Projeto AED - O TAD image8bit

Pedro Pinto nº115304 ; João Pinto nº104384

---

1. Introdução

No contexto da disciplina de Algoritmos e Estruturas de Dados, este projeto conduziu ao desenvolvimento do Tipo Abstrato de Dados (TAD) *image8bit*. Este TAD é capaz de lidar com imagens em tons de cinzento, onde cada pixel possui uma intensidade que varia de 0 a 255 (8 bits), conforme definido nas funções do ficheiro de interface. Para avaliar o desenvolvimento, utilizámos a ferramenta *imageTool*, aplicando testes específicos em cada caso. Contudo, para analisar a complexidade computacional das funções em destaque neste relatório, optou-se por desenvolver mecanismos de teste simples e eficientes. Para testar as funções *ImageLocateSubImage()* e *ImageBlur()*, basta executar os ficheiros *main.m* nos diretórios *testLocate* e *testBlur*, respetivamente. Este script *Matlab* possibilita diversos testes ajustáveis, alterando apenas algumas constantes. O script inicia executando um ficheiro *shell* (*execute_locateTests.sh* ou *execute_blurTests.sh*), que compila e executa testes no ficheiro em C (*imageTestLocate.c* ou *imageTestBlur.c*). Este ficheiro realiza testes sobre *image8bit.c*, gerando e processando diferentes imagens. Após a execução, o script *shell* escreve os resultados nos ficheiros *data_locateTests.txt* ou *data_blurTests.txt*. Esses resultados são lidos e processados pelo script *Matlab*, fornecendo gráficos conforme solicitado. Este módulo de teste (Figura 1) permite uma análise robusta e abrangente do desempenho do TAD *image8bit*. É importante destacar que a estrutura de testes principal é a mesma para ambas as funções.

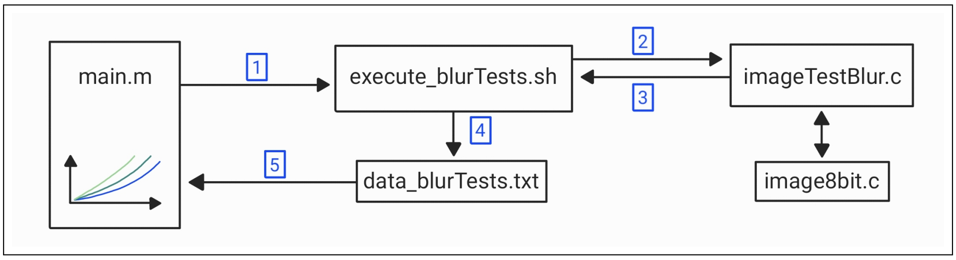
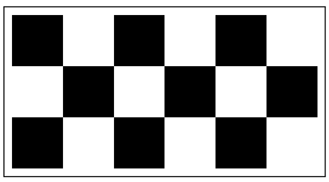


Figura 1. Módulo de testes desenvolvido para a função *ImageBlur()*.

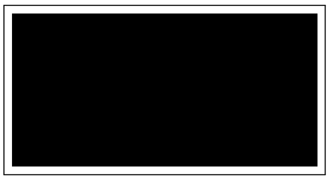
2. Análise da complexidade da função *ImageLocateSubImage()*

Desenvolvemos a função *ImageLocateSubImage()* que por sua vez vez invoca a função *ImageMatchSubImage()* também desenvolvida. Esta função recebe como argumento duas imagens, *img1* e *img2*, e procura *img2* dentro da *img1*. Se a função conseguir localizar a subimagem dentro da imagem é retornado 1 e a posição em que foi encontrada é guardada nas variáveis **px* e **py*. Considerando *img1* com $width$=w_1 e $height$=h_1 e a subimagem *img2* com $width$=w_2 e $height$=h_2 podemos fazer uma análise dos casos possíveis. Apresentamos um exemplo em baixo, em que *img1* tem w_1=6 e h_1=3 e a subimagem *img2* tem w_2=2 e h_2=1, onde iremos analisar o número de comparações para os diferentes casos.

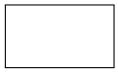


(a) *img1* - Melhor caso (encontro).

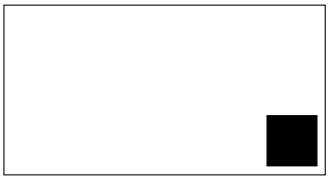
(b) *img2* - Melhor caso (encontro).



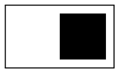
(c) *img1* - Melhor caso (não encontro).



(d) *img2* - Melhor caso (não encontro).



(e) *img1* - Pior caso (encontro).



(f) *img2* - Pior caso (encontro).

- 1) Melhor caso (imagem é encontrada) = $w_2 \times h_2$. No exemplo de cima apenas faria 2 comparações.
- 2) Melhor caso (imagem não é encontrada) = $(w_1 - w_2 + 1) \times (h_1 - h_2 + 1)$. No exemplo de cima, apenas faria uma comparação na função *ImageMatchSubImage()*. A função *ImageLocateSubImage()* invocava a anterior 15 ($(6 - 2 + 1) \times (3 - 1 + 1) = 15$) vezes seguidas.
- 3) Pior caso (imagem é encontrada) = Pior caso (imagem não é encontrada) = $w_2 {\cdot} h_2 {\times} (w_1 {-} w_2 {+} 1) {\times} (h_1 {-} h_2 {+} 1)$. No exemplo de cima, falhava sempre na segunda comparação na função *ImageMatchSubImage()*. Número de comparações realizadas = $2 \times (6 - 2 + 1) \times (3 - 1 + 1) = 30$.

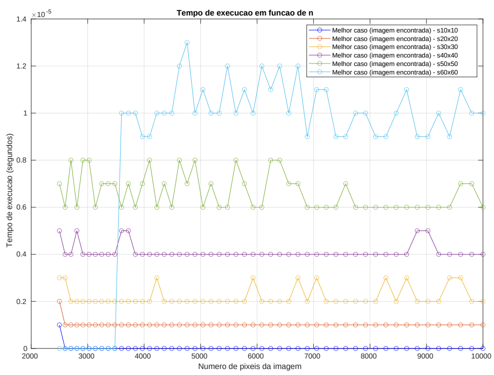
A dedução das expressões utilizadas em cima encontra-se na equação 1. Ao longo da dedução foi utilizado a expressão $\sum_{i=1}^{k} 1 = k$. O fator $w_2 \cdot h_2$ não é contabilizado na situação 2), uma vez que apenas faz uma comparação na função *ImageMatchSubImage()*.

$$\sum_{x=0}^{w_1-w_2} \sum_{y=0}^{h_1-h_2} \sum_{i=0}^{w_2-1} \sum_{j=0}^{h_2-1} 1 = \sum_{x=0}^{w_1-w_2} \sum_{y=0}^{h_1-h_2} w_2 \cdot h_2 = \sum_{x=0}^{w_1-w_2} w_2 \cdot h_2[(h_1 - h_2) + 1] = w_2 \cdot h_2[(h_1 - h_2) + 1] \times [(w_1 - w_2) + 1] \quad (1)$$

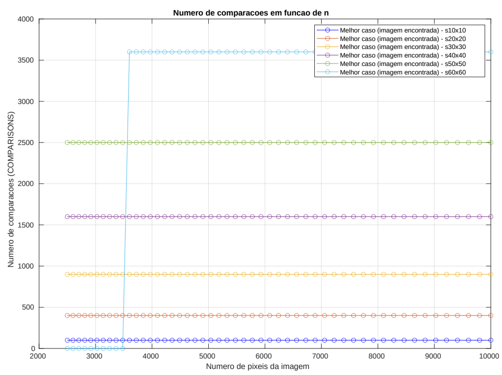
Representando por $C_T^E(h_1, w_1, h_2, w_2)$ o número de comparações total quando a imagem é encontrada (E) e $C_T^N(h_1, w_1, h_2, w_2)$ quando a imagem não é encontrada (N) temos:

$$C_T^E \in \begin{cases} \Omega(h_2 \cdot w_2) \\ \Theta\left(\frac{w_2 \cdot h_2[1+(w_1-w_2+1)\times(h_1-h_2+1)]}{2}\right) \\ \mathcal{O}(w_2 \cdot h_2 \times (w_1 - w_2 + 1) \times (h_1 - h_2 + 1)) \end{cases} \qquad C_T^N \in \begin{cases} \Omega((w_1 - w_2 + 1) \times (h_1 - h_2 + 1)) \\ \Theta\left(\frac{(w_2 \cdot h_2+1)\times(w_1-w_2+1)\times(h_1-h_2+1)]}{2}\right) \\ \mathcal{O}(w_2 \cdot h_2 \times (w_1 - w_2 + 1) \times (h_1 - h_2 + 1)) \end{cases}$$

1) Melhor caso (imagem é encontrada) = $w_2 \times h_2$. Neste caso, o número de comparações depende exclusivamente das dimensões de *img2*. De forma a analisarmos este caso, gerámos diferentes imagens quadradas *img1* com tamanhos distintos, desde *width* igual a 50 até 100, todas com cor preta. Em seguida, analisámos o número de comparações feitas quando a função é utilizada para encontrar *img2*, também de cor preta, dentro dessas imagens *img1*. A Figura 4b mostra os resultados para vários tamanhos de *img2*, desde *width* igual a 10 até *width* igual a 60. Por exemplo, podemos ver nesta Figura que quando *img2* tem *width*=*height*=50, há 2500 comparações, confirmando $w_2 \times h_2$. No entando, no caso da subimagem ser maior que a imagem a função percebe que não é possível a *img2* estar dentro de *img1*, portanto, o número de comparações é 0 e o tempo de execução também é muito próximo de 0, o que é visível nas Figuras 4a e 4b para s60x60. Na Figura 4a, vemos que o tempo de execução é limitado e depende apenas do tamanho de *img2*. Em resumo, aumentar o tamanho de *img2* implica um maior tempo de execução e um maior número de comparações. No entanto, para uma determinada *img2* o número de comparações é constante.

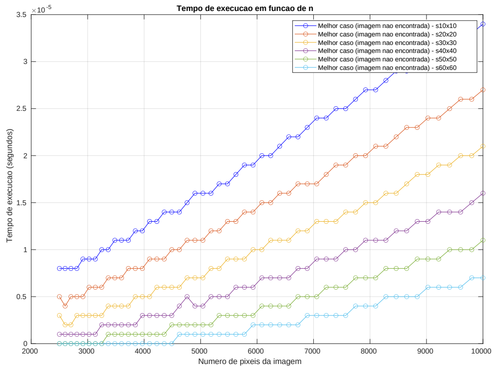


(a) Tempo de execução (s) em função do número de pixéis.

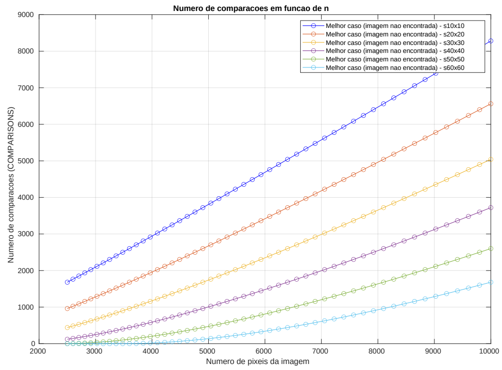


(b) Número de comparações em função do número de pixéis.

2) Melhor caso (imagem não é encontrada) $= (w_1 - w_2 + 1) \times (h_1 - h_2 + 1)$**.** De forma a ilustrarmos este caso, gerámos diferentes imagens quadradas *img1* com os mesmos tamanhos mencionados no caso 1). Estas *img1* têm cor preta e as *img2* são de cor branca. Os resultados obtidos para este caso encontram-se na Figura 5. Por exemplo, quando *img1* tem *width*=100 e *img2* tem *width*=30, o número de comparações é igual a $(100-30+1)\times(100-30+1) = 5041$, como podemos constatar na Figura 5b para s30x30 (ver último ponto da curva a amarelo).

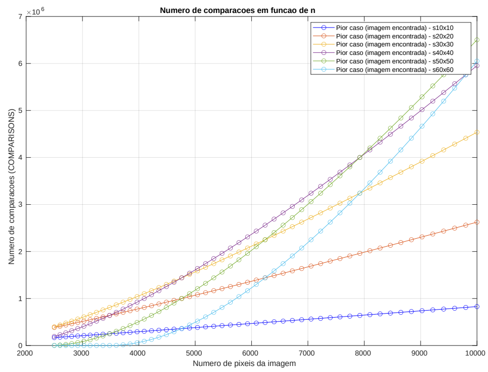


(a) Tempo de execução (s) em função do número de pixéis.

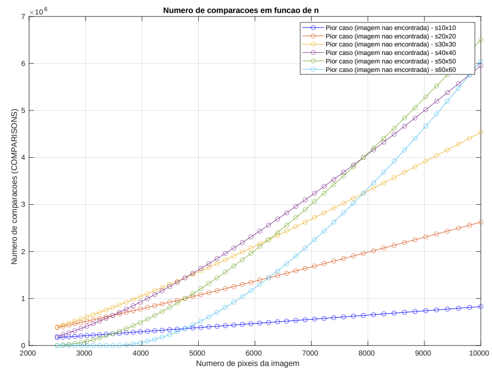


(b) Número de comparações em função do número de pixéis.

3) Pior caso (imagem é encontrada) = Pior caso (imagem não é encontrada) $= w_2 \cdot h_2 \times (w_1 - w_2 + 1) \times (h_1 - h_2 + 1)$**.** A expressão reflete que o número de comparações é igual quer a imagem seja encontrada quer não seja. Neste caso, gerámos diferentes imagens quadradas *img1* com os mesmos tamanhos mencionados nos casos anteriores. Em ambos os casos, encontrar ou não encontrar a imagem, as *img2* criadas têm cor preta e apenas no canto inferior direito o último pixel tem cor branca. Quando a imagem é encontrada as *img1* têm cor preta e apenas no canto inferior direito o último pixel tem cor branca. No caso em que a imagem não é encontrada, as *img1* têm apenas cor preta. Os resultados obtidos comprovam o esperado uma vez que o número de comparações é o mesmo nos dois casos, ver Figura 6a e 6b. Por exemplo, quando *img1* tem *width*=100 e *img2* tem *width*=60, o número de comparações é igual a $60 \cdot 60 \times (100-60+1) \times (100-60+1) = 6051600$ (ver o último ponto na curva azul clara).



(a) Nº de comparações quando a imagem é encontrada.



(b) Nº de comparações quando a imagem não é encontrada.

3. Análise da complexidade da função *ImageBlur()*

Desenvolvemos duas versões da função *ImageBlur()* sendo a segunda uma versão optimizada. A 1ª versão da função vai a cada pixel da imagem e vê os pixeís à volta consoante a janela de *Blur* considerada, dx e dy. Caso seja um pixel válido soma a uma variável denominada de *sum* e no final de percorrer todos os pixéis da janela faz uma média com o número de pixéis válidos, que se encontra armazenado na variável *count*. Esta função funciona mas não é óptima uma vez que depende da janela de *Blur* considerada. Conseguimos perceber esta dependência através da expressão do número de comparações, cuja dedução apresentamos. Usámos a expressão $\sum_{py=-dy}^{dy} 1 = (2 \cdot \sum_{py=1}^{dy} 1) + 1$.

$$\begin{aligned}
\sum_{x=0}^{w-1}\sum_{y=0}^{h-1}\sum_{px=-dx}^{dx}\sum_{py=-dy}^{dy} 1 &= \sum_{x=0}^{w-1}\sum_{y=0}^{h-1}\sum_{px=-dx}^{dx} [(2 \times \sum_{py=1}^{dy} 1) + 1] = \sum_{x=0}^{w-1}\sum_{y=0}^{h-1}\sum_{px=-dx}^{dx} (2 \cdot dy + 1) \\
&= \sum_{x=0}^{w-1}\sum_{y=0}^{h-1} [(\sum_{px=-dx}^{dx} 2 \cdot dy) + (\sum_{px=-dx}^{dx} 1)] = \sum_{x=0}^{w-1}\sum_{y=0}^{h-1} [2 \cdot dy(2 \cdot dx + 1) + 2 \cdot dx + 1] \\
&= (2 \cdot dy + 1)(2 \cdot dx + 1) \sum_{x=0}^{w-1}\sum_{y=0}^{h-1} 1 = w \cdot h \times (2 \cdot dy + 1) \times (2 \cdot dx + 1) \qquad (2)
\end{aligned}$$

Resumidamente, o número de comparações é proporcional a $w \cdot h \times (2 \cdot dy + 1) \times (2 \cdot dx + 1)$, ou seja, depende da janela de *Blur*. É proporcional uma vez que depende do número de comparações feitas na função *ImageValidPos()*.

A 2ª versão da função, independente da janela de *blur* considerada, inicialmente cria uma matriz (*array*) com as somas acumulativas de todos os pixéis. Em seguida, são introduzidas verificações para garantir que as coordenadas da janela (*rx*, *lx*, *by*, *ty*) permaneçam dentro dos limites da imagem. Caso alguma dessas coordenadas ultrapasse os limites, o divisor é ajustado para levar em conta apenas os pixéis presentes na área válida da janela. Após isso, são retiradas da matriz acumulativa quatro somas (*r_bottom*, *r_top*, *l_bottom*, *l_top*) com base nas coordenadas da janela (adaptada ao pixel em questão). Essas somas são utilizadas para calcular a média ponderada, onde o divisor é o produto do número de pixéis na largura e altura da janela. A média ponderada é obtida como a diferença entre as somas acumulativas relevantes divididas pelo divisor. O valor médio resultante é então arredondado para o número inteiro mais próximo e utilizado para atualizar o pixel na posição atual da imagem original. Esses passos são repetidos para todos os pixéis da imagem, proporcionando assim o efeito de desfoque desejado. Conseguimos perceber esta independência através da análise da expressão do número de comparações, cuja dedução apresentamos na equação 3.

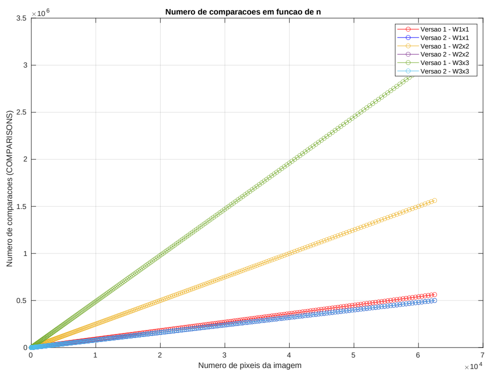
$$\left(\sum_{x=0}^{w-1} \cdot \sum_{y=0}^{h-1} 6\right) + \left(\sum_{y=0}^{h-1}\sum_{x=0}^{w-1} 1 + \sum_{x=0}^{w-1}\sum_{y=0}^{h-1} 1\right) = 6 \cdot hw + 2 \cdot hw = 8 \cdot hw \tag{3}$$

Resumidamente, o número de comparações é proporcional a $8 \cdot hw$, ou seja, não depende da janela de *Blur*.

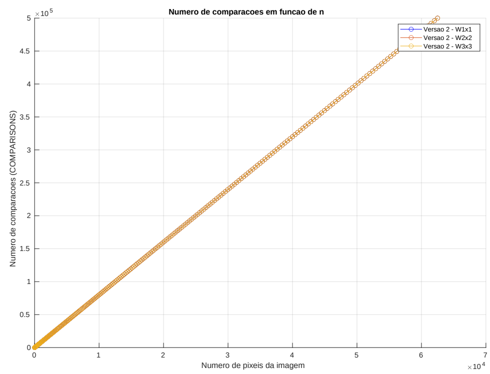
3.1. Dados experimentais - *BySize*

Gerámos diferentes imagens quadradas com tamanhos distintos (*BySize*), desde *width* igual a 1 até 250. Em seguida aplicámos a estas imagens as duas versões da função *ImageBlur()*. Na Figura 7a encontram-se os resultados obtidos para diferentes janelas de *Blur*, desde *dx=dy=1* até *dx=dy=3*. Como podemos observar, na versão 1 do algoritmo o número de comparações aumenta bastante quando a janela de *Blur* aumenta. Por outro lado, na versão 2 do algoritmo o número de comparações é independente da janela, como podemos observar na Figura 7b onde se encontram sobrepostos os resultados da versão 2 para as três janelas.

O número de comparações realizadas pelo **algoritmo 1** é descrito pela equação 2 e a sua validade é comprovada na Figura 7a. Por exemplo, quando a *img* tem *width=200* (40000 pixéis) se for utilizada uma janela dx=dy=2 (W2x2), o número de comparações previsto teoricamente é $200 \times 200 \times (2 \times 2 + 1) \times (2 \times 2 + 1) = 10^6$, o que é igual ao valor obtido experimentalmente (ver curva a amarelo). Por outro lado, o número de comparações realizadas pelo **algoritmo 2** é descrito pela equação 3, e a sua validade também é comprovada na Figura 7b. Por exemplo, quando a *img* tem *width=250* (62500 pixéis), se for utilizada uma janela dx=dy=3 (W3x3), o número de comparações previsto teoricamente é $8 \times 250 \times 250 = 5 \times 10^5$, o que é igual ao valor obtido experimentalmente visível no último ponto.



(a) Nº de comparações para a versão 1 e 2 do algoritmo.

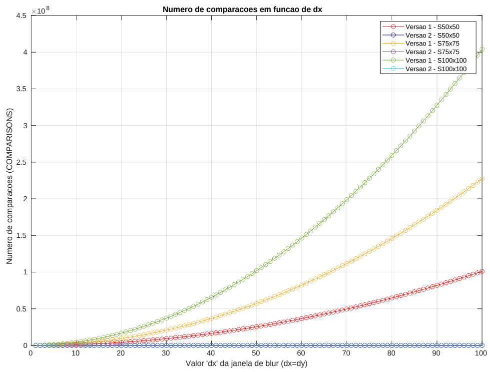


(b) Nº de comparações apenas para a versão 2 do algoritmo.

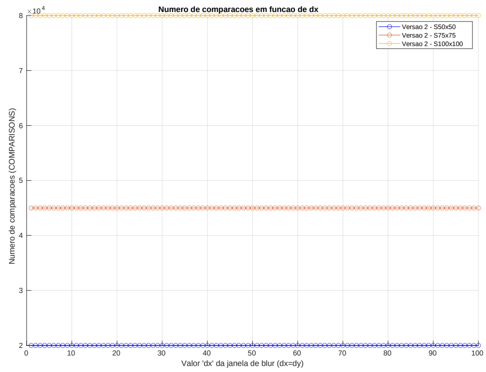
3.2. Dados experimentais - *ByWindow*

Nesta subsecção analisámos de forma mais detalhada o impacto do aumento da janela de *Blur* no número de comparações obtidos para a versão 1 e 2 do algoritmo. Variou-se o desde dx=dy=1 até dx=dy=100 (*ByWindow*) e registou-se o número de comparações para três imagens de tamanhos distintos. Como podemos observar na Figura 8a, apenas na versão 1 o número de comparações depende da janela considerada. Na versão 2 da função, o número de comparações para cada uma das imagens é independente da janela, sendo bastante notório na Figura 8b.

A expressão obtida na equação 2 é validada novamente. Na Figura 8a, quando a *img* tem *width=100* (10000 pixéis), se for utilizada uma janela dx=dy=70, o número de comparações previsto teoricamente é $100 \times 100 \times (2 \times 70 + 1) \times (2 \times 70 + 1) = 1.9881 \times 10^8$, o que é igual ao valor obtido experimentalmente (ver curva a verde para dx=70). Também comprovámos a validade da equação 3 para a versão 2. Na Figura 8b, quando a *img* tem *width=100* (10000 pixéis), se for utilizada uma janela dx=dy=70, o número de comparações previsto teoricamente é $8 \times 100 \times 100 = 8 \times 10^4$, o que é igual ao valor obtido experimentalmente (ver curva a amarelo da Figura 8b).

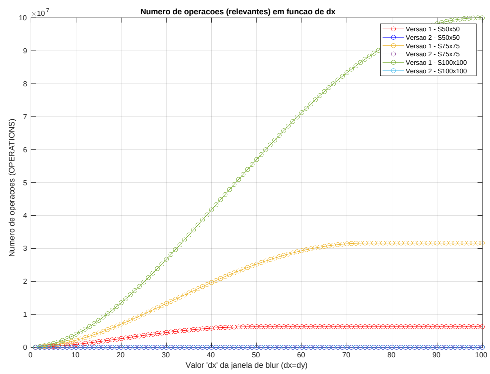


(a) Nº de comparações para a versão 1 e 2 do algoritmo.

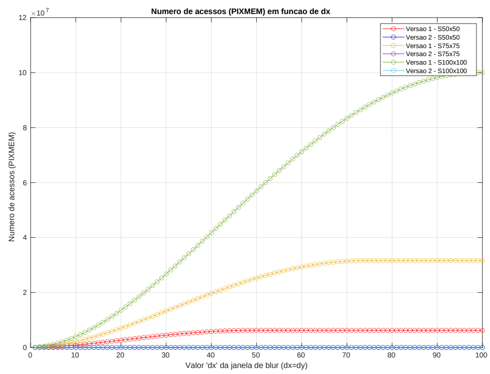


(b) Nº de comparações apenas para a versão 2 do algoritmo.

Analisou-se ainda a variação do número de operações e do número de acessos em função da janela considerada para ambas as versões desenvolvidas, ver Figuras 9a e 9b, respetivamente. Nas operações não contabilizámos divisões porque têm o mesmo crescimento em ambas as versões. As adições/subtrações (*operations*) é que têm uma diferença significativa de versão para versão. Como podemos observar nas Figuras 9a e 9b, para a versão 1 o valor do número de operações e de acessos acaba por estabilizar, a partir de dx=$width - 1$ (ver por exemplo curva a amarelo para dx=74).



(a) Nº de operações para a versão 1 e 2 do algoritmo.



(b) Nº de acessos para a versão 1 e 2 do algoritmo.

4. Conclusão

No âmbito deste trabalho concluímos o desenvolvimento das funções especificadas no ficheiro de implementação e garantimos o seu correto funcionamento. Desenvolvemos ainda duas funções *ImageLocateSubImage()* e *ImageBlur()* cuja complexidade foi analisada neste relatório. São ambos algoritmos deterministas, pois devolvem sempre o mesmo resultado quando executados com os mesmos dados de entrada. Para a função *ImageLocateSubImage()* foram analisados os casos possíveis e o respetivo número de comparações, C_T^E e C_T^N, ver expressões 2. Para o melhor caso quando a imagem é encontrada ($w_2 \times h_2$) concluiu-se que dependia unicamente das dimensões de *img2*, como podemos ver na Figura 4b. No melhor caso em que a imagem não é encontrada, o número de comparações é dado por $(w_1 - w_2 + 1) \times (h_1 - h_2 + 1)$, sendo portanto linear em relação a w_1 e h_1 considerando o tamanho de *img2* constante, como podemos constatar nas Figuras 5a e 5b. O pior caso em que a imagem é encontrada coincide ao pior caso quando a imagem não é encontrada e corresponde a $w_2 \cdot h_2 \times (w_1 - w_2 + 1) \times (h_1 - h_2 + 1)$.

Relativamente à função *ImageBlur()* foi feita uma análise formal de dois algoritmos distintos e comparada a diferença entre os dois. A principal optimização da versão 2 é o facto de não depender da janela de *Blur* considerada, ver equação 3. Apesar dos gráficos colocados neste relatório terem sido obtidos com imagens quadradas, também realizámos testes com outras imagens tendo-se verificado o funcionamento esperado. Dada a limitação do número de páginas deste relatório não foi possível analisar todos os gráficos obtidos. Os gráficos obtidos das duas funções estão nos diretórios *testBlur* e *testLocate*. Dentro do diretório *testBlur* os gráficos encontram-se separados por *bySize* e *byWindow*. Este trabalho foi desafiante tendo nos permitido conjugar diferentes linguagens de programação, de forma a podermos realizar os testes da forma mais eficiente e robusta possível, como explicado no esquema da Figura1.