One-dimensional Dilation Algorithms

Wim H. Hesselink, whh587
Bernoulli Institute, University of Groningen

May 6, 2022

1 Introduction

Given an input sequence $x(0)$, $x(1)$, ... and a window size W, the dilation or max-filter problem is to compute the outputs $y(n) = \max\{x(i) \mid n - W < i \leq n\}$. Let sequence x have length N. Then sequence y has length $N + W - 1$, because $y(n)$ is undefined (or $-\infty$) for $N + W - 1 \leq n$.

The shortest program is presented in Section 2. It is elegant, but in performance it cannot compete. The oldest other solution is HGW, which is due, independently, to Van Herk [4] and Gil and Werman [3]. It works for an arbitrary associative binary operator $\otimes$. It is treated in Section 3 below. In Section 4, the operator $\otimes$ becomes the max-operator again. Here, we treat three optimizations of HGW that have been proposed.

Section 5 reports on a performance experiment that compares the algorithm of Section 2, the HGW algorithm, and the three optimizations of HGW proposed. For $W \leq 32$, HGW has the best performance. For $64 \leq W \leq 512$, the algorithm of Gil and Kimmel [2] performs best, while the algorithm of Yuan and Atallah [5] wins for $W \geq 1024$. Conclusions are drawn in Section 6.

2 Dilation by a sliding window

The simplest solution that is reasonably efficient, uses a *queue* of indices of length $\leq N$ to hold the so-called right-maximal numbers. A number k is called *right-maximal below* n iff $n - W < k \leq n$ and $0 \leq k < N$, and $x(k) > x(j)$ for all j with $k < j \leq n$ and $j < N$. The loop invariant used is that the sequence $queue(i)$ with $p \leq i < q$ is the decreasing sequence of right-maximal numbers below n.

```
sliding(x, N, W, y) {
    int p := 0, q := 1;
    queue(0) := 0;
    for (n := 0 ; p < q ; n := n + 1) {
        y(n) := x(queue(p));
        if (queue(p) = n + 1 − W) p := p + 1;
        if (n < N − 1) {
            while (p < q  ∧  x(queue(q − 1)) ≤ x(n + 1))
                q := q − 1;
            queue(q) := n + 1;
            q := q + 1;
        }
    }
} .
```

In the code fragments of this note, a C-like syntax is used, except that equality is denoted by = and assignment by :=, and that ordinary parentheses are used for subscription of arrays.

The number of comparisons in *sliding* is bounded by $2N$. Yet, it turns out that *sliding* (even when optimized in various ways) requires about twice the time needed for any of the algorithms treated in Section 4 below.

3 The one-dimensional abstract dilation

As a preparation for max-filters and min-filters, we first discuss filtering a one-dimensional signal by means of an arbitrary associative operator $\otimes$ on some type T, compare [1]. In other words, the pair $(T, \otimes)$ is a semigroup. In the main examples, T is the type of the integers or reals, and $\otimes$ gives the maximum, the minimum, or the sum of two numbers.

If s is a nonempty sequence in T, and $p \le q$ are indices in the domain of s, the repeated product from p to q is

$$\bigotimes(p, q, x) = s(p) \otimes s(p+1) \otimes \ldots \otimes s(q) \ .$$

The order is relevant here, because $\otimes$ is not necessarily commutative. As $\otimes$ is associative, parentheses are unnecessary, and one has the rule

$$(0) \qquad p \le m < q \ \Rightarrow \ \bigotimes(p, q, s) = \bigotimes(p, m, s) \otimes \bigotimes(m+1, q, s) \ .$$

Let x be a sequence in T, with indices i bounded by $0 \le i < N$. The $\otimes$ filter of x with window size $W > 0$ is defined as follows. For completeness, we include the border values which are often ignored (the borders became relevant when we wanted to rotate the image). The result of the filter is the dilation sequence y given for $0 \le n < N + W - 1$, by

$$y(n) = \bigotimes(\max(0, n - W + 1), \min(n, N - 1), x) \ .$$

The central part of the dilation is characterized by

$$W - 1 \le n < N \ \Rightarrow \ y(n) = \bigotimes(n - W + 1, n, x) \ .$$

The dilation can be computed with the following HGW algorithm of Van Herk [4] and Gil and Werman [3]. The idea for efficient computation is to concentrate the computation at the *pivot points*. The pivot points are the positive multiples of W in the domain of x. For a pivot point c, one defines

$$\begin{aligned} &\mathit{Left}(c, k) = \bigotimes(c - 1 - k, c - 1, x) \text{ for } 0 \le k < W - 1 \ , \\ &\mathit{Right}(c, k) = \bigotimes(c, \min(c + k, N - 1), x) \text{ for } 0 \le k < W \ . \end{aligned}$$

Note that the operand $x(c)$ is needed in *Right* because $x(c+1)$ need not be defined.

Computation of the sequences $\mathit{Left}(c)$ and $\mathit{Right}(c)$ requires (at most) $2 \cdot W - 3$ operations of $\otimes$. These sequences determine dilation y from the pivot c to the next one because $y(c + W - 1) = \mathit{Right}(c, W - 1)$ and, by formula (0),

$$(1) \qquad 0 \le n < W - 1 \ \Rightarrow \ y(c + n) = \mathit{Left}(c, W - 2 - n) \otimes \mathit{Right}(c, n) \ .$$

The values of y to the left of the first pivot are determined by

$$0 \le n < W \ \Rightarrow \ y(n) = \bigotimes(0, \min(n, N - 1), x) \ .$$

Let c be the biggest pivot point. Then $c < N \le c + W$ and

$$(2) \qquad c + W \le n < N + W - 1 \ \Rightarrow \ y(n) = \bigotimes(n - W + 1, N - 1, x) \ .$$

If there are no pivot points, i.e., $N \leq W$, this formula is applicable for $c = 0$.

The above algorithm is now summarized in the procedure

```
HGW(x, N, W, y){
    makeRight(x, 0, N, W, y);
    for (c := W ; c < N ; c := c + W) {
        makeLeft(x, c − 1, W − 1, left);
        makeRight(x, c, N, W, right);
        merge(c, W, left, right, y);
    }
    makeTail(x, c, N, W, y)
}
```

where *left* and *right* are auxiliary arrays of lengths $W - 1$ and W, respectively. The auxiliary procedures *makeLeft* and *makeRight* establish $left = Left(c)$ and $right = Right(c)$.

```
makeLeft(x, c, w, left){
    left(0) := x(c);
    for (k := 1 ; k < w ; k := k + 1)
        left(k) := x(c − k) ⊗ left(k − 1);
}

makeRight(x, c, N, W, right){
    right(0) := x(c);
    for (k := 1 ; k < min(W, N − c) ; k := k + 1)
        right(k) := right(k − 1) ⊗ x(c + k);
    for ( ; k < W ; k := k + 1)
        right(k) := right(k − 1);
}
```

Procedures *merge* and *makeTail* apply the formulas (1) and (2), respectively.

```
merge(c, W, left, right, y){
    y(c + W − 1) := right(W − 1);
    for (k := 0 ; k < W − 1 ; k := k + 1)
        y(c + k) := left(W − 2 − k) ⊗ right(k);
}

makeTail(x, c, N, W, y){
    if (c < N + W − 1){
        y(N + W − 2) := x(N − 1);
        for (k := N + W − 3 ; c ≤ k ; k := k − 1)
            y(k) := x(k + 1 − W) ⊗ y(k + 1);
    }
}
```

4 Max filters

Considerable effort has been made to optimize the above program for the case that the operator $\otimes$ is the **max** operator. Three optimizations are considered here.

4.1 Merge by binary search

As the operator $\otimes = \mathbf{max}$, the sequences $Left(c)$ and $Right(c)$ are ascending. In the procedure *merge*, the test $B(k) = (left(W-2-k) < right(k))$ therefore satisfies $B(k) \Rightarrow B(k+1)$. As observed by Gil and Kimmel [2], the smallest k with $B(k)$ can therefore be determined by binary search. Procedure *merge* can thus be replaced by

```
mergeBS(c, W, left, right, y){
    r := −1; s := W − 1;
    while (r + 1 < s) {
        m = (r + s) div 2;    // integer division
        if (left(W − 2 − m) < right(m))   s := m;
        else r = m;
    }
    for (n := 0 ; n ≤ r ; n := n + 1)   y(c + n) := left(W − 2 − n);
    for ( ; n ≤ W ; n := n + 1)   y(c + n) := right(n);
}
```

4.2 Stopping halfway

The other optimizations proposed by Gil and Kimmel [2] and Yuan and Atallah [5] combine the computations of $Right(c)$ and $Left(c+W)$, because the same elements are involved. For simplicity, the other two optimizations are treated only with the precondition $W < N$. Cases with $N \leq W$ can be treated with HGW.

Gil and Kimmel use the observation that, when the computations of $Left(c+W)$ and $Right(c)$ are halfway, all elements at indices i with $c \leq i < c+W$ have been considered. Therefore, one of the two rows can be filled with identical values. The procedures *makeLeft* and *makeRight* are thus combined in

```
makeLeftRightGK(x, c, N, W, left, right){
    h := W div 2;
    left(0) := x(c + W − 1);
    for (k := 1 ; k < h ; k := k + 1)
        left(k) := max(x(c + W − 1 − k), left(k − 1));
    right(0) := x(c);
    for (k := 1 ; k < W − h ; k := k + 1)
        right(k) := max(right(k − 1), x(c + k));
    if (left(h − 1) ≤ right(W − h − 1)) {
        for (k := W − h ; k < W ; k := k + 1)
            right(k) := right(k − 1);
        for (k := h ; k < W − 1 ; k := k + 1)
            left(k) := max(x(c + W − 1 − k), left(k − 1));
    } else {
        for (k := W − h ; k < W ; k := k + 1)
            right(k) := max(right(k − 1), x(c + k));
        for (k := h ; k < W − 1 ; k := k + 1)
            left(k) := left(k − 1);
    }
}
```

The algorithm of Gil and Kimmel therefore needs a third allocated auxiliary array *aux*, to be swapped with *left*.

```
GiKi(x, N, W, y){
    makeLeftRightGK(0, W, aux, y);
    for (c := W ; c + W < N ; c := c + W){
        int *r := aux;
        aux := left ; left := r;
        makeLeftRightGK(c, W, aux, right);
        mergeBS(c, W, left, right, y);
    }
    makeRight(c, N, W, right);
    mergeBS(c, W, aux, right, y);
    makeTail(c + W, N, W, x, y);
} .
```

4.3 Optimal removal of comparisons

Yuan and Atallah [5] have taken the last step in removing superfluous comparisons. They propose an optimization of the procedure *makeLeftRight* in which the arrays *left* and *right* compete to find a maximal value $x(m)$ with $c \le m < c + W$. The paper [5] provides code that makes an index-out-of-bounds error. Figure 4.3 gives a version where this error is eliminated.

The algorithm uses three loops that preserve the loop invariants

$$J1: \quad 0 \le i < W - 1 \ \wedge \ (\forall\, k : 0 \le k \le i \ \Rightarrow \ \mathit{left}(k) = \mathit{Left}(c + W, k)) \ ,$$
$$J2: \quad 0 \le j < W \ \wedge \ (\forall\, k : 0 \le k \le j \ \Rightarrow \ \mathit{right}(k) = \mathit{Right}(c, k)) \ .$$

Indeed, the aim is to establish the postcondition $J1 \wedge J2 \wedge Q$ where

$$Q: \quad (i = W - 2 \ \wedge \ j = W - 1) \ .$$

The invariants are easily initialized by the setting i and j equal to 0 and giving $\mathit{left}(0)$ and $\mathit{right}(0)$ the appropriate values. The algorithm consists of three loops: a main loop followed by two auxiliary ones. The main loop has a body that increments i and builds *left* when $\mathit{left}(i) \le \mathit{right}(j)$, and otherwise increments j and builds *right*. The loop is broken when i should be incremented but has reached $W - 2$, or j should be incremented and has reached $W - 1$. At this point, Algorithm 1 of [5] has an index-out-of-bounds error. Specifically, the program does assignments $i := i + \delta$ and $j := j + \delta$, followed by inspection of $\mathit{left}(i)$ and $\mathit{right}(j)$, without guaranteeing that i and j are in the ranges required.

The algorithm uses a parameter s to indicate that the main loop body extends its array with at most s elements before the next comparison between $\mathit{left}(i)$ and $\mathit{right}(j)$. According to the paper [5], the optimal value for this parameter is $s = \lceil \sqrt{W - 1} \rceil$. The main loop terminates because every execution of its body terminates with a **break** statement or increases $i + j$ to reach the bound $W + s - 2$. After this loop, the remaining elements of *left* and *right* are obtained by copying previous values in two subsequent loops.

It is clear that the main loop preserves the predicates $J1$ and $J2$. Therefore, the predicates hold in the postcondition of the main loop.

This postcondition must also ensure that the two auxiliary loops preserve the invariants $J1$ and $J2$. For this purpose, the main loop gets three more invariants:

$$J3: \quad \forall\, k : c + W + s - 1 - i \le k \le c + W - 1 \ \Rightarrow \ x(k) \le \mathit{right}(j) \ ,$$
$$J4: \quad \forall\, k : c \le k \le c + j - s \ \Rightarrow \ x(k) \le \mathit{left}(i) \ ,$$
$$J5: \quad i + j \le W + s - 2.$$

These predicates hold initially because $i = j = 0$ and $s > 0$. Predicate $J3$ is threatened only when i is incremented. This only happens in the first inner loop. This first inner loop has the precondition $\mathit{left}(i) \le \mathit{right}(j)$, or equivalently

```
makeLeftRightYA(x, c, N, W, left, right, s) {
    left(0) := x(c + W − 1);
    right(0) := x(c);
    i := j := 0;
    while (i + j < W + s − 2) {
        δ := min(s, W + s − 2 − i − j);
        if (left(i) ≤ right(j)) {
            if (i = W − 2) break;
            upb := min(i + δ, W − 2);
            for ( ; i < upb ; i := i + 1)
                left(i + 1) := max(x(c + W − 2 − i), left(i));
        } else {
            if (j = W − 1) break;
            upb := min(j + δ, W − 1);
            for ( ; j < upb ; j := j + 1)
                right(j + 1) := max(right(j), x(c + j + 1));
        }
    }
    for ( ; i < W − 2 ; i := i + 1) left(i + 1) := left(i);
    for ( ; j < W − 1 ; j := j + 1) right(j + 1) := right(j);
} .
```

Figure 1: An improved version of the algorithm of Yuan and Atallah

$$\forall\, k : c + W - 1 - i \le k \le c + W - 1 \;\Rightarrow\; x(k) \le \mathit{right}(j)\ .$$

The inner loop increments i with at most s, and therefore preserves $J3$. An analogous argument proves preservation of $J4$ (by the second inner loop). Preservation of $J5$ is obvious.

Let M be the maximal value $x(k)$ for the interval $c < k < c + W$, and let m be an index in this interval with $x(m) = M$. We claim that the main loop establishes the postcondition

$$Q1: \qquad M \le \mathit{left}(i) \;\wedge\; M \le \mathit{right}(j)\ .$$

Indeed, at the first **break** statement, predicate $J1$ implies $M = \mathit{left}(i) \le \mathit{right}(j)$. At the second **break** statement, predicate $J2$ implies $M \le \mathit{right}(j) \le \mathit{left}(i)$. It therefore remains to show that ordinary termination of the main loop establishes $Q1$. Because of $J5$, the postcondition of ordinary termination satisfies $i + j = W + s - 2$. Assume that $\mathit{left}(i) < M$. Then $J4$ implies that

$$\forall\, k :\; c \le k \le c + j - s \;\Rightarrow\; x(k) < M\ .$$

It follows that $c + j - s + 1 \le m$ and hence $c + W - 1 - i \le m \le c + W - 1$. Using $J1$, we get $\mathit{left}(i) = M$, a contradiction. Similarly, assume $\mathit{right}(j) < M$. Then $J3$ gives

$$\forall\, k :\; c + W + s - 1 - i \le k \le c + W - 1 \;\Rightarrow\; x(k) < M\ ,$$

and hence $c < m \le c + W + s - 2 - i = c + j$, implying $M \le \mathit{right}(j)$, a contradiction. This proves that the main loop establishes the postcondition $Q1$.

Predicate $Q1$ clearly implies

$$(\forall\, k : c < k < c + W \;\Rightarrow\; x(k) \le \mathit{left}(i) \;\wedge\; x(k) \le \mathit{right}(j))\ .$$

This ensures that the two auxiliary loops preserve the predicates $J1$ and $J2$. This concludes the proof of the algorithm.

window	sliding	slidingOpt	HGW	HGWbs	Gil-Kim...	Yuan-At...
1	∗ 598	964	1580	1779	2027	1591
2	1591	∗ 1572	1667	2151	2434	2396
4	1974	∗ 1838	1877	2353	2404	2701
8	2415	2205	∗ 1872	2280	2171	2437
16	2479	2240	∗ 1693	1915	1908	2114
32	2478	2232	∗ 1577	1681	1628	1775
64	2471	2219	1427	1438	∗ 1414	1490
128	2470	2216	1339	1281	∗ 1255	1279
256	2466	2210	1283	1183	∗ 1151	1198
512	2467	2210	1255	1128	∗ 1088	1091
1024	2474	2208	1235	1092	1048	∗ 1045
2048	2459	2213	1222	1072	1027	∗ 1000
4096	2456	2219	1227	1069	1021	∗ 994
8192	2467	2232	1237	1077	1029	∗ 988

Figure 2: Timings in microseconds for 100000 items

5 Experiments

Some experiments were performed to compare the timing performance of the algorithms presented. The experiments were primarily set up to affirm the correctness of the output, by comparing it with the ground truth. This verification was included in all later experiments, outside of the timing sessions.

The timing experiment was done with a random input sequence of $N = 100000$ numbers. The window size W ranges over the powers of 2 from 1 to 8192. In each case, the shortest time of 100 runs is recorded, in microseconds. The results are shown in Figure 5. The first column gives the window size. The next column gives the timings for the algorithm of Section 2. The third column adds all optimizations we could invent. The other four columns correspond to HGW with the max operator, and the three optimizations presented in the sections 4.1, 4.2, 4.3, respectively.

For each window size, the table indicates the best timing by means of a $*$. It shows that HGW has the best timings for $8 \leq W \leq 32$. For $64 \leq W \leq 512$, the algorithm of Gil-Kimel performs best, while Yuan-Atallah wins when $W \geq 1024$. The paper of Yuan and Atallah [5] reports that their algorithm wins when $W \geq 33$. They have a very careful description of the experiment, with an input sequence of size 10^7. As their program has an index-out-of-bounds error, however, we are not convinced of reliability.

Our programs were programmed in C. They were executed on a MacBook Air with a 1.8 GHz Dual-Core Intel Core i5 (of mid 2012). They seem to perform 5 times as fast as those of [5].

6 Conclusions

It is a pity that the sliding window solution of Section 2 cannot compete in performance with the other solutions treated. Its optimized version has been included in Table 5 to indicate the scope for optimization. The corresponding program is too ugly to present.

The operator max is idempotent, i.e., $\max(x, x) = x$. This has been used in the classical papers [3, 2, 5] to treat *left* and *right* symmetrically. The symmetry induces a minor but unnecessary inefficiency, which we avoided in the treatment of these algorithms in Section 4.

The paper of Yuan and Atallah reports that their algorithm is optimal for $W \geq 33$. In our experiment the threshold is around 700. We don't know if the difference is related to the out-of-bounds error we found in their program, or to other differences in program, language, or compiler.

The parameter value $s = \lceil \sqrt{W-1} \rceil$ for the algorithm of Yuan and Atallah is chosen to minimize the number comparisons, but it is not always optimal for timing performance. For instance, for $250 < W < 1000$, the program of Section 4.3 with $s = \lceil W/4 \rceil$ has better timings than both Gil-Kimmel and Yuan-Atallah. The differences, however, are marginal.

References

[1] D. Z. Gevorkian, J. T. Astola, and S. M. Atourian. Improving Gil-Werman algorithm for running min and max filters. *IEEE Trans. Pattern Anal. Machine Intell.*, 19:526–529, 1997.

[2] J. Gil and R. Kimmel. Efficient dilation, erosian, opening, and closing algorithms. *IEEE Trans. Pattern Anal. Machine Intell.*, 24:1606–1617, 2002.

[3] J. Gil and M. Werman. Computing 2-D min, median, and max filters. *IEEE Trans. Pattern Anal. Machine Intell.*, 15:504–507, 1993.

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

[5] Hao Yuan and M. J. Atallah. Running Max/Min filters using 1 + o(1) comparisons per sample. *IEEE Trans. Pattern Anal. Machine Intell.*, 33:2544–2548, 2011.