Early Abandoning PrunedDTW and its application to similarity search

Matthieu Herrmann∗ Geoffrey I. Webb∗

Faculty of Information Technology, Monash University, Melbourne, Australia.

Abstract

The Dynamic Time Warping (“DTW”) distance is widely used in time series analysis, be it for classification, clustering or similarity search. However, its quadratic time complexity prevents it from scaling. Strategies, based on early abandoning DTW or skipping its computation altogether thanks to lower bounds, have been developed for certain use cases such as nearest neighbour search. But vectorization and approximation aside, no advance was made on DTW itself until recently with the introduction of PrunedDTW. This algorithm, able to prune unpromising alignments, was later fitted with early abandoning. We present a new version of PrunedDTW, “EAPrunedDTW”, designed with early abandon in mind from the start, and able to early abandon faster than before. We show that EAPrunedDTW significantly improves the computation time of similarity search in the UCR Suite, and renders lower bounds dispensable.

Keywords: Time Series, Similarity Measure, DTW

1 Introduction

DTW [16] is one of the most important similarity measure in time series analysis, extensively used in clustering [8], classification [21], and similarity search [14]. Classification with one nearest neighbour under DTW (“NN1-DTW”) was a widely used benchmark for a long time, and while state of the art classifiers are now significantly more accurate, some of them actually embed a NN1-DTW classifier as a component [5, 6, 7, 18].

A naive DTW implementation has space and time complexities both undesirably quadratic. How to take care of the former is well known [11] (and we do so in this article, cf section 2), while taking care of the later is more challenging. The common “early abandoning” strategy first tries to avoid DTW computations, and then tries to early abandon them. But it can only speed up scenarios where an upper bound is available (e.g. NN1 search), and not DTW itself. DTW has been sped up through vectorization [22, 11] or approximation [17], and more recently by PrunedDTW [19], an algorithm developed for scenarios where early abandoning isn’t applicable (e.g. when all pairwise distances are required). It was later applied to similarity search in the UCR USP Suite [20], fitted with some early abandon capabilities used in conjunction with lower bounding. However, the core of the algorithm was not revisited.

In this article, we present “EAPrunedDTW”, our version of PrunedDTW specifically designed to efficiently and effectively early abandon. We incorporated our algorithm in the UCR Suite and reproduced the experiments from [20], showing a significant speed improvement over previous implementations based on DTW and PrunedDTW. Moreover, we show that using lower bounds,
even if useful, becomes dispensable. This is especially relevant for other distances sharing the
same structure as DTW (see [9]) for which no efficient and tight lower bounds exist.

The next section introduces DTW and related work. We then present our algorithm in two
parts, sections 3 and 4. Experiments follow in section 5, and we conclude in section 6.

2 Background and related work

Given two times series $S$ and $T$ of respective length $l_S$ and $l_T$, and a cost function (usually the
squared Euclidean distance) over data points themselves denoted by application (e.g. $S(i)$), the
distance $\text{DTW}^{S,T}$ between $S$ and $T$ is defined by:

\begin{align}
\text{DTW}^{S,T} &= \text{DTW}^{S,T}(l_S, l_T) \\
\text{DTW}^{S,T}(0,0) &= 0 \\
\text{DTW}^{S,T}(i,0) &= +\infty \\
\text{DTW}^{S,T}(0,j) &= +\infty \\
\text{DTW}^{S,T}(i,j) &= \text{cost}(S(i), T(j)) + \min\left\{\begin{array}{l}
\text{DTW}^{S,T}(i-1,j) \\
\text{DTW}^{S,T}(i,j-1) \\
\text{DTW}^{S,T}(i-1,j-1)
\end{array}\right.
\end{align}

Figure 1: Equations defining $\text{DTW}^{S,T}$

We define by $M_{\text{cost}}^{S,T}(i,j) = \text{cost}(S(i), S(j))$ the cost matrix of $S$ and $T$. A warping path in
$M_{\text{cost}}^{S,T}$ is a continuous and monotonous path connecting $(1,1)$ to $(l_S, l_T)$. The value $\text{DTW}^{S,T}$ is
the cost of the optimal (cheapest) warping path which also represents the cost of the optimal
non linear alignment between $S$ and $T$. Such an alignment is illustrated by Figure 2b for
$S = (3, 1, 4, 4, 1, 1)$ and $T = (1, 3, 2, 1, 2, 2)$. The constraints on the path lead to the extremities
(at $(1, 1)$ and $l_S, l_T$) being aligned and no alignments crossing each other (monotonicity).

Let $S(1\ldots i)$ be the prefix of $S$ of length $i$. We defined the DTW matrix of the series $S$ and $T$ by
$M_{\text{DTW}}^{S,T}(i,j) = \text{DTW}^{S(1\ldots i), T(1\ldots j)}$. Figure 2a shows the matrix and optimal warping path
for $S$ and $T$ (this and subsequent figures are colored from intense green for the minimum value 0,
through to intense red for the maximum value 22). We are only interested in the DTW matrix’s
last cell at $(l_S, l_T)$, i.e. the value of $\text{DTW}^{S,T}$. This cell can be computed without keeping the
full matrix in memory. Equation 5 tells us that a cell’s value only depends on its “top”, “top left
diagonal”, and “left” neighbours. In a line by line, left to right scan, the two firsts are obtained
from the previous line, while the later is the previously computed cell. Hence, we only need to
keep the current and previous line in memory, limiting the space complexity to $O(l)$ as described
by Algorithm 1.

Commenting Algorithm 1. Memory allocation is limited by matching the line dimension with
the shortest series. Both arrays, representing the previous and current lines of the matrix, have
an extra cell for the vertical border ($j = 0$) and are initialized to $\infty$. The horizontal border
($i = 0$) is initially represented by the curr array, which is swapped for the prev array prior to any
computation (line 5). This is why we initialize the border cell $(0,0)$ in the extra cell of the curr
array (line 4). This cell has to be set back to $\infty$ after the first iteration. We repeatedly do so
(line 7), which is avoidable by computing the first line before the loop. However, our goal is to
introduce a structure we will build upon: a form of this assignment is required later.

\footnote{There may be several optimal paths with the same cost.}
Algorithm 1: $O(n)$ space complexity DTW$^S,T$.

Input: the time series $S$ and $T$

Result: DTW$^S,T$

1. $co \leftarrow$ shortest series between $S$ and $T$
2. $li \leftarrow$ longest series between $S$ and $T$
3. $(prev, curr) \leftarrow$ arrays of length $l_{co} + 1$ filled with $\infty$
4. $curr[0] \leftarrow 0$
5. for $i \leftarrow 1$ to $l_{li}$ do
6.   swap(prev, curr)
7.   $curr[0] \leftarrow \infty$
8.   for $j \leftarrow 1$ to $l_{co}$ do
9.       $c \leftarrow$ cost($li(i), co(j)$)
10.      $curr[j] \leftarrow c + \min(curr[j-1], prev[j], prev[j-1])$
11. return $curr[l_{co}]$

2.1 The Warping Window

DTW is commonly parametrized by a warping window $w$ constraining the warping path to deviate by at most $w$ cells from the diagonal [16]. A window of 0 is equivalent to the cost function being applied to the successive elements of the two series (e.g. the squared Euclidean distance), while a window of min($l_S, l_T$) is equivalent to DTW. The experiments section 5 require a warping window, but for clarity’s sake we present our algorithms for DTW only.

2.2 Early Abandoning

Early abandoning is commonly used to speed up scenarios requiring nearest neighbour search, like NN1 classification or similarity search. In those scenarios, we test a query $S$ against several candidates $T_1, T_2, \ldots, T_m$. After computing $v_1 = \text{DTW}^S,T_1$, we have a first upper bound $ub = v_1$ on the DTW cost between $S$ and its actual nearest neighbour, i.e. the nearest neighbour will have a cost lower than or equal to $ub$. We tighten $ub$ every time we find a better candidate. It
follows that we can early abandon the computation of $\text{DTW}_{S,T_k}$ as soon as we can determine that $v_k$ will strictly exceed $ub$. This strictness condition prevents from early abandoning ties. To implement DTW with early abandon it is enough to get the the minimum value of each curr line, and stop if it is strictly greater then $ub$. The UCR Suite also exploits the Keogh lower bound to further tighten $ub$.

Another form of early abandoning is lower bounding: The idea is to cheaply compute a lower bound $\text{lb}(S,T_k) \leq \text{DTW}_{S,T_k}$ and to skip any further computation if $\text{lb}(S,T_k) > ub$. Several lower bounds have been developed, and developing them is still an active field of research. The most commonly used lower bounds are LB_Kim and LB_Keogh.

2.3 PrunedDTW

Early abandoning requires an upper bound on the best candidates. PrunedDTW was designed for scenarios where no such upper bound exists, e.g. when all pairwise distances are required. In those cases, we can only speed-up DTW itself. PrunedDTW determines that some cells from the DTW matrix cannot be part of the optimal warping path and skips their computation. To do so, it relies on a pruning threshold which is an upper bound on the current alignment’s cost (and not on the cost of the alignment with the best candidates). This threshold can only depend on the series being aligned. PrunedDTW uses the squared Euclidean distance, which corresponds to the diagonal in the DTW matrix, which may be quite loose.

The next logical step was to use PrunedDTW in scenarios with upper bounds on best candidates as they are usually tighter. The UCR Suite has been modified to use PrunedDTW, giving birth to the UCR USP Suite. However, PrunedDTW inherits features from its origins as a pruning algorithm, and fails to fully exploit the possibilities of early abandoning. In the current work we seek to exploit those opportunities to the full.

2.4 Overheads and Run-Time Speed

The computation of DTW relies of two nested “tight” loops: the inner body is small but repeated many times. Early abandoning and pruning techniques modify those loops with extra checking and bookkeeping, adding a bit of “overhead” per iteration. The expectation is that the time saved by those techniques outweigh their overhead. It follows that for the same gain, the algorithm with the smallest overhead will be the most efficient. By carefully decomposing our algorithm in several stages, we seek to minimize these overheads while maximising pruning.

Comparing the run-time speed of different algorithms seems simple: run the candidates several times under the same conditions and compare the fastest (i.e. less noisy) results. There is not much to discuss if the differences are substantial (regardless of the direction), and clearly due to algorithmic changes. However, we are not comparing algorithms, but implementations of those algorithms. A “fast” implementation of a “slow” algorithm may be the best performer until we reach large enough data. To combat this problem we embed our DTW algorithm in the UCR Suite and make minimal modifications to the code.

3 Pruning and Early Abandoning from the left

In the previous section, we show how to compute $\text{DTW}_{S,T}$ line by line. Now, we show how to prune the computation “from the left”. As we scan a line, we look at discarding as many as

---

3 The cost of any valid alignment is an upper bound. Either the alignment is the optimal one, or the optimal one will have a lower cost.
possible of the leftmost cells of the DTW matrix. We do so by extending the “left border” up to the leftmost cell of the current line that could potentially be part of an optimal warping path. The Figures 3a and 3b present the pruning and early abandoning scenarios for the same $S$ and $T$ as before, using upper bounds of $ub = 9$ and $ub = 6$. As $DTW_{ST}^{S,T} = 9$, no early abandoning occurs in the first case, but it does in the second.

**Algorithm 2:** Pruning from the left.

```
Input: the time series $S$ and $T$, an upper bound $ub$
Result: $DTW_{S,T}$
1 $co \leftarrow$ shortest series between $S$ and $T$
2 $li \leftarrow$ longest series between $S$ and $T$
3 $(prev, curr) \leftarrow$ arrays of length $l_{co} + 1$ filled with $\infty$
4 $curr[0] \leftarrow 0$
5 next_start $\leftarrow 1$
6 for $i \leftarrow 1$ to $l_{li}$ do
7     swap($prev$, $curr$)
8     $j \leftarrow$ next_start
9     $curr[j-1] \leftarrow \infty$
10    while $j = next_start \land j \leq l_{co}$ do
11       $c \leftarrow cost(li(i), co(j))$
12       $curr[j] \leftarrow c + \min(prev[j], prev[j-1])$
13       if $curr[j] > ub$ then $next_start \leftarrow next_start + 1$
14       $j \leftarrow j + 1$
15    if $j > l_{co}$ then return $\infty$
16    while $j \leq l_{co}$ do
17       $c \leftarrow cost(li(i), co(j))$
18       $curr[j] \leftarrow c + \min(curr[j-1], prev[j], prev[j-1])$
19       $j \leftarrow j + 1$
20 return $curr[l_{co}]$
```

Figure 3: Representation of $M_{DTW}^{S,T}$ “from the left” pruning for $S = (3,1,4,4,1,1)$ and $T = (1,3,2,1,2,2)$ with two different upper bounds.
Looking at Figure 3, the grey cells are never computed (border or pruned cells), and the white cells with red edges represent “discard points”. Given a line and an upper bound $ub$, discard points represent a continuous sequence of cells greater than $ub$, starting at the left border. By monotonicity, all columns below discard points are guaranteed to be greater than $ub$. It follows that those columns can be skipped for the rest of the computation, and we do so by starting the next line after the current line’s last discard point. Ignoring the top border for now (we will take care of it in the next section), you can easily check that all the dependencies of the grey cells are either another grey cell, or a discard point. Note that $(1, 0)$ is a discard point while $(2, 0)$ is pruned. In this regard, the first discard point and pruned column naturally match the left border.

Looking at the discard point at $(3, 1)$ in Figure 3a with $ub = 9$, we see that all the dependencies (black arrows) of $(4, 1)$ are greater than $ub$, and the same goes until $(6, 1)$. It follows that we can start the following lines on the second columns, at $(3, 2)$, effectively skipping the cells below $(3, 1)$ and intuitively extending the border by one cell to the right. Note that the continuity condition, that all cells to the left of a discard point must also exceed $ub$, is paramount. Without it, the cell at $(3, 4)$ with value $14 > 9$ would be marked as a discard point and wrongfully prevent the optimal warping path from being computed.

Now looking at Figure 3b with $ub = 6$, we see that the discard points reach the end of the fifth line, leading to early abandoning. Under this strategy, early abandoning is the natural consequence of extending the border over the full length of a line. This is a very effective way of early abandoning, requiring little bookkeeping. In contrast, PrunedDTW [20] maintains the minimum across a row and abandons when that minimum exceeds $ub$.

Algorithm 2 shows how this works, in two stages. Lines now start at index $next_start$ instead of index 1. The first line still starts at 1, so we initialize $next_start$ to 1, and then update it by $j + 1$ after encountering a discard point at $(i, j)$. The first stage, represented by the first inner loop, computes successive discard points on the same line. The condition on line 10 enforces our continuity requirement: if $next_start$ did not increase, the previous cell is not a discard point, and so neither is the current cell. Also, while in this loop, we know that a cell’s left neighbour is greater than $ub$ and thus can be left out when looking for the smallest dependency (see line 12). In that regard, line 9 seems unnecessary – it isn’t as it also represents an assignment to the top left diagonal at the next iteration. If we reached the end of the line, we early abandon (line 15), else we go into the second stage of the algorithm, embodied by the second inner loop, which is a normal DTW computation.

### 4 Early Abandoned PrunedDTW

In the previous section, we intuitively extended the left border to the right, leading to early abandoning when it reaches the end of the line. Unfortunately, we cannot do the same thing for the top border. This is because our strategy implicitly relies on the computation order (line by line) by moving forward where we start a line (updating $next_start$). In a line by line scan “moving where a column starts” does not make sense. However, we can move backward where we end a line pruning cells up to its actual end. As before, we ensure that all pruned cells would be greater than an upper bound $ub$ by ensuring that their dependencies are greater than $ub$. Adding this “pruning from the right” to the algorithm gives us the full EAPrunedDTW algorithm.

Figures 4a and 4b present the EAPrunedDTW scenarios for $S$ and $T$, with $ub = 9$ and $ub = 6$. The blue cell shows an early abandoning position, and the black cells with red border are “pruning points”. In a line, a pruning point starts a continuous sequence of cells greater than $ub$.

---

4 This explains why Algorithm 2 uses the “$next_start$” variable rather than a “discard point” one.
(a) DTW Matrix for $S$ and $T$ with $\text{ub} = 9$

(b) DTW Matrix for $S$ and $T$ with $\text{ub} = 6$. The arrows show the dependencies actually evaluated.

Figure 4: Illustration of EAPrunedDTW. The blue cell represents the point of early abandoning, white cells represent discard points, and dark gray cells represent pruning points.

$\text{ub}$ until the actual end of the line. In this regard, we have another natural match, now between the first pruning point at $(0, 1)$ and the top border, as for all cells after $(0, 1)$ we indeed have $\infty > \text{ub}$.

Examining Figure 4b with $\text{ub} = 6$, the cell $(2, 2) = 8 > \text{ub}$ is not a pruning point as it is followed by cells below $\text{ub}$. Notice that we can prune after the pruning points at $(1, 4)$ and $(4, 3)$, but not after $(3, 3)$. This is because a pruning point at $(i, j)$ carries information for the next line: it tells us that a cell at $(i + 1, j') > j$ can only depend on its left neighbour, its top and top left neighbours being greater than $\text{ub}$. It follows that all cells following a cell $(i + 1, j') > \text{ub}$ will be greater than $\text{ub}$. This allow us to both create a pruning point at $(i + 1, j')$ and to prune the rest of the line. This explains why we prune after $(4, 3)$ but not after $(3, 3)$: the former can be pruned thanks to the latter! Moreover, cells after $(3, 3)$ must be computed in order to check the continuity requirement, allowing $(3, 3)$ to be a pruning point. Notice how, contrary to the discard points that extend the left border once and for all, pruning points create a “right border” that can go back and forth.

Given a pruning point at $(i, j)$, we discussed what happens to the cells at $(i + 1, j') > j$.

We now discuss the case of the cell immediately below, at $(i + 1, j)$. This point is of particular interest as it is where our left and right borders can “collide”. Either it depends on both its left and top left neighbours (e.g. $(2, 4)$), or only on its top left neighbour if it follows a discard point (e.g. $(5, 3)$ and $(1, 1)$). In the latter case, we early abandon if the cell is greater than $\text{ub}$ (e.g. the blue cell at $(5, 3)$) as this would put next_start in a pruned area, creating a “border collision”. Note that PrunedDTW does prune “from the left” and “from the right”, but does not take the border collision into account, early abandoning by checking the minimum value of the line. This allows EAPrunedDTW to abandon earlier than PrunedDTW.

EAPrunedDTW is described in algorithm 3 in four stages, building upon algorithm 2. We introduce two new variables. The first one, prev_pruning_point, represents the pruning point of the previous line and is initialized to 1 (for the first pruning point at $(0, 1)$, line 6). The second one, pruning_point, represents the pruning point being computed in the current line, and is initialized to 0. This concurs with our update strategy: we assume that a cell is a pruning point unless
Algorithm 3: Early Abandoning PrunedDTW.

Input: the time series S and T, an upper bound ub

Result: DTW(S,T)

1 $co \leftarrow$ shortest series between S and T
2 $li \leftarrow$ longest series between S and T
3 $(prev, curr) \leftarrow$ arrays of length $l_{co} + 1$ filled with $\infty$
4 $curr[0] \leftarrow 0$
5 next_start $\leftarrow 1$
6 prev_pruning_point $\leftarrow 1$
7 pruning_point $\leftarrow 0$
8 for $i \leftarrow 1$ to $l_{li}$ do
9   swap(prev, curr)
10  $j \leftarrow$ next_start
11  while $j = next_start \land j < prev_pruning_point$ do
12     $c \leftarrow$ cost($li(i), co(j)$)
13     $curr[j] \leftarrow c + \min(prev[j], prev[j-1])$
14     if $curr[j] \leq ub$ then pruning_point $\leftarrow j + 1$
15     else next_start $\leftarrow$ next_start + 1
16     $j \leftarrow j + 1$
17  end
18  while $j < prev_pruning_point$ do
19     $c \leftarrow$ cost($li(i), co(j)$)
20     $curr[j] \leftarrow c + \min(curr[j-1], prev[j], prev[j-1])$
21     if $curr[j] \leq ub$ then pruning_point $\leftarrow j + 1$
22     $j \leftarrow j + 1$
23  end
24  if $j < l_{co}$ then // At prev_pruning_point
25     $c \leftarrow$ cost($li(i), co(j)$)
26     if $j = next_start$ then
27        $curr[j] \leftarrow c + prev[j-1]$
28        if $curr[j] \leq ub$ then pruning_point $\leftarrow j + 1$
29        else return $\infty$
30     else
31        $curr[j] \leftarrow c + \min(curr[j-1], prev[j-1])$
32        if $curr[j] \leq ub$ then pruning_point $\leftarrow j + 1$
33        $j \leftarrow j + 1$
34     end
35  end
36  else if $j = next_start$ then
37     return $\infty$
38  end
39  while $j = pruning_point \land j < l_{co}$ do
40     $curr[j] \leftarrow c + curr[j-1]$
41     if $curr[j] \leq ub$ then pruning_point $\leftarrow j + 1$
42     $j \leftarrow j + 1$
43  end
44  prev_pruning_point $\leftarrow$ pruning_point
45  if $prev_pruning_point > l_{co}$ then return $curr[l_{co}]$
46  else return $\infty$

proven otherwise. In other words, if the current cell is less than $ub$, we assume that the next one
is a pruning point. Either it is, and pruning_point being left untouched will end up correctly set,
or it is not and pruning_point will be updated again.

The first two stages, embodied by the first two inner loops, are similar to the ones in Algorithm 2.
However, the loops now update pruning_point, only go up to prev_pruning_point, and we
early abandon in the third stage. The third stage corresponds to the column of the previous
pruning point (line 23). If we are still within bounds, we check if the previous cell is a discard point (next_start = j at line 25) tests if we reached this position while advancing next_start, i.e. while encountering discard point), we only look at the diagonal, early abandoning if the computed value is greater than ub. Else, we compute the DTW value based on the left and diagonal. In both cases we update pruning_point. If we are not within bound, we reached the end of the line. If we did so with the first loop (again tested with next_start = j), we are in the same early abandoning situation as in Algorithm 2 and we early abandon (line 34).

We finally reach the fourth stage, after the column prev pruning_point (line 35). The only possible dependency is now on the left, and we can skip the rest of the line as soon as we encounter a value greater than ub. We do this by checking pruning_point against j, ensuring that it is being updated. The final check before returning ensures that we computed the last cell of the line in the last iteration, as we may have stopped the computation in the last stage.

Decomposing the algorithm in several stages is the main underlying contribution of this paper. Whereas the standard DTW loop used by PrunedDTW requires that the minimum of three previous values be computed for each new cell of the matrix, our decomposition allows many cells to be updated considering only one (line 36) or two (line 14) previous values, saving considerable computation.

5 Experiments

We created the “UCR MON Suite”\(^5\) available online with the experimental results and analysis script. Just like the “UCR USP Suite”\(^6\) which uses PrunedDTW, it is a modification of the original “UCR Suite”\(^7\). It uses our EAPrunedDTW algorithm with support for a warping window and upper bound tightening (present in both the UCR and the UCR USP suites). Given EAPrunedDTW’s efficiency, it is also interesting to assess how it performs without lower bounding, so we added the fourth variant “UCR MON nolb”. Note that this last variant cannot tighten the upper bound, as this relies on information computed by lower bounds. We compare the four suites on the same computer equipped with an AMD Opteron 6338P at 2.3Ghz (RAM is not limiting).

We reproduced the experiments from the UCR USP paper\(^20\) over the same datasets. See \(^{20}\) for a detailed description, and how they have been obtained. The data originated from:

1. Freezing of Gait (FoG)
2. Athletic performance monitoring (Soccer)
3. Physical activity monitoring (PAMAP2)
4. MIT-BIH Arrhythmia Database (ECG)
5. Electrical load measurements (REFIT)
6. Photoplethysmography (PPG)

Each dataset is composed of a long reference \(R\) series, and 5 queries \(Q_{i \leq n \leq 5}\) of length 1024. For each \(Q_{n}\), the algorithm locates the closest subsequence in \(R\) under several conditions of query’s length and warping window ratio of that length. A query’s length can be 128, 256, 512 and 1024 (taking a prefix for lengths smaller than 1024), and the window’s ratio is taken among \{0.1, 0.2, 0.3, 0.4, 0.5\}. This generates 20 experiments per query, 100 experiments per dataset and so 600 experiments per algorithm. We note that our results for the UCR and UCR USP suites are in line with those published in \(^{20}\).

---

\(^5\)UCR from Monash University  
[https://github.com/MonashTS/UCR-Monash](https://github.com/MonashTS/UCR-Monash)  
\(^6\)UCR from the Universidade de Sao Paulo  
[https://sites.google.com/view/ucruspsuite](https://sites.google.com/view/ucruspsuite)  
\(^7\)[http://www.cs.ucr.edu/~eamonn/UCRsuite.html](http://www.cs.ucr.edu/~eamonn/UCRsuite.html)
We present the results in Figure 5. For each data set we indicate the proportion of the DTW computations that are pruned by each of the three lower bounds used in the UCR Suite, and the remaining proportion for which the DTW computation is performed. Note that our algorithm is only applied to these latter cases. The more computations that are pruned, the less opportunity our algorithm has to demonstrate its relative capacity.

UCR MON is overall the fastest, completing the experiments in 473150 seconds (5 days 11h25m50s), which is 8.778 times faster than UCR (4153309 seconds, i.e. 48 days 1h41m49s), and 2.036 times faster than UCR USP (963251 seconds, i.e. 11 days 3h34m11s, 4.312 times faster than UCR). Its advantage is clear for long series (length 1024, Figure 5a), where it is faster than UCR from 3.73 times (FoG) up to 9.72 times (PPG), and is faster than UCR USP from 1.93 (PPG) up to 3.6 (FoG). In some cases the overhead is not compensated by the gain, and our algorithm is slower but the difference stays overall small. Our algorithm is slower than UCR in 44 (7.3%) cases over 600 by an average of 0.971 seconds, and at most by 9.06 seconds. It is slower than UCR USP in 66 (11%) cases by an average of 1 seconds, and at most by 10.98 seconds. In comparison, UCR USP is slower than UCR in 108 (18%) cases, by an average of 55.11 seconds, and at most by 986.15 seconds.

The most interesting results come from UCR MON nolb. They show that lower bounds are dispensable, UCR MON nolb completing the experiments in 644652 seconds (7 days 11h04m12s). It is faster overall than both UCR (speedup 6.443) and UCR USP (speedup 1.494). It is however slower than UCR in 364 (60.06%) cases, by an average of 174.372 seconds, and at most 6392 seconds. Higher ratios of DTW computation lead to higher differences, especially as the window size and length increase. A major exception is the REFIT dataset: even if it has a higher ratio of DTW computations than PAMAP2, UCR MON nolb only starts to be competitive with a length of 1024. Another interesting observation is that varying the window size has less impact on our algorithms than the others. Again, this is very remarkable with REFIT. We explain this by the following: at first, a small window (around 0.1, 0.2) gives a fast result without leaving room for a lot of pruning. As the window increases (around 0.3, 0.4), more and more pruning becomes possible, compensating both for the associated overhead and the increased numbers of cells to compute. This continues up to a point (around 0.4, 0.5) where increasing the window is nearly fully compensated by pruning, with most extra cells being pruned. Further increases to the window result in ever more pruning, having little overall impact on runtime. Interestingly, UCR USP does not exhibit this behaviour.

6 Conclusion and future work

We present EAPrunedDTW, a new version of the PrunedDTW algorithm that supports highly effective pruning and early abandoning. We implemented it in the UCR Suite, and show it being currently the overall fastest available way of performing similarity search. The main innovations of EAPrunedDTW are to use the collision of the left and right borders to trigger early abandoning, saving the need to keep track of the minimum value so far, and a decomposition that allows many cells to be updated by considering only one or two previous values as opposed to the three previous values always required by PrunedDTW.

We also show that similarity search is tractable without using lower bounds, opening it to distance measures other than DTW. Indeed, many elastic measures share the same structure as DTW [9], only differing in their cost function, without having lower bounds that are as efficient to compute, if any. In essence we should be able to speed up most of elastic distances, improving computation time in most scenarios where best candidates are required. This includes nearest neighbour classification, which in turn is widely used in several ensemble classifiers. We note
Figure 5: Average runtimes per dataset of UCR, UCR USP, UCR MON and UCR MON without lower bound, by query length and window ratios. Also show effectiveness of lower bounds pruning per datasets, applicable for all but UCR_MON_nolb which is 100% DTW.
that the Elastic Ensemble was removed from a revision of Hive Cote due to its computation time
something that may not be necessary with our algorithm.

Our next steps will be to adapt our algorithm to other elastic distances and implement them in ensemble classifiers such as Elastic Ensemble [5], Proximity Forest [7] and TS-Chief [18].

References

[1] M. Bachlin et al. “Wearable Assistant for Parkinson’s Disease Patients With the Freezing of Gait Symptom”. In: IEEE Transactions on Information Technology in Biomedicine 14.2 (Mar. 2010), pp. 436–446.

[2] A. Bagnall et al. “A tale of two toolkits, report the third: on the usage and performance of HIVE-COTE v1.0”. In: arXiv:2004.06069 [cs, stat] (Apr. 25, 2020). arXiv: 2004.06069.

[3] A. L. Goldberger et al. “PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals”. In: Circulation 101.23 (June 13, 2000).

[4] M. Kachuee et al. “Cuff-less high-accuracy calibration-free blood pressure estimation using pulse transit time”. In: 2015 IEEE Int Symp on Circuits and Systems (ISCAS). 2015 IEEE Int Symp on Circuits and Systems (ISCAS). Lisbon, Portugal: IEEE, May 2015, pp. 1006–1009.

[5] J. Lines and A. Bagnall. “Time series classification with ensembles of elastic distance measures”. In: Data Mining and Knowledge Discovery 29.3 (May 2015), pp. 565–592.

[6] J. Lines, S. Taylor, and A. Bagnall. “Time Series Classification with HIVE-COTE: The Hierarchical Vote Collective of Transformation-Based Ensembles”. In: ACM Transactions on Knowledge Discovery from Data 12.5 (July 20, 2018), pp. 1–35.

[7] B. Lucas et al. “Proximity Forest: an effective and scalable distance-based classifier for time series”. In: Data Mining and Knowledge Discovery 33.3 (May 2019), pp. 607–635.

[8] M. Luczak, “Hierarchical clustering of time series data with parametric derivative dynamic time warping”. In: Expert Systems with Applications 62 (Nov. 2016), pp. 116–130.

[9] P.-F. Marteau. “Time Warp Edit Distance with Stiffness Adjustment for Time Series Matching”. In: IEEE Transactions on Pattern Analysis and Machine Intelligence 31.2 (Feb. 2009), pp. 306–318.

[10] G.B. Moody and R.G. Mark. “The impact of the MIT-BIH Arrhythmia Database”. In: IEEE Engineering in Medicine and Biology Magazine 20.3 (June 2001), pp. 45–50.

[11] A. Mueen and E. Keogh. “Extracting Optimal Performance from Dynamic Time Warping”. In: Proc 22nd ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining - KDD ’16. the 22nd ACM SIGKDD Int Conf. San Francisco, California, USA: ACM Press, 2016, pp. 2129–2130.

[12] D. Murray et al. “A data management platform for personalised real-time energy feedback”. In: Int Conf Energy Efficiency in Domestic Appliances and Lighting. 2015, pp. 1293–1307.

[13] S. A. Pettersen et al. “Soccer video and player position dataset”. In: Proc 5th ACM Multimedia Systems Conf - MMSys ’14. the 5th ACM Multimedia Systems Conf. Singapore, Singapore: ACM Press, 2014, pp. 18–23.
[14] T. Rakthanmanon et al. “Searching and mining trillions of time series subsequences under dynamic time warping”. In: Proc 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’12. the 18th ACM SIGKDD international conference. Beijing, China: ACM Press, 2012, p. 262.

[15] A. Reiss and D. Stricker. “Introducing a New Benchmarked Dataset for Activity Monitoring”. In: 2012 16th Int Symp on Wearable Computers. 2012 16th Annual Int Symp on Wearable Computers (ISWC). Newcastle, United Kingdom: IEEE, June 2012, pp. 108–109.

[16] H. Sakoe and S. Chiba. “Dynamic programming algorithm optimization for spoken word recognition”. In: IEEE Transactions on Acoustics, Speech, and Signal Processing 26.1 (Feb. 1978), pp. 43–49.

[17] S. Salvador and P. Chan. “Toward accurate dynamic time warping in linear time and space”. In: Intelligent Data Analysis 11.5 (Oct. 10, 2007), pp. 561–580.

[18] Ahmed Shifaz et al. “TS-CHIEF: A Scalable and Accurate Forest Algorithm for Time Series Classification”. In: Data Mining and Knowledge Discovery 34.3 (2020), pp. 742–775.

[19] D. F. Silva and G. E. A. P. A. Batista. “Speeding Up All-Pairwise Dynamic Time Warping Matrix Calculation”. In: Proc 2016 SIAM Int Conf on Data Mining. Proc 2016 SIAM Int Conf on Data Mining. Society for Industrial and Applied Mathematics, June 30, 2016, pp. 837–845.

[20] D. F. Silva et al. “Speeding up similarity search under dynamic time warping by pruning unpromising alignments”. In: Data Mining and Knowledge Discovery 32.4 (July 2018), pp. 988–1016.

[21] X. Wang et al. “Experimental comparison of representation methods and distance measures for time series data”. In: Data Mining and Knowledge Discovery 26.2 (Mar. 2013), pp. 275–309.

[22] L. Xiao et al. “Parallelizing Dynamic Time Warping Algorithm Using Prefix Combinations on GPU”. In: IEEE, Nov. 2013, pp. 294–299.