Running Max/Min Filters using $1+o(1)$ Comparisons per Sample

Hao Yuan and Mikhail J. Atallah

Accepted by IEEE Transactions on Pattern Analysis and Machine Intelligence

Copyright Notice:

© 2011 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

The IEEE published final version of this work:
http://dx.doi.org/10.1109/TPAMI.2011.183
Running Max/Min Filters using $1 + o(1)$ Comparisons per Sample

Hao Yuan, Member, IEEE, and Mikhail J. Atallah, Fellow, IEEE

Abstract—A running max (or min) filter asks for the maximum or (minimum) elements within a fixed-length sliding window. The previous best deterministic algorithm (developed by Gil and Kimmel, and refined by Coltuc) can compute the 1D max filter using $1.5 + o(1)$ comparisons per sample in the worst case. The best known algorithm for independent and identically distributed input uses $1.25 + o(1)$ expected comparisons per sample (by Gil and Kimmel). In this work, we show that the number of comparisons can be reduced to $1 + o(1)$ comparisons per sample in the worst case. As a consequence of the new max/min filters, the opening (or closing) filter can also be computed using $1 + o(1)$ comparisons per sample in the worst case, where the previous best work requires $1.5 + o(1)$ comparisons per sample (by Gil and Kimmel); and computing the max and min filters simultaneously can be done in $2 + o(1)$ comparisons per sample in the worst case, where the previous best work (by Lemire) requires 3 comparisons per sample. Our improvements over the previous work are asymptotic, that is, the number of comparisons is reduced only when the window size is large.

Index Terms—Mathematical morphology, erosion, dilation, opening, closing.

1 INTRODUCTION

Given an input sequence $x_0, \ldots, x_{n-1}$ and the window size $p > 1$, the 1D running max filter problem is to compute the outputs

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

for $0 \leq i \leq n - p$. The 1D running min filter problem is defined in a similar way (by changing max to min). In the $d$-dimensional case, a $d$-dimensional cube is used as a window. Throughout this work, we assume that the dimension is one unless otherwise specified. Following previous work [6], [7], [14], the computation model is the comparison model, i.e., only the number of comparisons for comparing input elements is counted, and the comparisons between indices (e.g., as part of iterations) are not counted. This means that the inputs can be drawn from any totally ordered set, rather than a restricted universe like $\{0, 1, \ldots, 255\}$. Usually, $n$ is very large compared to $p$, so the complexity is measured by the number of comparisons per output (or sometimes called per sample) in terms of $p$.

Running max/min filters are fundamental operators in morphological image processing [13]. The max filter corresponds to the dilation operator over gray-scale images using a flat and linear structuring element, and the min filter corresponds to the erosion operator. Because all the algorithms for the running max filter apply to the min filter, we will only discuss the running max filter.

In this work, we also consider the opening and closing filters, which are two basic morphological image processing operators [13]. The opening filter is obtained by first applying the min filter, and then applying the max filter to the result of the previous min filtering. Similarly, the closing filter is obtained by first applying the max filter, and then applying the min filter to the result of the previous max filtering.

1.1 Previous Work

A naive implementation of the max filter requires $p - 1$ comparisons per sample. Pitas [11] gave an algorithm that uses $O(\log p)$ comparisons per sample for the max filter. An algorithm that does not depend on the window size was given by van Herk [14] and independently by Gil and Werman [6]. Their algorithm (HGW for short) requires $3 - 4/p$ comparisons per sample. Gevorkian, Astola and Atourian [5] considered independent and identically distributed (i.i.d.) inputs, and presented an algorithm that uses $2.5 - 3.5/p + 1/p^2$ expected comparisons per sample. The previous best known worst-case algorithm was due to Gil and Kimmel [7]. Their algorithm (GK for

1. Throughout this work, we use $\log$ to represent $\log_2$.
short) requires $1.5 + \log p \left/ p \right. + O\left(\frac{1}{p}\right)$ comparisons per sample. The GK algorithm was further improved by Coltuc [3]. Coltuc’s algorithm saves about $1.5\left/ (p + 1\right)$ comparisons per sample compared to the GK algorithm, but the order is still $1.5 + \log p \left/ p \right. + O\left(\frac{1}{p}\right)$. For i.i.d. inputs, Gil and Kimmel also presented an algorithm that requires $1.25 + \log p \left/ p \right. + O\left(\frac{1}{p}\right)$ expected comparisons per sample.

Note that the HGW algorithm can compute semigroup sums in a sliding window using $3 - 4/p$ semigroup operations per sample. The semigroup case was improved to $3 - 6/(p + 1)$ operations per sample by Coltuc [3].

For computing both the maximum and minimum elements in the sliding window, Lemire [10] presented an algorithm that uses $3$ comparisons per sample in the worst case, which is slightly better than running the GK algorithm twice. When the input is i.i.d., Gil and Kimmel [7] presented an algorithm that uses $2 + \sqrt{p} + \log p \left/ p \right. + O\left(\frac{1}{p}\right)$ comparisons per sample.

The community of algorithm and theory studied a more general version of the problem — Range Minimum Query (RMQ). The RMQ problem is to preprocess an input array, such that the minimum element within any query window must be returned efficiently. A linear time and space preprocessing algorithm for 1D RMQ to achieve constant-time query answering was first given by Gabow, Bentley and Tarjan [4], using a linear reduction to the Nearest Common Ancestor (NCA) problem [9] on the Cartesian tree [15]. The Cartesian tree (defined by Vuillemin [15]) is built on top of the input array to completely capture the information necessary to determine the solution for any range minimum query on the original array. It was shown that any range minimum query can be translated to an NCA query on the precomputed Cartesian tree in constant time. The first constant-time NCA solution with linear preprocessing was given by Harel and Tarjan [9], and much effort was spent on simplifying the solution [1], [2], [12]. Multidimensional RMQ was studied by the authors [16]. Note that using the Cartesian tree approach [15], the 1D max filter can be solved using at most 2 comparisons per sample.

The $d$-dimensional max/min filter can be computed by applying the 1D filter $d$ times [6], using a structuring element decomposition approach [8]. Therefore, the GK algorithm can be used to compute the 2D max (or min) filter by $3 + o(1)$ comparisons per sample, and more generally $d(1.5 + o(1))$ comparisons per sample for $d$-dimensional cases.

The previous best result for the 1D opening/closing filter was $1.5 + O\left(\log^2 p \left/ p \right.\right)$ by Gil and Kimmel [7]. They extended the result to the $d$-dimensional cases using $(2d - 1)(1.5 + o(1))$ comparisons per sample.

### 1.2 Our Contribution

In this work, we present a simple improvement based on the GK algorithm for 1D max filter. Our new algorithm achieves $1 + \frac{2}{\sqrt{p}} + \log p \left/ p \right. + O\left(\frac{1}{p}\right)$ comparisons per sample in the worst case, which improves over the GK algorithm when $p \geq 20$, and over the Coltuc’s algorithm when $p \geq 24$. As a consequence, the worst-case complexity to compute both the maximum and minimum elements in a sliding window becomes $2 + \frac{4}{\sqrt{p}} + \log p \left/ p \right. + O\left(\frac{1}{p}\right)$ comparisons per sample, which improves the solution of Lemire [10] when $p \geq 37$. This means that our improvements over previous work are asymptotic, i.e., the number of comparisons is reduced only for large $p$’s.

Based on our improvement for the 1D max/min filter, the $d$-dimensional max (or min) filter can be computed in $d(1 + o(1))$ comparisons per sample in the worst case. Using the techniques of Gil and Kimmel [7], 1D opening (or closing) filter can be computed in $1 + O\left(\frac{1}{\sqrt{p}}\right)$ comparisons per sample in the worst case, and the $d$-dimensional opening and closing can be computed in $(2d - 1)(1 + o(1))$ comparisons per sample in the worst case.

We will first review the GK algorithm in Section 2, and then present our result in Section 3. Section 4 presents an experimental evaluation. Section 5 concludes.

### 2 Gil and Kimmel’s Algorithm

We will first review the HGW algorithm, on which the GK algorithm is built. All the given presentations are for the 1D max filter. The HGW algorithm splits the input sequence into overlapping segments of length $2p - 1$, where the segments are centered at positions $ip - 1$ for $1 \leq i \leq n/p$. Consider a segment centered at position $c$, i.e.,

$$x_{c-p+1}, x_{c-p+2}, \ldots, x_c, \ldots, x_{c+p-2}, x_{c+p-1}.$$ 

The HGW algorithm generates $p$ outputs $y_{c-p+1}, y_{c-p+2}, \ldots, y_c$ in two stages — a preprocessing stage and a merge stage. In the preprocessing stage, prefix maximums and suffix maximums are computed for each block. This consists of computing

$$P_c(k) = \max\{x_c, x_{c+1}, \ldots, x_{c+k}\},$$
$$S_c(k) = \max\{x_c, x_{c-1}, \ldots, x_{c-k}\},$$

for $0 \leq k \leq p-1$. Then in the merge stage, $p-2$ outputs can be obtained by merging the prefix maximums and suffix maximums in the following way: for $1 \leq k \leq p-2$,

$$y_{c-p+1+k} = \max\{S_c(p - 1 - k), P_c(k)\}. \quad (1)$$

We can get the remaining two outputs by

$$y_{c-p+1} = S_c(p - 1),$$
$$y_c = P_c(p - 1).$$
Straightforward implementation of the preprocessing stage uses $2(p-1)$ comparisons (based on the fact that $P_c(k) = \max\{P_c(k-1), x_{c+k}\}$ and $S_c(k) = \max\{S_c(k-1), x_{c-k}\}$), and the merge stage uses $p-2$ comparisons. Therefore, the total number of comparisons is $3p-4$ for generating $p$ outputs.

The GK algorithm improves both the preprocessing and merge stages of the HGW algorithm. In the preprocessing stage, the saving of comparisons is achieved by considering the computations of $P_c(\cdot)$ and $S_{c+p}(\cdot)$ together when processing the elements $x_c, x_{c+1}, \ldots, x_{c+p}$. Let $q = \lceil (p+1)/2 \rceil$. The GK algorithm first computes $P_c(k)$ for $k = 0, 1, \ldots, p-1$ using $q-1$ comparisons, and then computes $S_{c+p}(k)$ for $k = 0, 1, \ldots, p-1$ using $p-q$ comparisons. If $P_c(q-1) \geq S_{c+p}(p-q)$, then no element in $\{x_{c+j} \mid q \leq j < p\}$ can be greater than $P_c(q-1)$. In such a case, we have $P_c(j) = P_c(q-1)$ for $q \leq j < p$ (note that no comparison is required), and only $q-1$ more comparisons are required to finish computing $S_{c+p}(k)$ for $p-q < k < p$. Similarly, if $P_c(q-1) \leq S_{c+p}(p-q)$, we have $S_{c+p}(j) = S_{c+p}(p-q)$ for $p-q < j < p$, and $p-q$ more comparisons are required to finish the computation of $P_c(k)$ for $q \leq k < p$. This GK improvement reduces the total number of comparisons for the preprocessing stage to $1.5p - \frac{(p \mod 2)}{2}$.

For the merge stage, the GK algorithm reduces the number of comparisons employing the following observation: In equation (1), there is an index $k'$ such that $y_{c-p+1+k} = S_c(p-1-k)$ for all $1 \leq k \leq k'$ and $y_{c-p+1+k} = P_c(k)$ for all $k' \leq k \leq p-2$. Therefore, a binary search for $k'$ using $\lceil \log(p-1) \rceil$ comparisons is sufficient to do the merge. Combining the improvements for the preprocessing stage and the merge stage, the amortized number of comparisons for $p$ outputs is $1.5p - \frac{(p \mod 2)}{2} + \lceil \log(p-1) \rceil$, which is equivalent to $1.5 + \frac{\log 2}{p} + O(\frac{1}{p})$ per sample.

### 3 Improved Algorithm

Our improvement is obtained by improving the preprocessing stage of the GK algorithm. When considering the computations of $S_{c+p}(\cdot)$ and $P_c(\cdot)$ together, the GK algorithm achieves saving of comparisons by dividing the sequence $x_c, x_{c+1}, \ldots, x_{c+p}$ into two halves, and then computing the prefix maximums of the first half and suffix maximums of the second half. The half that has a smaller maximum will extend the maximum computation into the other half by doing more comparisons to finish the computation. The worst case for the GK algorithm happens when the maximum is at or near the boundaries of the sequence $x_c, x_{c+1}, \ldots, x_{c+p}$ in which case many comparisons are redundant. For example, if the maximum is located at $x_{c+3}$ (assume that it is at the left half), then the prefix maximum computations for $P_c(i)$ (where $3 < i < q$) cost too many comparisons.

This preprocessing strategy can be refined by adaptively advancing the prefix/suffix maximum computations, so that not many comparisons are done after the prefix/suffix maximum touches the actual maximum of $x_c, x_{c+1}, \ldots, x_{c+p}$. At the beginning, a current prefix index $i$ and a current suffix index $j$ are both set to 0. The index $i$ is used to track the progress of prefix maximum computation, and its semantic meaning is that $P_c(k)$ has already been computed for $0 \leq k \leq i$. Similarly, the index $j$ is used to track the progress of suffix maximum computation, and it means that $S_{c+p}(k)$ has already been computed for $0 \leq k \leq j$. Note that initially, $P_c(0) = x_c$ and $S_{c+p}(0) = x_{c+p}$.

Let $s \geq 1$ be a default step size (a parameter whose value will be fixed to $\lceil \sqrt{p-1} \rceil$ later). Our algorithm will adaptively advance the value of $i$ and $j$ until $i + j = p + s - 1$. In each advancement (except the last one), either $i$ or $j$ will be advanced by $s$. In the last advancement (right before $i+j$ reaches $p+s-1$), the step size is $s$ if $(p-1) \mod s = 0$; or the step size is $(p-1) \mod s$ if $(p-1) \mod s \neq 0$. More specifically, except in the last advancement, if $P_c(i) \leq S_{c+p}(j)$, then $P_c(k)$ for $i < k \leq i+s$ will be computed, and after that $i$ will have increased by $s$. Similarly, if $P_c(i) > S_{c+p}(j)$, then $S_{c+p}(k)$ for $j < k \leq j+s$ will be computed, and after that $j$ will have increased by $s$. In the last advancement, similar computation is done except that a different step size $(p-1) \mod s$ may be used (see the beginning part of this paragraph).

There are $\lceil \frac{p+s-1}{s} \rceil$ advancements. So the number of comparisons spent on determining whether to advance $i$ or $j$ (by comparing $P_c(i)$ to $S_{c+p}(j)$) is $\lceil \frac{p+s-1}{s} \rceil$. At each time when $i$ (or $j$) is increased by $s$, there are $\delta$ comparisons for computing $\delta$ entries of $P_c(\cdot)$ (or $S_{c+p}(\cdot)$). Because $i+j$ is 0 at the beginning, and is changed to $p+s-1$ at the end, $p+s-1$ comparisons are made other than the comparisons for determining whether to advance $i$ or $j$. So the total number of comparisons so far is

$$p + s - 1 + \left\lceil \frac{p + s - 1}{s} \right\rceil.$$  

After $i+j = p + s - 1$, no comparison is required and we will show that

$$P_c(k) = P_c(i) \text{ for } i < k \leq p, \quad (2)$$

$$S_{c+p}(k) = S_{c+p}(j) \text{ for } j < k \leq p. \quad (3)$$

The preprocessing stage is finished, and the correctness is based on Lemma 1 (given below). The pseudo-code of this algorithm is given in Algorithm 1. Line 5 sets the $\delta$ to be the advancement step size. Lines 7 to 10 advance the prefix computations, and lines 12 to 15 advance the suffix computations. Note that in Algorithm 1, some special boundary cases are considered when $i$ or $j$ reaches $p-1$, but these cases do not affect the correctness and complexity of the algorithm.

**Lemma 1:** When $i+j = p+s-1$, equations (2) and (3) hold.
Algorithm 1: Improved Preprocessing Stage

**Input:** \(x_c, x_{c+1}, \ldots, x_{c+p}\) and a parameter \(s\)

**Output:** \(P_c(k)\) and \(S_{c+p}(k)\) for \(0 \leq k \leq p - 1\)

1. \(P_c(0) \leftarrow x_c\)
2. \(S_{c+p}(0) \leftarrow x_{c+p}\)
3. \(i \leftarrow 0, j \leftarrow 0\)
4. **while** \(i + j < p + s - 1\) **do**
5. \(\delta \leftarrow \min\{s, p + s - 1 - i - j\}\)
6. **if** \(i < p - 1\) **and** \(P_c(i) \leq S_{c+p}(j)\) **then**
7.   **for** \(k = i + 1\) **to** \(\min\{i + \delta, p - 1\}\) **do**
8.     \(P_c(k) \leftarrow \min\{P_c(k - 1), x_{c+k}\}\)
9.   **end**
10. \(i \leftarrow i + \delta\)
11. **else**
12.   **for** \(k = j + 1\) **to** \(\min\{j + \delta, p - 1\}\) **do**
13.     \(S_{c+p}(k) \leftarrow \max\{S_{c+p}(k - 1), x_{c-p+k}\}\)
14.   **end**
15. \(j \leftarrow j + \delta\)
16. **end**
17. **while** \(i < p - 1\) **do**
18. \(i \leftarrow i + 1\)
19. \(P_c(i) \leftarrow P_c(i - 1)\)
20. **end**
21. **while** \(j < p - 1\) **do**
22. \(j \leftarrow j + 1\)
23. \(S_{c+p}(j) \leftarrow S_{c+p}(j - 1)\)
24. **end**

**Proof:** Without loss of generality, assume that all the input elements are distinct (ties can be broken by considering their indices). After each advancement, let \(l\) be the index of the maximum element of \(\{x_k \mid k \in [c, c+i]\} \cup \{c+p-j, c+p\}\) (the set of elements considered so far), then we must have either \(c + i - s < l \leq c + i\) or \(c + p - j < l \leq c + p - j + s\). In other words, \(l\) is within a distance of \(s - 1\) from either \(c + i\) or \(c + p - j\). This is because the algorithm always advances \(i\) if the current prefix maximum is smaller, or advances \(j\) if the current suffix maximum is smaller.

When \(i + j = p + s - 1\), because of \(c + i + 1 \geq c + p - j\), the \(x_l\) is the maximum element of \(\{x_k \mid k \in [c, c+p]\}\). According to the previous paragraph, \(l \leq c + i\) or \(l < c + p - j + s\). The latter inequality implies that:

\[ l \leq c + p - j + s - 1 = c + (p + s - 1 - j) = c + i. \]

Therefore, we always have \(l \leq c + i\). This means that no element in \(\{x_k \mid k \in [c+i+1, c+p]\}\) can be greater than the current prefix maximum, hence equation (2) holds. By a similar argument, equation (3) holds.

If \(s = \lceil \sqrt{p-1} \rceil\), then the total number of comparisons for the preprocessing stage is bounded by:

\[
\begin{align*}
p + s - 1 + \left\lceil \frac{p + s - 1}{s} \right\rceil &= p + \left\lceil \sqrt{p - 1} \right\rceil - 1 + \left\lceil \frac{p - 1}{\sqrt{p - 1}} \right\rceil + 1 \\
&\leq p + \left\lceil \sqrt{p - 1} \right\rceil + \left\lceil \frac{p - 1}{\sqrt{p - 1}} \right\rceil \\
&= p + 2 \left\lceil \sqrt{p - 1} \right\rceil.
\end{align*}
\]

Combining with the merge stage of the GK algorithm (which requires \(\lceil \log(p-1) \rceil\)) comparisons by a binary search), the total amortized number of comparisons per sample of Algorithm 1 is bounded by:

\[
\begin{align*}
C_1 &= \frac{1}{p} \cdot \left( p + 2 \left\lceil \sqrt{p - 1} \right\rceil + \lceil \log(p-1) \rceil \right) \\
&\leq \frac{1}{p} \cdot (p + 2\sqrt{p} + \log p + 3) \\
&\leq 1 + \frac{2}{\sqrt{p}} + \frac{\log p}{p} + O \left( \frac{1}{p} \right) \\
&= 1 + O \left( \frac{1}{\sqrt{p}} \right) \\
&= 1 + o(1).
\end{align*}
\]

This establishes the following theorem:

**Theorem 1:** The 1D running maximum (or minimum) filter can be computed in \(1 + o(1)\) comparisons per sample in the worst case by a deterministic algorithm.

Note that the GK algorithm is very similar to ours when \(s = q - 1 = \lfloor (p-1)/2 \rfloor\).

It was shown that the \(d\)-dimensional max/min filter can be computed by applying the 1D filter \(d\) times \([6]\), using a structuring element decomposition approach \([8]\). More specifically, the following lemma holds:

**Lemma 2:** Let \(C_1(p)\) be the number of comparisons per sample of an 1D filter for a 1D window of size \(p\), then a \(d\)-dimensional max/min filter with window size \(p_1 \times p_2 \times \cdots \times p_d\) can be computed by \(\sum_{i=1}^{d} C_1(p_i)\) comparisons per sample.

Based on Lemma 2, we have Corollary 1.

**Corollary 1:** The \(d\)-dimensional maximum (or minimum) filter can be computed deterministically using \(d(1+o(1))\) comparisons per sample in the worst case.

Gil and Kimmel \([7]\) showed that the opening/closing filter can be computed using \(C_1 + O(\log^2 p/p)\) comparisons per sample. Therefore, the number of comparisons per sample for computing the opening (or closing) filter is at most

\[
\begin{align*}
C_1 + O \left( \frac{\log^2 p}{p} \right) &= 1 + O \left( \frac{1}{\sqrt{p}} \right) + O \left( \frac{\log^2 p}{p} \right) \\
&= 1 + O \left( \frac{1}{\sqrt{p}} \right) \\
&= 1 + o(1).
\end{align*}
\]
Corollary 2: The 1D opening (or closing) filter can be computed deterministically using $1 + o(1)$ comparisons per sample in the worst case.

Gil and Kimmel [7] shows that $d$-dimensional opening (or closing) can be computed by running 1D min filter $d - 1$ times, 1D max filter $d - 1$ times, and 1D opening (or closing) filter once. So we have Corollary 3.

Corollary 3: The $d$-dimensional opening (or closing) filter can be computed deterministically using $(2d - 1)(1 + o(1))$ comparisons per sample in the worst case.

4 EXPERIMENTAL EVALUATION

The comparison model does not capture other costs, like comparisons between indices, memory accesses, branch mispredictions, etc. Therefore, we do not expect our algorithm to beat existing methods in actual implementations even when the window size is large, unless the comparison between two input elements is very expensive. So here, we only show the experiment result for the case where the costs to compare input elements are high. One such a case is when two 128-bit floating-point numbers are compared in a 64-bit processor without hardware supports for 128-bit floating-point computations. Note that for the cases when the comparisons of input elements are cheap, we do not observe that our algorithm outperforms previous ones even for large window sizes.

We implemented the 1D max filter using the HGW algorithm, GK algorithm, and our algorithm in C++. The experiment is conducted on a laptop computer with an Intel Core i5-540M 2.53GHz processor and 2GB main memory. The operating system is Ubuntu 11.04 64-bit version, and the compiler is the GNU Compiler Collections (GCC) 4.5.2. The compilation option that we used is “-O2”. The compiled binary file is in the 64-bit mode. The data type is the quadruple precision floating-point number (e.g., __float128 in the C++ of GCC), which has 128 bits. The comparisons for quadruple precision floating-point numbers are very expensive, because they are currently implemented using a software emulation.

We generated $10^7$ random input numbers, and tested the three algorithms on the input with window sizes from 2 to 100. The running times are shown in Figure 1. Our algorithm (denoted by YA in the figure) outperforms the GK algorithm when $p \geq 33$ in this experiment.

5 CONCLUDING REMARKS

In this work, we asymptotically improve the state-of-the-art for computing running max/min filters. Some questions remain open: Is it possible to simultaneously compute the 1D maximum and minimum filters with less than $2C_1$ (e.g., 2 or even 1.5) comparisons per sample? Is it possible to compute the 2D maximum (or minimum) filter with less than $2C_1$ (e.g., 2 or even 1.5) comparisons per sample? Can the 2D opening or closing be computed by 3 or less comparisons per sample?
ACKNOWLEDGMENTS

We would like to thank the referees for their helpful comments and suggestions.

REFERENCES


