Algorithms for Fast Computation of Matrix Profiles of Time Series Under Unnormalized Euclidean Distances
Zhang, Jing; Nikovski, Daniel N.
TR2022-040 May 19, 2022

Abstract
We propose a novel algorithm to compute the Matrix Profile (MP) under the unnormalized distance (useful for value-based similarity search) using two suitable data structures (double-ended queue and segment tree). The algorithm has the same time/space complexity as the state-of-the-art algorithm for typical MP computation under the normalized '2 distance (useful for shape-based similarity search). We validate its efficiency and effectiveness through extensive numerical experiments and two real-world applications – anomaly detection in the NYC taxi data, as well as in a simulated communication network. Depending on data and applications, we show that the value-based similarity search could perform almost equally well as the shape-based similarity search, and sometimes outperform the latter significantly. Index Terms—Matrix Profile, unnormalized Euclidean distance, double-ended queue, segment tree, anomaly detection

International Conference on Applied Statistics and Data Analytics 2022
Algorithms for Fast Computation of Matrix Profiles of Time Series Under Unnormalized Euclidean Distances

Jing Zhang and Daniel Nikovski
Mitsubishi Electric Research Labs
{jingzhang, nikovski}@merl.com

Abstract—We propose a novel algorithm to compute the Matrix Profile (MP) under the unnormalized $\ell_\infty$ distance (useful for value-based similarity search) using two suitable data structures (double-ended queue and segment tree). The algorithm has the same time/space complexity as the state-of-the-art algorithm for typical MP computation under the normalized $\ell_2$ distance (useful for shape-based similarity search). We validate its efficiency and effectiveness through extensive numerical experiments and two real-world applications — anomaly detection in the NYC taxi data, as well as in a simulated communication network. Depending on data and applications, we show that the value-based similarity search could perform almost equally well as the shape-based similarity search. We propose novel algorithms to compute the MP. Numerical results from extensive comparative studies and two real-world applications are presented in [V]. We make a few concluding remarks in [VI] The Appendix [VI] contains detailed descriptions of some subroutines required by the major algorithm proposed in [III].

II. NOTATION AND DEFINITION.

Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a real-valued time series with length $n$, where $X[i] = x_i \in \mathbb{R}$ is the value sampled at time instance $i$, $i = 0, 1, \ldots, n-1$. Denote by $m$ the length of a subsequence (window size), which satisfies $1 \leq m \leq n$. Let $X_{j-1, j+m-1} \equiv [x_j, x_{j+1}, \ldots, x_{j+m-1}]$ denote the $j$th subsequence of $X$, $0 \leq j \leq n-m$.

[Matrix Profile of Time Series] The Matrix Profile of $X$ is a new time series $Y = [y_0, y_1, \ldots, y_{n-m}]$, where

$$y_j \equiv \min_{0 \leq j' \leq n-m, j \neq j'} d(X_{j-1, j+m-1}, X_{j'-1, j'+m-1}), \quad (1)$$

where $d(\cdot, \cdot)$ is the $\ell_p$ distance.

In other words, the Matrix Profile (MP) of $X$ at time instance $j$ is the distance between the $j$th subsequence and its nearest-neighbor subsequence in $X$. Note that, for economy of space, while still capturing the essence of the MP, in Def. [II] we ignore the other component of the MP (i.e., the corresponding indices of the nearest-neighbor subsequence) from the original definition proposed in [I]. It is also worth pointing out that, unlike in [I], we do not normalize the subsequences before computing their distances. In addition, without loss of generality, throughout the paper we only use the $\ell_\infty$ distance (i.e., $p = \infty$; $d(X_{j-1, j+m-1}, X_{j'-1, j'+m-1}) = \max_{0 \leq k \leq m-1} |X_{j+k} - X_{j'+k}|$) to demonstrate the operation of our algorithms. It should not require much effort to adapt our proposed algorithms to the $\ell_p$ ($1 \leq p < \infty$) cases.

III. ALGORITHM.

To facilitate computing the distances between subsequences in [I], we rotate the original time series $X$ to obtain $n - 1$ new time series, denoted by $X^{(i)} = [x_{i+1}, x_{i+2}, \ldots, x_{n-1}, x_0, x_1, \ldots, x_i]$, $i = 0, 1, \ldots, n-2$. (Note, however, that in our actual implementation, we do not need to keep all these rotated time series in memory; instead, using only one array to store them is enough.) Then, given a subsequence index $j$ ($0 \leq j \leq n-m$) (i.e., time instance),
all the subsequences to be compared could be extracted from these rotated time series at the same time instance. To illustrate this, let us take a look at the example shown in Table I for which we have $n = 10$, $m = 4$, $j = 2$. To obtain $y_2$ in (1), we only need to compute the distances between the subsequence $X_{2,...,5}$ and subsequences $X_{2,...,5}^{(i)}, i \in \{i|i \in \mathbb{N}, 0 \leq i \leq 8\} \setminus \{i \in \mathbb{N}, 3 < i < 7\}$, and find the minimum. Note that the subsequences $X_{2,...,5}^{(i)}, i = 4, 5, 6$, are not valid ones in the original time series $X$, because they violate the monotonicity of time indices in $X$.

It is not hard to verify that, to obtain $y_j$ in (1) for general cases, where we have $1 \leq m \leq n$ and $0 \leq j \leq n - m$, it is sufficient to compute the distances between the subsequences $X_{j,...,j+m-1}$ and subsequences $X_{j,...,j+m-1}^{(i)}, i \in \{i|i \in \mathbb{N}, 0 \leq i \leq n - 2\} \setminus \{i \in \mathbb{N}, n - m - j - 1 < i < n - j - 1\}$, and find the minimum.

**TABLE I: Example of Filtering Out Invalid Subsequences**

\[
\begin{array}{cccccccccccc}
X & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 \\
X^{(0)} & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_0 \\
X^{(1)} & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_0 & x_1 \\
X^{(2)} & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_0 & x_1 & x_2 \\
X^{(3)} & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_0 & x_1 & x_2 & x_3 \\
X^{(4)} & x_5 & x_6 & x_7 & x_8 & x_9 & x_0 & x_1 & x_2 & x_3 & x_4 \\
X^{(5)} & x_6 & x_7 & x_8 & x_9 & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 \\
X^{(6)} & x_7 & x_8 & x_9 & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\
X^{(7)} & x_8 & x_9 & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\
X^{(8)} & x_9 & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\
\end{array}
\]

Based on the observations above, to obtain the MP under the $\ell_\infty$ distance, let us first compute the element-wise distances of the original time series $X$ and the rotated ones. We summarize the steps as Procedure “EleWiseDistToRotTimeSeries”: see VI-E It is worth noting that the returned array $eleDist$ is again a new time series with the same length as the original $X$.

We also notice that the key to the MP computation is a subroutine that computes the maximum values of sliding windows given a time series $Z$ (the aforementioned $eleDist$ actually; we rewrite it as $Z$ to make the description concise) and a window size $m$. Specifically, we need to find the maximum value of the subsequence $Z_{j,...,j+m-1}, \forall 0 \leq j \leq n - m$. Clearly, a brute-force approach could simply search the maximum value for each and every $m$-sized window, leading to a time complexity in the order of $O(m(n - m + 1)) = O(mn)$, which is dependent on the window size $m$ and would be too slow for a large $m$. To speed up the computation, we apply two types of data structures: the double-ended queue and the segment tree.

### A. Finding Sliding Maxima Using a Double-ended Queue.

In this subsection, we apply the double-ended queue to find maximum values of the sliding windows. We summarize the steps as Procedure “SlidMaxDeque”: see VI-A Lines 9 through 11 remove indices of elements that are not from the current sliding window. Lines 12 through 14 remove from the double-ended queue the indices of all elements that are smaller than the current element $X[i]$. Noting each element of the given $X$ would only be added into the queue once and removed from the queue once, we see that the time complexity of the procedure is $O(n)$.

#### B. Finding Sliding Maxima Using a Segment Tree.

In this subsection, we apply the segment tree to find maximum values of the sliding windows. The segment tree was originally designed for fast updating an array (list) and conducting range queries [3]. Considering our goal in this paper, we only make use of the fast range-query property. In particular, given a time series with length $n$, the time complexity of building a segment tree would be $O(n \log n)$, and each range query (e.g., searching the maximum of a subsequence) would take $O(\log n)$ time only. Note that this time complexity does not depend on the query size $m$. For a fairly large $m$ and $n$, using the segment tree method would be faster than the brute-force by a factor $\frac{m \log n}{n^2} - 1 \approx O(m)$.

Before searching the range maxima (i.e., sliding maxima) of the original time series $X$, we need to construct a segment tree from $X$ and implement a subroutine for range query (i.e., searching range maximum in our case) as preprocessing steps; see Procedures “SegmentTree” (VI-B) and “Range-Maximum” (VI-C). The remaining steps are summarized as Procedure “SlidMaxSegTree.” see VI-D

#### C. Computing the Matrix Profile.

Finally, we summarize the steps of computing the MP using the double-ended queue or segment tree data structures as Alg. I where we use the argument $option$ to indicate which to choose: when $option$ equals “deque,” we use the double-ended queue, and when $option$ equals “segtree,” we use the segment tree. It is not hard to figure out that the total time complexity is $O(n^2)$ for the “deque” option and $O(n^2 \log n)$ for the “segtree” option, and the space complexity is $O(n)$ for both options. Note that these complexities are the same as those for the state-of-the-art typical MP algorithms (CPU version only) [1], [4].

**Algorithm 1 Computation of Matrix Profile Using Double-ended Queue or Segment Tree**

1: **procedure** MatrixProfileDequeSegTree($X, m, option$) 
2: $n \leftarrow \text{length of } X$ 
3: initialize $MP$ as an array with length $n - m + 1$ and each element being a large enough float number (e.g., $1.0 \times 10^6$) 
4: for $i = 0, 1, \ldots, n - 2$ do 
5: $eleDist \leftarrow \text{EleWiseDistToRotTimeSeries}(X, i)$ 
6: if $option$ equals “deque” then 
7: $slidMax \leftarrow \text{SlidMaxDeque}(eleDist, m)$ 
8: else if $option$ equals “segtree” then 
9: $ST \leftarrow \text{SegmentTree}(eleDist)$ 
10: $slidMax \leftarrow \text{SlidMaxSegTree}(ST, m)$ 
11: end if 
12: for $j = 0, 1, \ldots, n - m$ do 
13: if $i \leq n - m - j - 1$ or $i \geq n - j - 1$ then 
14: $MP[j] \leftarrow \min(MP[j], slidMax[j])$
For cases where $1 \leq p < \infty$, the segment tree method still applies, except that the range query should be about the sum rather than the maximum. The double-ended queue alternative, however, would not generalize to such cases; instead, one could apply the typical dynamic programming approach (see, e.g., [2]). It is also worth noting that the segment tree could also be replaced by the binary indexed tree [5]; the two have similar properties in terms of fast range queries.

IV. NUMERICAL RESULTS.

A. CPU Time for Matrix Profile Computation.

First, we conduct a comparative study on the CPU time of different algorithms for the Matrix Profile (MP) computation. The tested time series had a length $n = 2000$ and was generated randomly by sampling from a normal distribution $\mathcal{N}(5.0, 10.0)$. We record the CPU time of three algorithms (i.e., the brute-force approach, the one using the segment tree data structure, and the one using the double-ended queue data structure) for MP computation, varying the window size $m$, such that $m \in \{10, 30, 50, \ldots , 490\}$. We implement the algorithms in Python and run them on a desktop computer with an Intel(R) Core(TM) i7-4770 CPU and 16 GB of system memory.

The results are shown in Fig. 1. It can be seen that our proposed algorithm is much faster than the brute-force one; its CPU time is independent of the window size $m$, whereas the CPU time of the brute-force algorithm is approximately proportional to $m$. Clearly, this is consistent with our complexity analysis of Alg. 1 and the brute-force baseline algorithm.

Fig. 1: CPU time vs. window sizes for different algorithms for Matrix Profile computation ($n = 2000$).

B. Application in Anomaly Detection.

Next, we apply the proposed algorithm for Matrix Profile computation to two anomaly detection scenarios.

1) NYC Taxi Data: The NYC taxi data set [6] contains time-stamped numbers of NYC taxi passengers, where 8 anomalies occur during the Independence day, Labor day, Labor day parade, NYC marathon, Thanksgiving, Christmas, New Years day, and a snow storm. The raw data is from the NYC Taxi and Limousine Commission [4]. The data file included here consists of aggregating the total number of taxi passengers into 30 minute buckets between 7/1/2014 12:00:00 AM and 1/31/2015 11:30:00 PM. Before the MP computations, we conduct preprocessing by normalizing the raw numbers over the whole time range; note that this is different than normalizing the query windows separately.

To see performance differences between applying the MP under unnormalized and normalized distances (hence, value-based and shape-based similarity search, respectively), we compute MP values for both cases using the same window size $m = 24$ hours (i.e., 1 day). The CPU times for both cases are close and less than 10 minutes using the same hardware environment described in IV-A; for the MP under unnormalized $\ell_\infty$ distance, we implement Alg. 1 in Python with the “deque” option, and for the MP under normalized $\ell_2$ distance, we use the Python library STUMPY [7].

The obtained MP values are shown in Fig. 2, where the top subplot shows the MP values under unnormalized $\ell_\infty$ distance (value-based similarity search), whereas the bottom subplot shows the MP values under normalized $\ell_2$ distance (shape-based similarity search). It can be seen that for this application, both the value-based similarity search and the shape-based alternative would lead to approximately equally good anomaly detection results. Note that the red shaded areas map to the ground-truth anomalies where we observe that the MP reaches peak values, indicating “discords” (i.e., anomalies).

2) Communication Network Flow Data: To generate network traffic (flow) data, we conduct simulations using the software package SADIT [7], which, based on the fs-simulator

https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

https://github.com/TDAmeritrade/stumpy
can efficiently generate flow-level network traffic datasets with annotated anomalies.

We simulate a network (see Fig. 3) consisting of an internal network involving 8 normal users (CT1-CT8) and 1 server (SRV) that stores some sensitive data, and 3 Internet nodes (INT1-INT3) that visit the internal network via a gateway (GATEWAY). We consider the network with a day-night pattern where the flow size follows a log-normal distribution. The simulation settings we use are exactly the same as those in \[9\] [IV.B.2]. We generate 10 normal samples, each being a multidimensional time series lasting for one week (0 h – 168 h). To generate one more test sample, we consider an anomaly where node CT2 increases its mean flow size by 30% at 59 h and the increase lasts for 80 minutes before the mean returns to the normal value (i.e., the anomalies are on the time interval [212400, 217200] (in seconds)). This kind of anomaly could be related to the case where an attacker tries to exfiltrate sensitive information (e.g., user accounts and passwords) through SQL injection \[10\].

When dealing with the flow data, we only use the feature “flow size,” and downsample the original time series by a factor of 10. Before computing the MP, we conduct a preprocessing step by normalizing the whole time series; note again that this is different than normalizing the query windows (corresponding to \[9\] [IV.B.1]) the CPU times for computing the MP under unnormalized and normalized distances are close and less than 20 minutes using the same hardware/software environment. The window size for both cases is taken as \( m = 360 \) (corresponding to 3600 seconds (i.e., 1 hour) in the original data).

The resulting MP values are shown in Fig. 4 where the top subplot shows those under unnormalized \( \ell_\infty \) distance (value-based similarity search), whereas the bottom subplot shows those under normalized \( \ell_2 \) distance (shape-based similarity search). It is seen from Fig. 4 that the value-based similarity search would be able to successfully detect the anomalies, whereas the shape-based alternative would not. Note that peak MP values correspond to “discords” or, in other words, anomalies.

![Image](https://via.placeholder.com/150)

Fig. 3: Simulation setting (this figure is from \[11\]).

Fig. 4: Matrix Profile applied to communication network anomaly detection; top: MP under unnormalized distance (value-based similarity search), bottom: MP under normalized distance (shape-based similarity search). The red shaded area maps to the original time interval [212400, 217200] (in seconds) where the anomalies are.

V. Conclusion.

Using two carefully selected data structures (double-ended queue and segment tree), we have proposed a novel algorithm to compute the Matrix Profile (MP) under the unnormalized \( \ell_\infty \) distance (leading to value-based similarity search). The algorithm has the same time/space complexity as the state-of-the-art algorithm for typical MP computation under the normalized \( \ell_2 \) distance (leading to shape-based similarity search), and is validated through numerical experiments. Also, our algorithm is readily adaptable to the MP computation under unnormalized \( \ell_p \) \((1 \leq p < \infty)\) distances (see Remark \[ILL-C\]).

We also show through specific real-world applications that the value-based similarity search could perform almost equally well as the shape-based similarity search, and sometimes outperform the latter significantly. Of course, this depends on the data and applications.

Possible future work includes adapting the proposed algorithm to a GPU version, thus leveraging the benefits of parallel computing, and to a Pan Matrix Profile (PMP) version \[12\], thus enabling analysis with varying query sizes. Also, we might consider additional applications.

VI. Appendix.

A. Finding Sliding Maxima Using Double-ended Queue.

1: \textbf{procedure} SlidMaxDeque\((Z, m)\)
2: \hspace{1em} \( n \leftarrow \text{length of } Z \)
3: \hspace{1em} \textbf{if} \( m \) equals \( 1 \) \textbf{then}
4: \hspace{2em} \textbf{return} \( Z \)
5: \hspace{1em} \textbf{end if}
6: \hspace{1em} \text{initialize } Q \text{ as a new empty double-ended queue}
7: \hspace{1em} \text{initialize } slidMax \text{ as a new empty vector}
8: \hspace{1em} \textbf{for} \( l = 0, 1, \ldots, n - 1 \) \textbf{do}
9: \hspace{2em} \textbf{if} \( Q \) is nonempty and \( Q[0] \leq l - m \) \textbf{then}
10: \hspace{3em} \text{remove the leftmost element from } Q
11: \hspace{2em} \textbf{end if

\textbf{end if

\textbf{end for

\textbf{return} \( Z \)
while $Q \neq \emptyset$ and $Z[l] > Z[Q[\ell - 1]]$ do
remove the rightmost element from $Q$
end while
append $l$ to the rightmost of $Q$
if $l \geq m - 1$ then
append $Z[Q[0]]$ to the rightmost of $slidMax$
end if
end for
return $slidMax$
end procedure

B. Construction of Segment Tree.

1. procedure SegmentTree($X$) [13]
2. $n \leftarrow$ (length of $X$)
3. initialize a new zero array $ST$ with length $2n$
4. copy $X$ to the second half of $ST$
5. for $l = n - 1 \ldots 1$
6. $ST[l] \leftarrow$ max($ST[2 \times l], ST[2 \times l + 1]$)
7. end for
8. return $ST$
9. end procedure

C. Computation of Range Maximum in Segment Tree.

1. procedure RangeMaximum($ST, left, right$) [13]
2. $n \leftarrow$ (length of $ST$)/2
3. $left \leftarrow left + n$, $right \leftarrow right + n$
4. initialize max with a small enough float number (e.g., $1.0 \times 10^{9}$)
5. while $left < right$ do
6. if $left$ is odd then
7. $max \leftarrow$ max($max, ST[left]$)
8. $left \leftarrow left + 1$
end if
if $right$ is odd then
right $\leftarrow$ right $-$ 1
max $\leftarrow$ max($max, ST[right]$)
end if
left $\leftarrow left/2$, right $\leftarrow right/2$
end while
return $max$
end procedure

D. Finding Sliding Maxima Using Segment Tree.

1. procedure SlidMaxSegTree($ST, m$)
2. $n \leftarrow$ (length of $ST$)/2
3. initialize $slidMax$ as a new empty vector
4. for $j = 0, 1, \ldots, n - m$ do
5. $rm \leftarrow$ RangeMaximum($ST, j, j + m$)
6. append $rm$ to the rightmost of $slidMax$
end for
8. return $slidMax$
9. end procedure

E. Computation of Element-wise Distance to Rotated Time Series.

1. procedure EleWiseDistToRotTimeSeries($X, i$)
2. $n \leftarrow$ length of $X$