A Fast Algorithm for Running Computation of Center Weighted Rank Order Filters

Bogdan P. Dobrin, T. George Campbell and Moncef Gabbouj

Signal Processing Laboratory
Tampere University of Technology
P.O.Box 553, SF-33101 Tampere, Finland

Abstract-This paper presents a fast center weighted rank order filtering algorithm with logarithmic worst case time complexity with respect to the window size. The method is based on a data structure, called an unbalanced "double heap" which naturally supports partition of the window samples into two different "heaps", greater and smaller than the *r*th sample. The algorithm is designed for any window size and any weight of the center point and needs linear memory dependent only on the window width.There are no restrictions here with respect to the window size, which may be any odd or even number.

1. INTRODUCTION

One of the most popular classes of nonlinear filters is the class of rank order-based filters. These filters perform well in many situations where linear filters fail. Various types of rank order-based filters have been proposed during the last decade and these filters usually fall into one of two categories: hybrid order statistic filters and generalized stack filters. A natural extension of rank order filters are weighted rank order filters which have greater flexibility in specification than simple rank order filters.

A very well known particular rank order filter is, in fact, the median filter (which has received much attention lately and its theoretical properties are rather well understood [1] [2]), and its natural extension, weighted median [3], is also a particular case of weighted rank order filter. The center weighted Median filter is of interest because it is relatively gentle filter and can be made idempotent [4].

The idea of a rank order filter is to slide a window, usually of length n=(2k+1), over the data values. At each point, the output is the rth largest element in the window. These two integers n and r are the parameters which define the rank order filter. The weighted rank order filter is defined by taking the rth largest value after duplicating each sample from the window with a certain integer number, called *weight*, generally dependent of the position of the sample inside the sliding window.

The traditional method of finding the rth largest sample is to sort the samples of the entire window each time a new sample enters the window. This method, however, ignores the high correlation of the ordering between the two consecutive windows and requires $O(n \log n)$ operations. There are also methods for finding the rth order statistic on a set of n numbers without actually sorting them. The theoretical limitation of this approach have been shown to be $O(n)$ [5]. In [6], Astola and Campbell introduced an algorithm for the running median which operates in $O(\log n)$ time which can be extended to arbitrary rank order filters.

In this paper we present the extension of this algorithm to an arbitrary rank. In addition we show how a weight can be introduced to compute a single weighted rank order filter. Because this weight is usually applied to the center sample this filter is often referred to as a center weighted rank order filter.

We first describe the data structure used by the main part of the algorithm, followed by the description of the algorithm itself.

2. THE DATA STRUCTURE

The initialization part of our algorithm builds the data structure for the specific input parameters n, w and t, which are integer with t ranging from 1 (for the minimum largest sample in the window) to $n+w-1$ (for the maximum largest sample).

We define a data structure called *unbalanced double heap* whose behavior is very suited for partitioning the samples from the running window [7]. This double heap consists of two eventually incomplete binary trees T1 and T2 which have common root and satisfy the following properties:

- T1 and T2 are balanced binary trees, i.e. any two

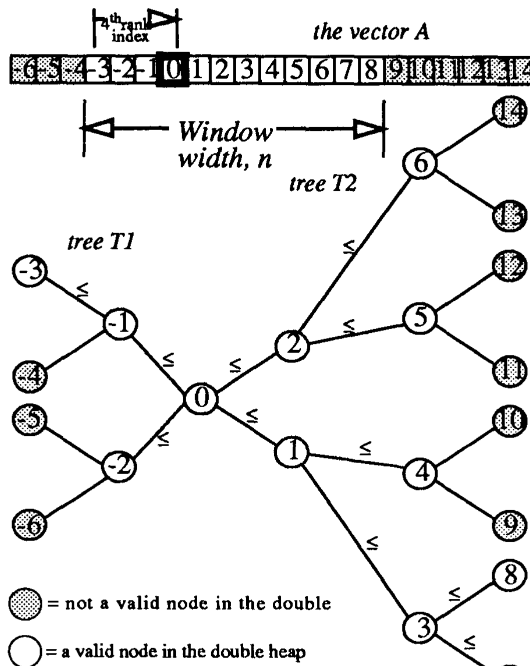

Figure 1. The corresponding addresses in the vector A, for every node in the double heap.

subtrees which contain the root and at least one leaf node have the same height or differ only by one unit. It follows that T1 and T2 might not be completely filled on the deepest level, but the others levels are fully occupied (each node is, in fact, a sample from the running window).

- In T1, no node has a value which is less than either of its children.
- In T2, no node has a value which is greater than either of its children.

It is obviously that the common root is the rth largest node if and only if T1 has r nodes in it (counting the root node). An example of incomplete unbalanced double heap can be seen in Figure 1. It follows that such a double heap structure is completely specified by two parameters: n, the window size and r, the rank of the common root.

Let us call this structure ***rof{n, r}***.

As shown in [6] such a structure can be implemented in a vector A whose size is exactly the window size n, and the zero index element, being the common root. All the elements from the double heap are stored in this array and must satisfy

$$A[i] \le A\left[-\left\lfloor \frac{-i-1}{2} \right\rfloor\right] \quad \text{for } lowerbound \le i \le -1 \quad (1)$$

$$A[i] \ge A\left[\left\lfloor \frac{i-1}{2} \right\rfloor\right] \quad \text{for } 1 \le i \le upperbound \quad (2)$$

Where *lowerbound* and *upperbound* satisfy the following equations

$lowerbound = -(r-1)$ and

$upperbound = n-r$.

It follows that A[0] is the rth largest element because it is greater than (or equal to) others r-1 elements and in the same time is less than (or equal to) others $(n-r)$ elements.

If we replace an element of the double heap with a new one which violates the rules (1) or (2). By repeatedly exchanging this element, as shown in [6], with a suitable adjacent element, we can restore the double heap property.

In addition to this, we must maintain a First In First Out doubly linked list whose state at any moment reflects the positions of all members in the double heap and in the same time preserves the order in which the members enter and leave the filter window.

3. THE ALGORITHM

We will present here a general algorithm for computing center weighted rank order filter, without restriction on the number of samples (n), in the running window, the overall filter rank (t), or the weight of the central point (w).

These parameters should satisfy the following:

n, t, w are strictly positive integers with n>1,

w>1 (for center weighted rank order filter) and $1 \le t \le (n+w-1)$. If w=1 we have a not-weighted rank order filter.

The data structure, required by the algorithm is completely specified by two formal parameters:

n, the window size and r, the rank. Let us denote such a data structure by ***rof{n, r}***. This structure is generated and initialized by the following:

function *Generate(rof{n,r})* **returning** the vector corresponding to one double heap and initializes all its n elements to a constant value (the background, for example).

We describe the functions and procedures used to maintain the double heap and finally the general algorithm.

procedure *Update*(A) **returns** the vector A in a correct state after inserting the *new_sample* and also performs the operations required for the double linked list, to maintain the order of the nodes as they enter and leave the filter window. The index i is a variable which initial refers the oldest value currently in the double heap.

We use the following functions to address adjacent nodes of A[i], in the double heap, used by *Update*(A).

Table 1.

```
procedure Update(A)
begin
1. A[i] is replaced by new_sample
2. while i<0 and A[i]>A[Child(i)] do Swap(i,Child(i))
3. while i>0 and A[i]<A[Parent(i)] do
                Swap(i,Parent(i))
4. while ( i<0 or i=0 ) and Exists(Mother(i))
    and A[i]<A[IndOfLarger(Father(i),Mother(i))] do
      Swap(i,IndOfLarger(Father(i),Mother(i)))
5. while ( i>0 or i=0 ) and Exists(Daughter(i))
    and A[i]>A[IndOfSmaller(Son(i),Daughter(i))] do
      Swap(i,IndOfSmaller(Son(i),Daughter(i)))
6. while ( i<0 or i=0 ) and Exists(Father(i))
      and A[i]<A[Father(i)] do Swap(i,Father(i))
7. while ( i>0 or i=0 ) and Exists(Son(i))
      and A[i]>A[Son(i)] do Swap(i,Son(i))
end.
```

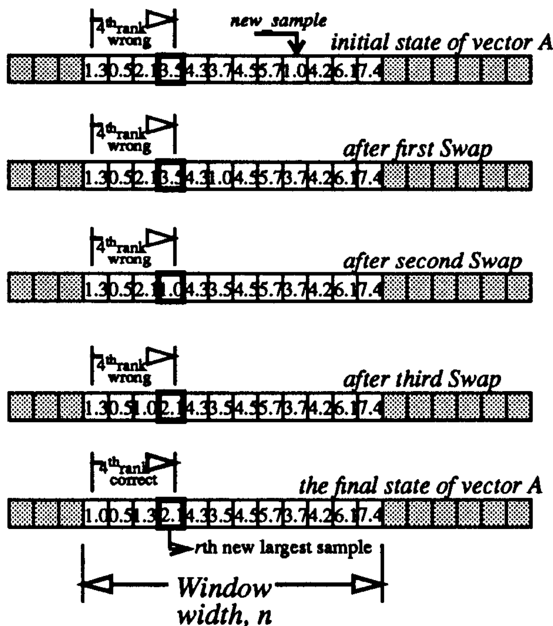

Figure 3. The steps performed by *Update* procedure.

When $i \le 0$ we will use
function *Father(i)* **returns** the value *2i-1*,
function *Mother(i)* **returns** the value *2i-2*
and for $i < 0$ we will use
function *Child(i)* **returns** the value $-\left\lfloor \frac{-i-1}{2} \right\rfloor$.
When $i \ge 0$ we will use
function *Son(i)* **returns** the value *2i+1*,
function *Daughter(i)* **returns** the value *2i+2*
and for $i > 0$ we will use

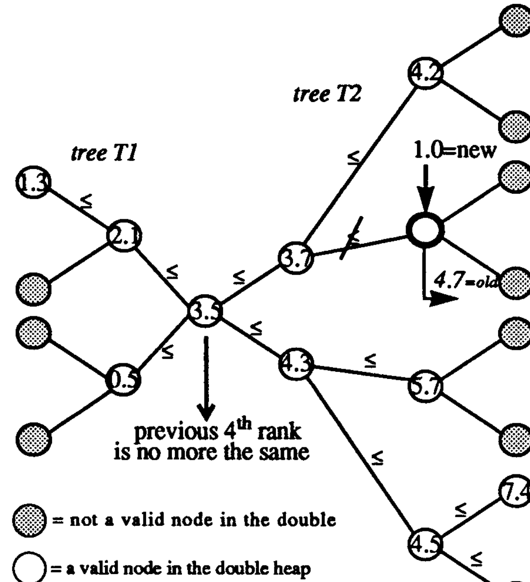

Figure 2. The initial state of the double heap.

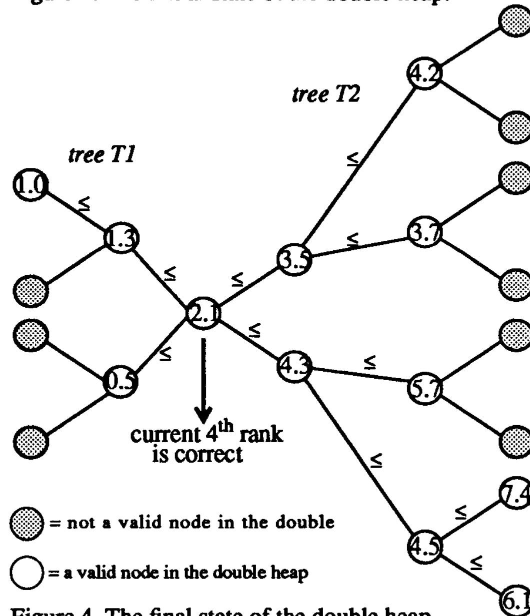

Figure 4. The final state of the double heap.

function *Parent(i)* **returns** the value $\left\lfloor \frac{i-1}{2} \right\rfloor$.

We use for *Update*(A), the following set of functions:
function *Exists(i)* **returns** the boolean value *true* if the index *i* is between the *lowerbound* and the *upperbound*, otherwise *false*.
function *IndexOfLarger(i,j)* **returns** the index of the larger of the nodes A[i] and A[j] and similar
function *IndexOfSmaller(i,j)* **returns** the index of the smaller of the nodes A[i] and A[j]. And finally,

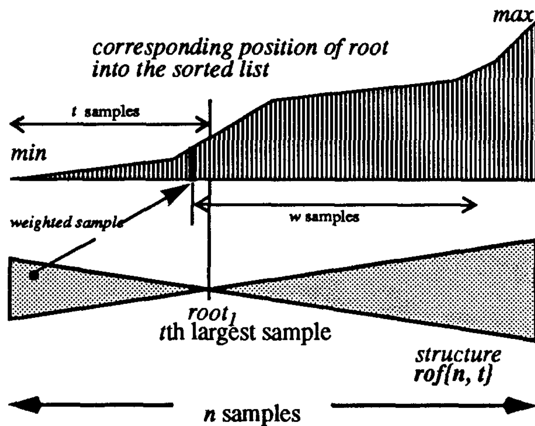

Figure 5. The case (c1) when $t < min(n, w)$

function *Swap(i,j)* **returns** the value j into the variable i and also exchanges the values of the nodes A[i] and A[j] in vector A and makes the corresponding changes into the doubly referred list, preserving the order of the members as they enter and leave the filter window.

The *Update* procedure is defined in Table1, where i is, initial, the index of the oldest sample in the window. The center sample value is denoted by c. The algorithm that compute the single weighted running rank order filter is as follows.

1. **Read** n, t, w
2. (c0) **if** w=1 **then** B=*Generate(**rof[n, t]**)*.
   (c1) **else if** t<min(n,w) **then** A=*Generate(**rof[n, t]**)*.
   (c2) **else if** t>max(n,w) **then**
   A=*Generate(**rof[n, t-w+1]**)*.
   (c3) **else if** w<n **then**
   A_1=*Generate(**rof[n, t]**)*.
   A_2=*Generate(**rof[n, t-w+1]**)*.
   (c4) **else** don't generate data structure.
3. **case** situation **of**
   (c0) : *Update(*B*)*; *output_sample*=B[0].
   (c1) : *Update(*A*)*; *output_sample*=min(c,A[0]).
   (c2) : *Update(*A*)*; *output_sample*=max(c,A[0]).
   (c3) : *Update(*A_1*)*; *Update(*A_2*)*;
   output_sample=min(A_1[0],max(c,A_2[0])).
   (c4) : *output_sample*=c. {no computation required}
4. **while** not *EndOfFile* **do** step 3.

When a new sample enters the window (and replaces the oldest one), we just pick up the common root sample (which is the tth largest sample in the case (c0) and (c1), $(t\text{-}w\text{+}1)$st larger sample in the case (c2) and so on) and compare it with the value of the center sample, c.

We should notice that in the third case we have two

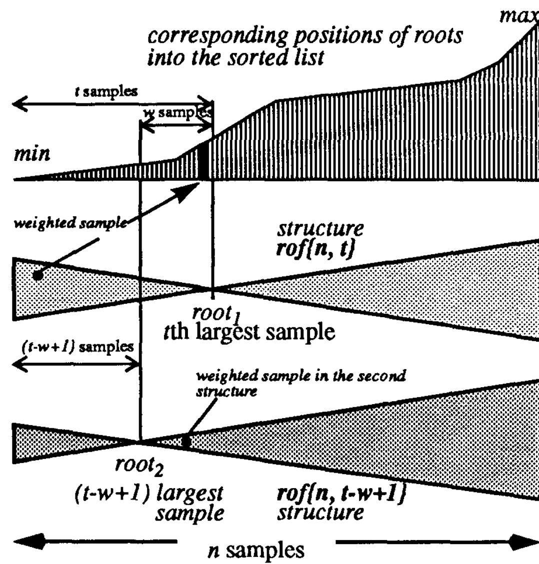

Figure 6. The case (c3) when $min(n, w) \le t \le max(n, w)$

double heaps performing in parallel, which find the tth order statistic and the $(t\text{-}w\text{+}1)$st order statistic from the current window, simultaneously. Obviously $root_1$=A_1[0] is greater than or equal to $root_2$=A_2[0] because the threshold t corresponding to A_1 is greater than the threshold $t\text{-}w\text{+}1$ corresponding to A_2.

The time required for updating the two double heaps is still logarithmic with respect to window size n. The memory requirements for this case are still linearly dependent on n (it is just twice that for a normal double heap).

4. DISCUSSION

We will compute the number of comparisons required, in the worst case, for maintaining the ***rof[n,r]*** data structure.

The worst case is encountered when the new element, which enters the window, replaces a leaf node and wonders to another leaf node, through the common root.

Without lose of generality, we can assume $l_1 \le l_2$, where l_1 is the height of the tree T1 and l_2 the height of the tree T2 (counting also the root as a valid level). Then the worst case requires $(l_1 + l_2 - 2)$ calls of *Swap* and $(l_1 + 2 \cdot l_2 - 3)$ comparisons.

Because the number of levels of a binary tree is logarithmic on the number of nodes then all the above equations exhibit a logarithmic dependency on the numbers of samples in the double heap, i.e. $O(\log n)$, Thus for the first rank order filter, i.e. the minimum or-

der statistic, we have in the worst case

$3\lfloor \log_2 n \rfloor$ comparisons. (3)

For the $(k+1)$ st order statistic (assuming that n is an odd number $n = (2k+1)$), we obtain the median filter which requires the most time of all the rank order filters of the same window width. It requires

$3\lfloor \log_2(n+1) - 1 \rfloor$ comparisons, (4)

which is always greater than (3) for large n.

Any other rank order filter requires, in the worst case, a number of operations between these two extreme cases, showed above.

Finally, for the center weighted rank order filter the worst case is the (c3) situation and the computation required under the most pessimistic assumption is twice as stated above plus a constant overhead (because we have two double heaps to maintain, each of them consisting of exactly n nodes, and the parameter r doesn't change the order of complexity). So the order of complexity, required for any center weighted rank order filter using this algorithm is $O(\log n)$ and is not dependent on the weight, w of the center sample from the running window or the filter rank, t.

5. CONCLUSIONS

In this paper we showed an algorithm for fast computation of center weighted rank order filter. This filter could be of practical value when implementing impulsive noise removal on critical signals, where a long filter window is required. The results indicates that there may be a possibility to implement a general weighted median filter with some further study.

6. REFERENCES

[1] J.W. Tukey, "Nonlinear (nonsuperposable) methods for smoothing data" in *Conf.Rec.,1974 EASCON*, p. 673.

[2] N.C. Gallagher, Jr. and G.L. Wise, "A theoretical analysis of median filters" *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-29, pp. 1136-1141, Dec. 1981.

[3] B.I. Justusson "Noise Reduction by Median Filtering" Proc. 4th Int. Joint Conf. Pattern Recognition, Kyoto, Japan, pp. 502-504, Nov. 1978.

[4] P. Haavisto, M. Gabbouj, and Y. Neuvo, "Median based Idempotent filters," Journal of Circ.,Syst., and Comp., vol. 1,no.2, pp. 125-148, June 1991.

[5] D.E. Knuth, *Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.

[6] Jaakko T.Astola and T.George Campbell, "On Computation of the Running Median" *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-37,No.4, pp. 572-574, April 1989.

[7] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.