n)*.

O *Worst Case* verifica-se quando a imagem onde se vai procurar a sub-imagem tem todos os píxeis com o mesmo valor de cinzento e a sub-imagem tem todos os píxeis com esse valor de cinzento, exceto o último pixel, que terá obrigatoriamente de ter um tom diferente. Posto isto, a complexidade pode ser calculada da seguinte forma:

$$\sum_{x=1}^{w1-w2+1} \sum_{y=1}^{h1-h2+1} \sum_{i=1}^{h2} \sum_{j=1}^{w2} 1 = (w1-w2+1)(h1-h2+1)h2w2 = (w1h1w2h2-w1w2h^2 2+...+h2w2)|$$

Assumindo N como o número total de píxeis da imagem original, N=w1h1, e n=w2h2, e w2h2<=w1h1, a complexidade desta função é O(Nn) e pode ser simplificada em $O(N^2)$.

Testes

```
# Não otimizado:
 # Locate image (size: 11988 - window 111x108) in image (size: 48174 - window 222x217)

# Best Case = encontra (Sucess = 1 FOUND(0,0))
#          time         caltime          pixmem         pixcomp     iterações
       0.000048        0.000071           23976           11988           11989

# Worst Case = não encontra (Sucess = 0 NOT FOUND)
#          time         caltime          pixmem         pixcomp     iterações
       0.527185        0.780190       295384320       147692160       147704480
```

Fig 1 - Testagem da função ImageLocateSubImage, com os respectivos Best Case e Worst Case

Função ImageLocateSubImage - 2ª Versão

O método de otimização da função *ImageLocateSubImage* envolve a criação de quatro tabelas correspondentes às dimensões das imagens. Duas dessas tabelas representam a soma acumulativa dos valores dos píxeis anteriores, enquanto as outras duas representam a soma acumulativa dos valores dos pixeis anteriores ao quadrado. Essa abordagem permite a comparação das duas somas, garantindo que ambas tenham o mesmo número de píxeis com determinado tom de cinzento e evitando píxeis diferentes entre as imagens.

Se as somas forem iguais, a função invoca ImageMatchSubImage, que compara as somas normais das linhas e colunas. Caso alguma soma seja diferente, a função retorna 0. Mesmo após esses testes, as imagens podem ser diferentes, então é realizada uma comparação pixel a pixel da sub-imagem. Se todos os píxeis forem iguais, a função retorna 1. Essa otimização visa melhorar o desempenho do código, especialmente para imagens de maiores dimensões.

No final, a função *ImageLocateSubImage* retorna 1, atribuindo o respetivo valor de x e y correspondentes ao primeiro pixel da imagem aos ponteiros px e py.

Complexidade temporal

Procedendo-se à análise da complexidade temporal desta função, deparamo-nos com um cenário de complexidade do tipo *Best Case* e *Worst Case*.

No Best Case chegamos a dois resultados, que dão um número de comparações bastante semelhante, o primeiro Best Case, verifica-se quando se encontra uma imagem correspondente logo no início:

$$2+\sum_{y=1}^{h2} 1+\sum_{x=1}^{w2} 1+\sum_{x=1}^{w2}\sum_{y=1}^{h2} 1 = 2+h2+w2+w2*h2$$

Com ordem de complexidade O(N), sendo N=w2*h2.

O 2º caso é quando nunca encontra um match, mas w2*h2 têm de ser superior a ¼ de w1*h1

$$\sum_{y=1}^{h1-h2+1}\sum_{x=1}^{w1-w2+1} 1 = (h1-h2+1)(w1-w2+1)|$$

Com Ordem de complexidade O((h1-h2+1)(w1-w2+1)).

O *Worst Case* agora tornou-se numa imagem muito específica, mas a sua complexidade seria:

$$\sum_{x=1}^{w1-w2+1}\sum_{y=1}^{h1-h2+1}\sum_{i=1}^{w2} 1+\sum_{x=1}^{w1-w2+1}\sum_{y=1}^{h1-h2+1}\sum_{j=1}^{h2} 1+\sum_{x=1}^{w1-w2+1}\sum_{y=1}^{h1-h2+1}\sum_{j=1}^{h2}\sum_{i=1}^{w2} 1+\sum_{x=1}^{w1-w2+1}\sum_{y=1}^{h1-h2+1} 2$$

$$= (w1-w2+1)(h1-h2+1)(w2*h2+w2+h2+2) = w2h2(w1-w2+1)(h1-h2+1)+...$$

Como w2h2(w1-w2+1)(h1-h2+1) é o termo de maior grau, podemos concluir que a ordem complexidade é *O(w2h2(w1-w2+1)(h1-h2+1))*.

Complexidade espacial

O espaço ocupado na memória vai ser sempre igual ao número de píxeis da imagem*2 + o número de píxeis da subimagem*2, pois é alocada memória para os 4 arrays de somas.

Em 2*w1*h1 + 2*w2*h2, como w2*h2 nunca é maior que w1*h1, a ordem de complexidade é *O(n)*, sendo n=w1*h1.

Testes

```
# Otimizado:
 # Locate image (size: 307200 - window 640x480) in image (size: 1920000 - window 1600x1200)

# Best Case ?=? encontra (Sucess = 1 FOUND(0,0))
#          time         caltime          pixmem         pixcomp     iterações
       0.033907        0.048891         7296000          308322         4762721

# Best Case ?=? não encontra (Sucess = 0 NOTFOUND)
#          time         caltime          pixmem         pixcomp     iterações
       0.036848        0.053132         6681600          692881         5147281

# Otimizado:
 # Locate image (size: 480000 - window 800x600) in image (size: 1920000 - window 1600x1200)

# Best Case ?=? encontra (Sucess = 1 FOUND(0,0))
#          time         caltime          pixmem         pixcomp     iterações
       0.039671        0.057202         8160000          481402         5281401

# Best Case ?=? não encontra (Sucess = 0 NOTFOUND)
#          time         caltime          pixmem         pixcomp     iterações
       0.035236        0.050807         7200000          481401         5281401

# Otimizado:
 # Locate image (size: 691200 - window 960x720) in image (size: 1920000 - window 1600x1200)

# Best Case ?=? encontra (Sucess = 1 FOUND(0,0))
#          time         caltime          pixmem         pixcomp     iterações
       0.039993        0.057667         9216000          692882         5915281

# Best Case ?=? não encontra (Sucess = 0 NOTFOUND)
#          time         caltime          pixmem         pixcomp     iterações
       0.033951        0.048955         7833600          308321         5530721
```

Fig 2 - Testes da função ImageLocateSubImage Otimizada

Função ImageBlur - 1ª Versão

A nossa primeira abordagem a este problema, consistiu em criar uma cópia da imagem original e fazer dois ciclos “for”, que iteram, respetivamente, pelos valores de x e y dos elementos da imagem, de maneira a iterar por todos os píxeis.

Cada pixel terá a sua janela de filtragem, calculada através do valor dos argumentos “dx” e “dy” passados para a função, criando assim a seguinte janela, representada pela matriz: [x-dx, x+dx ; y-dy, y+dy]. Para aplicar o filtro, criámos dois ciclos “for” que iteram por todos os pixeis dessa janela, e se a posição desses píxeis pertencer às dimensões da imagem, o valor do tom de cinzento irá ser tido em conta para o cálculo do filtro. O valor do tom de cinzento de cada pixel resultante da filtragem é calculado da seguinte forma: *((soma do valor de cada pixel)+(número de pixeis tidos em conta)/2)/(número de pixeis tidos em conta)*.

No final deste processo, o valor resultante da filtragem, irá substituir o valor anterior na imagem original, na respetiva posição, e o processo é repetido até chegar ao último pixel.

Complexidade temporal

Procedendo-se à análise da complexidade temporal desta função. O número de iterações desta expressão é dado pela seguinte expressão:

$$\sum_{x=1}^{w}\sum_{y=1}^{h}\sum_{i=x-dx}^{x+dx}\sum_{j=y-dy}^{y+dy} 1 = w*h*(2dx+1)*(2dy+1) = 4dxdywh + 2dxwh + 2dywh + wh$$

Assumindo que n = número de píxeis total, resultante de w*h, logo n=w*h e como dx e dy têm valores sempre menores ou iguais a w/2 e h/2, respetivamente, pode-se concluir que a ordem de complexidade é $O(ndxdy)=O(n*w/2*h/2)=O(n*n/4)=O(n^2/4)=O(n^2)$.

Complexidade espacial

O espaço ocupado na memória vai ser sempre igual ao número de píxeis da imagem, pois é alocada memória para uma imagem com as mesmas dimensões da original.

Logo, a ordem de complexidade é $O(n)$, sendo n=w*h.

Testes

```
# Não otimizado:
 # BLUR image (size: 48174 - window 222x217) com dx = 10 e dy = 10
#          time         caltime          pixmem         pixcomp     iterações
       0.047406        0.069354        20387266               0        21244734

# Não otimizado:
 # BLUR image (size: 48174 - window 222x217) com dx = 20 e dy = 20
#          time         caltime          pixmem         pixcomp     iterações
       0.185058        0.270737        73741836               0        80980494
```

Fig 3 - Testagem da função ImageBlur para uma imagem com tamanho 222x217

```
# Não otimizado:
 # BLUR image (size: 1920000 - window 1600x1200) com dx = 10 e dy = 10
#          time         caltime          pixmem         pixcomp     iterações
       1.931034        2.773607       846024100               0       846720000

# Não otimizado:
 # BLUR image (size: 1920000 - window 1600x1200) com dx = 20 e dy = 20
#          time         caltime          pixmem         pixcomp     iterações
       7.620328       10.945321      3185240400               0      3227520000
```

Fig 4 - Testagem da função ImageBlur para uma imagem com tamanho 1600x1200

Função ImageBlur - 2ª Versão

Depois de testar a função anterior, e identificar a sua ineficiência para imagens de dimensões maiores, tentámos uma nova abordagem, de modo a que não fosse necessário iterar pelos píxeis da janela de filtragem por cada pixel da imagem para os cálculos.

Deste modo, concluímos que podíamos criar uma tabela com as mesmas dimensões da imagem, em que cada pixel correspondente terá o valor da respetiva soma acumulativa dos valores dos píxeis anteriores. Desta forma, serão evitados cálculos intermédios na aplicação da filtragem de cada imagem, pois os valores necessários já estão armazenados na tabela.

Sendo o nome da tabela “sumtable”, para calcular o valor de filtragem de cada pixel, basta apenas aceder aos respectivos elementos da tabela e aplicar a expressão: *sumtable(x+dx, y+dy) + sumtable(x-dx, y-dy) - sumtable(x-dx, y+dy) - sumtable(x+dx, y-dy).*

Nesta versão, tal como na anterior, terá de se efetuar este processo para todos os píxeis da imagem, mas melhora-se imenso o desempenho sem os cálculos intermédios.

Complexidade temporal

O número de iterações desta expressão é dado pela seguinte expressão:

$$\sum_{x=1}^{w}\sum_{y=1}^{h} 1 + \sum_{x=1}^{w}\sum_{y=1}^{h} 1 = w*h + w*h = 2\,w*h$$

Assumindo n como o número total de pixeis, e n=w*h, como 2 é uma constante, a ordem de complexidade temporal desta função é *O(n)*.

Complexidade espacial

O espaço ocupado na memória vai ser sempre igual a 2 vezes o número de píxeis da imagem, pois é alocada memória para uma imagem com as mesmas dimensões da original e alocada memória para uma tabela com as mesmas dimensões da imagem, 2*w*h.

Logo, como se pode omitir a constante 2 e sendo n=w*h, a ordem de complexidade é *O(n)*.

Testes

```
# Otimizado:
 # BLUR image (size: 48174 - window 222x217) com dx = 10 e dy = 10
#          time         caltime          pixmem         pixcomp     iterações
       0.000531        0.000776           96348               0           96348

# Otimizado:
 # BLUR image (size: 48174 - window 222x217) com dx = 20 e dy = 20
#          time         caltime          pixmem         pixcomp     iterações
       0.000469        0.000686           96348               0           96348
```

Fig 5 - Testagem da função ImageBlur otimizada para uma imagem com tamanho 222x217

```
# Otimizado:
 # BLUR image (size: 1920000 - window 1600x1200) com dx = 10 e dy = 10
#          time         caltime          pixmem         pixcomp     iterações
       0.026015        0.037367         3840000               0         3840000

# Otimizado:
 # BLUR image (size: 1920000 - window 1600x1200) com dx = 20 e dy = 20
#          time         caltime          pixmem         pixcomp     iterações
       0.026141        0.037547         3840000               0         3840000
```

Fig 6 - Testagem da função ImageBlur otimizada para uma imagem com tamanho 1600x1200

Conclusão

Os resultados registados, levaram-nos a perceber o impacto que a complexidade das funções têm no tempo de execução.

Para melhorar este aspeto, experimentámos vários métodos diferentes, como por exemplo, no *ImageLocateSubImage*, onde foram tentados 5 métodos diferentes, como a utilização da função *Threshold* ou a transformação das imagens em *strings* e compará-las, até chegarmos às versões otimizadas apresentadas. A fiabilidade da nossa análise formal é comprovada, devido à compatibilidade com os resultados da testagem das funções.

Acima de tudo, foi-nos permitido pôr em prática e aprofundar ainda mais, os conhecimentos adquiridos ao longo do semestre na disciplina, o que vai ser de extrema utilidade, no nosso futuro na área de informática.