# Efficient Dilation, Erosion, Opening and Closing Algorithms *

JOSEPH (YOSSI) GIL RON KIMMEL

Department of Computer Science

Technion–Israel Institute of Technology

Technion City, Haifa 32000, Israel

Submitted to IEEE Transactions on Pattern Analysis and Machine Intelligence

**Abstract**

We propose an efficient and deterministic algorithm for computing the one dimensional dilation and erosion (max and min) sliding window filters. For a $p$-element sliding window, our algorithm computes the 1D filter using $1.5+o(1)$ comparisons per sample point. Our algorithm constitutes a deterministic improvement over the best previously known such algorithm, independently developed by van Herk [25] and by Gil and Werman [12] (the HGW algorithm). The results presented in this paper constitute also an improvement over the Gevorkian, Astola and Atourian [9] (GAA) variant of the HGW algorithm. The improvement over the GAA variant is also in the computation model. The GAA algorithm makes the assumption that the input is independently and identically distributed (the i.i.d. assumption), whereas our main result is deterministic.

We also deal with the problem of computing the dilation and erosion filters simultaneously, as required e.g., for computing the unbiased morphological edge. In the case of i.i.d. inputs we show that this simultaneous computation can be done more efficiently then separately computing each. We then turn to the *opening* filter, defined as the application of the min filter to the max filter, and give an efficient algorithm for its computation. Specifically, this algorithm is only slightly slower than the computation of just the max filter. The improved algorithms are readily generalized to two dimensions (for a rectangular window), as well as to any higher finite dimension (for a hyper-box window), with the number of comparisons per window remaining constant.

For the sake of concreteness, we also make a few comments on implementation considerations in a contemporary programming language.

* A preliminary version of this paper was published in the proceedings of ISMM'00 [11].

# 1 Introduction

In signal and image analysis one often encounters the problem of min (or max) computation in a window with $p$ elements in the one-dimensional (1D) case, or $p \times p$ elements in the two dimensional (2D) case. In mathematical morphology [20], the result of such an operator is referred to as the erosion (or dilation) of the signal with a *structuring element* given by a pulse of width $p$.

The *unbiased morphological edge* is obtained by subtracting the filtered min result from the filtered max result. This filter has numerous applications in image processing and analysis (see e.g., [15, 18]). To appreciate the visual effect of the morphological edge detector on actual images, consider Figure 1 which gives three examples of edge detection, using a window sized $p \times p$, for $p = 2, 4, 8, 16$.[1]

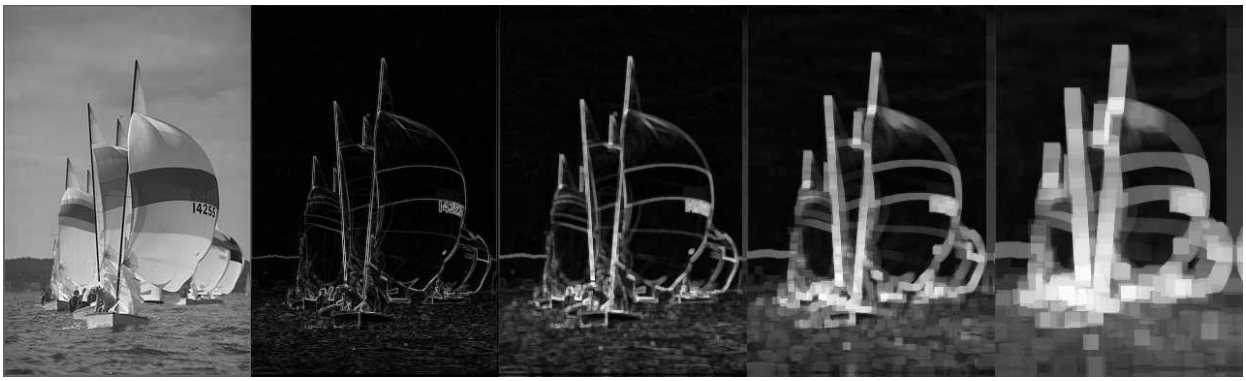

Figure 1: The effect of the unbiased morphological edge filter.
(Original image is shown on left frame, followed by the filtered image using rectangular windows sized $2 \times 2$, $4 \times 4$, $8 \times 8$, and $16 \times 16$.)

As can be seen from the figure, edges are accentuated by the morphological edge detection filter. These 'morphological gradients' need further processing to provide useful information, see e.g., [21, pp. 116-119]. It should be emphasized that even though the larger windows do not appear to highlight the edges as clearly as the smaller windows, they are useful as a pre-processing stage for scale-space analysis of images [2, 3, 8, 16, 19, 24]. Therefore, we are interested in the problem

[1]Note that when $p$ is even the filter is not centered at the pixels of the original image. Thus, the result of the application of the filters is not, strictly speaking, translation invariant. The filtered image represents a half a pixel shift with respect to the original image. Our algorithms are equally applicable for both odd and even $p$.

of computing the min and max problems for a *wide range* of window size $p$.

Filtering out image components smaller than a certain threshold is carried out relying on the min and max filters with a suitable window size. The *closing* (respectively *opening*) filter is obtained by feeding the results of the max (resp. min) filter to the min (resp. max) filter. Figure 2 gives an example of the application of the opening and closing filters. We see that the closing filter eliminates small dark image components, while the opening eliminates small white regions. In both filters, the size of the window determines the size of the image components that can be removed.

Figure 2: The effect of the opening (top) and closing(bottom) filters.
(Original image is shown on left frame, followed by the filtered image using rectangular windows sized $2 \times 2$, $4 \times 4$, $8 \times 8$, and $16 \times 16$.)

Gevorkian, Astola and Atourian [9] mention other applications of the min and max filters in pattern analysis, adaptive signal processing and morphological analysis. Morphological operations were found to be useful in 2D and 3D image processing and analysis. Applications include *micro-chip manufacturing* where machine vision techniques are used by inspection tools for qual-

ity control and fault detection, feature detection for *character recognition*, 3D data analysis such as *medical images*, graphics applications, and video processing. Also, the above mentioned basic morphological operations are by now core technology in many standard software tools such as Matlab.

Recently, a patent that deals with efficient applications of such operations was filed [7], where a pyramidal morphological 'structuring element' is applied to images by a two scans procedure. Actually, it can be shown that a pyramidal and a cone shaped structuring elements operating on an image can be obtained as the result of a version of Danielson's [6] distance map two pass algorithm. Our work deals with flat 'structuring elements' which are more useful in the general morphological setting.

The one-dimensional version of our most basic problem can be formulated as follows:

1D MAX-FILTER: Given a sequence $x_0, \ldots, x_{n-1}$, and an integer $p > 1$, compute

$$y_i = \max_{0 \le j < p} x_{i+j}$$

for $i = 0, \ldots, n-p$.

(Note the above definition does not include the border, i.e., $i > n - p$. In actual image processing the border usually receives some special treatment, e.g., mirroring, periodic condition, etc. Since our focus is algorithmic, the border is tacitly ignored henceforth.)

The 1D MIN-FILTER problem is similarly defined. For the remainder of this article, we will devote our attention mainly to max computations. Clearly, all results are equally applicable to min computation.

As usual in filtering, we assume that $p \ll n$. As an efficiency measure of algorithms for this problem we use the coefficient $C_1$, defined as the number of comparison operations per sample (or output) point as $n$ goes to infinity.

A trivial algorithm for the 1D MAX-FILTER problem gives

$$C_1 = p - 1.$$

On the other hand, since it is impossible to compute the filter without examining each input point at least once, there is a trivial information theoretical lower bound for the problem of

$$C_1 \ge 1.$$

We are unaware of any stronger lower bound for this problem.

Pitas [17] describes two non-trivial algorithms for the problem. The first such algorithm achieves $C_1 = O(\lg p)$. [2] Pitas's second algorithm achieves $C_1 = 3 + o(1)$ on the average for i.i.d. input.[3]

Note that the worst case performance of both of these algorithms depends on the window size.

van Herk [25] and later but independently Gil and Werman [12] gave an algorithm (HGW) for computing the max filter whose performance does not depend on $p$. Gil and Werman description of the algorithm is slightly more general, showing that the $p$ sized filter of any semi-ring operation, $\diamond$, can be computed using $3 - 4/p$ applications of $\diamond$ per sample point. Since max is a semi-ring operation, we have that

$$C_1 = 3 - 4/p.$$

In the special case that the semi-ring operation is max, and assuming i.i.d. input signal. Gevorkian, Astola and Atourian [9] gave an algorithm (GAA) that improves the HGW algorithm, achieving

$$E(C_1) = 2.5 - 3.5/p.$$

The expectation here is with respect to input distribution.

We note that for many applications, such an assumption is clearly invalid. In many natural signals, the probability that $x_{i+} > x_i$ is greater than $0.5$ if it is given that $x_i > x_{i-1}$. In the worst input case, such as an almost monotonically increasing signal, the performance of the algorithm of GAA is the same as the HGW algorithm.

In this paper, we describe an algorithm achieving further reduction,

$$C_1 = 1.5 + \frac{\lg p}{p} + O(1/p).$$

This improvement is *deterministic* and does not make any assumptions on the input distribution.

[2] We use $\lg(\cdot)$ to denote $\log_2(\cdot)$

[3] Here and henceforth we use the familiar $o(f(p))$ notation for the family of functions $g(p)$, such that

$$\lim_{p\to\infty} \frac{g(p)}{f(p)} = 0.$$

Thus, $o(1)$ are those functions which tend to zero as $p$ tends to infinity.

The morphological edge detector and other applications require the simultaneous computation of the min and max in each window, as summarized in the following problem definition.

1D MAX-MIN-FILTER: Given a sequence $x_0, \ldots, x_{n-1}$, and an integer $p > 1$, compute

$$y_i = \max_{0 \le j < p} x_{i+j}$$

$$z_i = \min_{0 \le j < p} x_{i+j}$$

for $i = 0, \ldots, n-p$.

We give an algorithm which solves 1D MAX-MIN-FILTER problem faster than solving 1D MAX-FILTER and 1D MIN-FILTER separately. Let $C_1^m$ be the number of comparisons per input sample for solving 1D MAX-MIN-FILTER. Then, our algorithm achieves

$$E(C_1^m) \approx 2 + 2.3466 \frac{\lg p}{p},$$

for the special case of independent input distribution, i.e., the expectation is with regard to input distribution. In the worst case this algorithm does not improve on the independent computation of the Min- and Max filters. However, for natural images, the algorithm makes such an improvement, although not to the extent possible for randomized inputs.

The problem posed by the opening filter is similar to 1D MAX-MIN-FILTER, since in both it is required to compute both a Min-Filter and a Max-Filter. However, the fact that in the opening filter this filters are computed sequentially, where the results of one filter are the input of the other, makes it much easier. Let $C_1^o$ be the number of comparisons per input sample for computing the opening filter. Then, we show that

$$C_1^o \le C_1 + O\left(\frac{\lg^2 p}{p}\right).$$

Clearly, the same result holds for the closing filter.

A 1D max filter can be extended to a rectangular window 2D max filter [12]. The extension is carried out by first applying the 1D filter along the rows, and then feeding the result to a 1D filter running along the columns. Let $C_2$ be the number of comparison operations required per input point for computing the 2D max filter. We have that

$$C_2 = 2C_1,$$

and more generally,

$$C_d = dC_1,$$

where $C_d$ is defined accordingly for the $d$-dimensional filter. We similarly have that

$$C_d^m = dC_1^m$$
$$C_d^o = dC_1^o.$$

**Outline** The remainder of this paper is organized as follows. Section 2 reviews the HGW algorithm. Our main result which improves this algorithm is described in Section 3. This section also makes a few comments on a randomized version of the algorithm and on an actual implementation of the algorithms in languages such as C [14]. In Section 4 we give our algorithm for the 1D MAX-MIN FILTER PROBLEM. The efficient algorithm for computing the opening (and closing) filter is described in Section 5. Finally, Section 6 gives the conclusions and mentions a few open problems.

# 2 The van Herk-Gil-Werman algorithm

The van Herk-Gil-Werman (HGW) algorithm is based on splitting the input signal to overlapping segments of size $2p-1$, centered at

$$x_{p-1}, x_{2p-1}, x_{3p-1}, \ldots$$

Let $j$ be the index of an element at the center of a certain segment. The maxima of the $p$ windows which include $x_j$ are computed in one *batch* of the HGW algorithm as follows: First, define $R_k$ and $S_k$ for $k = 0, \ldots, p-1$:

$$\begin{aligned} R_k = R_k(j) &= \max(x_j, x_{j-1}, \ldots, x_{j-k}), \\ S_k = S_k(j) &= \max(x_j, x_{j+1}, \ldots, x_{j+k}). \end{aligned} \tag{1}$$

Now, the $R_k$'s and the $S_k$'s can be merged together to compute the max filter:

$$t_k = \max(x_{j-k}, \ldots, x_j, \ldots, x_{j+p-k-1}) = \max(R_k, S_{p-k-1}), \tag{2}$$

for $k = 1, \ldots, p-2$. In addition, we have

$$\max(x_{j-p-1}, \ldots, x_j) = R_{p-1},$$
$$\max(x_0, \ldots, x_{j+p-1}) = S_{p-1}.$$

There are two stages to the HGW algorithm:

**Preprocessing** Computing all $R_k$ and $S_k$ from their definition (1), and noting that $R_k = \max(R_{k-1}, x_{j-k})$ and $S_k = \max(S_{k-1}, x_{j+k})$ for $k = 1, \ldots, p-1$. This stage is carried out in $2(p-1)$ comparisons.

**Merge** Merging the $R_k$ and $S_k$ together using (2). This stage requires another $p-2$ comparisons.

Since this procedure computes the maximum of $p$ windows in total, we have that the amortized number of comparisons per window is

$$\frac{2(p-1)+p-2}{p} = 3 - \frac{4}{p}.$$

For large $p$, we have that the preprocessing step requires two comparison operations per element, while the merge step requires one more such comparison.

# 3 The Improved Algorithms for the Max-Filter

In this section we show how the two steps of the HGW algorithm can be carried out more efficiently.

## 3.1 An efficient preprocessing computation

Let us now deal with the preprocessing step of the HGW algorithm. The observation behind the GAA algorithm is that preprocessing computation can be made more efficient for randomized input, using the fact that in the HGW algorithm, the suffixes $S_k$ of one segment overlap with the prefixes $R_k$ of the following segment. Specifically, the problem that needs to be solved is

PREFIX-SUFFIX MAX: Given a sequence $x_0, \ldots, x_p$, compute all of its *prefix maxima*:

$$s_k = S_k(0) = \max(x_0, \ldots, x_k),$$

for $k = 0, \ldots, p-1$, and all its *suffix maxima*:

$$r_k = R_k(p) = \max(x_k, \ldots, x_p),$$

for $k = 1, \ldots, p$.

Note that this problem does not call for computing the overall maximum of the input

$$s_p = r_0 = \max(x_0, \ldots, x_p).$$

The original HGW algorithm makes $2(p-1)$ comparisons in solving the PREFIX-SUFFIX MAX problem. We propose the following efficient solution for this problem. Let

$$q = \left\lfloor \frac{p+1}{2} \right\rfloor = \frac{p}{2} + \frac{p \bmod 2}{2}. \tag{3}$$

In the *first part* of the modified implementation, compute all $s_k$, for $k = 0, \ldots, q-1$, using $q-1$ comparisons and $r_k$ for $k = q, \ldots, p$, using $p-q$ comparisons. The total number of comparisons in the first stage is then $p-1$.

The *second part* of the modified implementation of the preprocessing stage begins in comparing $s_{q-1}$ and $r_q$. If $r_q \geq s_{q-1}$, then we know that the overall maximum falls in one of $x_q, \ldots, x_p$. Therefore, it is unnecessary to further compute the value of $r_{q-1}, r_{q-2}, \ldots, r_1$. Instead, the algorithm outputs

$$r_{q-1} = r_{q-2} = \ldots = r_1 = r_q,$$

and continues to compute $s_q, \ldots, s_{p-1}$ in

$$p - q = \frac{p}{2} - \frac{p \bmod 2}{2} \tag{4}$$

comparisons.

A similar situation occurs if $r_q \leq s_{q-1}$, in which case it is unnecessary to compute $s_q, \ldots, s_{p-1}$. In this case $r_1, \ldots, r_{q-1}$ are computed in

$$q - 1 = \frac{p}{2} + \frac{p \bmod 2}{2} - 1 \tag{5}$$

comparisons.

The number of comparisons in the second part is given by the maximum of (4) and (5)

$$\frac{p}{2} - \frac{p \bmod 2}{2}.$$

The total number of comparisons in the more efficient algorithm for the preprocessing stage of PREFIX-SUFFIX MAX is

$$(p-1) + 1 + \frac{p}{2} - \frac{p \bmod 2}{2} = 1.5p - \frac{p \bmod 2}{2}. \tag{6}$$

Consider Algorithm 1 demonstrating the improved pre-processing stage for $p = 9$. Again, the implementation carries no hidden overhead, with the comparisons dominating the computation. Notice also that the assignment operations in instructions 3, 4, 14 and 20 can be executed in parallel on suitable architecture.

## 3.2 An efficient merge procedure

We first show how to improve the merge step, by reducing the amortized number of comparisons in it from 1 to

$$\frac{\lg p}{p} + o(1).$$

In this step, we compute

$$\max(R_k, S_{p-k-1}), \tag{7}$$

for $k = 1, \ldots, p-2$. Observing that

$$R_{p-2} \geq R_{p-1} \geq \ldots \geq R_1,$$
$$S_{p-2} \geq S_{p-1} \geq \ldots \geq S_1,$$

we can eliminate most of these comparisons. Suppose that for some specific $i$ it was found that

$$R_i \geq S_{p-i-1},$$

then for all $k > i$, we have that

$$R_k \geq R_i \geq S_{p-i-1} \geq S_{p-k-1},$$

**Algorithm 1** Solving PREFIX-SUFFIX MAX, $p = 9$ ($q = 5$) using 13 comparisons instead of 16.

**Input:** $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9$.

**Output:** $s_0, s_1, ..., s_8$ and $r_1, ..., r_9$,

*// Initialization*

$s_0, \ldots, s_8 \leftarrow x_0, \ldots, x_8$

$r_1, \ldots, r_9 \leftarrow x_1, \ldots, x_9$

*// First part ($p - 1 = 8$ comparisons)*

**if** $s_0 > s_1$ **then** $s_1 \leftarrow s_0$

**if** $s_1 > s_2$ **then** $s_2 \leftarrow s_1$

**if** $s_2 > s_3$ **then** $s_3 \leftarrow s_2$

**if** $s_3 > s_4$ **then** $s_4 \leftarrow s_3$

**if** $r_9 > r_8$ **then** $r_8 \leftarrow r_9$

**if** $r_8 > r_7$ **then** $r_7 \leftarrow r_8$

**if** $r_7 > r_6$ **then** $r_6 \leftarrow r_7$

**if** $r_6 > r_5$ **then** $r_5 \leftarrow r_6$

*// Second part (5 comparisons)*

**if** $r_5 > s_4$ **then** // $\max(x_0, \ldots, x_9) = \max(x_5, \ldots, x_9)$

    $r_1, r_2, r_3, r_4 \leftarrow r_5$

    **if** $s_4 > s_5$ **then** $s_5 \leftarrow s_4$

    **if** $s_5 > s_6$ **then** $s_6 \leftarrow s_5$

    **if** $s_6 > s_7$ **then** $s_7 \leftarrow s_6$

    **if** $s_7 > s_8$ **then** $s_8 \leftarrow s_7$

**else** // $\max(x_0, \ldots, x_9) = \max(x_0, \ldots, x_4)$

    $s_5, s_6, s_7, s_8 \leftarrow s_4$

    **if** $r_5 > r_4$ **then** $r_4 \leftarrow r_5$

    **if** $r_4 > r_3$ **then** $r_3 \leftarrow r_4$

    **if** $r_3 > r_2$ **then** $r_2 \leftarrow r_3$

    **if** $r_2 > r_1$ **then** $r_1 \leftarrow r_2$

**end if**

and therefore there is no need to do the comparisons of (7) for all $k > i$. Similarly, if it is determined that

$$R_i \leq S_{p-i-1},$$

then we do not need to do the comparisons of (7) for all $k < i$.

The optimized procedure for doing the merge step is therefore by a binary search. We start by setting $i = \lceil (p-2)/2 \rceil$, and then continue with the remaining half of the problem size.

Consider for example Algorithm 2, demonstrating the binary search method for the merge stage in the case $p = 9$. Instruction 1–13 implement the binary search, which terminates in a jump to a label $L_i$, where $i$ is the smallest such that $t_i = \max(R_i, S_{8-i}) = R_i$. Each such jump is carried out after exactly 3 instructions in this case. Instructions 14–21 compute the values of $t_i$'s by proper assignments to $R_i$'s, without any additional comparisons. The total number of assignments is any where between 0 and 7.

Notice that the implementation in Algorithm 2 is very straightforward and carries no additional overhead. In fact, this implementation can be translated directly to a VLSI layout, machine code, or higher level programming language. This efficiency is achieved by unrolling loops and more generally pre-computing all values, including labels, which are dependent solely on $p$.

In general, we have that binary search method reduces the number of comparisons from $p-2$ to $\lg p + O(1)$. In fact, it can be easily checked that the maximal number of comparisons is exactly $\lceil \lg(p-1) \rceil$. The amortized contribution of the improved merge step to the complexity is therefore

$$\frac{\lceil \lg(p-1) \rceil}{p}. \tag{8}$$

## 3.3 Comparison Complexity

In examining algorithms 2 and 1 together, we conclude that in the case $p = 9$, we are able to reduce the number of comparisons from 23 in the case of the HGW algorithm to 16, i.e., improving the complexity by about $30\%$. The improvement is even greater for larger values of $p$.

More generally, we can combine (8) and (6) to compute the amortized comparison complexity of our improved algorithm.

**Algorithm 2** Efficient merge for $p = 9$ using 3 comparisons instead of 7.

**Require:** $R_1 \leq R_2 \leq R_3 \leq R_4 \leq R_5 \leq R_6 \leq R_7$ and $S_1 \leq S_2 \leq S_3 \leq S_4 \leq S_5 \leq S_6 \leq S_7$

**Ensure:** $t_1 = \max(R_1, S_7)$, $t_2 = \max(R_2, S_6)$, $t_3 = \max(R_3, S_5)$, $t_4 = \max(R_4, S_4)$, $t_5 = \max(R_5, S_3)$, $t_6 = \max(R_6, S_2)$, and $t_7 = \max(R_7, S_1)$

*// Values $t_1, \ldots, t_7$ are returned in variables $R_1, \ldots, R_7$*

1: **if** $R_4 \geq S_4$ **then**
2: **if** $R_2 \geq S_6$ **then**
3: **if** $R_1 \geq S_7$ **then** jump $L_1$ **else** jump $L_2$
4: **else**
5: **if** $R_3 \geq S_5$ **then** jump $L_3$ **else** jump $L_4$
6: **end if**
7: **else** // $R_4 < S_4$
8: **if** $R_6 \geq S_2$ **then**
9: **if** $R_5 \geq S_3$ **then** jump $L_5$ **else** jump $L_6$
10: **else**
11: **if** $R_7 \geq S_1$ **then** jump $L_7$ **else** jump $L_8$
12: **end if**
13: **end if**
14: $L_8$: $R_7 \leftarrow S_1$
15: $L_7$: $R_6 \leftarrow S_2$
16: $L_6$: $R_5 \leftarrow S_3$
17: $L_5$: $R_4 \leftarrow S_4$
18: $L_4$: $R_3 \leftarrow S_5$
19: $L_3$: $R_2 \leftarrow S_6$
20: $L_2$: $R_1 \leftarrow S_7$
21: $L_1$:

**Theorem 1** *There exists a deterministic algorithm for the* 1D MAX-FILTER *problem, achieving*

$$C_1 = 1.5 + \frac{\lceil \lg(p-1) \rceil}{p} - \frac{p \bmod 2}{2p} \leq 1.5 + \frac{\lceil \lg(p-1) \rceil}{p} = 1.5 + \frac{\lg p}{p} + O(\frac{1}{p}) \tag{9}$$

Can we improve on this result? An information theoretical lower bound for the number of comparisons required to solve PREFIX-SUFFIX MAX, is $p + \lg p - O(1)$. This bound is derived as follows. A compact output of an algorithm for the problem uses $p + \lg p - O(1)$ bits comprised as follows:

1. $\lg p$ bits to designate the location of the overall maximum (for simplicity, we assume that $p$ is a power of 2) ,

2. a single bit for each location prior to the maximum, designating whether the corresponding element changes the prefix maxima, and

3. a single bit for each location following to the maximum, designating whether the corresponding element changes the suffix maxima.

Moreover, there are distinct inputs which produce all the bit combinations of this compact representation. Thus, in order to make the distinction between these inputs, the algorithm is forced to make at least

$$p + \lg p - O(1) \tag{10}$$

comparisons.

Although we are unable to meet the lower bound (10), we can come even closer to it in an important special cases. Suppose that in an input to the PREFIX-SUFFIX MAX problem, the overall maximum is located at a random location $\ell$ in the input sequence. (This does not necessarily mean obeys the input the i.i.d. condition.)

In the first part of the preprocessing stage, we maintain a record of the index $\ell_1$ at which the value stored at $s_i$, $i = 1, \ldots q-1$ was found. Similarly, we keep a record of the index $\ell_2$ at which the value stored at $r_i$, $i = p-1, \ldots, q$ was found.

Then, once the comparison between $s_{q-1}$ and $r_q$ is made, we know whether $\ell = \ell_1$ or $\ell = \ell_2$. All that remains is to proceed to compute outputs $s_q, s_{q+1}, \ldots, s_{\ell-1}$ in the case that $s_{q-1} < r_q$,

or $r_{q-1}, r_{q-2}, \ldots, r_{\ell+1}$ otherwise. The expected number of comparisons in this completion stage is

$$\begin{aligned}
\frac{1}{p+1}\left(\sum_{j=q-1,\ldots,0}(q-1-j)+\sum_{j=q,\ldots,p}(j-q)\right) &= \\
&= \left(\sum_{i=0}^{q-1} i+\sum_{i=0}^{p-q} i\right) \\
&= \frac{q(q-1)+(p-q)(p-q+1)}{2(p+1)} \\
&= \frac{p^2+2q^2-2pq+p-2q}{2(p+1)} \\
&= \frac{p^2-(p \bmod 2)}{4(p+1)} \\
&= \frac{p}{4}-\frac{1}{4}+\frac{1}{4(p+1)}-\frac{(p \bmod 2)}{4(p+1)} \\
&\leq \frac{p}{4}.
\end{aligned} \tag{11}$$

In general, it cannot be assumed that arbitrary input to the PREFIX-SUFFIX MAX problem will have its maximum at a random location. For example, the maximum will always occur at an end point for monotonic inputs. [4] When this assumption holds, then the amortized expected number of comparisons in this version of the preprocessing stage is therefore at most

$$\frac{(p-1)+1+p/4}{p} = 1.25$$

Combining the above with (8) we obtain:

**Theorem 2** *There exists an algorithm for the* 1D MAX-FILTER *problem, achieving*

$$E(C_1) \leq 1.25 + \frac{\lceil \lg p - 1 \rceil}{p} + \leq 1.25 + \frac{\lg p}{p}$$

*when the input is i.i.d.*

[4] For some inputs it should be possible to gain better performance by choosing at random the starting point for segmentation. Segments will be centered at positions indexed $\tau, \tau+p, \tau+2p, \ldots$, where $\tau$ is an integer selected at random in the range $[0, \ldots, p-1]$. Such a random selection *does not degrade* the overall efficiency due to the assumption that $p \ll n$, since only one random number in the range $[0, .., p-1]$ needs to be computed.

## 3.4 Implementation of the algorithms

The model of computation we have used here is that of comparisons. Few words are in place regarding the implementation of the algorithms in contemporary programming languages in actual use such as C [14], in machine language, or even in hardware.

The basic assumption is that $p$ is quite small and is either fixed or selected from a small set of pre-determined values. Therefore, an implementation should unroll all definite loops whose number of iterations is dependent solely on $p$. By doing so, the comparisons of the loop control variable are eliminated, just as the overhead of manipulating it. This principle means that the computation of the prefix maxima in the lower half and the suffix maxima in the upper half of each segment should be implemented using loop unrolling. An examples of the output of this loop unrolling is given in Algorithm 1.

The second part of the randomized version of the preprocessing stage is more difficult to implement without an explicit loop structure. Specifically, if the overall maximum occurs (say) in the upper half, it is required to complete the computation of the prefix maxima, stopping exactly at the location of the maximum. One way of implementing this without checking the location of the maximum in each iteration is by using "computed goto" of Fortran or functions pointers in C. There are no more than $p/2$ possible locations of the maximum in the upper half of the input. For each such location, there is a chunk of code which unrolls the loop up to that point. There are similar chunks of code designed to deal with the case of the maximum falling in the lower half of the input. Now, in the first part of the preprocessing, whenever the maximum is updated as a result of examination of a certain input value, the algorithm also updates a pointer to the appropriate chunk to be executed in case the maximum is found in the current location.

Now, when the maximum of the upper half is found to be greater than the maximum of the lower half, all that should be done is to use this pointer in order to make a jump to the appropriate chunk of code. Admittedly, hand coding of such an implementation could be tedious. Fortunately, with the advent of C++ [23], it is possible to employ its rich template mechanism in order to have such code generated automatically. The techniques of doing so are beyond the scope of this paper. The interested reader is referred to e.g., [10] and the references thereof for examples of applying the template mechanism for non-trivial compile-time computation and code generation.

In the merge step, the flow of control in the implementation should follow the comparisons tree implicit in the binary search. Specifically, each node in the tree should correspond to an if-then-else construct in the code. These constructs would have the same nesting as the tree nodes. Algorithm 2 shows how this is done for a fixed $p$. Again, the template mechanism can be used to reduce the bulk of the burden of generating the code from the implementor.

Finally, it should be noted that in a highly optimized implementation of the second part of the preprocessing stage, if the overall maximum is found in the lower half, all prefix maxima of the upper half are equal to this overall maximum, and there is *no need* to actually *produce* or *store* them in an auxiliary array . Instead, these values could be inlined into the code of the merge step.

# 4 Efficient Algorithm for Simultaneously Computing the Max and Min Filters

In this section we deal with the 1D MAX-MIN FILTER problem, and show how computing the min and max filters simultaneously can be done more efficiently than an independent computation of both. We start again with the HGW algorithm. The gain comes from partitioning the input signal into pairs of consecutive elements, and comparing the values in each pair. The greater value in each pair carries on the maximum computation while the lesser one carries one to the minimum computation.

## 4.1 The Prefix Max-Min Problem

Let us first consider the following problem,

PREFIX MAX-MIN: Given a sequence $x_0, \ldots, x_{q-1}$, compute

$$M_k = \max(x_0, \ldots, x_k),$$
$$m_k = \min(x_0, \ldots, x_k),$$

for $k = 0, \ldots, q-1$.

The straightforward solution for PREFIX MAX-MIN uses a total of $2(q-2)+1$ comparisons. For the sake of simplicity we assume that all elements in the input sequence are distinct. Analyzing this

problem from an information theoretical point of view the algorithm is tantamount to classifying each element $x_i$, $i > 2$, into one of three categories. Element $x_i$ may increase the running prefix maximum, i.e., $x_i > M_{k-1}$, and therefore $M_k = x_i$. If this is not the case, then $x_i$ may decrease the running minimum, i.e., $x_i < m_{k-1}$. The third case is that $m_{k-1} \leq x_i < M_{k-1}$, and therefore no changes are made to be made to the running prefix minimum or maximum. Also, there are only two possible cases for $x_1$, while there is exactly one case for $x_0$. Thus, we obtain

$$1 + (q-2)\lg 3 \approx 1.58496q,$$

as an information theoretic lower bound for the number of comparisons for the PREFIX MAX-MIN problem.

We do not know of a general way of bringing the amortized number of comparisons from $2 - o(1)$ closer to the $\lg 3$ lower bound, or alternatively, proving a stronger lower bound. However, if the distribution of the input elements is independent, we can do even better than the lower bound (which obviously holds for worst case inputs). This improvement is carried out as follows. Suppose that $M_i$ and $m_i$ were already computed. Then, to compute $M_{i+1}$, $M_{i+2}$, $m_{i+1}$ and $m_{i+2}$, we apply the following *incorporate-next-input-pair* algorithm.

**Algorithm *incorporate-next-input-pair:*** *Extend the result of a solution to* PREFIX MAX-MIN *to include input elements $x_{i+1}$ and $x_{i+2}$, using the four following comparisons:*

1. Compare $x_{i+1}$ and $x_{i+2}$. Assume, without loss of generality, that $x_{i+1} \geq x_{i+2}$.
2. Compare $M_i$ with $x_{i+1} = \max(x_{i+1}, x_{i+2})$.
3. Compare $m_i$ with $x_{i+2} = \min(x_{i+1}, x_{i+2})$.
4. At this stage, the algorithm has determined both $M_{i+2}$ and $m_{i+2}$. Specifically,

$$M_{i+2} = \max(x_{i+1}, M_i)$$
$$m_{i+2} = \min(x_{i+2}, m_i).$$

There are four cases to consider in computing $m_{i+1}$ and $M_{i+1}$.

(a) *No changes:*

$$x_{i+1} \leq M_i \text{ and } x_{i+2} \geq m_i$$

No more comparisons need to be done in this case, and the algorithm simply outputs

$$M_{i+2} = M_{i+1} = M_i$$
$$m_{i+2} = m_{i+1} = m_i.$$

(b) *Changes to both the maximum and the minimum:*

$$x_{i+1} \geq M_i \text{ and } x_{i+2} \leq m_i.$$

Again, no more comparisons need to be done in this case, and the algorithm outputs

$$M_{i+2} = M_{i+1} = x_{i+1},$$
$$m_{i+1} = m_i,$$
$$m_{i+2} = x_{i+2}.$$

(c) *Change to the maximum:*

$$x_{i+1} \geq M_i \text{ and } x_{i+2} \geq m_i.$$

The algorithm outputs

$$M_{i+2} = M_{i+1} = x_{i+1},$$
$$m_{i+2} = m_{i+1} = m_i.$$

without any additional comparisons.

(d) *Possible change to the minimum:*

$$x_{i+1} \leq M_i \text{ and } x_{i+2} \leq m_i.$$

This is the only case in which an additional comparison is required: The algorithm first outputs

$$M_{i+2} = M_{i+1} = M_i,$$
$$m_{i+2} = x_{i+2},$$

and then determines $m_{i+1}$ by comparing $x_{i+1}$ with $M_i$. If $x_{i+1} < m_i$ then

$$m_{i+1} = x_{i+1},$$

otherwise,

$$m_{i+1} = m_i.$$

Thus, in the worst case, the algorithm makes four comparisons for each pair $x_{i+1}$ and $x_{i+2}$, where $i > 0$ is odd, which does not improve on the two comparisons per element by the trivial algorithm. The fourth comparison however is needed only in case

$$x_{i+2} < m_i = \min_{0 \le j \le i}(x_i), \tag{12}$$

or in the dual case, namely when the first comparison yields $x_{i+1} \le x_{i+2}$, and

$$x_{i+2} < M_i = \max_{0 \le j \le i}(x_i). \tag{13}$$

With i.i.d. (independent input distribution) the probability of (12) or (13) holding is $1/(i+3)$, for all $i > 0$. Let $u = \lfloor q/2 \rfloor - 1 = (q - (q \bmod 2)) - 1$. Then in the last application of the above algorithm we deal with the pair $x_{2u}$ and $x_{2u+1}$. In total, $F_q$, the *expected* (with regard to input distribution) number of times the fourth comparison is made is given by

$$F_q = \frac{1}{4} + \frac{1}{6} + \frac{1}{8} + \cdots + \frac{1}{2u+2} = (H_{u+1} - 1)/2, \tag{14}$$

where $H_u$ is the $u$th harmonic number. It is well known that

$$\lim_{u \to \infty} H_u = \ln u + \gamma \tag{15}$$

where $\gamma \approx 0.577216$ is Euler's constant (also called Mascheroni's constant). Combining (14) and (15) we have

$$\begin{aligned} F_q &= \frac{\ln(u+1)}{2} + \frac{\gamma}{2} - 0.5 + o(1) \\ &\approx \frac{\ln(u+1)}{2} - 0.211392 + o(1) \end{aligned} \tag{16}$$

It is also known that

$$\ln u + \gamma \le H_u \le \ln u + 1 \tag{17}$$

from which we obtain

$$\begin{aligned} F_q &\le \frac{\ln(u+1)}{2} \\ &\le \frac{\ln q - 1}{2}. \end{aligned} \tag{18}$$

Other than these, in solving PREFIX MAX-MIN, there are $u$ applications of *incorporate-next-input-pair*, in which $3u$ comparisons are made, one comparison in which $x_0$ is compared with $x_1$ to determine $M_0$, $M_1$, $m_0$ and $m_1$, and finally, and only if $q$ is odd, two comparisons to determine $M_{q-1}$ and $m_{q-1}$. The number of these comparisons is

$$1 + 3u + 2(q \bmod 2) = \frac{3q}{2} - 2 + \frac{q \bmod 2}{2}. \tag{19}$$

Adding (18) and (19) we have that the expected total number of comparisons in our solution to PREFIX MAX-MIN is at most

$$\frac{3q}{2} + \frac{\ln q}{2} - 2, \tag{20}$$

and the expected amortized number of comparisons per element is at most

$$1.5 + \frac{\ln q}{2q} - 2/q.$$

It should be noted that one cannot hope to improve much on this result. The reason is that solving PREFIX MAX-MIN also yields the maximum and the minimum of the whole input. However, computing both these values cannot be done in less than $\lceil 3p/2 \rceil$ comparisons [5, page 187] even for randomized inputs.

## 4.2 Computing the Min-Max Filter

We now employ algorithm *incorporate-next-input-pair* in the pre-processing stage of the modified HGW algorithm adapted for finding both the minimum and the maximum filters. Specifically, we are concerned in this stage in finding an efficient algorithm to the PREFIX-SUFFIX MAX-MIN problem, defined as computing the maximum and the minimum of all prefixes and all suffixes of an array of size $p + 1$. Such an efficient algorithm is obtained by partitioning the input array into two halves. In the lower half which comprises $q = \lfloor (p+1)/2 \rfloor = p/2 + (p \bmod 2)/2$ elements we repetitively apply *incorporate-next-input-pair* to compute the prefix maxima and the prefix minima in this half. A similar computation is carried out in the upper half with $p - q + 1 = \lceil (p+1)/2 \rceil$ elements of the input array, except that algorithm *incorporate-next-input-pair* is mirrored to compute the *suffix* minima and the suffix maxima in this half. The total expected

number of comparisons so far can be computed from (20):

$$\begin{aligned}\frac{3q}{2}+\frac{3(p+1-q)}{2}+\frac{\ln q}{2}+\frac{\ln(p+1-q)}{2}-4 &= \frac{3p}{2}+\frac{\ln(\lfloor (p+1)/2\rfloor\lceil (p+1)/2\rceil}{2}-2.5\\ &\le \frac{3p}{2}+\frac{\ln\left(\frac{p+1}{2}\right)^2}{2}-2.5\\ &= \frac{3p}{2}+\ln(p+1)-\ln 2-2.5\\ &\le \frac{3p}{2}+\ln p-2.5.\end{aligned} \tag{21}$$

Once this computation is done, we carry on as before to produce the rest of the required output. In two more comparisons we find out where the maximum and the minimum of the whole array occur. If the maximum occurs in the lower (resp. upper) half then it remains to compute the suffix (resp. prefix) maxima from the mid-point down-to (resp. up-to) the location of the maximum. From (11) we have that this computation costs another 0.25 comparison per input element. A similar completion stage must be carried out for the minimum prefixes or suffixes, using another 0.25 amortized comparisons. All that remains to do is the merge step, which has to be carried out twice, once for the minimum and once for the maximum. The number of comparisons for the merge is at most $2\lg p$. Combining this bound with (21) we obtain:

**Theorem 3** *There exists an algorithm for the* 1D MIN-MAX FILTER *problem, that at the worst case makes twice the number of comparisons as that of Theorem 2. For i.i.d. input, the amortized number of comparisons that the algorithm makes is*

$$\begin{aligned}C_1^m &< 2+2\frac{\ln p+\lg p}{p}\\ &= 2+(2+\frac{\ln 2}{2})\frac{\lg p}{p}\\ &\approx 2+2.3466\frac{\lg p}{p}.\end{aligned}$$

Stated differently, we have that asymptotically for large $p$, and for i.i.d. one comparison per element is required to compute each of the minimum and the maximum filters, provided they are computed together.

## 4.3 Performance on Natural Images

Natural images are far from being random inputs. It is therefore important to check the performance of the algorithm of Theorem 3 in natural images. The important factor is $K_p$, the number of times the prefix maximum (or the prefix minimum) is changed in a window of size $p$. Clearly, $1 \leq K_p \leq p$. With randomized input, the expected value of $K_p$ is $H_p \approx \ln p + \gamma$, which gives rise to an asymptotic saving of $0.5$ comparison per input value. If, on the other hand, the input is monotonically increasing then $K_p = p$. This is a worst case input in which no savings at all can be made in computing the min and max filters together. More generally, the amortized number of comparisons that are saved by the iterative application of *incorporate-next-input-pair* is in the order of

$$\frac{p - K_p}{p}.$$

Figure 3 shows the average value of $K_p$ (using max computation) for $p = 2, \ldots, 100$ in the lighthouse image. In this, and all subsequent figures, the average was computed by examining all one-dimensional row windows in the image. Only windows which entirely fall in the image were considered. For comparison purposes, this figure, just as all the ones to follow, shows the rate at which $H_p$ increases with $p$.

Figure 3 left frame shows the average value of $K_p$ for the red, green and blue components of each pixel. It is interesting to note that these three channels behave quite similarly, and as we shall see next, very much like the behavior of the illumination in grey-level images. For small values of $p$, $K_p$ and $H_p$ are close, and $K_p$ appears to increase in a logarithmic rate. For larger values of $p$, $K_p$ appears to increase at a linear rate, with

$$9 \leq K_{100} \leq 12.$$

It is also interesting to note that the rate of increase of $K_p$ is the fastest in the red channel, and slowest in the blue channel.

Figure 3, right frame, is similar to Figure 3, left frame, except that it depicts the rate of increase of $K_p$ for minimum computation.

We witness again the same phenomena: The rate of increase in $K_p$ is faster than that of $H_p$, it is slowest to change in the blue and fastest in the red. Curiously, we have a slightly better ratio for

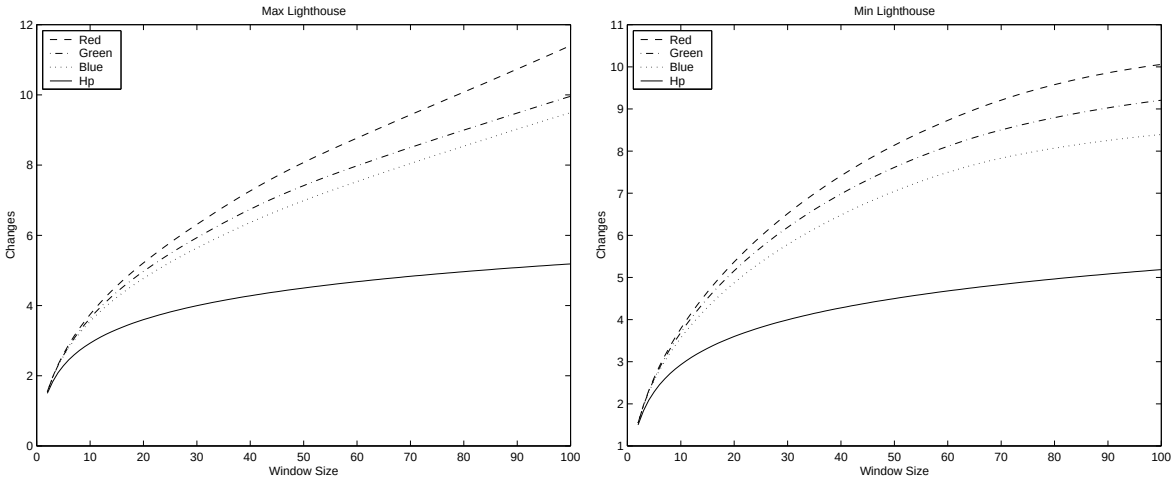


Figure 3: Left: Average number of times the prefix maximum is changed in the lighthouse image vs. window size. Right: Average number of times the prefix minimum is changed in the lighthouse image vs. window size.

$K_p/p$ for the minimum computation

$$8 \leq K_{100} \leq 10.$$

To gain better understanding of the rate of increase of $K_p$ in natural images, we run similar experiments for six grey-level images. The results are depicted in Figure 4 left frame for the maximum computation, and in Figure 4 right frame for the minimum computation. The experiments were conducted this time only for the green channel. It was our experience that in all these images $K_p$ increased slightly faster for red, and slightly slower for blue.

As can be seen from these two figures, $K_p$ is always faster to increase than $H_p$. Still, even for large windows we have that $K_p$ is only a small fraction of $p$. A slower rate of increase in $K_p$ for minimum rather than for maximum could not be observed. For example, $K_p$ of the "Sails" image increases at the slowest rate for minimum prefix, and at the fastest rate for maximum prefix.

One may conjecture that the rate of increase of $K_p$ is logarithmic, but with a base of logarithm less than $e$. To check this Figures 4 is redrawn in semi-logarithmic scale in Figure 5. This conjecture is false as can easily be seen from these two frames.

In conclusion, we find that for natural images the algorithm behind Theorem 3 gives rise to an amortized saving of about $0.9$ comparison compared to the independent computation of the min

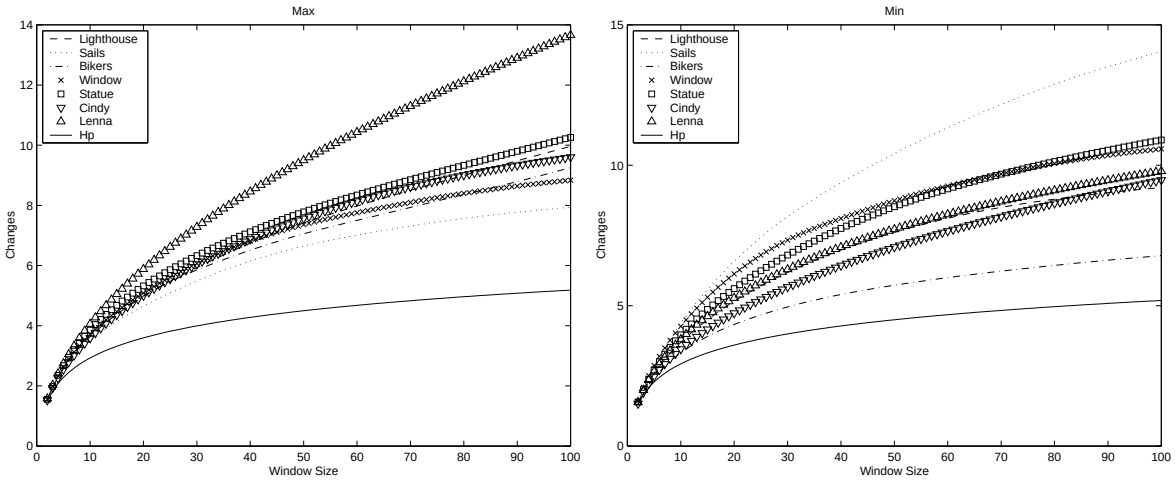


Figure 4: Left: Average number of times the prefix maximum is changed in seven different natural images vs. window size. Right: Average number of times the prefix minimum is changed in seven different natural images vs. window size.

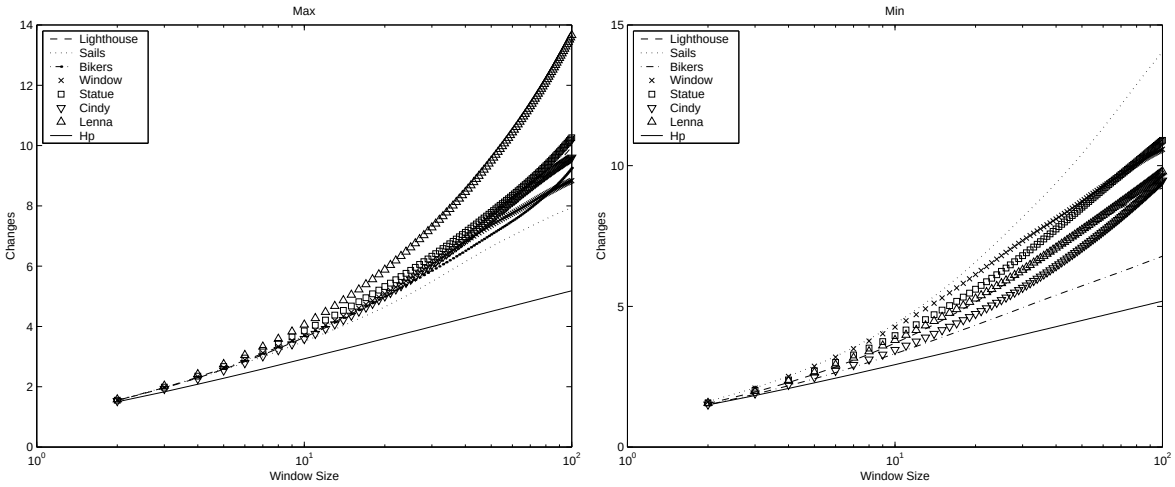


Figure 5: Left: Average number of times the prefix maximum is changed in seven different natural images vs. window size (semi-logarithmic scale). Right: Average number of times the prefix minimum is changed in seven different natural images vs. window size (semi-logarithmic scale).

and max filters.

# 5 An Efficient Algorithm for the Opening and Closing Filters

In this penultimate section of the paper we turn to describing how the opening (and closing) filter can be computed more efficiently than a mere sequential application of the Max-Filter and then the Min-Filter.

To understand the improvement, consider for a moment the problem of computing the prefix-minimum, in the case that the input of length $p$ is given as a sequence of $L$ monotonically increasing or decreasing segments. Suppose that the prefix-minimum has been computed up to a point $i$, i.e., that the value of $m_i = \min(x_0, \ldots, x_i)$ is known, and that $x_{i+1}, \ldots, x_{i+k}$ is a monotonically decreasing segment of the input of length $k$. Then, in order to compute $m_{i+1}, \ldots, m_{i+k}$, all that is required is to find the smallest $\ell$ such that $m_\ell < m_i$. This $\ell$ can be easily found using a binary search in $\lceil \lg k \rceil$ comparisons. We then have

$$m_{i+j} = \begin{cases} m_i & \text{if } j < \ell \\ x_{i+j} & \text{if } \ell \leq j \leq k. \end{cases}$$

If on the other hand $x_{i+1}, \ldots, x_{i+k}$ is a monotonically *increasing* sequence, all that is required in order to compute $m_{i+1}, \ldots, m_{i+k}$ is to compare $x_{i+1}$ and $m_i$. In this case we have that

$$m_{i+1} = m_{i+2} = \cdots = m_{j+k} = \min(x_{i+1}, m_i).$$

Using Lagrange multipliers we obtain that the number of comparisons is bounded above by

$$L \left\lceil \lg \frac{p}{L} \right\rceil. \tag{22}$$

Recall now the improved merge step described in Section 3.2. Each iteration of the binary search algorithm generates about half of the outputs of the max-filter that remained to be computed. Note that all values generated in one such iteration are consecutive in the output. Further, since these values are obtained from computing either $R_i$ or $S_i$, they are either monotonically increasing or monotonically decreasing. Thus an application of the modified max filter algorithm also partitions each stretch of $p$ outputs into no more than $\lceil \lg p \rceil$ monotonic segments.

The improved opening filter algorithm is thus obtained by first applying the modified HGW max-filter algorithm, while preserving this partitioning of the output. Then, the results are feed into the modified HGW min-filter algorithm. The partitioning information is then used for an efficient implementation of the preprocessing stage in which prefix- and suffix-minima are computed. It follows from (22) that the preprocessing stage can be done in $O(\lg^2 p)$ comparisons. Since the merge step can be done in $O(\lg p)$ comparisons, we obtain:

**Theorem 4** *There exists an algorithm which computes the opening filter, achieving*

$$C_1^o = C_1 + O\left(\frac{\lg^2 p}{p}\right).$$

In other words, asymptotically computing the opening filter is not more expensive than computing just the max-filter. When going to more than one dimension, unlike the erosion and dilation, the opening and closing operations are not separable, and thus do not enjoy the same computational efficiency as the one dimensional opening and and closing. Nevertheless, one could still use the one dimensional efficiency to accelerate these operations. The order of operations in this case could be the following:

- Apply the MAX FILTER on the rows. For non-i.i.d. signals, this operation takes $C_1$ comparisons.
- Apply the MIN-MAX FILTER on the columns of the result of the previous step. For non-i.i.d. signals, this operation takes $C_1 + o(1)$ comparisons.
- Apply the MIN FILTER on the rows of the result of the the previous step. For non-i.i.d. it takes $C_1$ comparisons.

That is, for two dimensional images, instead of using $4C_1 = 6$ comparisons per element, we spend only $3C_1 = 4.5$ comparisons per element (or $3.75$ comparisons instead of $5$ for i.i.d. signals). For the general $n$-dimensional case, we spend $(n-1)C_1$ comparisons, exploiting the fact that at least for one dimension we can enjoy the efficiency of the 1D-MIN-MAX FILTER.

# 6 Conclusions and Open Problems

We presented improvements of the HGW algorithm for running min and max filters. The average computational complexity was shown to be $1.25+o(1)$ per element for randomized input, and $1.5+o(1)$ for a deterministic algorithm (worst case input). These improvements, which come close to the best known lower bound for the problem, were enabled by careful examination of the redundancies in the preprocessing and the merge steps of the HGW algorithm.

We continued to study a related problem, namely the computation of the min and the max filter together. We found that for i.i.d. input, it is possible to compute the minimum and the maximum filters together in $2 + o(1)$ comparisons per data point. This is less than $2.5 + o(1)$ comparisons required by applying twice the best max filter algorithm.

The opening and closing filters which are similar to the problem of computing the min- and max-filters together, can be computed much more efficiently. We found algorithms for these filters using $1.5 + o(1)$ comparisons deterministically, or $1.25 + o(1)$ comparisons when the input is i.i.d.

All separable algorithms like erosion and dilation are readily extendible to higher dimensions.

We leave the following open questions for further research:

1. In image processing, the selection of a coordinate system is usually arbitrary and unrelated to the geometry of the objects being presented. Therefore, it seems more natural to use a circle rather than a square as the shape of the window. However, the extension of the 1D algorithm for a 2D-circle case needs further thought. By using a heap data structure to represent a sliding window in the shape of a circle of radius $p$, we can compute the filter in

   $$O(p \lg p)$$

   comparisons per window; in each move of the center of the circle, the data structure is updated by adding $O(p)$ points and removing $O(p)$ points. If pixel values are drawn from some small finite domain, then it is possible to use a dynamic moving histogram [13, 4] data structure supporting insertions and deletions in $O(1)$ time. The amortized cost is then reduced to $O(p)$. It is interesting and important to find more efficient *accurate* algorithms for this problem, with and without assuming that pixel values are bounded. (Previous results [1, 22] give approximations to this problem.)

2. We know of no deterministic or randomized algorithm which computes for worst case input the MAX-MIN FILTER more efficiently than computing the min and max separately. There is however an interesting property of algorithm *incorporate-next-input-pair* which might be used in trying to meet this challenge: The fourth comparison is only required for the computation of the *interim* $M_{i+1}$ and $m_{i+1}$ output values. Thus, a repetitive application of this algorithm can carry on to its next iteration, while delaying the fourth comparison of the current iteration for later. It might be possible to use this observation to obtain an efficient algorithm for the MAX-MIN FILTER problem which does not presume any input distribution. For example, in the preprocessing stage one may apply *incorporate-next-input-pair* to compute the prefix maxima of the greater elements of each pair in the lower half as well as the prefix minimum of the lesser elements of these pairs. A similar computation is carried out in the upper half of the input array, except that algorithm *incorporate-next-input-pair* is mirrored to compute the suffix minima (resp. maxima) of the lesser (resp. greater) elements of each pair in the upper half. The computation of the skipped values could be done later on and only if necessary.

3. A related algorithmic problem is that of solving PREFIX MAX-MIN problem in less than $2p + O(1)$ comparisons. We find this problem fascinating since it is possible to solve either the PREFIX MAX (or the PREFIX MIN) problem in the same number of comparisons it takes to compute just the overall maximum (minimum). On the other hand, computing both the overall maximum and the overall minimum can be done in a smaller number of comparisons than what is required for computing them independently. Our inability to make similar saving for the problem of computing the PREFIX MAX together with PREFIX MIN leads us to suspect that there is an $2p + O(1)$ lower bound for the PREFIX MAX-MIN problem. It might be possible to derive such a bound using a technique similar to that of the proof that computing the maximum and minimum of $p$ values requires $\lceil 3p/2 \rceil$ comparisons.

4. As mentioned above, the Max-Filter algorithms do not assume any input distribution. For some applications it could be useful to produce an algorithm for this problem which works better in the case of i.i.d. input.

Our results do not seem to be directly applicable to the more difficult problem of computing the median filter. However, it might similar techniques might be used to improve the constants, or even the asymptotic complexity of the currently best median filter algorithm [12] which runs in $O(\log^2 p)$ time per filtered point.


## Acknowledgments

Stimulating discussions of both authors with Reuven Bar-Yehuda of the Technion during the writeup of this paper are gratefully acknowledged. We also thank the reviewers for their detailed comments that help us enhance the clarity of the paper. The second author is grateful to Renato Keshet from HP Labs. Israel, for intriguing discussions on efficient morphological operators.


# References


[1] E. Breen and P. Soille. Generalization of van Herk recursive erosion/dilation algorithm to lines at arbitrary angles. In *Proc. DICTA'93: Digital Image Computing: Techniques and Applications*, Sydney, Dec. 1993. Australian Pattern Recognition Society.

[2] R. W. Brockett and P. Maragos. Evolution equations for continuous–scale morphology. In *Proceedings IEEE international Conference on Acoustics, Speech, and Signal Processing*, pages 1–4, San Francisco, California, March 1992.

[3] R. W. Brockett and P. Maragos. Evolution equations for continuous-scale morphological filtering. *IEEE Transactions on Signal Processing*, 42(12):3377–3386, 1994.

[4] B. Chaudhuri. An efficient algorithm for running window pixel gray level ranking in 2d images. *Pattern Recognition Letters*, 11(2):77–80, 1990.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[6] P. Danielson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

[7] M. Davis. Efficient methods of performing morphological operations. *US patent*, US5960127, 1999.

[8] L. Dorst and R. Boomgaard. Morphological signal processing and the slope transform. *Signal Processing*, 38:79–98, 1994.

[9] D. Z. Gevorkian, J. T. Astola, and S. M. Atourian. Improving Gil-Werman algorithm for running min and max filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5):526–529, May 1997.

[10] J. Y. Gil and Z. Gutterman. Compile time symbolic derivation with C++ templates. In *Proceedings of the fourth Conference on Object-Oriented Technologies and Systems (COOTS'98)*, Santa Fe, New Mexico, May 1998. USENIX.

[11] J. Y. Gil and R. Kimmel. Efficient dilation, erosion, opening and closing algorithms. In J. Goutsias, L. Vincent, and D. S. Bloomberg, editors, *Mathematical Morphology and Its Applications to Image and Signal Processing*, pages 301–310. Kluwer Academic Publishers, Boston, 2000.

[12] J. Y. Gil and M. Werman. Computing 2-D min, median, and max filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):504–507, May 1993.

[13] T. Huang, G. Yang, and G. Tang. A fast two-dimensional median filtering algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 27(1):13–18, 1979.

[14] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, second edition, 1988.

[15] J. Lee, R. Haralick, and L. Shapiro. Morphologic edge detection. *IEEE Journal of Robotics and Automation*, 3(3):142–156, 1987.

[16] P. Maragos. Slope transforms: theory and application to nonlinear signal processing. *IEEE Transactions on Signal Processing*, 43(4):864–877, 1995.

[17] I. Pitas. Fast algorithms for running ordering and max/min recalculations. *IEEE Transactions on Circuits and Systems*, CAS-36(6):795–804, June 1989.

[18] J. Rivest, P. Soille, and S. Beucher. Morphological gradients. *Journal of Electronic Imaging*, 2(4):326–336, 1993.

[19] G. Sapiro, R. Kimmel, D. Shaked, B. Kimia, and A. M. Bruckstein. Implementing continuous–scale morphology via curve evolution. *Pattern Recognition*, 26(9):1363–1372, 1993.

[20] J. Serra. *Image analysis and mathematical morphology*. Academic Press, New York, 1982.

[21] P. Soille. *Morphological Image Analysis*. Springer-Verlag, Berlin, Heidelberg, New York, 1999.

[22] P. Soille, E. Breen, and R. Jones. Recursive implementation of erosions and dilations along discrete lines at arbitrary angles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18:5, 1996.

[23] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[24] B. M. ter Haar Romeny, editor. *Geometric–Driven Diffusion in Computer Vision*. Kluwer Academic Publishers, The Netherlands, 1994.

[25] M. van Herk. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recognition Letters*, 13:517–521, 1992.