International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2020): 7.803

A Very Efficient Algorithm for Calculating the Max and Min from Running Data

Rushabh Sharadchandra Sankalcar

Under the Guidance of Nayana Shenvi
Associate Professor, ETC Department, Goa College of Engineering and Mr. Rupesh Porob

Abstract: ***I present an algorithm for calculating the running maximum and minimum value of a one-dimensional sequence over a sliding window. The above-mentioned algorithm stores an ordered list of data elements which may have the potential to become maxima or minima across the data window at some future time instant. This algorithm has a number of advantages over competing algorithms like reducing processing time and storage requirements for long data windows. It also finds the minimum and maximum at the same time with minimum number of comparisons. It is observed that by using this new algorithm the number of comparisons reduces drastically in finding maximum and minimum value of a one-dimensional sequence.***

Keywords: Running sequence, maximum, minimum, effective algorithm, MINMAX, MAXLIST, comparisons

1. Introduction

Computing the running maximum and minimum over a sliding data window is useful in many signals and image processing tasks. For example, in morphological signal processing running maximum and minimum calculation is used [1]. In image processing, filtering is a main application. The problem of running max/min calculation is a subset of the general task of determining the ranked order of a sequence across a sliding data window [2, 3, 4…]. Running max/min calculation is much simpler because i) only one output needs to be calculated per sample time and ii) the maximum or minimum value appears at either end of a running sorted list. Existing approaches for calculating the running maximum/minimum include three algorithms. The first method maintains a sorted list of L numbers and updates this list using a dividend-conquer strategy; this algorithm requires a maximum of approximately ($\log_2$L) comparisons and L storage locations at each time instant [2]. The second method is MAXLINE algorithm; it compares the maximum value across the data window at the previous time instant, the old data value leaving the window, and the new data value entering the window. The MAXLINE algorithm is extremely efficient for uniformly-distributed signals. However, for these signals, the MAXLINE algorithm performs L + 1 comparison in a single time step on an average of once every L+1time samples [2]. The third method, termed the MAXLIST algorithm, uses the fact that only a small portion of a running sorted list needs to be maintained to retain the true maximum or minimum value across a sliding data window [3]. By incorporating both sorting and timing information, a pruned list of ordered data values is created and maintained. Although the length of this list is often much shorter than the length of the data window, but due to high insertion and deletion link-list are used to store this list as they have minimum insertion and deletion time. But in link list each data requires two memory locations. So, if the length of the link-list is L then the memory space required will be 2L which causes a problem in devices which have limited memory space. To avoid such problems, I have proposed an effective algorithm named MAXMIN which not only reduces the time and the number of comparisons but also uses limited number of memory space. By the use of circular array, we will require only one memory space per data. Furthermore the number of comparisons can be reduced if we have as idea from where the comparison of the data should begin i.e., from the front or rear end. This is also taken care by the proposed algorithm. Also, by finding the max and min and the same time we can also reduce the number of comparisons.

2. The Algorithm

Consider Figure 1 to understand the operation of the algorithm. It shows a data window of length L = 10 acting upon a data sequence at five consecutive time instants. The numbers inside the boxes indicate data positions, and the arrows above the boxes indicate the values of the data. At the first time instant, the maximum is located at position #1, and the second-largest datum is located at position #4 and the third at #5. However, since the running data window causes data at the leftmost edge of the window to exit first, the sample at position #2 and #3 will never be the maximum across the window as, when the data at #1 exits the window the sample at position #4will be the largest which will exit the window after position #2 and #3. Thus, we construct a list which consists of the potential maxima of the window at future time instants, as shown will consists of #1, #4 and #5. Similar will be in the case of finding the minimum. The minimum list will consist of data at position at #2, #3 and #5. The rest will not be included as they don't have the potential to be maximum at any given time. Note that the last most elements had the potential to be maximum and minimum at the same time so both the list have the last most data included in them.

To update this list, consider the window at time t = 1. Since only the items within the list can become maxima or minimum, we compare the data with only the list we have made. The list to be compared will be based on the value of the data at position #6. If it lies between the max list, we will compare with that if not with the min list. But the #6 will be added to both the list. The list in which we don't have to compare we just add the data at the end of that list. So, in this we can c the data at #6 lies between the min list so we compare it with that list only. From where to begin the com-

parison will depend whether the difference between the front or rear value if smaller. Whichever value is smaller the comparison will happen from that end to reduce the number of comparisons. So, in the minimum list #6 has the potential to be minimum but now #3 can never be minimum so the undated list will have only #2 and #6.

As in the case of maximum list #6 is added at the end as it has the potential to be maximum but not before #4 and #5. So, the maximum list will now have #4, #5 and #6 as #1 has left the window making #4 the maximum.

For t=2 data at position #7 falls between the min list. So, we just add it at the end. But in the case of minimum list, it is the smallest element in the list as #2 has left the window so the list will have only #7.

For t=3 data at position #8 falls between the max list. So, depending of which difference is list we start comparing from that end we get can eliminate #6 and #7 as they will never be max as #8 has entered the window. But in the case of minimum list #8 will just be added at the end.

For t=4 data at position #9 falls between the max list. So, by comparing we can eliminate #8 as it will never be max and add #9 to the list. But in the case of minimum list #9 will just be added at the end.

For t=5 data at position #10 doesn't fall between the max list so we just add it at the end. But in the case of min list if we comparing, we can eliminate #9 as it will never be min as #10 is added to the list, so not min list will only have #7, #8 and #10.

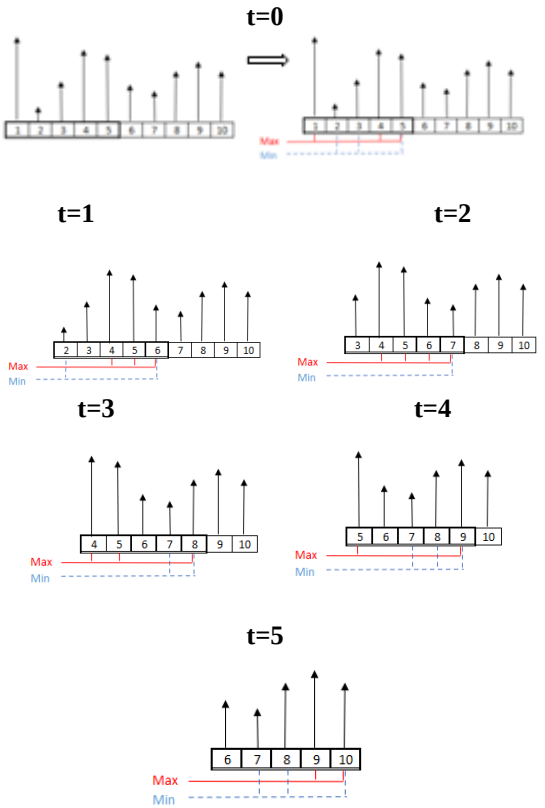

Figure 1: An example illustrating the MINMAX algorithm

3. Analysis

We now determine the computational complexity and memory requirements of the MAXMIN algorithm by just analysing the outputs. In MAXLIST algorithm at a time we can calculate min or max. Consider a time instance T in which the length of the pruned max and min list is L_{max} and L_{min} respectively. So, the number of comparisons for worse case to find max and min will be L_{max}-1 andL_{min}-1. Adding both we will have (L_{max}+L_{min}-2) comparisons to find the min and max at a time instance T.

Considering the same example for MINMAX algorithm, depending on which rang the incoming new data falls into we have to compare it with that list only. So, in this case if we observe number of comparisons will either be L_{max}-1 or L_{min}-1 depending on which range the new incoming data falls into. This reduces the number of comparisons for worst case scenario by L_{max}-1 or L_{min}-1.

Although the pruned list id much smaller than the actual window we still have to compare all the elements in it. So, the number of comparisons may be L_{max}-1 or L_{min}-1 depending on which category it falls, as explained above. This comparison according to experimental theses [4] should be from back to front. Thus, reducing the number of compressions. Although this is very effective in most cases but not all.

To avoid this, I have also proposed an algorithm in which before stating the comparison from back we have to compare the highest and lowest value from the table i.e., the extreme front and extreme rear. Depending on which difference is smaller we will begin the comparison from there hence reducing the number of comparisons.

Doing this will further reduce the execution time making the MAXMIN algorithm more effective. The amount of memory required to store both these max and min list will be equal to the window size L each, as we are using circular array. If we had used link list for the same, each data used two memory locations and in worst case if L=L_{max}= L_{min} then the memory locations required in total will be 4L which will be twice the location required in the MAXMIN algorithm. In MINMAX algorithm the required locations will be the same i.e., 2L for all cases.

4. Results

We have compared the data of MAXLINE and MAXMIN algorithm. These results show a clear-cut idea in the number of comparisons required for both the algorithms hence giving us the time requires for the execution when the data and the results are identical.

Figure 2 show a compression of data of MAXMIN and MINLIST when we calculate the min and max of the given data. The data has a window size of 30 showing the comparisons required by both for time instance of T=15.

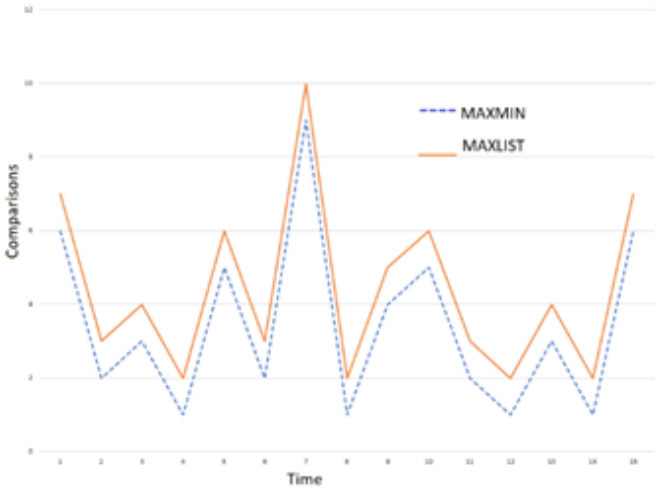

Figure 2: Comparison of MINLIST and MINMAX with a data of window size 30 over a period of 15 insertions

From Figure 2 we can observe that the total number of comparisons to find the min and max in the given data is always less for MINMIX as compared to the MINLIST. This behaviour is observed even when we increase the window size.

Figure 3 shows the average number of comparisons for MINMAX and MINLIST for data with different window size.

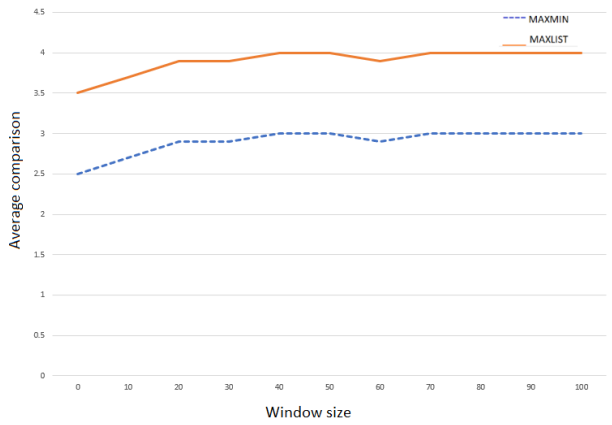

Figure 3: Average comparison of MINLIST and MINMAX with a data of varying window size

We can observe that if we want to find the minimum and maximum of a give data, by using the MINLIST algorithm on an average we require 4 comparisons but by using the MINMAX algorithm we require an average of 3 comparisons.

Reduction in comparisons lead to reduction of processing time hence increasing the time efficiency of the system in which the algorithm is deployed.

Furthermore we can reduce the number of comparisons by figuring out from which side of the min and max array we should start comparison.

In MINLIST algorithm it is shown that by comparing from the back it reduces the number of comparisons but it's not always true. In cases where the incoming data is the new maximum or minimum in such cases if we compare from the back, we have to traves through the entire array which won't be efficient. This problem is taken care by the MINMAX algorithm.

Figure 4 shows the number of comparisons required to find the minimum/maximum of a data with window size 30 for a period of T=15 for both the algorithms.

We can observe that on an average, the number of comparisons taken by the MAXMIN algorithm is less compared to the MINLIST algorithm.

To get a better comparison we have taken an average of the comparisons taken by the two algorithms when the window size is varied.

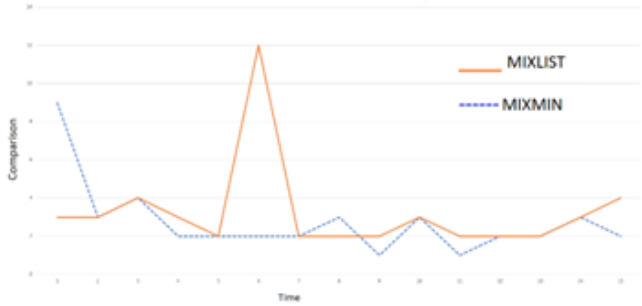

Figure 4: Comparison of MINLIST and MINMAX with a data of window size 30 over a period of 15 insertions

To get a better analysis we have taken an average of the comparisons taken by the two algorithms when the window size is varied.
Figure 5 shows the average number of comparisons for MINMAX and MINLIST for data with different window size.

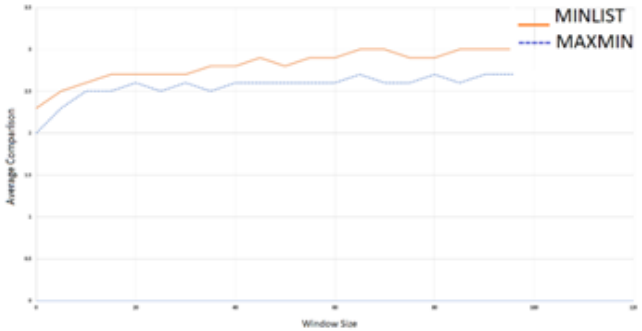

Figure 5: Average comparison of MINLIST and MINMAX with varying window size

As we can observe the average number of comparisons for MINLIST is 3 where as for the MINMAX algorithm its around 2.6. This shows that by using the MINMAX algorithm we can reduce the execution time.

5. Conclusion

We can observe from the results that the number of comparisons for the MINMAX algorithm is less as compared to the MAXLIST algorithm hence reducing the execution time. Also, the space required to store the pruned list is equal to the window size in all cases for MINMAX but in the case of

MAXLIST the worst case can take a space twice of the window size. So, in conclusion we can say that in the case where the space is limited and execution time needs to be effective, MINMAX is a suitable algorithm.

References

[1] J. P. Serra, Image Analysis and Mathematical Morphology, Academic Press.
[2] I. Pitas, "Algorithms for running and max/min calculation,". IEEE Trans. Circuits Syst.
[3] S. C. Douglas, "ALGORITHM FOR RUNNING MAX/MIN CALCULATION", Department of Electrical Engineering. University of Utah Salt Lake City.
[4] Chris J. Myers and Hao Zheng, "ASYN-CHRONOUS IMPLEMENTATION OF MAXLIST ALGORITHM" Electrical Engineering Department. University of Utah Salt Lake City
[5] T.S. Huang, G.J. Yang, and G.Y. Tang, "Two-dimensional median filtering algorithm,". IEEE Trans. Acoust., Speech, Signal Processing.
[6] L.E. Lucke and K.K. Parhi. "Parallel pro-cessing for rank order and stack filters,". IEEE Trans. Signal Processing.
[7] S.C. Douglas, "Running max/min calculation by pruned list,". IEEE Trans. Signal Pro-cessing.
[8] S.C. Douglas, "Analysis and implementation of max-NLMS filter," Conf. on Signals, Systems, and Computers, Pacific Grove.
[9] I.S. Gradshteyn and M. Ryzhik, Table of Integrals, Series, and Products.