STREAMING MAXIMUM-MINIMUM FILTER USING NO MORE THAN 3 COMPARISONS PER ELEMENT

Daniel Lemire
University of Quebec at Montreal (UQAM), UER ST
100 Sherbrooke West, Montreal (Quebec), H2X 3P2 Canada
lemire@acm.org

Abstract. The running maximum-minimum (MAX-MIN) filter computes the maxima and minima over running windows of size w. This filter has numerous applications in signal processing and time series analysis. We present an easy-to-implement online algorithm requiring no more than 3 comparisons per element, in the worst case. Comparatively, no algorithm is known to compute the running maximum (or minimum) filter in 1.5 comparisons per element, in the worst case. Our algorithm has reduced latency and memory usage.

ACM CCS Categories and Subject Descriptors: F.2.1 Numerical Algorithms and Problems

Key words: Design of Algorithms, Data Streams, Time Series, Latency, Monotonicity

1. Introduction

The maximum and the minimum are the simplest form of order statistics. Computing either the global maximum or the global minimum of an array of n elements requires $n - 1$ comparisons, or slightly less than one comparison per element. However, to compute simultaneously the maximum and the minimum, only $3\lceil n/2 \rceil - 2$ comparisons are required in the worst case [Cormen *et al.* 2001], or slightly less than 1.5 comparisons per element.

A related problem is the computation of the running maximum-minimum (MAX-MIN) filter: given an array $a_1, \ldots, a_n$, find the maximum and the minimum over all windows of size w, that is max / $\min_{i \in [j, j+w)} a_i$ for all j (see Fig. 1.1). The running maximum (MAX) and minimum (MIN) filters are defined similarly. The MAX-MIN filter problem is harder than the GLOBAL MAX-MIN problem, but a tight bound on the number of comparisons required in the worst case remains an open problem.

Running maximum-minimum (MAX-MIN) filters are used in signal processing and pattern recognition. As an example, Keogh and Ratanamahatana [2005] use a precomputed MAX-MIN filter to approximate the time warping distance between two time series. Time series applications range from music retrieval [Zhu and Shasha 2003] to network security Sun *et al.* [2004]. The unidimensional MAX-MIN filter can be applied to images and other bidimensional data by first applying the unidimensional on rows and then on columns. Image processing applications include cancer diagnosis [He *et al.* 2005], character [Ye *et al.* 2001] and handwriting [Ye

Received August 13, 2018.

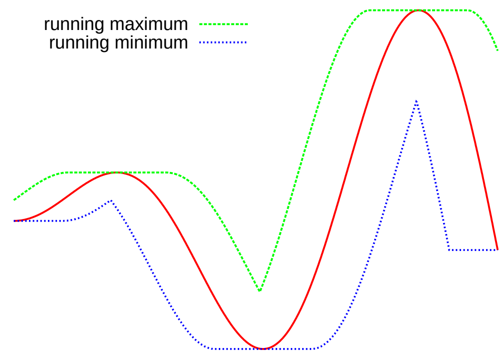


Fig. 1.1: Example of a running MAX-MIN filter.

et al. 2001] recognition, and boundary feature comparison [Taycher and Garakani 2004].

We define the *stream latency* of a filter as the maximum number of data points required after the window has passed. For example, an algorithm requiring that the whole data set be available before the running filter can be computed has a high stream latency. In effect, the stream latency is a measure of an algorithm on the batch/online scale.

We present the first algorithm to compute the combined MAX-MIN filter in no more than 3 comparisons per element, in the worst case. Indeed, we are able to save some comparisons by not treating the MAX-MIN filter as the aggregate of the MAX and MIN filters: if x is strictly larger than k other numbers, then there is no need to check whether x is smaller than any of these numbers. Additionally, it is the first algorithm to require a constant number of comparisons per element without any stream latency and it uses less memory than competitive alternatives. Further, our algorithm requires no more than 2 comparisons per element when the input data is monotonic (either non-increasing or non-decreasing). We provide experimental evidence that our algorithm is competitive and can be substantially faster (by a factor of 2) when the input data in piecewise monotonic. A maybe surprising result is that our algorithm is arguably simpler to implement than the recently proposed algorithms such as Gil and Kimmel [2002] or Droogenbroeck and Buckley [2005]. Finally, we prove that at least 2 comparisons per element are required to compute the MAX-MIN filter when no stream latency is allowed.

2. Related Work

Pitas [1989] presented the MAX filter algorithm MAXLINE requiring $O(\log w)$ comparisons per element in the worst case and an average-case performance over independent and identically distributed (i.i.d.) noise data of slightly more than 3 comparisons per element. Douglas [1996] presented a better alternative: the MAX filter

TABLE I: Worst-case number of comparisons and stream latency for competitive MAX-MIN filter algorithms. Stream latency and memory usage (buffer) are given in number of elements.

algorithm	comparisons per element (worst case)	stream latency	buffer
naive	$2w-2$	**0**	$O(1)$
van Herk [1992], Gil and Werman [1993]	$6-8/w$	w	$4w+O(1)$
Gil and Kimmel [2002]	$3+2\log w/w$ $+O(1/w)$	w	$6w+O(1)$
New algorithm	**3**	**0**	$2w+O(1)$

algorithm MAXLIST was shown to average 3 comparisons per element for i.i.d. input signals and Myers and Zheng [1997] presented an asynchronous implementation.

More recently, van Herk [1992] and Gil and Werman [1993] presented an algorithm requiring $6-8/w$ comparisons per element, in the worst case. The algorithm is based on the batch computation of cumulative maxima and minima over overlapping blocks of $2w$ elements. For each filter (MAX and MIN), it uses a memory buffer of $2w + O(1)$ elements. We will refer to this algorithm as the VAN HERK-GIL-WERMAN algorithm. Gil and Kimmel [2002] proposed an improved version (GIL-KIMMEL) which lowered the number of comparisons per element to slightly more than 3 comparisons per element, but at the cost of some added memory usage and implementation complexity (see Table I and Fig. 2.2 for summary). For i.i.d. noise data, Gil and Kimmel presented a variant of the algorithm requiring $\approx 2+(2+\ln 2/2)\log w/w$ comparisons per element (amortized), but with the same worst case complexity. Monotonic data is a worst case input for the GIL-KIMMEL variant.

Droogenbroeck and Buckley [2005] proposed a fast algorithm based on anchors. They do not improve on the number of comparisons per element. For window sizes ranging from 10 to 30 and data values ranging from 0 to 255, their implementation has a running time lower than their VAN HERK-GIL-WERMAN implementation by as much as 30%. Their GIL-KIMMEL implementation outperforms their VAN HERK-GIL-WERMAN implementation by as much as 15% for window sizes larger than 15, but is outperformed similarly for smaller window sizes, and both are comparable for a window size equals to 15. The Droogenbroeck-Buckley MIN filter pseudocode alone requires a full page compared to a few lines for VAN HERK-GIL-WERMAN algorithm. Their experiments did not consider window sizes beyond $w = 30$ nor arbitrary floating point data values.

3. Lower Bounds on the Number of Comparisons

Gil and Kimmel [2002] showed that the PREFIX MAX-MIN ($\max / \min_{i\leq j} a_i$ for all j) requires at least $\log 3 \approx 1.58$ comparisons per element, while they conjectured that at least 2 comparisons are required. We prove that their result applies directly to

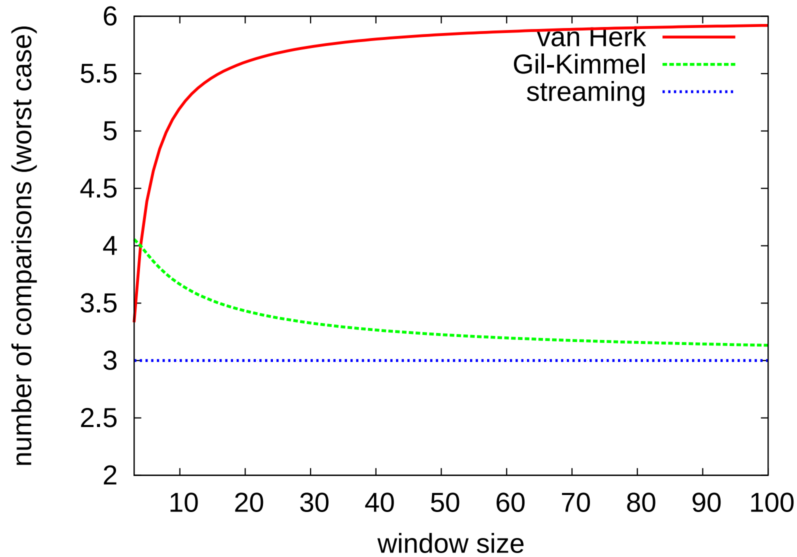


Fig. 2.2: Worst-case number of comparisons per element with the VAN HERK-GIL-WERMAN (van Herk) algorithm, the GIL-KIMMEL algorithm, and our new streaming algorithm (less is better).

the MIN-MAX filter problem and show that 2 comparisons per element are required when no latency is allowed.

THEOREM 1. *The* MIN-MAX *filter problem requires at least 2 comparisons per element when no stream latency is allowed, and* $\log 3$ *comparisons per element otherwise.*

PROOF. Let array values be distinct real numbers. When no stream latency is allowed, we must return the maximum and minimum of window $(i-w,i]$ using only the data values and comparisons in $[1,i]$. An adversary can choose the array value a_i so that a_i must be compared at least twice with preceding values: it takes two comparisons with a_i to determine that it is neither a maximum nor a minimum ($a_i \in (\min_{j\in(i-w,i]} a_j, \max_{j\in(i-w,i]} a_j)$). Hence, two comparisons per element are required in the worst case.

Next we assume stream latency is allowed. Browsing the array from left to right, each new data point a_i for $i \in [w,n]$ can be either a new maximum ($a_i = \max_{j\in(i-w,i]} a_j$), a new minimum ($a_i = \min_{j\in(i-w,i]} a_j$), or neither a new maximum or a new minimum ($a_i \in (\min_{j\in(i-w,i]} a_j, \max_{j\in(i-w,i]} a_j)$). For any ternary sequence such as MAX-MAX-MIN-NOMAXMIN-MIN-MAX-..., we can generate a corresponding array. This means that a MIN-MAX filter needs to distinguish between more than 3^{n-w} different partial order over the values in the array a. In other words, the binary decision tree must have more than 3^{n-w} leaves. Any binary tree having l leaves has height at least $\lceil \log l \rceil$. Hence, our binary tree must have height at least $\lceil \log 3^{n-w} \rceil \geq (n-w)\log 3$, proving our result. □

By the next proposition, we show that the general lower bound of 2 comparisons per element is tight.

Proposition 1. *There exists an algorithm to compute the* MIN-MAX *filter in no more than 2 comparisons per element when the window size is 3 ($w = 3$), with no stream latency.*

Proof. Suppose we know the location of the maximum and minimum of the window $[i-3, i-1]$. Then we know the maximum and minimum of $\{a_{i-2}, a_{i-1}\}$. Hence, to compute the maximum and minimum of $\{a_{i-2}, a_{i-1}, a_i\}$, it suffices to determine whether $a_{i-1} > a_i$ and whether $a_{i-2} > a_i$. □

4. The Novel Streaming Algorithm

To compute a running MAX-MIN filter, it is sufficient to maintain a *monotonic wedge* (see Fig. 4.3). Given an array $a = a_1, \ldots, a_n$, a monotonic wedge is made of two lists U, L where U_1 and L_1 are the locations of global maximum and minimum, U_2 and L_2 are the locations of the global maximum and minimum in (U_1, ∞) and (L_1, ∞), and so on. Formally, U and L satisfy $\max_{i>U_{j-1}} a_i = a_{U_j}$ and $\min_{i>L_{j-1}} a_i = a_{L_j}$ for $j = 1, 2, \ldots$ where, by convention, $U_0 = L_0 = -\infty$. If all values of a are distinct, then the monotonic wedge U, L is unique. The location of the last data point n in a, is the last value stored in both U and L (see U_5 and L_4 in Fig. 4.3). A monotonic wedge has the property that it keeps the location of the current (global) maximum (U_1) and minimum (L_1) while it can be easily updated as we remove data points from the left or append them from the right:

- to compute a monotonic wedge of $a_2, a_3, \ldots, a_n$ given a monotonic wedge U, L for $a_1, a_2, \ldots, a_n$, it suffices to remove (pop) U_1 from U if $U_1 = 1$ or L_1 from L if $L_1 = 1$;
- similarly, to compute the monotonic wedge of $a_1, a_2, \ldots, a_n, a_{n+1}$, if $a_{n+1} > a_n$, it suffices to remove the last locations stored in U until $a_{\mathrm{last}(U)} \geq a_{n+1}$ or else, to remove the last locations stored in L until $a_{\mathrm{last}(L)} \leq a_{n+1}$, and then to append the location $n+1$ to both U and L.

Fig. 4.4 provides a visual example. In Step A, we begin with a monotonic wedge. In Step B, we add value a_i to the interval. This new value is compared against the last value (a_{i-1}) and since $a_i > a_{U_5}$, we remove the index U_5 from U. Similarly, because $a_i > a_{U_4}$, we also remove U_4. In Step C, the index i is appended to both U and L and we have a new (extended) monotonic wedge. Algorithm 1 would further remove L_1, consider the next value forward, and so on.

Algorithm 1 and proposition 2 show that a monotonic wedge can be used to compute the MAX-MIN filter efficiently and with few lines of code.

Proposition 2. *Algorithm 1 computes the* MAX-MIN *filter using no more than 3 comparisons per element.*

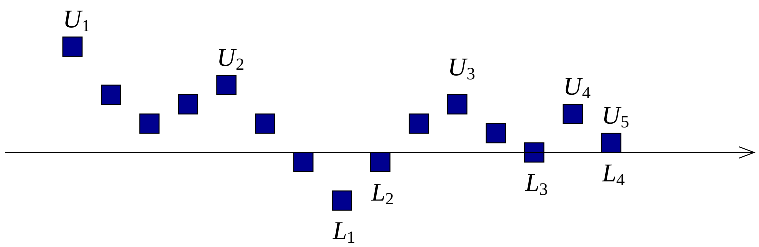


Fig. 4.3: Example of a monotonic wedge: data points run from left to right.

Proof. We prove by induction that in Algorithm 1, U and L form a monotonic wedge of a over the interval $[\max\{i-w,1\},i)$ at the beginning of the main loop (line 5). Initially, when $i=2$, $U,L=\{1\}$, U,L is trivially a monotonic wedge. We have that the last component of both U and L is $i-1$. If $a_i > a_{i-1}$ (line 11), then we remove the last elements of U until $a_{\text{last}(U)} \geq a_{n+1}$ (line 11) or if $a_i \leq a_{i-1}$, we remove the last elements of L until $a_{\text{last}(L)} \leq a_{n+1}$ (line 15). Then we append i to both U and L (line 17). The lists U,L form a monotonic wedge of $[\max\{i-w,1\},i]$ at this point (see Fig. 4.4). After appending the latest location i (line 17), any location $j<i$ will appear in either U or L, but not in both. Indeed, $i-1$ is necessarily removed from either U or L. To compute the monotonic wedge over $[\max\{i-w+1,1\},i+1)$ from the monotonic wedge over $[\max\{i-w,1\},i]$, we check

Algorithm 1 Streaming algorithm to compute the MAX-MIN filter using no more than 3 comparisons per element.

1: **INPUT:** an array a indexed from 1 to n
2: **INPUT:** window width $w>2$
3: $U,L \leftarrow$ empty double-ended queues, we append to "back"
4: append 1 to U and L
5: **for** i in $\{2,\ldots,n\}$ **do**
6:     **if** $i \geq w+1$ **then**
7:         **OUTPUT:** $a_{\text{front}(U)}$ as maximum of range $[i-w,i)$
8:         **OUTPUT:** $a_{\text{front}(L)}$ as minimum of range $[i-w,i)$
9:     **if** $a_i > a_{i-1}$ **then**
10:         pop U from back
11:         **while** $a_i > a_{\text{back}(U)}$ **do**
12:             pop U from back
13:     **else**
14:         pop L from back
15:         **while** $a_i < a_{\text{back}(L)}$ **do**
16:             pop L from back
17:     append i to U and L
18:     **if** $i = w + \text{front}(U)$ **then**
19:         pop U from front
20:     **else if** $i = w + \text{front}(L)$ **then**
21:         pop L from front

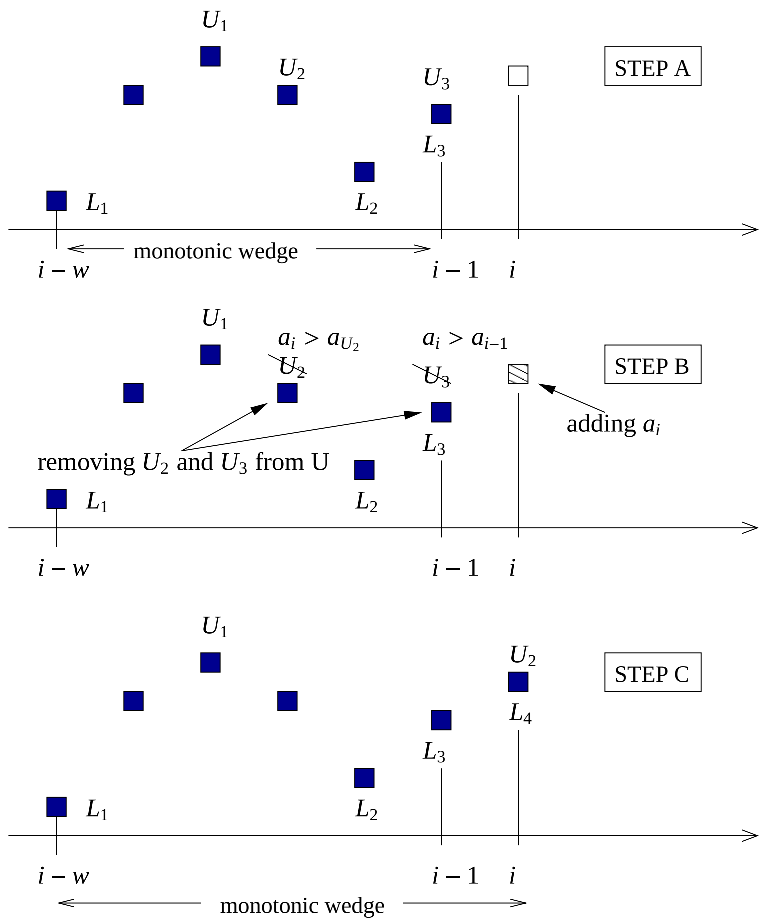


Fig. 4.4: Algorithm 1 from line 5 to line 17: updating the monotonic wedge is done by either removing the last elements of U or the last elements of L until U, L form a monotonic wedge for $[\max\{i - w, 1\}, i]$.

whether the location $i - w$ is in U or L at line 18 and if so, we remove it. Hence, the algorithm produces the correct result.

We have to prove that the algorithm will not use more than 3 comparisons per element on average, no matter what the input data is. Firstly, each new element i will be compared once against the previous element at line 9. Secondly, in the worst case, for each element, the condition of the while loop (either at line 11 or line 15) will evaluate once to false when the element is first encountered and once to true on a subsequent iteration through the main loop. Therefore, in the worst case, the algorithm uses at most 3 comparisons per element. □

While some signals such as electroencephalograms (EEG) resemble i.i.d noise, many more real-world signals are piecewise quasi-monotonic [Lemire *et al.* 2005].

While one Gil-Kimmel variant [Gil and Kimmel 2002] has a comparison complexity of nearly 2 comparisons per element over i.i.d noise, but a worst case complexity of slightly more than 3 comparisons for monotonic data, the opposite is true of our algorithm as demonstrated by the following proposition.

Proposition 3. *When the data is monotonic, Algorithm 1 computes the* max-min *filter using no more than 2 comparisons per element.*

Proof. If the input data is non-decreasing or non-increasing, then the conditions at line 11 and line 15 will never be true. Thus, in the worse case, for each new element, there is one comparison at line 9 and one at either line 11 or line 15. □

The next proposition shows that the memory usage of the monotonic wedge is at most $w+1$ elements. Because U and L only store the indexes, we say that the total memory buffer size of the algorithm is $2w + O(1)$ elements (see Table I).

Proposition 4. *In Algorithm 1, the number of elements in the monotonic wedge ($size(U) + size(L)$) is no more than $w + 1$.*

Proof. Each new element is added to both U and L at line 17, but in the next iteration of the main loop, this new element is removed from either U or L (line 10 or 14). Hence, after line 14 no element in the w possible elements can appear both in U and L. Therefore $\text{size}(U) + \text{size}(L) \leq w + 1$. □

5. Implementation and Experimental Results

While interesting theoretically, the number of comparison per element is not necessarily a good indication of real-world performance. We implemented our algorithm in C++ using the STL `deque` template. A more efficiently data structure might be possible since the size of our double-ended queues are bounded by w. We used 64 bits floating point numbers ("double" type). In the pseudocode of Algorithm 1, we append i to the two double-ended queues, and then we systematically pop one of them (see proof of proposition 2). We found it slightly faster to rewrite the code to avoid one pop and one append (see appendix). The implementation our algorithm stores only the location of the extrema whereas our implementation of the van Herk-Gil-Werman algorithm stores values. Storing locations means that we can compute the arg max / min filter with no overhead, but each comparison is slightly more expensive. While our implementation uses 32 bits integers to store locations, 64 bits integers should be used when processing streams. For small window sizes, Gil and Kimmel [2002] suggests unrolling the loops, essentially compiling w in the code: in this manner we could probably do away with a dynamic data structure and the corresponding overhead.

We ran our tests on an AMD Athlon 64 3200+ using a 64 bit Linux platform with 1 Gigabyte of RAM (no thrashing observed). The source code was compiled using the GNU GCC 3.4 compiler with the optimizer option "-O2".

We process synthetic data sets made of 1 million data points and report wall clock timings versus the window width (see Fig. 5.5). The linear time complexity of the naive algorithm is quite apparent for $w > 10$, but for small window sizes ($w < 10$), it remains a viable alternative. Over i.i.d. noise generated with the Unix `rand` function, the VAN HERK-GIL-WERMAN and our algorithm are comparable (see Fig. 5(b)): both can process 1 million data points in about 0.15 s irrespective of the window width. For piecewise monotonic data such as a sine wave (see Fig. 5(a)) our algorithm is roughly twice as fast and can process 1 million data points in about 0.075 s. Our C++ implementation of the GIL-KIMMEL algorithm [Gil and Kimmel 2002] performed slightly worse than the VAN HERK-GIL-WERMAN algorithm. To insure reproducibility, the source code is available freely from the author.

6. Conclusion and Future Work

We presented an algorithm to compute the MAX-MIN filter using no more than 3 comparisons per element in the worst case whereas the previous best result was slightly above $3 + 2\log w/w + O(1/w)$ comparisons per element. Our algorithm has lower latency, is easy to implement, and has reduced memory usage. For monotonic input, our algorithm incurs a cost of no more than 2 comparisons per element. Experimentally, our algorithm is especially competitive when the input is piecewise monotonic: it is twice as fast on a sine wave.

We have shown that at least 2 comparisons per element are required to solve the MAX-MIN filter problem when no stream latency is allowed, and we showed that this bound is tight when the window is small ($w = 3$).

Acknowledgements

This work is supported by NSERC grant 261437. The author wishes to thank Owen Kaser of the University of New Brunswick for his insightful comments.

References

CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L., ET AL. 2001. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

DOUGLAS, S.C. 1996. Running max/min calculation using a pruned ordered list. *IEEE Transactions on Signal Processing 44*, 11, 2872–2877.

DROOGENBROECK, M. AND BUCKLEY, M. J. 2005. Morphological Erosions and Openings: Fast Algorithms Based on Anchors. *J. Math. Imaging Vis. 22*, 2-3, 121–142.

GIL, J. AND KIMMEL, R. 2002. Efficient Dilation, Erosion, Opening, and Closing Algorithms. *IEEE Trans. Pattern Anal. Mach. Intell. 24*, 12, 1606–1617.

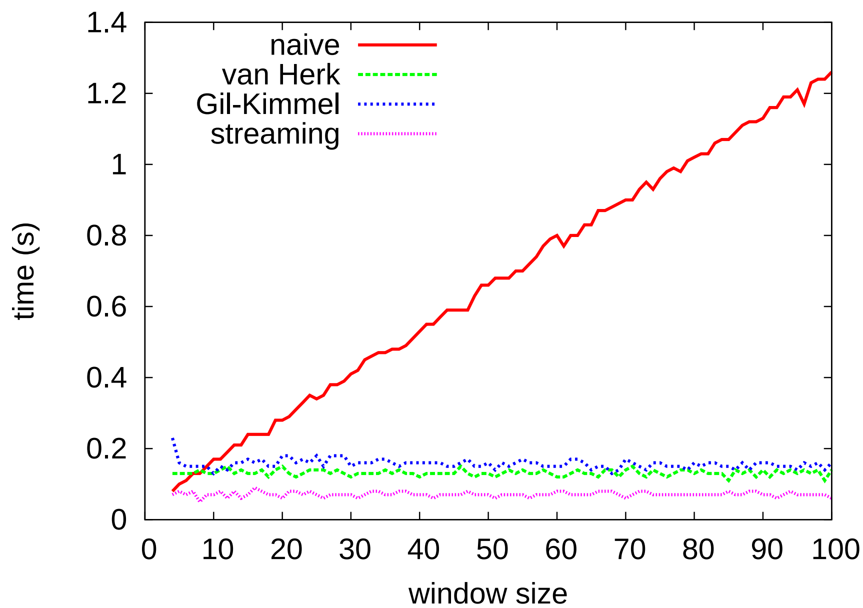
GIL, J. AND WERMAN, M. 1993. Computing 2-D Min, Median, and Max Filters. *IEEE Trans. Pattern Anal. Mach. Intell. 15*, 5, 504–507.

GIL, JOSEPH (YOSSI) AND KIMMEL, RON. 2002. Data filtering apparatus and method. US Patent Number 6,952,502.

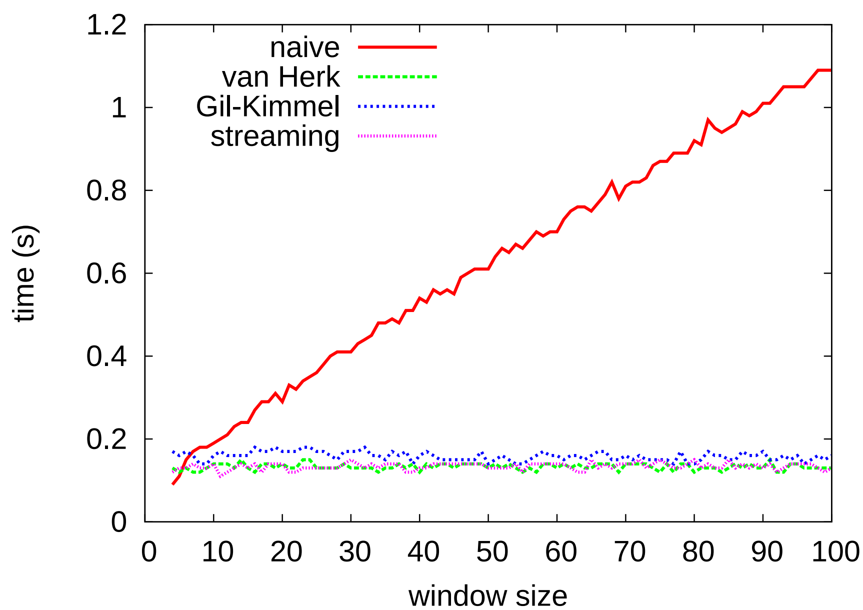
HE, Y.-L., TIAN, L.-F., ZHU, C.-M., CHEN, P., LI, B., AND MAO, Z.-Y. 2005. Development of intelligent diagnosis and report system based on whole body bone SPECT image. In *Machine Learning and Cybernetics 2005*, 5437–5441.

KEOGH, E. AND RATANAMAHATANA, C.A. 2005. Exact indexing of dynamic time warping. *Knowledge and Information Systems 7*, 3, 358–386.

LEMIRE, D., BROOKS, M., AND YAN, Y. 2005. An Optimal Linear Time Algorithm for Quasi-Monotonic Segmentation. In *ICDM'05*, 709–712.



(a) Input data is a sine wave with a period of 10 000 data points.



(b) Input data is i.i.d. noise with a uniform distribution.

Fig. 5.5: Running time to compute the MAX-MIN filter over a million data points using the naive algorithm, our VAN HERK-GIL-WERMAN (van Herk) implementation, our GIL-KIMMEL implementation, and our streaming implementation (less is better).

MYERS, C. AND ZHENG, H. 1997. An Asynchronous Implementation of the MAXLIST Algorithm. In *ICASSP'97*.

PITAS, I. 1989. Fast algorithms for running ordering and max/min calculation. *IEEE Transactions on Circuits and Systems 36*, 6, 795–804.

SUN, H., LUI, JCS, AND YAU, DKY. 2004. Defending against low-rate TCP attacks: dynamic detection and protection. In *ICNP 2004*, 196–205.

TAYCHER, L. AND GARAKANI, A. 2004. Machine vision methods and systems for boundary feature comparison of patterns and images. US Patent Number 6,687,402.

VAN HERK, MARCEL. 1992. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recogn. Lett. 13*, 7, 517–521.

YE, X., CHERIET, M., AND SUEN, C. Y. 2001. A generic method of cleaning and enhancing handwritten data from business forms. *International Journal on Document Analysis and Recognition 4*, 2, 84–96.

YE, X., CHERIET, M., AND SUEN, CY. 2001. Stroke-model-based character extraction from gray-level documentimages. *IEEE Transactions on Image Processing 10*, 8, 1152–1161.

ZHU, Y. AND SHASHA, D. 2003. Warping indexes with envelope transforms for query by humming. In *SIGMOD'03*, 181–192.

Appendix: C++ source code for the streaming algorithm

```cpp
// input: array a, integer window width w
// output: arrays maxval and minval
// buffer: lists U and L
// requires: STL for deque support
deque<int> U, L;
for(uint i = 1; i < a.size(); ++i) {
 if(i>=w) {
   maxval[i-w] = a[U.size()>0 ? U.front() : i-1];
   minval[i-w] = a[L.size()>0 ? L.front() : i-1];
 }// end if
 if(a[i] > a[i-1]) {
  L.push_back(i-1);
  if(i == w+L.front()) L.pop_front();
  while(U.size()>0) {
    if(a[i]<=a[U.back()]) {
     if (i == w+U.front()) U.pop_front();
     break;
    }// end if
    U.pop_back();
  }// end while
 } else {
  U.push_back(i-1);
  if (i == w+U.front()) U.pop_front();
  while(L.size()>0) {
    if(a[i]>=a[L.back()]) {
     if(i == w+L.front()) L.pop_front();
     break;
    }//end if
    L.pop_back();
  }//end while
 }// end if else
}// end for
maxval[a.size()-w] = a[U.size()>0 ? U.front() : a.size()-1];
minval[a.size()-w] = a[L.size()>0 ? L.front() : a.size()-1];
```