

|  |
| --- |
|  |

## Relatório do Projeto de Algoritmos

## e Estruturas de Dados

Trabalho realizado por:

Tomás Brás Nº 112665

Afonso Ferreira Nº 113084

**Profº Pedro Lavrador**

**Profº Joaquim Machado**

Índice

[Relatório do Projeto de Algoritmos 1](#_gjdgxs)

[e Estruturas de Dados 1](#_sojwjhj1tkbu)

[**Análise a complexidade da função ImageLocateSubImage() 3**](#_5j2ygwlzs8ma)

[Testes realizados e respectivos resultados 3](#_30j0zll)

[Pior caso 3](#_55x4gjbtej3l)

[Procura Normal 3](#_neahmyo69t39)

[Melhor Caso 4](#_c8nz0k2wl58s)

[￼ 4](#_ioiwnh360ljm)

[**Análise formal da complexidade 4**](#_3shtzip3pzvn)

[Pior Caso: 4](#_ed2k7uhp6jyk)

[Melhor Caso: 4](#_1y5tols3ts77)

[**Testes realizados e respetivos resultados 5**](#_y5fhrwynz58w)

[**Estratégias algorítmicas utilizadas e respetivos resultados 5**](#_ky3tmh3fj9ay)

[Função OldBlur (menos eficiente) 5](#_clr2mni0bqr1)

[Função Blur (mais eficiente) 6](#_nbolcpxto06j)

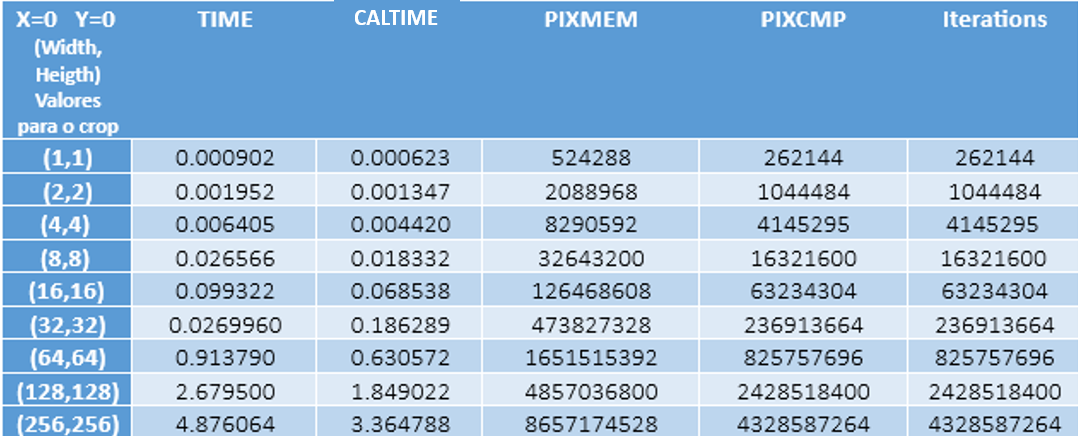
Introdução

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.

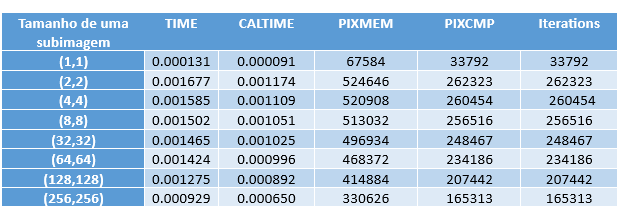
## **Análise a complexidade da função ImageLocateSubImage()**

### Testes realizados e respectivos resultados

### Pior caso



### Procura Normal

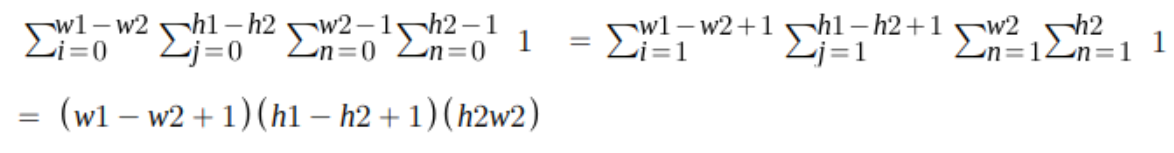


### Melhor Caso

### ￼

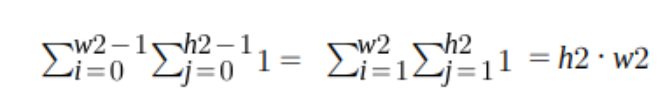
## **Análise formal da complexidade**

### 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:*

### 

### Melhor Caso:

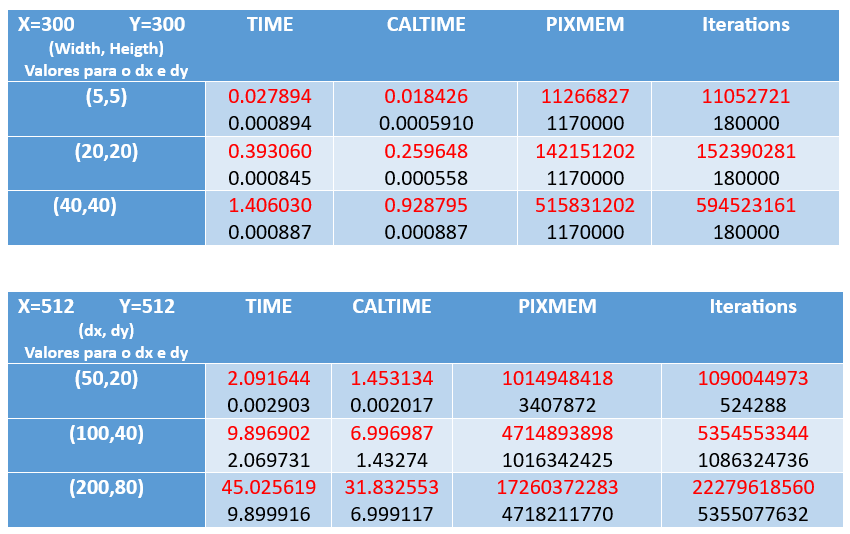
O melhor caso da função ImageLocateSubImage() será quando a sub-imagem for encontrada na primeira iteração, ou seja, na posição (0,0). O número de acesso aos pixels (**PIXCMP**) será igual ao número de píxeis da sub-imagem. Irá comparar cada pixel e verificar se têm o mesmo valor de cinzento. Irá se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



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 pelo que preferimos voltar ao método original.

**Análise a complexidade da função ImageBlur()**

## Testes realizados e respetivos resultados



## **VERMELHO - Função OldBlur (não otimizada)**

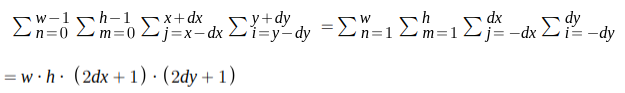
**PRETO - Função Blur (otimizada)**

## 

## **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][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:

### 

### 

### 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).



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)