deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

Relatório do Projeto Nº1 de Algoritmos e Estruturas de Dados

Trabalho realizado por:

Tomás Brás **Nº 112665**
Afonso Ferreira **Nº 113084**

Algoritmos e Estruturas de Dados

Prof. Pedro Lavrador

Prof. Joaquim Madeira

Ano Letivo 2023/2024

Índice

Introdução

Neste relatório, após a conclusão do desenvolvimento do tipo abstrato de dados da função **ImageLocateSubImage()** e **ImageBlur()** realizámos uma sequência de testes de modo a verificar o número de comparações e o tempo que o programa demora a executá-los. O nosso objetivo com estas informações é, para cada função, realizar uma análise formal da complexidade do algoritmo utilizado.

Análise da complexidade da função ImageLocateSubImage()

Testes realizados e respectivos resultados

Pior caso

Tamanho da subimagem	TIME	CALTIME	PIXMEM	PIXCMP	Iterations
(1,1)	0.000902	0.000623	524288	262144	262144
(2,2)	0.001952	0.001347	2088968	104484	104484
(4,4)	0.006405	0.004420	8290592	4145295	4145295
(8,8)	0.026566	0.018332	32643200	16321600	16321600
(16,16)	0. 099322	0.068538	126468608	63234304	63234304
(32,32)	0.269960	0.186289	473827328	236913664	236913664
(64,64)	0.913790	0.630572	1651515392	825757696	825757696
(128,128)	2.679500	1.849022	4857036800	2428518400	2428518400
(256,256)	4.876064	3.364788	8657174528	4328587264	4328587264

Procura Normal

Procura de subimagens de diferentes tamanhos na imagem mandrill_512x512.pgm que se encontra dentro da diretoria ./pgm/medium/.

Tamanho de uma subimagem	TIME	CALTIME	PIXMEM	PIXCMP	Iterations
(1,1)	0.000131	0.000091	67584	33792	33792
(2,2)	0.001677	0.001174	524646	262323	262323
(4,4)	0.001585	0.001109	520908	260454	260454
(8,8)	0.001502	0.001051	513032	256516	256516
(16,16)	0.001465	0.001025	496934	248467	248467
(32,32)	0.001424	0.000996	468372	234186	234186
(64,64)	0.001275	0.000892	414884	207442	207442
(128,128)	0.000929	0.000650	330626	165313	165313
(256,256)	0.000500	0.000359	263648	131824	131824

Melhor Caso

OBJ

Tamanho de uma subimagem	TIME	CALTIME	PIXMEM	PIXCMP	Iterations
(1,1)	0.000004	0.000003	2	1	1
(2,2)	0.000003	0.000002	8	4	4
(4,4)	0.000002	0.000001	32	16	16
(8,8)	0.000002	0.000002	128	64	64
(16,16)	0.000002	0.000002	512	256	256
(32,32)	0.000003	0.000002	2048	1024	1024
(64,64)	0.000004	0.000003	8192	4096	4096
(128,128)	0.000010	0.000007	32768	16384	16384
(256,256)	0.000023	0.000016	131072	65536	65536

Análise formal da função ImageLocateSubImage()

Pior Caso:

O pior caso acontece quando a imagem original tem uma só com cor e a sub-imagem tem todos os pixels com a mesma cor, exceto o último que terá outra tonalidade. Isto fará com que a função percorra todas as janelas, e compare todos os pixels, o que resulta na seguinte complexidade:

$$\sum_{i=0}^{w1-w2}\sum_{j=0}^{h1-h2}\sum_{n=0}^{w2-1}\sum_{n=0}^{h2-1} 1 = \sum_{i=1}^{w1-w2+1}\sum_{j=1}^{h1-h2+1}\sum_{n=1}^{w2}\sum_{n=1}^{h2} 1$$

$$= (w1-w2+1)(h1-h2+1)(h2w2) = w1h1h2w2 - w1(h2)^2w2 + w1w2h2 - (w2)^2h1h2 +$$

$$+ (w2)^2(h2)^2 - (w2)^2h2 + h1w2h2 - (h2)^2w2 + h2w2$$

$\theta(N \cdot n)$, *sendo N o número de pixeis da imagem grande e n o número de pixeis da subimagem*

Melhor Caso:

O melhor caso da função ImageLocateSubImage() será quando a sub-imagem for encontrada na primeira iteração, ou seja, na posição (0,0). O número de acesso aos pixels (**PIXCMP**) será igual ao número de píxeis da sub-imagem. Irá comparar cada pixel e verificar se têm o mesmo valor de cinzento. Irá se verificar que têm o mesmo valor, trocando o valor dos ponteiros x e y para 0, 0.

$$\sum_{i=0}^{w2-1}\sum_{j=0}^{h2-1} 1 = \sum_{i=1}^{w2}\sum_{j=1}^{h2} 1 = h2 \cdot w2$$

$O(n)$, *sendo n o numero de pixeis da subimagem*

Outras tentativas de algoritmos:

Durante a realização desta função tentámos aplicar algoritmos que otimizassem mais o código. Um destes era calcular uma hashtable da subimagem e ir comparando às hashtables das janelas da imagem grande. Se fossem iguais, passaríamos à comparação pixel a pixel como é feito no imageMatchSubImage(). Isto apesar de ser eficaz em alguns casos, tornou-se mais lento em diversos exemplos, acedendo à memória muitas mais vezes e com mais iterações, sendo o único ponto positivo, o baixo número de comparações entre pixels. Por estas razões, optámos pelo método realizado posteriormente.

Análise a complexidade da função ImageBlur()

Testes realizados e respetivos resultados

Imagem 300x300 (dx, dy)	TIME	CALTIME	PIXMEM	Iterations
(5,5)	0.027894 0.000894	0.018426 0.0005910	11266827 1170000	11052721 180000
(20,20)	0.393060 0.000845	0.259648 0.000558	142151202 1170000	152390281 180000
(40,40)	1.406030 0.000887	0.928795 0.000887	515831202 1170000	594523161 180000

Imagem 512x512 (dx, dy)	TIME	CALTIME	PIXMEM	Iterations
(50,20)	2.091644 0.002903	1.453134 0.002017	1014948418 3407872	1090044973 524288
(100,40)	7.919775 0.003577	5.676454 0.002564	3706162178 3407872	4284916633 524288
(200,80)	37.399469 0.002795	26.80585 0.002003	12570576898 3407872	16990715953 524288

VERMELHO - Função OldBlur (não otimizada)

PRETO - Função Blur (otimizada)

Estratégias algorítmicas utilizadas e respetivos resultados

Função OldBlur (menos eficiente)

Ao ler o que faria a função Blur, pela primeira vez a nossa ideia inicial, foi iterar por todos os pixels da função e depois iterar pela janela de cada pixel ([x-dx, x+dx]x[y-dy,y+dy] (sendo x e y a coordenada do pixel) somando todos os valores da mesma. No fim iteramos por todos os pixels e substituímos pelo valor da média da janela de cada um.

Isto apesar de funcionar não é viável quando temos imagens/janelas muito grandes, visto que a complexidade da função é:

$$\sum_{n=0}^{w-1}\sum_{m=0}^{h-1}\sum_{j=x-dx}^{x+dx}\sum_{i=y-dy}^{y+dy} = \sum_{n=1}^{w}\sum_{m=1}^{h}\sum_{j=-dx}^{dx}\sum_{i=-dy}^{dy}$$

$$= w \cdot h \cdot (2dx+1)\cdot(2dy+1) = O(4 \cdot N \cdot dx \cdot dy + N)$$

$$\theta(N \cdot dx \cdot dy), \text{ sendo } N \text{ o número de pixeis da imagem.}$$

Os testes na tabela acima mostram o porquê desta função não ser viável, por exemplo na segunda tabela quando o (dx, dy) = [200,80) ou seja, uma janela relativamente grande, a função já demora por volta de 37 segundos a correr e com um número muito elevado de iterações. Por esta razão, decidimos procurar uma solução melhor.

Função Blur (mais eficiente)

Ao repararmos que a função era muito ineficiente e com uma grande complexidade, tentámos arranjar uma solução melhor, e foi aí que nos deparámos com o algoritmo Box Blur, um algoritmo que nos permitiu reduzir a complexidade para O(N).

$$\sum_{i=0}^{w} \sum_{j=0}^{h} 1 + \sum_{i=0}^{w} \sum_{j=0}^{h} 1 = 2wh$$

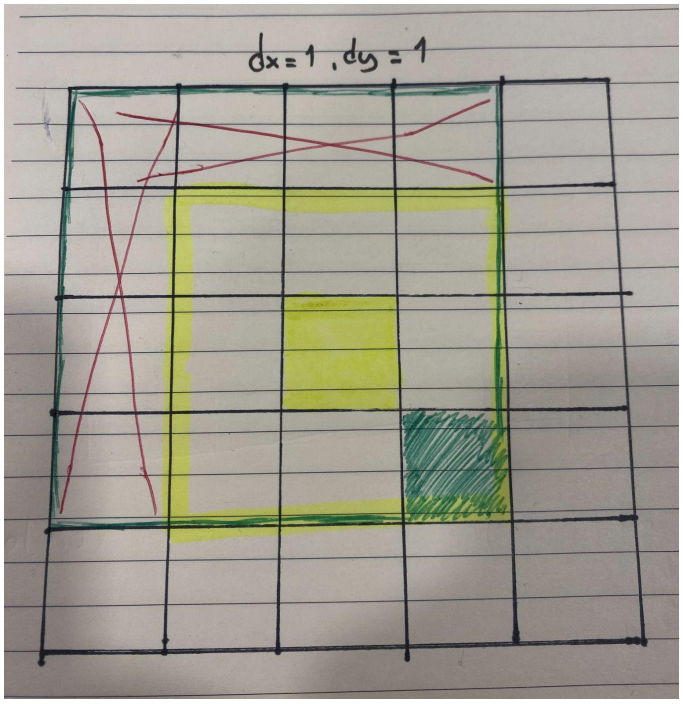
$$O(N), \text{ sendo } N \text{ o número de pixeis}$$

O algoritmo usado foi, tal como foi dito anteriormente, o box blur. Basicamente vamos criar um novo array onde iremos armazenar a soma da tonalidade de todos os pixels anteriores na imagem. Assim, para cada pixel vão ser calculadas as somas cumulativas dos pixels anteriores (lado esquerdo e superior), pois os que se encontram à sua direita são os que vão ser determinados posteriormente. Na adição é necessário subtrair o valor do pixel da diagonal esquerda, devido a alguns valores terem sido adicionados duas vezes.

Isto é muito útil, porque para alterar o valor de cada pixel tendo em conta os valores de filtragem (dx e dy) apenas temos que subtrair ao último pixel as colunas e linhas que não se encontram dentro do intervalo de pixels (dx-x dx+x) (dy-y dy+y). A imagem seguinte é um exemplo de como a função calcula o valor da tonalidade de um pixel.

p(3,3) = p(4,4) - p(1,4)-p(4,1) + p(1,1)

(necessitamos de somar p(1,1) visto que foi removido duas vezes)



Conclusão

Concluindo, o desenvolvimento destas funções foi um sucesso, pois as duas estão funcionais e viáveis para uso. Este projeto foi bastante interessante, pois nos ensinou que muitas vezes a primeira solução não é a mais correta e temos que ir à procura de opções para otimizar a mesma. Para além disso, mostrou-nos a importância de realizar testes e os analisar para entender melhor o algoritmo e se podemos alterar algum aspeto que nos irá beneficiar.