MPdist Haskell Implementation: A distance metric for Time Series

Asif Mallik (am5086)

December 2021

1 MPdist

MPdist (Matrix Profile distance) is a distance metric that captures the similarity between different time series [1]. Among its various features are its ability to be able to compare time series of varying lengths, being robust to anomalies, missing data and other issues common to time series data, very efficient to compute. In brief, it compares the similarity of subsequences of their time series in order to determine whether they share motifs or not and hence whether they are similar. The definition of MPdist makes use of a recently developed concept in time series data mining known as Matrix Profile.

2 Matrix Profile

Defining a matrix profile rigorously would require at least five prerequisite definitions. So, for interest of brevity, we define it informally. A matrix profile with respect to time series A and B of window size m, is a list representing the minimal euclidean distance of each m-length subsequence of A to any m-length subsequence of B. As a result, note that the matrix profile seen as an operation is non-commutative. The "matrix" in matrix profile refers to the distance matrix one would have to compute if they were to naively try to compute the matrix profile. A join similarity matrix is an extension of this concept that makes it more symmetric by concatenating the matrix profile of A with respect to B. Yeh et al. outlines the algorithm for computing the matrix profile for any given pair of time series [2].

MPdist between time series A and B can then be defined as the nth percentile of the join similarity matrix profile of time series A and B. n here is a parameter to MPdist, so more accurately, MPdist is a family of distance metrics parameterized by window size m and similarity percentile n. Given a matrix profile, it is simple to compute the MPdist which amounts to doing a linear time selection on the resulting matrix profile. Thus, the bulk of the project has been spent writing and optimizing STOMP and STAMP (in particular, STOMP).

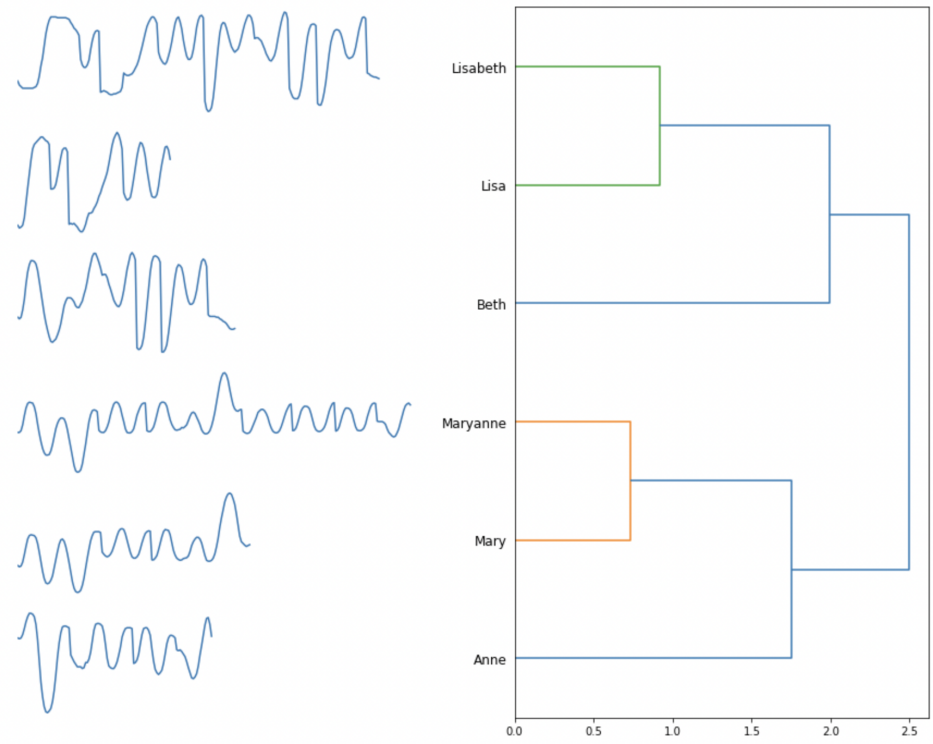


Figure 1: Time series clustered by MPdist

Figure 1 shows a clustering of different handwriting of names using MPdist. It can be seen it correctly finds Maryanne, Mary and Anne to be similar and similarly with Beth, Lisa and Lisabeth as they share parts of the name. However, it finds the two groups to be dissimilar. The actual distances can be verified by running the program according to the instructions given in the README file.

3 STOMP and STAMP algorithm

STAMP and STOMP are two algorithms for computing the matrix profile between two time series. The STAMP algorithm works by iterating through the indices of one of the time series to select as a starting index for the subsequence. Then, in each iteration, it computes a sequence of sliding dot products with the selected subsequence as the starting point so as to compute the minimum distance over it. At the ith iteration the minimum distance over these sliding distances becomes the ith entry of the matrix profile. The novelty in this algorithm is the use of Fast Fourier Transform and the sliding dot product trick in order to avoid recomputing. Overall, its complexity is $O(n^2 \log n)$ with n being the length of the time series. STOMP is even faster with a runtime of $O(n^2)$ by

using utilizing the diagonal dependence structure of the dot product matrix of the subsequences. Namely,

$$QT_{i,j} = QT_{i-1,j-1} - T_{i-1}T_{j-1} + T_{i+m-1}T_{j+m-1}$$

With this, we are able to apply dynamic programming and simultaneously split the computation of such a matrix in parallel into its diagonals whose computations are independent of each other. A full treatment of all the modifications and optimizations made to STOMP to make it as fast as possible can be found in Zhu et al. [3]

4 The Implementations

4.1 MPdist Algorithm

Despite being the topic of my project, the actual algorithm for computing the MPdist is very simple given the matrix profile algorithm. I parameterize mpdist using the type of a matrix profile function that would return the matrix profile.

```
1  mpdist :: (IFArray -> IFArray -> Int -> [Float]) -> Int -> Float
   ↪  -> [Float] -> [Float] -> Float
2
3  mpdist mpFunc m p tAList tBList = getKPartition k pABBA where
4      k =  min (ceiling (p * (fromIntegral (nA + nB)))) (nA - m + 1
       ↪  + nB - m)
5      pABBA = pAB ++ pBA
6      pAB = mpFunc tA tB m
7      pBA = mpFunc tB tA m
8      tA = toZeroIndexedArray tAList
9      tB = toZeroIndexedArray tBList
10     nA = length tAList
11     nB = length tBList
```

The function compute the matrix profile for both A with respect to B and B with respect to A, then joins the two together and then computes the bottom kth value in the matrix profile. The k depends on the parameter p which specifies the proportion of the subsequence of the two that define similarity between time series.

4.2 STAMP

The following is how I define the STAMP algorithm. The bulk of the computation takes place in Mueen's Algorithm for Similarity Search which here is called mass which computes the sliding distance for each subsequence of time series A. Another important function is the computeMeanStd which computes the rolling means and standard deviations of time series A.

```
stamp ::  IFArray -> IFArray -> Int -> [Float]

stamp tA tB m = res  where
    res =  map (minimum . (mass (elems tB) (elems meansB) (elems
    ↪  stdsB))) rollingA
    rollingA = rollingWindow m (elems tA)
    (meansB, stdsB) = computeMeanStd tB m
```

MASS mostly relies on the sliding dot product function and computeMeanStd, using those to compute distances:

```
mass :: [Float] -> [Float] -> [Float] -> [Float] -> [Float]

mass t meanT varT q = map (distanceComp meanQ varQ m') (zip3
↪  meanT varT qt) `using` parBuffer 100 rdeepseq where
    meanQ = meanQ' ! 0
    varQ = varQ' ! 0
    (meanQ', varQ') = computeMeanStd (toZeroIndexedArray q) m
    qt = slidingDotProduct q t
    m' = fromIntegral m
    m = length q
```

The sliding product computation adds the most computational complexity to the entire algorithm as its runtime is $O(n \log n)$ and it internally relies on FFT in order to avoid redundant computation:

```
slidingDotProduct :: [Float] -> [Float] -> [Float]

slidingDotProduct q t = elems (ixmap (m-2, n-2) succ qt) where
    qt = irfft (arrZipWith (*) q_raf t_af)
    q_raf = rfft (listArray (0, 2*n-1) q_ra)
    t_af = rfft (listArray (0,2*n-1) t_a)
    t_a = t ++ (take n (repeat 0))
    q_ra = (reverse q) ++ (take (2 * n - m) (repeat 0))
    n = length t
    m = length q
```

The rolling mean and standard deviation is computed using a variant of the Welford online algorithm for computing variance which is called by compute-MeanStd internally.

```
welford :: Int -> Int -> IFArray -> ([Float], [Float])

welford k w a
    | k < w = let fw = fromIntegral w in
            let t = take w (elems a) in
            let mean = (sum t)/fw in
```

```
            ([mean], [(sum (map (\x -> x*x) t))/fw - (mean **
            ↪ 2)])
    | otherwise = ((newMean : prevMeans), (newVar : prevVars))
    ↪ where
            (prevMean : _) = prevMeans
            (prevVar : _) = prevVars
            newMean = prevMean + (a!k - a!(k-w))/(fromIntegral w)
            newVar = prevVar + (a!k - a!(k-w)) * (a!k - newMean +
            ↪ a!(k-w) - prevMean)/(fromIntegral w)
            (prevMeans, prevVars) = welford (k-1) w a
```

4.3 STOMP (initial implementation)

Now we look at the code for the STOMP algorithm that I first implemented without much attempt at making it efficient. First, we have the STOMPTopt function (opt stands for optimized, the paper introduces changes to the original STOMP throughout for better performance [3]).

```
stompOpt :: Array Int Float -> Array Int Float -> Int -> [Float]
stompOpt tA tB m = map (\x -> sqrt (abs (2 * (fromIntegral m) *
↪ (1-x)))) pearsons where
    pearsons = minDiag tA meansA stdsInvA meansDecA tB meansB
    ↪ stdsInvB meansDecB m [(-((length tA) - m + 1) +
    ↪ 1)..((length tB) - m )]
    (meansA, stdsInvA, meansDecA) = preprocessT tA m
    (meansB, stdsInvB, meansDecB) = preprocessT tB m
```

The function does not do much other than other call minDiag which returns the final maximum correlation for each of the subsequences which is then used to compute the minimum distance.

Internally, the minDiag recursively combines each of the diagonal with the previous result by taking their elementwise maximum Pearson correlation, after padding each of them appropriately, as can be seen below. As mentioned earlier, this is the natural place for parallelization so, I added a parallel evaluation strategy here as seen below:

```
minDiag :: IFArray -> IFArray -> IFArray -> IFArray -> IFArray ->
↪ IFArray -> IFArray -> IFArray -> Int -> [Int] -> [Float]
minDiag tA meansA stdsInvA meansDecA tB meansB stdsInvB meansDecB
↪ m (k:diags) =
    zipWith max paddedPearsons otherDiagPearsons where
    otherDiagPearsons = minDiag tA meansA stdsInvA meansDecA tB
    ↪ meansB stdsInvB meansDecB m diags
    paddedPearsons = take (length tA - m + 1) ((take (-k) (repeat
    ↪ (-1))) ++ pearsons ++ (repeat (-1)))
    pearsons = reverse rPearsons `using` rparWith rdeepseq
```

```
7       (cov, rPearsons) = computePearsons tA meansA stdsInvA
        ↪  meansDecA tB meansB stdsInvB meansDecB m k [upperI,
        ↪  (upperI-1)..lowerI]
8       upperI = (min ((length tA) - m) ((length tB) - m - k))
9       lowerI = (max 0 (-k)) + 1
10
11  minDiag tA _ _ _ _ _ _ _ m [] = take (length tA - m + 1) (repeat
    ↪  (-1))
```

The computePearson function individually computes the specific and also pursues a recursive method internally.

5 Intermediate Results

STAMP performs very poorly in terms of speed. Even computing the MPdist of two time series of five thousand data points takes very long compared (over ten minutes) to the naive implementation of STOMP (under a minute). While I am sure that this can be significantly improved with parallelization and better memory management, empirical results from Zhu et. al. [3] and its time complexity points to it being inherently slower than STOMP. So, I decide not to spend any more time optimizing STAMP and instead focus on STOMP. In the rest of the paper, **I will be exclusively be focusing on STOMP for further parallelization efforts**.

Now, we look at some analysis of the result for STOMP. I tried running this in many configurations, but most notably, I ran MPdist on two time series of size 10,000. I tried seeing what would happen if I ran it with differing number of cores as seen in figure 2. Seeing that majority of the time was being spent on garbage collection, I tried to see what would happen if I changed increaseed the heap size from the default of 1 megabyte to 100 megabytes. Predictably, this caused a speedup across core count as seen in figures 3.

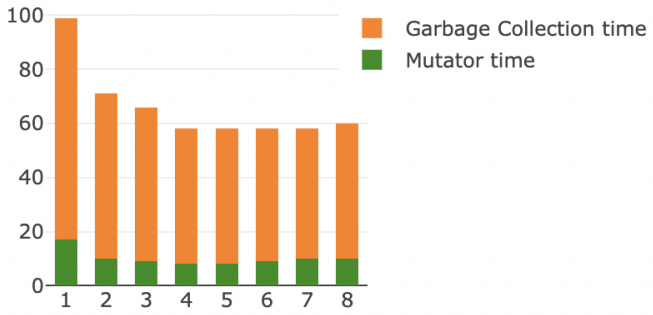


Figure 2: Runtime decomposed into garbage collection and mutator times for default heap size (one megabyte)

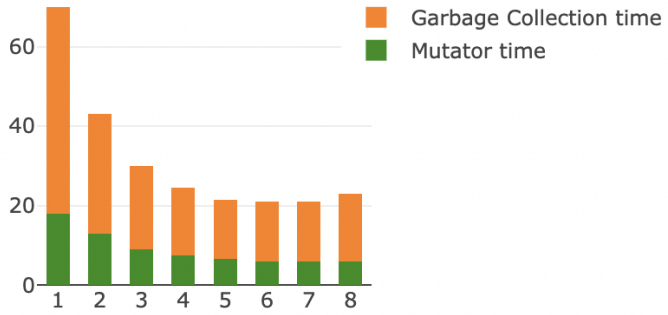
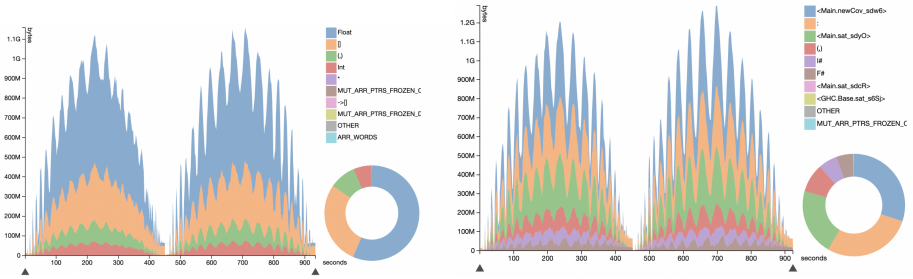


Figure 3: Runtime decomposed into garbage collection and mutator times for heap size hundred megabytes

In order to find the reason behind such high proportion of time being allocated to garbage collection, I looked at both threadscope and heap profiles. While threadscope was helpful in showing the timeline of when garbage collection was taking place, it was not helpful at all in figuring out which function or types were responsible. So, for this, I looked to heap profiles and I include two such graphs in figure 4 that I analyzed. As seen in these graphs, the main culprit types for the garbage collection are float, list, tuple and integers. This was contrary to what I had expected - I was expecting to see arrays dominate here but in hindsight it makes sense given the many intermediate lists that are generated for computation whereas the arrays are reused. Figure 4b gives even finer detail on the source of the heap allocation due to its description. In particular, the source that really stands out, both as readable and as having the highest allocation is newCov. This gives a precise starting point to try to optimize during the next step.



(a) Breaks down heap allocation by type (b) Breaks down heap allocation by closure description

Figure 4: Heap allocation profiling

6 Optimizations

From the above heap profiles, I guessed that the garbage collection time is amplified due to my extensive use of long lazy evaluations. As a result, when the time comes to do garbage collection, the garbage collector has to traverse through a much longer chain whereas if strict evaluation was used, the intermediate variable they could have been collected long time ago. So instead I wanted to switch to strict evaluations when there was possibility of very long chain of dependencies. I did this in multiple places such as in stompOpt I opt for strict parallel evaluation of preprocessed auxillary time series:

```
1 stompOpt tA tB m = map (\x -> sqrt (abs (2 * (fromIntegral m) *
  ↪  (1-x)))) pearsons where
2     pearsons = minDiag tA meansA stdsInvA meansDecA tB meansB
      ↪  stdsInvB meansDecB m [(-((length tA) - m + 1) +
      ↪  1)..((length tB) - m )]
3     (meansA, stdsInvA, meansDecA) = preprocessT tA m
      ↪  `using` parTuple3 rdeepseq rdeepseq rdeepseq
4     (meansB, stdsInvB, meansDecB) = preprocessT tB m
      ↪  `using` parTuple3 rdeepseq rdeepseq rdeepseq
```

Another place I use this is in computeFullPearsons, a wrapper function I introduced for computePearsons to change the structure of minDiag (I elaborate this at the end of this section).

```
1 computeFullPearsons tA meansA stdsInvA meansDecA tB meansB
  ↪  stdsInvB meansDecB m k =
2     take (length tA - m + 1) ((take (-k) (repeat (-1))) ++
      ↪  pearsons ++ (repeat (-1))) where
3         pearsons = reverse rPearsons
```

```
4           (cov, rPearsons) = computePearsons tA meansA stdsInvA
            ↪  meansDecA tB meansB stdsInvB meansDecB m k [upperI,
            ↪  (upperI-1)..lowerI] `using` rdeepseq
5           upperI = (min ((length tA) - m) ((length tB) - m - k))
6           lowerI = (max 0 (-k)) + 1
```

Another point of optimization is the large number of sparks that were made (one for every diagonal, of which there are around 20,000 when comparing two time series of 10,000). So I decided to consolidate the diagonals into larger chunks which required me to rewrite minDiag. I combined this idea with that of strict evaluation so that I ended up computing the minimum distance (maximum correlation) in two steps, one within each of the chunks evaluated strictly, and one outside it when consolidating them to get the overall minimum:

```
1  minDiag :: IFArray -> IFArray -> IFArray -> IFArray -> IFArray ->
   ↪  IFArray -> IFArray -> IFArray -> Int -> [Int] -> [Float]
2
3  minDiag tA meansA stdsInvA meansDecA tB meansB stdsInvB meansDecB
   ↪  m diags = maxPearsons reducedPearsonMatrix where
4
       ↪  reducedPearsonMatrix = map maxPearsons (chunk 100 pearsonMatrix)
       ↪  `using` parList rdeepseq
5      pearsonMatrix = map (computeFullPearsons tA meansA stdsInvA
       ↪  meansDecA tB meansB stdsInvB meansDecB m) diags
6      maxPearsons = foldl (zipWith max) floorPearsons
7      floorPearsons = take (length tA - m + 1) (repeat (-1))
```

These optimizations has led to a significant reduction in the heap allocations at a given time as seen in figure 5. There is a 6 times reduction in the peak heap allocation from 1.2 gigabytes to 200 megabytes. Furthermore, newCov is nowhere to be seen and so are types corresponding to integers, tuples. List construction and floats are still there, likely because they are unavoidable part of the computation of the matrix profile as the algorithm is structured. This reduction in garbage collection is reflected in figure 6 which shows the final runtimes.

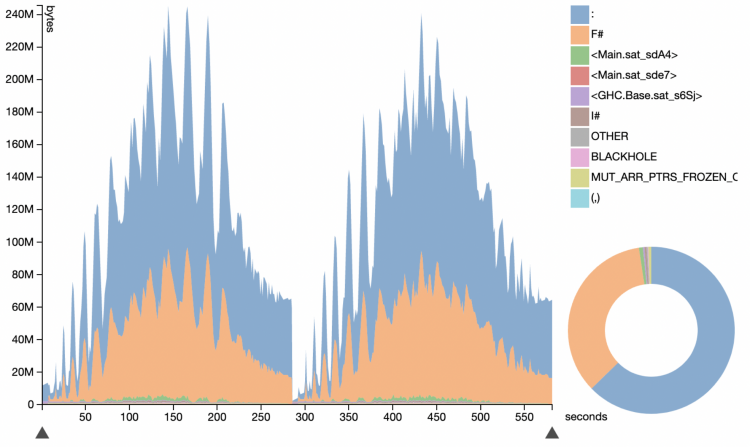


Figure 5: Heap allocation profiling after optimization

6.1 Other failed attempts

These are attempts I have made to reduce garbage collection further but they backfired in the form of increased time spent on garbage collection

- I tried moving a lot of repeated computations that were inside computePearsons but for some reason that only increased runtime
- I tried strict evaluations in many other locations to see if I shave off more garbage collection time but this led to even slower runtime, likely because it interferes with the optimization made by GHC when there isn't a good rationale to
- Write strict version of foldl - for some reason it led to slower runtime

6.2 Other ideas

Here I list some of my other ideas and speculations to try to reduce the garbage collection time that I did not get the time to implement

- To try to reduce the garbage collection are to use unboxed types for lists and floats as that is currently the biggest chunk of the allocated type.
- To compute the list of terms A to D before running computePearsons instead of computing them each time which leads to duplication
- Find ways to avoid conversion between lists and array by directly applying the same function in the current space

7 Final Results and Discussions

The final result shows that after the optimizations made, the MPdist function runtimes are fairly close to the ideal ones we would expect from a perfect parallelization strategy of the algorithm. We get around a 50% reduction in runtimes across core counts. There is still significant garbage collection going on but it forms a much smaller proportion of the total time than compared to before the optimizations. The discrepancy between our result and the ideal could be due to unavoidable serial computation such as from input-output and MPdist computation itself (all parallelization takes place within STOMP).

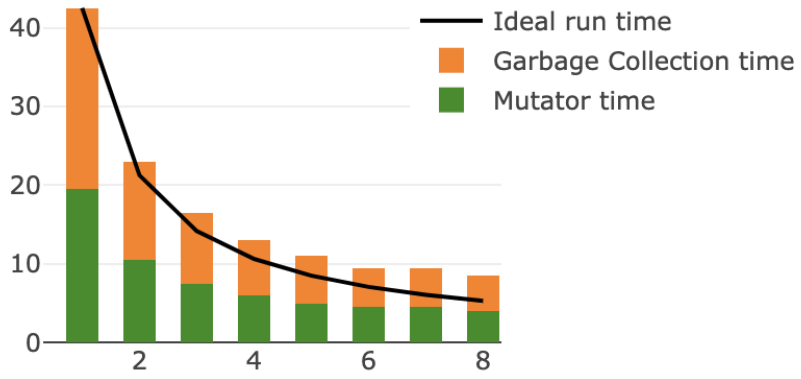


Figure 6: Final run time of MPdist on two time series of lengths 10,000 with heap allocation of 100mb

Future direction for this line of work would involve further trying to reduce the garbage collection size using unattempted ideas mentioned in the previous section. Another interesting direction would be to implement online versions of the algorithm such as STOMPI in Haskell and seeing whether garbage collection works better with those kind of algorithms for matrix profile.

References

[1] Shaghayegh Gharghabi, Shima Imani, Anthony Bagnall, Amirali Darvishzadeh, and Eamonn Keogh. Matrix profile xii: Mpdist: a novel time series distance measure to allow data mining in more challenging scenarios. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 965–970. IEEE, 2018.

[2] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn

Keogh. Matrix profile i: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In *2016 IEEE 16th international conference on data mining (ICDM)*, pages 1317–1322. Ieee, 2016.

[3] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk, and Eamonn Keogh. Matrix profile ii: Exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins. In *2016 IEEE 16th international conference on data mining (ICDM)*, pages 739–748. IEEE, 2016.