Early Abandoning and Pruning for Elastic Distances including Dynamic Time Warping*

Matthieu Herrmann, Geoffrey I. Webb

Monash University, Australia

{matthieu.herrmann,geoff.webb}@monash.edu

3 June 2021

Abstract

Nearest neighbor search under elastic distances is a key tool for time series analysis, supporting many applications. However, straightforward implementations of distances require $O(n^2)$ space and time complexities, preventing these applications from scaling to long series. Much work has been devoted to speeding up the NN search process, mostly with the development of lower bounds, allowing to avoid costly distance computations when a given threshold is exceeded. This threshold, provided by the similarity search process, also allows to early abandon the computation of a distance itself. Another approach, is to prune parts of the computation. All these techniques are orthogonal to each other. In this work, we develop a new generic strategy, "EAPruned", that tightly integrates pruning with early abandoning. We apply it to six elastic distance measures: DTW, CDTW, WDTW, ERP, MSM and TWE, showing substantial speedup in NN search applications. Pruning alone also shows substantial speedup for some distances, benefiting applications beyond the scope of NN search (e.g. requiring all pairwise distances), and hence where early abandoning is not applicable. We release our implementation as part of a new C++ library for time series classification, along with easy to use Python/Numpy bindings.

1 Introduction

Nearest neighbor (NN) search under elastic distances is a major tool in time series analysis, supporting many applications, including classification [12], subsequence search [25], regression [27], clustering [17], and outlier detection [2].

Unfortunately, elastic distances have quadratic time complexity with respect to length, incurring costly computation and preventing many applications from scaling to long series. This has mostly been addressed through lower bounding,

---

which seeks to identify cases where the quadratic distance computation can be avoided (see Section 2.1). Two main strategies have been developed for directly speeding up distance computations: "pruning" and "early abandoning" (see Section 2.2). In this paper we develop a new algorithm, "EAPruned", that tightly integrates both pruning and early abandoning, thereby substantially reducing computation and often rendering lower bounding superfluous.

EAPruned is a generic strategy, applicable to a broad class of elastic distances. We investigate the effectiveness of our approach with six key elastic distance measures: DTW, CDTW, WDTW, ERP, MSM and TWE. To enable fair comparison, we implemented the key alternative algorithms for these six measures in C++ and compared their run times using NN classification over the UCR archive (in its 85 univariate datasets version). We show that EAPruned offers significant speedups: from 3.93 to 39.23 times faster than simple implementations, and from 1.85 to 8.44 times faster than implementations with the usual early abandoning scheme. We also show that our algorithm remains effective in pruning-only uses, i.e. when early abandoning is not applicable We then show, in the cases of DTW and CDTW, that lower bounding is complementary to EAPruned. Finally, we show that our algorithm is also effective in settings beyond NN classification, with an application to sub-sequence search.

The rest of this paper is organised as follows. Section 2 introduces the context of this work and presents the six distances. The related work is presented in Section 3, and EAPruned itself is described in Section 4. We then present our experimental results in Section 5, and conclude in Section 6.

2   Background

NN classification has historically been the workhorse of time series classification, which led to the development of various elastic distances. The most widely used of these is the Dynamic Time Warping (DTW) distance, introduced in 1971 by [20] along with its constrained variant CDTW [19]. Guided by the UCR time series archive [4], time series classification has made enormous progress in the past decade, including the rise of "ensemble classifiers", i.e. classifiers combining an ensemble of other classifiers. The "Elastic Ensemble" ("EE", see [12]), introduced in 2015, was one of the first classifiers to be consistently more accurate than NN-DTW over a wide variety of tasks. EE combines eleven NN classifiers based on eleven previously developed elastic distances (see Section 2.3).

Most elastic distances have parameters. At train time, EE fine tunes these parameters using cross validation. At test time, the query's class is determined through a majority vote between the component classifiers. "Proximity Forest" ("PF", see [14]) uses the same set of NN classifiers as EE, but deploys them within an ensemble of random decision trees, for which splits are determined by distance to exemplars of each class. Both EE and PF solely work in the time domain, leading to poor accuracies when discriminant features are in other domains. This was addressed by their respective evolution into "HIVE-COTE" ("HC", see [13]) and "TSCHIEF" (see [23]), combining more classifiers working

2

in different domains (interval, shapelets, dictionary and spectral, see [13] for an overview).

While EE provided a qualitative jump in accuracy, it also required a quantitative jump in computation time. For example, [28] reports 17 days to learn and classify the UCR archive's ElectricDevice benchmark. Indeed, given series of length L, naive elastic distance algorithms have $O(L^2)$ space and time complexities. This is only compounded by an extensive search for the best parametrization through leave-one-out cross validation. For a training set of N times series, searching among M parameters ($M = 100$ for EE), EE has a training time complexity in $O(M.N^2.L^2)$. HIVE-COTE not only expands on EE, it actually embeds it as a component – or did. Recently, EE was dropped from HIVE-COTE (see [1]) because of its computational cost. This caused an average drop of 0.6% in accuracy, and beyond a 5% drop on some datasets (tested on the UCR archive). The authors considered that this loss in accuracy was a necessary price to pay for the resulting speedup (the authors only report that the new version is "significantly faster"). The implication is that if NN classifiers can be sped up sufficiently, it will be possible for EE to rejoin HIVE-COTE, resulting in a substantial further lift in accuracy in what is currently the most accurate time series classifier.

Speeding up the computation of NN search under elastic distances is also of interest for several other reasons. First, TS-Chief [23], which is another state-of-the-art classifier, still relies on NN classifiers. Second, this will benefit other applications relying on NN search, such as clustering [17], outlier detection [2], regression [27], and sub-sequence search [25]. Third, a disproportionate amount of research has been spent on DTW and CDTW compared to other distances, which is reinforced by the lack of efficient lower bounds for their alternatives. Our algorithm provides substantial speed-up for all six presented distances, and for any further distance that follows a similar structure (see Section 2).

In this paper, we will only consider univariate time series, although EAPruned is also applicable to multivariate series. We denote series by Q (for a query), C (for a candidate), S, and T. Their length is denoted by L, using subscript such as L_S to disambiguate the lengths of different series. Subscripts C_i are used to distinguish multiple candidates. The elements $s_1, s_2, \ldots s_{L_S}$ are the elements of the series $S = (s_1, s_2, \ldots s_{L_S})$. The element s_i is the i-th element of S with $1 \leq i \leq L_S$.

2.1 Similarity Search and Lower Bounding

Nearest neighbor search is a branch of similarity search, i.e. a search solely relying on the similarity between any pair of objects. It is a common application of elastic distances, and we use it to demonstrate the efficacy of our algorithm. Given a query Q and a set of n candidates $\mathcal{C} = \{C_1, C_2, \ldots C_n\}$, the nearest neighbor of Q under a distance D is a candidate C_{nn} with $d_{nn} = D(Q, C_{nn})$ such that $\forall C \in \mathcal{C}, d_{nn} \leq D(Q, C)$. Algorithm 1 presents this process.

Alg. 1: NN search
$(d_{\mathrm{nn}}, C_{\mathrm{nn}}) \leftarrow (\infty, \emptyset)$
for $C \in \mathcal{C}$ **do**
$\quad$$d \leftarrow D(Q, C)$
$\quad$**if** $d < d_{nn}$ **then**
$\quad\quad$$(d_{\mathrm{nn}}, C_{\mathrm{nn}}) \leftarrow (d, C)$
return $(d_{\mathrm{nn}}, C_{\mathrm{nn}})$

Alg. 2: Lower bounded NN search
$(d_{\mathrm{nn}}, C_{\mathrm{nn}}) \leftarrow (\infty, \emptyset)$
for $C \in \mathcal{C}$ **do**
$\quad$**if** $LB(Q, C) < d_{nn}$ **then**
$\quad\quad$$d \leftarrow D(Q, C)$
$\quad\quad$**if** $d < d_{nn}$ **then**
$\quad\quad\quad$$(d_{\mathrm{nn}}, C_{\mathrm{nn}}) \leftarrow (d, C)$
return $(d_{\mathrm{nn}}, C_{\mathrm{nn}})$

Note that at any point in time during the execution of Algorithm 1, d_{nn} is a monotonously decreasing upper bound on the end result. We will refer to this upper bound as the "cut-off". Any candidate C is discarded if $D(Q, C) \geq d_{\mathrm{nn}}$. Lower bounding exploits this fact to speed up the NN search (Algorithm 2). A lower bound LB of a distance D is an approximation of D such that $\mathrm{LB}(Q, C) \leq D(Q, C)$. $\mathrm{LB}(Q, C) \geq d_{\mathrm{nn}} \models D(Q, C) \geq d_{\mathrm{nn}}$, allowing us to skip its computation as its result will be discarded.

An ideal lower bound is fast (usually in $O(L)$, when distances are in $O(L^2)$) and tight (as close as possible to the actual distance). Lower bounding is shown to significantly speedup NN search in several domains [9, 28, 18]. Lower bounds have mainly been developed for DTW and CDTW, two widely used examples being LB Kim [22] and LB Keogh [9]. They also exist for other elastic distances [28], and remain an active field of research [29].

Another branch of similarity search is range queries, in which we want to find all candidates within a given distance of a given exemplar. Range queries benefit from both lower bounding and early abandoning (see Section 2.2), quickly discarding candidates beyond a cut-off, which in this case is the maximum requested distance.

2.2 Pruning and Early Abandoning

"Pruning" and "early abandoning" are generic concepts. "Pruning" refers to identifying and avoiding unproductive operations. "Early abandoning" refers to abandoning the whole computation as soon as it can be established through an "abandoning criterion" that an exact result is not required. In this paper, we will say that the distance computation algorithms are pruned and early abandoned, meaning that they support pruning and early abandoning. When describing a computation, we will also say that some operations are pruned or early abandoned, meaning that the actual act of pruning or early abandoning is taking place.

A pruned distance computation always returns an exact similarity score, unlike the case for an early abandoned one, which may abandon before determining the similarity score. Moreover, early abandoning requires an abandoning criterion as an extra parameter. Hence, early abandoned distances require special support from their caller whereas pruned distances can be used in place of their

straightforward counterparts. NN search easily provides this support: the abandoning criterion is the same cut-off used by lower bounding (see Section 2.1), and signaling early abandoning by returning ∞ is immediately compatible with the NN search algorithm.

The usual way DTW and other elastic distance computation algorithms have been early abandoned until now [16] is by monitoring the minimal cost of the distance at any point on the current boundary of the computed paths, and abandoning when it exceeds the cut-off. This approach is presented in Algorithm 4, Section 2.4. Of the numerous elastic distance measures, to the best of our knowledge pruning has previously only be developed for DTW (see Section 3) and is explained in Section 4.

2.3   Presentation of Elastic Distances

Many elastic distances share a common form, captured by Equations 1a to 1d.

$$M_{D(S,T)}(0,0) = 0 \tag{1a}$$

$$M_{D(S,T)}(i,0) = \text{InitVBorder} \quad \text{with} \quad M_{D(S,T)}(i,0) \le M_{D(S,T)}(i+1,0) \tag{1b}$$

$$M_{D(S,T)}(0,j) = \text{InitHBorder} \quad \text{with} \quad M_{D(S,T)}(0,i) \le M_{D(S,T)}(0,i+1) \tag{1c}$$

$$M_{D(S,T)}(i,j) = \min \begin{cases} M_{D(S,T)}(i-1,j-1) + \text{Canonical} \\ M_{D(S,T)}(i-1,j) + \text{AlternateRow} \\ M_{D(S,T)}(i,j-1) + \text{AlternateColumn} \end{cases} \tag{1d}$$

Elastic distances following this form include DTW, CDTW, WDTW, ERP, MSM, and TWE, which, when coupled with taking the first derivative of the series [10] before applying these distances (leading to DDTW, DCDTW, and DWDTW), account for nine of the 11 distance measures used by both EE and PF. The 2 remaining distances, LCSS and SQED, do not share the same form, hence do not benefit from our EAPruned approach.

An elastic distance D computes an optimal, minimal, alignment cost $D(S,T)$ between two series S and T by minimizing the cumulative cost of aligning their individual points. A "cost matrix" $M_{D(S,T)}$ is a 0-indexed matrix with dimension $(1 + L_S, 1 + L_T)$ used to carry this computation. A cell $M_{D(S,T)}(i,j)$ represents the minimal cumulative cost of aligning the fist i points of S with the first j points of T. It follows that the cell $M_{D(S,T)}(L_S, L_T)$ holds the cost $D(S,T)$. Equations 1a to 1d give a generic form to compute a cost matrix. An example of a cost matrix for DTW and the corresponding individual point alignments are shown Figure 1. An alignment between the points s_i and t_j is represented by the couple (i,j). In the cost matrix $M_{D(S,T)}$, the successive alignments $(1,1), \ldots (i,j) \ldots (L_S, L_T)$ form a path called the "warping path" (cells with black borders in Figure 1b). Conversely, reading the warping path gives the successive alignment of the individual points. See how the horizontal section of the warping path line 5 in Figure 1b corresponds to the fifth point of S being aligned thrice in Figure 1a.

Equations 1a to 1c initialise the borders. They are monotonously increasing, i.e. $M_{D(S,T)}(i,0) \leq M_{D(S,T)}(i+1,0)$. Equation 1d computes the value of every other cell (i,j) by taking the minimum of three alignment costs plus their respective dependency. The "Canonical" alignment depends on the top left diagonal cell $(i-1,j-1)$ and represents an alignment between two new points. For example, the cell $(3,2)$ in Figure 1b represents t_2 being aligned with s_3 in Figure 1a. The "AlternateRow" alignment depends on the top cell $(i-1,j)$ and represents an alignment between a new point along the lines, and reusing the last aligned point along the columns. For example, the cell $(4,2)$ in Figure 1b represents t_2 being aligned with s_4 in Figure 1a, when t_2 was already aligned with s_3. The "AlternateColumn" alignment depends on the left cell $(i,j-1)$, and is the symmetric of the AlternateRow alignment for the column. Alternate alignments happen either because the canonical alignment is too expensive, or because the series have differing lengths. In several instances, the canonical and alternate costs depend on a "cost" function between s_i and t_j. It usually is the squared Euclidean distance or the L1 norm, but other norms are acceptable.

These generic equations give us the following guarantees.

1. The optimal alignment cost is computed. Note that several optimal warping paths with the same minimal cost may exist.

2. Series extremities are aligned with each other $((1,1)$ and $(L_S, L_T))$. A valid warping path starts from the top left cell and reaches the bottom right cell.

3. The optimal warping path is continuous and monotonous, i.e. for two of its consecutive cells $(i_1,j_1) \neq (i_2,j_2)$, we have $i_1 \leq i_2 \leq i_1 + 1$ and $j_1 \leq j_2 \leq j_1 + 1$. Graphically, no alignment (dotted line in Figure 1a) crosses each other.

2.3.1 Dynamic Time Warping

The Dynamic Time Warping (DTW, see Equations 2a to 2d) distance was first introduced as a speech recognition tool [20]. Compared to the classic Euclidean distance, DTW handles distortion and disparate lengths. Figure 1 illustrates how DTW aligns two series S and T, along with the corresponding warping in the $M_{\text{DTW}}(S,T)$ cost matrix.

$$M_{\text{DTW}}(0,0) = 0 \tag{2a}$$

$$M_{\text{DTW}}(i,0) = +\infty \tag{2b}$$

$$M_{\text{DTW}}(0,j) = +\infty \tag{2c}$$

$$M_{\text{DTW}}(i,j) = \text{cost}(s_i,t_j) + \min \begin{cases} M_{\text{DTW}}(i-1,j-1) \\ M_{\text{DTW}}(i-1,j) \\ M_{\text{DTW}}(i,j-1) \end{cases} \tag{2d}$$

(a) DTW(S,T) alignments with costs.   (b) $M_{\mathrm{DTW}}(S,T)$ cost matrix and warping path.

Figure 1: $M_{\mathrm{DTW}(S,T)}$ with warping path and alignments between the series $S = (3, 1, 4, 4, 1, 1)$ and $T = (1, 3, 2, 1, 2, 2)$.   We have DTW$(S,T) = M_{\mathrm{DTW}(S,T)}(6,6) = 9$.

2.3.2   Constrained DTW

In CDTW, the "constrained" variant of DTW, the warping path is restricted to a subarea of the matrix. Different constraints exist [19, 7]. We focus on the popular Sakoe-Chiba band [19], also known as "Warping Window" (or "window" for short), which also appears in ERP (Section 2.3.4). The window is a parameter w controlling how far the warping path can deviate from the diagonal. Given a matrix M, a line index $1 \leq l \leq L$ and a column index $1 \leq c \leq L$, we have $|l - c| \leq w$. For example with $w = 1$, the warping path can only step one cell away from each side of the diagonal (Figure 2a).

A window of 0 is akin to the squared Euclidean distance (actually is, if the "cost" function between point also is), while a window of L is equivalent to DTW (no constraint). With a correctly set window, NN-CDTW can achieve better accuracy than NN-DTW by preventing spurious alignments. It is also faster to compute than DTW, as cells beyond the window are ignored. However, the window parameter must be set. Finally, not all window size are valid: if the series are of disparate lengths, a window can be too small to allow any alignment (Figure 2b).

2.3.3   Weighted DTW

Weighted Dynamic Time Warping (WDTW, see [8]) imposes a soft constraint on the warping path. The cells (l, c) of the M_{WDTW} cost matrix are weighted according to their distance to the diagonal $d = |l - c|$. A large weight decreases the chances of a cell to be on an optimal path. The weight is computed according to Equation 3. The parameter g controls the penalization, and usually lies within

7

(a) CDTW cost matrix with $w = 1$ for S and T of equal length

(b) CDTW cost matrix with $w = 1$ for S and U of disparate lengths.

Figure 2: Example of CDTW cost matrices with a window of 1. In the second case, the window it too small to allow an alignment between the last two points at $(6, 8)$.

$0.01 - 0.6$ [8].

$$w(d) = \frac{1}{1 + \exp^{-g \times (d - L/2)}} \tag{3}$$

2.3.4 Edit Distance with Real Penalty

The Edit distance with Real Penalty (ERP, see Equations 4a to 4d) is a metric designed as a L1-norm supporting local time shifting, or alternatively as a DTW variant fulfilling the triangular inequality [3]. It is parameterized by a warping window (see Section 2.3.2) and a "gap value" g. Compared to DTW, which reuses a previous point to compute an alternate cost (e.g. in Figure 1, the point 5 of S is used thrice), ERP computes an alternate cost based on g (Equation 4d). This property allows to recover the triangular inequality absent from DTW [3]. ERP requires its borders to be computed (Equations 4b and 4c).

$$M_{\mathrm{ERP}}(0, 0) = 0 \tag{4a}$$

$$M_{\mathrm{ERP}}(i, 0) = M_{\mathrm{ERP}}(i - 1, 0) + \mathrm{cost}(s_i, g) \tag{4b}$$

$$M_{\mathrm{ERP}}(0, j) = M_{\mathrm{ERP}}(0, j - 1) + \mathrm{cost}(g, t_j) \tag{4c}$$

$$M_{\mathrm{ERP}}(i, j) = \min \begin{cases} M_{\mathrm{ERP}}(i - 1, j - 1) + \mathrm{cost}(s_i, t_j) \\ M_{\mathrm{ERP}}(i - 1, j) + \mathrm{cost}(s_i, g) \\ M_{\mathrm{ERP}}(i, j - 1) + \mathrm{cost}(t_i, g) \end{cases} \tag{4d}$$

2.3.5 Move-Split-Merge

Move-Split-Merge (MSM, see Equations 6a to 6d) is a metric developed to overcome shortcomings in other elastic distances [26]. Compared to ERP, it is robust to translation[1]. MSM uses its own cost function C_c to compute the cost of the alternate alignments (Equation 5). It takes as arguments the new point (np) of the alternate alignment, and the two previously considered points (x and y) of each series. MSM is parameterized by a penalty c involved in the computation of C_c. The authors showed that MSM is competitive against DTW, CDTW and ERP for NN classification.

$$
C_c(\text{np}, x, y) = \begin{cases} c & \text{If } x \le \text{np} \le y \text{ or } x \ge \text{np} \ge y \\ c + \min \begin{cases} |\text{np} - x| \\ |\text{np} - y| \end{cases} & \text{otherwise} \end{cases}
\tag{5}
$$

$$
M_{\text{MSM}}(0, 0) = 0
\tag{6a}
$$
$$
M_{\text{MSM}}(i, 0) = +\infty
\tag{6b}
$$
$$
M_{\text{MSM}}(0, j) = +\infty
\tag{6c}
$$
$$
M_{\text{MSM}}(i, j) = \min \begin{cases} M_{\text{MSM}}(i-1, j-1) + |s_i - t_j| \\ M_{\text{MSM}}(i-1, j) + C(s_i, s_{i-1}, t_j) \\ M_{\text{MSM}}(i, j-1) + C(t_j, s_i, t_{j-1}) \end{cases}
\tag{6d}
$$

2.3.6 Time Warp Edit Distance

The Time Warp Edit distance (TWE, see Equations 8a to 8d) was designed to take timestamps, i.e. when a value is recorded, into account [15]. This matters for series with non-uniform sampling rates. The i-th timestamp of a series S is denoted by $\tau_{S,i}$. Our current implementation does not use timestamps, assuming a constant sampling rate, and we always have $\tau_{s,i} = i$. TWE defines its own cost functions (Equations 7) with two parameters. The first one, ν, is a "stiffness" parameter weighting the timestamp contribution ($\nu = 0$ is similar to DTW). The second one, λ, is a constant penalty added to the cost of the alternate alignments ("delete" in TWE terminology — deleteA for the lines, deleteB for the columns). The cost of the alternate case is the cost between the two current points, plus their timestamp difference, plus the λ penalty. The canonical alignment ("match") cost is the sum of the cost between the two current and the two previous points, plus a weighted contribution of their respective timestamps difference.

---

[1]In ERP, the gap cost is given by $\text{cost}(s_i, g)$. If S is translated, the gap cost also changes while the canonical alignment cost remains unchanged, making ERP translation-sensitive.

$$\text{match:} \gamma_M = \text{cost}(s_i, t_j) + \text{cost}(s_{i-1}, t_{j-1}) + \nu(|\tau_{s,i} - \tau_{t,j}| + |\tau_{s,i-1} - \tau_{t,j-1}|)$$
$$\text{deleteA:} \gamma_A = \text{cost}(s_i, s_{i-1}) + \nu|\tau_{s,i} - \tau_{s,i-1}| + \lambda$$
$$\text{deleteB:} \gamma_B = \text{cost}(t_j, t_{j-1}) + \nu|\tau_{t,j} - \tau_{t,j-1}| + \lambda \tag{7}$$

$$M_{\text{TWE}}(0,0) = 0 \tag{8a}$$
$$M_{\text{TWE}}(i,0) = +\infty \tag{8b}$$
$$M_{\text{TWE}}(0,j) = +\infty \tag{8c}$$
$$M_{\text{TWE}}(i,j) = \min \begin{cases} M_{\text{TWE}}(i-1, j-1) + \gamma_M \\ M_{\text{TWE}}(i-1, j) + \gamma_A \\ M_{\text{TWE}}(i, j-1) + \gamma_B \end{cases} \tag{8d}$$

2.4 Algorithms for the Common Structure

All the distances presented in Section 2.3 share the structure captured by Equations 1a to 1d. These equations are implemented by Algorithm 3, with a $O(L)$ space complexity. Indeed, a cell (i,j) only depends on the previous row (at $(i-1, j-1)$ or $(i-1, j)$), or on its left neighbor in the current row (at $(i, j-1)$). It follows that a row by row implementation only requires two rows of the matrix at any time, achieving linear space complexity. Using the shortest series along the columns minimizes the required row length, further reducing the memory footprint.

Alg. 3: Generic linear space complexity for a distance D.

    Input: the time series S and T
    Result: Cost $D(S, T)$
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 $l_{co} + 1$
4   curr[0] $\leftarrow$ 0
5   curr[$1 — L$] $\leftarrow$ InitHBorder
6   **for** $i \leftarrow 1$ **to** L_{li} **do**
7   swap(prev, curr)
8   curr[0] $\leftarrow$ InitVBorder
9   **for** $j \leftarrow 1$ **to** L_{co} **do**
10   curr[j] $\leftarrow \min \begin{cases} \text{prev}[j\text{-}1] & + \text{Canonical} \\ \text{prev}[j] & + \text{AlternateRow} \\ \text{curr}[j\text{-}1] & + \text{AlternateColumn} \end{cases}$
11 **return** curr[l_{co}]

Two arrays represent the current row (curr) and the previous row (prev). The arrays are swapped at each iteration of the outer loop (line 7), the current row

becoming the previous row, and the array formerly used for the previous row being assigned for use as the new current row. Initially, the horizontal border is stored in the curr array (line 5). After the first swap, it will be in prev, acting as the previous row for the first line. The vertical border is gradually computed for each new row (line 8). Finally, the inner loop computes from left to right the value of the cells of the current row (line 9).

Algorithm 4 builds upon Algorithm 3, adding support for a warping window w and the usual early abandoning technique: it monitors the boundary of the current pass through the cost matrix and abandons when all values on the boundary exceed a cut-off value. In this case the boundary is the current row. Thus, after each row is done, the algorithm looks at the minimum value of the row and abandons if it is above the cut-off.

Alg. 4: Generic distance with window and early abandoning.

Input: the time series S and T, a warping window w, a cut-off value cutoff
Result: Cost $D(S,T)$

1 co $\leftarrow$ shortest series between S and T
2 li $\leftarrow$ longest series between S and T
3 **if** $w < L_{\text{li}} - L_{\text{co}}$ **then return** $+\infty$
4 (prev, curr) $\leftarrow$ arrays of length $L_{\text{co}} + 1$ filled with $+\infty$
5 curr[0] $\leftarrow 0$
6 curr[$1 - w+1$] $\leftarrow$ InitHBorder
7 **for** $i \leftarrow 1$ **to** L_{li} **do**
8   swap(prev, curr)
9   jStart $\leftarrow \max(1, i - w)$
10   jStop $\leftarrow \min(i + w, L_{\text{co}})$
11   curr[jStart $- 1$] $\leftarrow$ **if** jStart $== 1$ **then** InitVBorder **else** $+\infty$
12   minv $\leftarrow +\infty$
13   **for** $j \leftarrow$ jStart **to** jStop **do**
14         $v \leftarrow \min \begin{cases} \text{prev[}j\text{-1]} & + \text{Canonical} \\ \text{prev[}j\text{]} & + \text{AlternateRow} \\ \text{curr[}j\text{-1]} & + \text{AlternateColumn} \end{cases}$
15   minv $\leftarrow \min(\text{minv}, v)$
16   curr[j] $\leftarrow v$
17   **if** minv $>$ cutoff **then return** $+\infty$
18 **return** curr[L_{co}]

This algorithm also shows how to handle a window w. It first checks whether w permits an alignment or not (line 3). The horizontal border is only initialized up to $w + 1$ (line 6), and the inner loop is caped within the window around the diagonal (from jStart to jStop, lines 9, 10 and 13). The vertical border is only computed while the window covers the first column (line 11). More interestingly, the arrays are now initialized to $+\infty$. To understand why, let us consider the cell $(2,3)$ for M_{CDTW} in Figure 2a. It depends on $(1,3)$ (AlternateRow case), which is outside the window. By initializing the arrays to $+\infty$, we implicitly set this cell, and all the upper right triangle outside of the window, to $+\infty$. The

lower triangle is implicitly set to $+\infty$ line 11, after the window stops covering the first column. This explains why we assign to curr[jStart − 1] and not to curr[θ]. Only the diagonal edge of the triangle is set to $+\infty$, which is enough for the next cell's AlternateColumn case to be properly ignored (e.g. in Figure 2a the cell $(3, 2)$ ignores $(3, 1)$).

3   Related Work

Most previous work on speeding up NN search under elastic distances has focused on improving lower bounds for CDTW (and DTW, which is a special case of CDTW) [22, 9, 16, 28, 29]. Another approach to directly speeding up elastic distances is through approximation. To our knowledge, this has only been done for CDTW with FastDTW (see [21], and [30] for a counterpoint). Our approach computes exact CDTW (and other distances, unless early abandoned), which is useful when approximation is not desirable.

Pruning was first developed for CDTW in 2016 with PrunedDTW [24]. It aimed at speeding up all pairwise calculations of exact DTW when early abandoning is not an option [31]. It was subsequently extended to incorporate early abandoning [25] using the classical early abandoning technique of monitoring the boundary of the progress through the cost matrix until the minimum value on the boundary exceeds the cut-off (similar to Algorithm 4). Both PrunedDTW and its early abandoned version tighten the cut-off during the computation, which we do not cover in this paper.

EAPruned uses the same overarching pruning strategy as PrunedDTW. Hence, we will first present EAPruned before giving a comparison with PrunedDTW (Section 4.4). The main differences are that EAPruned tightly integrates pruning and abandoning through a strategy of abandoning when pruning leaves no path through the matrix open, and reduces computational effort by organizing the computation in specialized stages, pruning in a more efficient way. Finally, EAPruned generalizes to the class of distances encompassed by Equations 1a to 1d.

4   The EAPruned technique

We first present EAPruned's two strategies for pruning, "Pruning from the left" and "Pruning on the right". We then present the EAPruned algorithm (Algorithm 5) and finally present the PrunedDTW technique in the light of EAPruned. Note that when we illustrate EAPruned, we use DTW with a warping window for the sake of simplicity.

EAPruned utilizes a cut-off value, such as usually provided by the similarity search process (Section 2.1). If a cut-off value is not applicable, EAPruned can be used as a prune-only algorithm, just like PrunedDTW, using an upper bound based on the diagonal of the cost matrix (e.g. the squared Euclidean distance in the case of DTW). Note that any warping path through the cost matrix can

(a) $M_{\mathrm{DTW}(S,T)}$ with cutoff $= 9$.

(b) $M_{\mathrm{DTW}(S,T)}$ with cutoff $= 6$.

Figure 3: Illustration of $M_{\mathrm{DTW}(S,T)}$ "from the left" pruning with two different cut-off values, the second one leading to early abandoning. The arrows represent the dependencies of a cell.

be used as an upper bound: either it is an optimal path, or an optimal path will have a lower cost. Such an upper bound allows pruning (some cells of the cost matrix won't be computed) but not early abandoning (allowing at least the corresponding path to be computed).

4.1 Pruning From the Left

While computing the cost matrix row by row, from left to right, we look at discarding as many as possible of its leftmost cells. We define the "discard zone" as the "discarded cells" topped by a "discard point", i.e. the cells (i, j) such that $\exists_{1 \le k < i}$ such that $(i - k, j)$ is a discard point. In the ith row, discard points are all cells $(i, l), \ldots, (i, k)$ such that $\forall_{j \in [l,k]}, M_{D(S,T)}(i, j) > $ cutoff and $M_{D(S,T)}(i, k + 1) \le$ cutoff, where (i, l) is the leftmost non discarded cell in the row, and cutoff is the cut-off value. In Figure 3, the white cells with red borders are discard points, and the discard zone is made of the grey cells below them. If $(i, k + 1)$ is out of bound, the computation is early abandoned (Figure 3b).

Only the discard points are computed: the discarded cells are pruned (i.e. never computed). Pruning is enabled by the fact that these cells ultimately depend only on discard points, which by definition are above the cut-off. Because a cell can only have a cost greater than (or equal to) its smallest dependency, cells in the discard zone can only have a cost greater than the cut-off, hence can be ignored. As an example, take the dependencies of the cell $(4, 1)$ in Figure 3a. Starting on the left border, a cell only has a top dependency. Hence, as soon as a border cell is above the cut-off (i.e. $(0, 1)$), the remainder of the column can be discarded. In turn, this creates the same border-like conditions for the next column, starting at the next row. We only need to check the top dependency for cells $(i > 1, 1)$, stopping as soon as possible. The cell $(3, 1)$ is above the

13

(a) $M_{\text{DTW}(S,T)}$ with cutoff $= 9$ (b) $M_{\text{DTW}(S,T)}$ with cutoff $= 6$

Figure 4: Illustration of $M_{\text{DTW}(S,T)}$ "from the left" and "on the right" pruning with two different cut-off values. The blue cell represents the point of early abandoning, white cells represent discard points, and dark gray cells represent pruning points.

cut-off, allowing to discard the remainder of the column, including $(4, 1)$.

When proceeding row by row, pruning from the left is implemented by starting the next row after the discarded columns. Discarding all the columns in a row, such as in Figure 3b, leads to early abandoning. Note that in this case, the discard points from $(5, 3)$ up to $(5, 6)$ need to check both their top and diagonal dependencies. Most distances have their borders initialized to ∞, starting the discard zone at $(1, 0)$ on the left border. For computed borders, the discard zone starts at the first line i such that $M_{D(S,T)}(i, 0) > $ cutoff. If a window w is used, the discard zone starts at the line $w + 1$, unless started earlier (see Algorithm 5 line 14).

4.2 Pruning on the Right

In the previous section, we extended a discard zone from the left border to the right, relying on a row by row evaluation order. The same is not applicable to the top border, as this would require a column by column evaluation order. However, we can still find further cells such that all their dependencies are above the cut-off. Looking back at Figure 3b, the cell $(1, 4)$ is above the cut-off of 6. In conjunction with the top border, all the following cells in the row only depend on cells above the cut-off, hence can be pruned. Doing so allows us to "prune on the right".

In the previous section, we mentioned that a discard point creates the "same border-like conditions for the next column, starting at the next row", perpetuating the pruning process. Similarly, we have to identify a condition akin to the top border. This condition is a continuous block of cells above the cut-off value, reaching the end of their row, i.e. in the ith row $\exists_{1 \le p_i \le L}, \forall_{p_i \le j \le L}, (i, j) \le$ cutoff,

with L being the length of the rows. The start p_i of such a block is called a "pruning point". In Figure 4b, pruning points are represented in dark grey with red borders. The first one is located on the top border at $(0, 1)$, another one is at $(1, 4)$. Pruning points provide information for the next row[2]. On the next row, as soon as a cell located after the pruning point is above the cut-off, the remainder of the row can be pruned, i.e. in the ith row, all the cells $(i, j > c)$ such that $\exists_{c > p_{i-1}}, M_{D(S,T)}(i, c) > \mathsf{cutoff}$. Note that the cell just under the pruning point is not pruned because of its diagonal dependency (e.g the cell $(1, 1)$ Figure 4b).

Pruning points move back and forth across rows. To determine their position, a cell (i, j) below the cut-off will always assume that the next cell is the pruning point of the row. If $(i, j + 1) > \mathsf{cutoff}$, and so on up to (i, L_{co}), then $(i, j + 1)$ indeed is a pruning point. Else, we assume that $(i, j + 2)$ is the pruning point, repeating the same logic. Note that there is only one and only one pruning point, although it may be out of bounds at $L_{\mathsf{co}} + 1$, in which case it does not appear in the figures. Let us look at some examples in Figure 4b. The cell $(2, 2) > \mathsf{cutoff}$ is not a pruning point because some following cells are below the cut-off. The cell $(3, 3)$ and all its following cells are above the cut-off: it is a pruning point. Note that the row must be fully evaluated to ensure that $(3, 3)$ indeed is the row's pruning point. It contributes to enabling pruning on the fourth row, starting at $(4, 4)$.

Finally, we have to address what happens when both pruning strategies "collide" (Figure 4b, in blue). Without pruning on the right, the cell $(5, 3)$ would be a discard point. Because it is below a pruning point, we know that the rest of the row is above the cut-off, meaning that the full row is pruned, leading to early abandoning. If we evaluate a technique by the number of saved computation, EAPruned saves 16 cells over 36 while the usual early abandoning strategy (Algorithm 4) only saves the last line, i.e. 6 cells. Also, if the cell at $(1, 1)$ is above the cut-off, EAPruned immediately early abandons while Algorithm 4 still evaluates the full row. Finally, even with a cut-off of 9 preventing from early abandoning, EAPruned still allows to save the computation of 5 cells (Figure 4a).

4.3   The EAPruned Algorithm

We now present the EAPruned algorithm (Algorithm 5), which is applicable to any distance matching the common structure described in Section 2.3. We present the algorithm with a window and computed borders, covering the most complex case. EAPruned exploits resulting properties of pruning to further reduce the computational effort. For example, cells after the pruning point of the previous row can ignore their top and diagonal dependencies as they are known to be above the cut-off. To do so, we split the computation of a row in several stages:

---

[2]Similar to discard points, instructing the next rows to skip their column.

1. Compute the value of the left border (computed or outside the window). A computed border may require the top dependency.

2. Compute discard points until a non-discard point or the pruning point. Depends on the top and diagonal cells.

3. Compute non-discard point until the pruning point. Depends on the top, diagonal and left cells.

4. Deal with the cell at the pruning point. Depends on the diagonal (always) and left (unless was a discard point) cells.

5. Compute the cells after the pruning point. Depends on the left cell.

In Algorithm 5, discard points are represented by the next_start variable. The pruning point from the previous row, used to prune in the current row, is represented by the pruning_point variable. Finally, the next_pruning_point variable holds the pruning point being currently computed (it is assigned to pruning_point after a row is done, line 39).

Beginning a new row, we first determine the index of the first and last columns. Then, the first stage updates the left border, computing its value or applying the window (line 14). The second stage (line 16) computes discard points, which require the row to be bordered on the left by a value above the cut-off (tested line 17). The condition in the for loop ensures that all discard points form a continuous block. As soon as a value below the cut-off is found, we jump to the second stage as we cannot have any more discard points. As explained in the previous section, next_pruning_point is set (line 20) to the next column index. Note that next_start can only be updated in the second stages while next_pruning_point must be checked for update after every cost computation.

The third stage (line 21) computes the cost taking all dependencies into account until we reach pruning_point. If pruning_point is out of bounds, the third stage completes the row and the following stages are skipped over. We enter the fourth stage (line 25). If the row is not done yet, we check if the left cell is a discard point or not. If it is (line 27), then we only need to check the diagonal dependency, early abandoning if the resulting cost is above the cut-off. It not (line 30), both the left and diagonal dependencies are checked. Finally, we early abandon if the discard points reached the end of the row (line 34).

At the fifth and final stage (line 35), only the left dependency is checked as, both the diagonal and top dependencies are known to be above the cut-off. The loop stops as soon as we find a cost above the cut-off, pruning the rest of the row.

4.3.1 On the Complexity of EAPruned

Our experiments show that EAPruned achieves significant speed up in several similarity search tasks (see Section 5). As it only allocates two buffers of length

Alg. 5: Generic EAPruned Algorithm.

Input: the time series S and T, a warping window w, a cut-off point cutoff
Result: Cost $D(S,T)$ or ∞ if early abandoned

1   co $\leftarrow$ shortest series between S and T
2   li $\leftarrow$ longest series between S and T
3   **if** $w < L_{\texttt{li}} - L_{\texttt{co}}$ **then return** $+\infty$
4   (prev, curr) $\leftarrow$ arrays of length $L_{\texttt{co}} + 1$ filled with $+\infty$
5   curr$[0] \leftarrow 0$
6   curr$[1 - w{+}1] \leftarrow$ InitHBorder
7   next_start $\leftarrow 1$
8   pruning_point $\leftarrow 1$
9   **for** $i \leftarrow 1$ **to** $L_{\texttt{li}}$ **do**
10   swap(prev, curr)
11   jStart $\leftarrow \max(i - w, \text{next\_start})$
12   jStop $\leftarrow \min(i + w, L_{\texttt{co}})$
13    $j \leftarrow$ jStart
14   /* Stage 1: init the vertical border   */
15   curr[jStart $- 1] \leftarrow$ **if** jStart $== 1$ **then** InitVBorder **else** $+\infty$
16   /* Stage 2: discard points up to, excluding, the pruning point   */
17   **if** curr[jStart-$1] >$ cutoff **then**
18   **for** j **to** pruning_point $- 1$ **while** $j =$ next_start **do**
19   curr$[j] \leftarrow \min \begin{cases} \text{prev}[j\text{-}1] & + \text{Canonical} \\ \text{prev}[j] & + \text{AlternateRow} \end{cases}$
20   **if** curr$[j] \leq$ cutoff **then** next_pruning_point $\leftarrow j + 1$ **else** next_start++
21   /* Stage 3: continue up to, excluding, the pruning point   */
22   **for** j **to** pruning_point $- 1$ **do**
23   curr$[j] \leftarrow \min \begin{cases} \text{prev}[j\text{-}1] & + \text{Canonical} \\ \text{prev}[j] & + \text{AlternateRow} \\ \text{curr}[j\text{-}1] & + \text{AlternateColumn} \end{cases}$
24   **if** curr$[j] \leq$ cutoff **then** next_pruning_point $\leftarrow j + 1$
25   /* Stage 4: at the pruning point   */
26   **if** $j \leq$ jStop **then**
27   **if** $j =$ next_start **then**
28   curr$[j] \leftarrow$ prev$[j\text{-}1] +$ Canonical
29   **if** curr$[j] \leq$ cutoff **then** next_pruning_point $\leftarrow j + 1$ **else return** $+\infty$
30   **else**
31   curr$[j] \leftarrow \min \begin{cases} \text{prev}[j\text{-}1] & + \text{Canonical} \\ \text{curr}[j\text{-}1] & + \text{AlternateColumn} \end{cases}$
32   **if** curr$[j] \leq$ cutoff **then** next_pruning_point $\leftarrow j + 1$
33   j++
34   **else if** $j =$ next_start **then return** $+\infty$
35   /* Stage 5: after the pruning point   */
36   **for** j **to** jStop **while** $j =$ next_pruning_point **do**
37   curr$[j] \leftarrow$ curr$[j\text{-}1] +$ AlternateColumn
38   **if** curr$[j] \leq$ cutoff **then** next_pruning_point $\leftarrow j + 1$
39   pruning_point $\leftarrow$ next_pruning_point
40   **return** curr$[L_{\texttt{co}}]$

L, its space complexity is in $O(L)$. The time complexity depends on the cut-off; at worst, it remains in $O(L^2)$, at best, it can be as low as $O(1)$ (e.g. in the case of DTW which does not initialize its buffers) or $O(L)$ (with buffer initialization). This is due to the unpredictable nature of early abandoning under a cut-off. If the provided cut-off never allows to prune and early abandon (i.e. it is too high), the full cost matrix will be computed, resulting in a quadratic complexity. In this case, it is a "worse" quadratic complexity than the one from Base due to EAPruned's additional overheads.

Hence, time complexity under early abandoning is a moving target, sitting between the best and worst case scenarios. When performing a NN classification, the speed up will depends on the order of evaluation. If the first candidate is the actual nearest neighbor of a query, the average complexity will be close to $O(1)$ (or $O(L)$ with buffer initialization) for all following distance computations. On the other hand, if the candidates are ordered from the furthest to the nearest of a given query, the computation will never be early abandoned, and probably not pruned much. The average complexity will then be closer to $O(L^2)$.

4.4 Comparison with The PrunedDTW algorithm

PrunedDTW was not designed with early abandoning in mind, and hence never considered the case of a cutoff preventing any alignment. Furthermore, its extension with early abandoning [25] did not consider pruning in that case either, instead early abandoning the classical manner (as done in Algorithm 4). Algorithm 6 presents the early abandoned version of PrunedDTW. This pseudo code is based on the implementation rather than the pseudo code from [25] that differs from the implementation and appears to be incomplete. Note that Algorithm 6 uses cutoff$'$, a tightened cut-off value based on the original cutoff and on an array stored during the lower bounding process representing a lower bound on the remaining alignment for each row of the matrix ("cumLB"). cutoff$'$ is tightened several times during computation (lines 3 and 31). We refer the reader to [25] for an explanation of the technique.

EAPruned uses the same overarching strategy as PrunedDTW, which can be tracked by following some shared variable names (e.g. next_start and pruning_point). However, EAPruned has the ability to prune more cells than Pruned-DTW. PrunedDTW in essence misses all the stage 4 pruning from Algorithm 5. It only checks what happens after the pruning point (line 25), and not at the pruning point, instead relying on the minimum value in the line to early abandon (lines 21 and 32). More importantly, PrunedDTW only uses the discard points to determine the start of the next line (line 8). By using this information earlier (computed at stage 2, used at stage 4), EAPruned is also able to early abandon earlier.

EAPruned not only prunes more cells, it also does so in a much more efficient way thanks to its staged approach (Section 5.2). PrunedDTW always checks all the dependencies of a cell when computing its cost (starting at line 16). Doing so requires "sanitizing" every access in order to exclude pruned cells. Hence, it not only tests unnecessary dependencies, it actually spends time to do so. This

Alg. 6: Early Abandoned PrunedDTW Algorithm.

Input: the time series S and T of length L, a warping window w, a cut-off point
cutoff, cumulative lower bound values cumLB

Result: Cost $\text{DTW}(S,T)$ or ∞ if early abandoned

1 $(\text{next\_start}, \text{pruning\_point}) \leftarrow (0, 0)$

2 $(\text{prev}, \text{curr}) \leftarrow$ arrays of length L filled with $+\infty$

3 $\text{cutoff}' \leftarrow \text{cutoff} - \text{cumLB}[w + 1]$ `// cutoff tightening`

4 **for** $i \leftarrow 0$ **to** $L - 1$ **do**

5 $\text{min\_cost} \leftarrow \infty$

6 $(\text{foundSC}, \text{prunedEC}) \leftarrow (\text{False}, \text{False})$

7 $\text{next\_pruning\_point} \leftarrow i + w + 1$

8 $\text{jStart} \leftarrow \max(0, \text{next\_start}, i - w)$

9   **for** $j \leftarrow \text{jStart}$ **to** $\min(i + w, L - 1)$ **do**

10   `/* First cell in the cumulative matrix`   `*/`

11   **if** $i = 0 \wedge j = 0$ **then**

12 $\text{curr}[0] \leftarrow \text{cost}(s_i, t_j)$

13 $\text{min\_cost} \leftarrow \text{curr}[0]$

14 $\text{foundSC} \leftarrow \text{True}$

15   **continue**

16   `/* Compute cost excluding invalid cells`   `*/`

17   **if** $j = \text{jStart}$ **then** $y \leftarrow \infty$ **else** $y \leftarrow \text{curr}[j - 1]$

18   **if** $i = 0 \vee j = i + w \vee j \geq \text{last\_pruning}$ **then** $x \leftarrow \infty$ **else** $x \leftarrow \text{prev}[j]$

19   **if** $i = 0 \vee j = 0 \vee j > \text{last\_pruning}$ **then** $z \leftarrow \infty$ **else** $z \leftarrow \text{prev}[j - 1]$

20 $\text{curr}[j] \leftarrow \text{cost}(s_i, t_j) + \min(x, y, z)$

21 $\text{min\_cost} \leftarrow \min(\text{min\_cost}, \text{curr}[j])$

22   `/* Pruning criteria`   `*/`

23   **if** $\text{foundSC} = \text{False} \wedge \text{curr}[j] \leq \text{cutoff}'$ **then** $(\text{next\_start}, \text{foundSC}) \leftarrow (j, \text{True})$

24   **if** $\text{curr}[j] > \text{cutoff}'$ **then**

25   **if** $j > \text{pruning\_point}$ **then**

26 $(\text{last\_pruning}, \text{prunedEC}) \leftarrow (j, \text{True})$

27   **break**

28   **else** $\text{next\_pruning\_point} \leftarrow j + 1$

29   `/* End of inner for loop - Early abandoning and updates`   `*/`

30   **if** $i + w < L - 1$ **then**

31 $\text{cutoff}' \leftarrow \text{cutoff} - \text{cumLB}[i + w + 1]$

32   **if** $\text{min\_cost} > \text{cutoff}'$ **then return** ∞

33 $\text{swap}(\text{curr}, \text{prev})$

34   **if** $\text{next\_start} > 0$ **then** $\text{prev}[\text{next\_start} - 1] \leftarrow \infty$

35   **if** $\text{prunedEC} = \text{False}$ **then** $\text{last\_pruning} \leftarrow i + w + 1$

36 $\text{pruning\_point} \leftarrow \text{next\_pruning\_point}$

37 `/* Check if the last row was pruned before returning`   `*/`

38 **if** $\text{prunedEC} = \text{True}$ **then** $\text{curr}[j] \leftarrow \infty$

39 **return** $\text{curr}[L]$

has a significant impact in a loop body literally executed billions of times across our benchmark.

EAPruned is also more direct and arguably simpler, getting rid of more than half the temporary variables. EAPruned was first developed for DTW. Its clean structure naturally leads to a the generalized version presented in this paper. To the best of our knowledge no generalisation of PrunedDTW to other distance measures has been proposed.

5   Experiments

We evaluate EAPruned in the context of NN search, which naturally supports early abandoned distances. In this context, all the presented implementations of a distance produce the exact same NN search results. Hence, our experiments are about execution speed (and not, e.g., accuracy). All our experiments have been run under the same conditions, on a computer equipped with 64GB of RAM (enough memory to fit all the relevant data) and an AMD Opteron 6338P at 2.4Ghz. The C++ source code for all the implementations (including PrunedDTW) is available on github [5].

Our 3 experiments evaluate "EAPruned" (Algorithm 5) distance implementations against several others: "Base" are the classical double buffered implementations, without early abandoning (Algorithm 3); "EABase", are Base implementations with the usual early abandoning technique (Algorithm 4). "Prune" are the same as "EAPruned", but always use their own computed cut-off based on the diagonal of the cost matrix (only allowing to prune, not to early abandon); Finally, "PrunedDTW" and "PrunedDTW+EA" are the reference C++ implementations from [24] and [25]. Our experiments provide the information required by "PrunedDTW+EA" to tighten the cut-off (Algorithm 6).

5.1   Evaluation Under NN Classification

We compare the timings of the different distance implementations under NN classification. We use the default train/test splits of the 85 datasets from the UCR Archive [4]. The Figure 5 presents the results in hours per distance. We use realistic parameters for the distances, using the one found by EE for each dataset. This is particularly relevant for distances with a warping window, as small windows (which is the most common case) greatly impact the runtime, e.g. CDTW Base is ≈ 12 times faster than DTW Base (Figures 5c and 5b).

EAPruned is overall the most efficient implementation (Figure 5a), being ≈ 7.61 times faster than Base, and ≈ 2.86 times faster than EABase. This is confirmed when looking at each individual distances (Figure 5), with a speedup ranging from ≈ 3.93 (TWE) up to ≈ 39.23 (WDTW) compared to Base, and from ≈ 1.85 (TWE) up to ≈ 8.44 (WDTW) compared to EABase. Note that only EAPruned achieves a speed up for CDTW compared to the Base version. Looking at pruning only scenarios (e.g. when all pairwise distance computations

Figure 5: Accumulated timings in hours of NN classification over 85 datasets from the UCR archive, using parameters discovered by EE.

21

Dataset	Base	Pruned	EABase	EAPruned
NonInvasiveFetalECGThorax2	16.72	9.53	3.81	0.38
NonInvasiveFetalECGThorax1	19.23	7.34	3.87	0.36
HandOutlines	19.82	10.05	6.89	1.33
UWaveGestureLibraryAll	21.05	17.15	8.44	3.47
StarLightCurves	63.01	47.63	19.58	7.62
total	139.83	91.70	42.59	13.16

Table 1: The 5 slowest datasets, timings in hours.

Dataset	Base	Pruned	EABase	EAPruned
ItalyPowerDemand	0.029	0.024	0.015	0.012
Coffee	0.035	0.029	0.032	0.017
SonyAIBORobotSurface1	0.051	0.028	0.037	0.016
BirdChicken	0.060	0.065	0.056	0.030
BeetleFly	0.060	0.050	0.046	0.032
total	0.235	0.195	0.186	0.107

Table 2: The 5 fastet datasets, timings in minutes.

are required), always choosing Prune over Base may not be the best choice. If Pruned is overall ahead of Base, this is not the case for CDTW and TWE (Figures 5c and 5g).

We presented the timings for 85 datasets. However, the majority of the computation time comes from the slowest datasets. The 5 slowest datasets (5.8% of the datasets) make up for $\approx 72.5\%$ of the total Base time (Table 1, in hours). Slower datasets have long series, favouring EAPruned, e.g. StarLightCurves contains 1000 train series and 8236 test series of length 1024. In this circumstances, EAPruned achieves a greater speedup over Base (≈ 10.6) and EABase (≈ 3.23).

EAPruned remains beneficial when looking at the 5 fastest datasets (Table 2, in minutes), albeit with a smaller speedup (≈ 2.19 for Base, ≈ 1.73 for EABase). This comes at no surprise as datasets with smaller series (ItalyPowerDemand contains 67 train series and 1029 test series of length 24) offers less pruning opportunities, hence less chances for EAPruned to make up for its overhead. It nonetheless is the fastest implementation for all the dataset in the archive.

5.2 Evaluation Under NN Classification with Lower Bounds

The previous experiment evaluates NN classification only using the distances. However, most NN classification scenarios are sped up using lower bounding. Under these circumstances, is EAPruned still beneficial? In the following experiment, we focus on DTW and CDTW for which efficient lower bounds exist. We repeat the previous experiment for CDTW and DTW (Figures 6 and 7) without lower bound ("lb-none"), the LB-Keogh lower bound ("lb-keogh"), and

Figure 6: Comparison of NN timings in hours of various CDTW implementations over the UCR Archive, under various lower bounds. Window parameters obtained from EE.

Figure 7: Comparison of NN timings in hours of various DTW implementations over the UCR Archive, under various lower bounds.

cascading two applications of LB-Keogh ("lb-keogh2", reversing their arguments as $Keogh(a, b) \neq Keogh(b, a)$, see [16]). Note that LB-Keogh requires to compute the envelopes of the series. For a given run, envelopes are only computed once, in $O(L)$ using Lemire's algorithm [11], keeping this overhead to a minimum. We also use the occasion to compare EAPruned with "PrunedDTW" and "PrunedDTW+EA".

In the DTW case, EAPruned is more than 4 times faster than the EABase with lb-keogh2, one of the fastest configurations known until now. Lower bounding EAPruned still offers $\approx 10\%$ speedup, which is interesting as envelopes computed over a window as wide as the series does not contain much information (see how the CDTW benefits way more from lower bounding in the Base and EABase cases than DTW). In the CDTW case, EAPruned without lower bounding is on par with Base and EABase with lower bounds. Without lower bounds, PrunedDTW+EA comes second, but loses its advantage in other cases. PrunedDTW is actually slower than the Base version. It is also slower than our Pruned version even though we do not tighten the cut-off during computation. Lower bounding EAPruned provides a further ≈ 2 times speed up.

Our results indicate that lower bounding — at least with lb-Keogh — complements EAPruned.

FoG

Proportion pruned by:
lb_Kim:	30.49%
lb_Keogh1:	13.19%
lb_Keogh2:	3.21%
Computed by DTW:	53.11%
Ref length:	1,724,584

Soccer

Proportion pruned by:
lb_Kim:	74.55%
lb_Keogh1:	11.28%
lb_Keogh2:	7.71%
Computed by DTW:	6.46%
Ref length:	1,998,606

PAMAP2

Proportion pruned by:
lb_Kim:	96.87%
lb_Keogh1:	1.30%
lb_Keogh2:	0.27%
Computed by DTW:	1.56%
Ref length:	3,657,033

ECG

Proportion pruned by:
lb_Kim:	59.20%
lb_Keogh1:	24.18%
lb_Keogh2:	8.00%
Computed by DTW:	8.62%
Ref length:	27,950,000

REFIT

Proportion pruned by:
lb_Kim:	88.70%
lb_Keogh1:	9.06%
lb_Keogh2:	0.48%
Computed by DTW:	1.76%
Ref length:	78,596,631

PPG

Proportion pruned by:
lb_Kim:	67.99%
lb_Keogh1:	8.87%
lb_Keogh2:	3.05%
Computed by DTW:	20.09%
Ref length:	333,570,000

Legend: UCR — UCR_USP — UCR+EAPruned

Figure 8: Comparison of subsequence search time, in second, between the UCR Suite, the UCR-USP Suite, and the UCR Suite using EAPruned, when increasing the length of the queries.

5.3 Sub-sequence Search

The two previous experiments present NN classification results, which are an application of NN search. Another kind of similarity search is sub-sequence search. Given a query, sub-sequence search locates the closest match (in our case, under DTW) within another, usually very long, reference series. The UCR suite [18] is the first scalable tool for the task, deploying several optimisations, including a custom early abandoned DTW. It later evolved into the UCR-USP suite, including "PrunedDTW+EA" [25]. We replaced the DTW implementation of the UCR suite with our own EAPruned DTW, and replicated the experiments from [25]. We refer the reader to the original paper for an in-depth presentation of the technique and the datasets.

The results are presented in Figure 8, in seconds. For each dataset, we show the length of the reference series, and how much DTW computations are saved using three increasingly tighter lower bounds. The version using our EAPruned implementation is always the fastest. Moreover, it scales better with long queries. Overall, the UCR Suite took 4153309 seconds (48 days, 1h:41m49s) to complete, the UCR-USP Suite took 963251 seconds (11 days, 3h34m11s), and UCR+EAPruned took 473150 seconds (5 days, 11h25m50s), providing the fastest known implementation for this task.

24

6 Conclusion

EAPruned is an efficient algorithm for computing elastic distances that efficiently integrates pruning and early abandoning. We implemented EAPruned for six key elastic distances used by state-of-the-art ensemble classifiers, and compared the timings with existing techniques.

We show experimentally, using the standard UCR archive, that EAPruned supports the fastest known NN classifiers. Not only is EAPruned alone competitive against other techniques using lower bounds, it further benefits from them. We also show that pruning alone can be beneficial for some distances, allowing the algorithm to be applied productively even when early abandoning is not applicable. Caution is advised, however, as the overheads of pruning may exceed the benefits in some cases. Finally, we show that our algorithm can be successfully applied to other instances of similarity search, like sub-sequence search where its application leads to the fastest known tool in its class.

In light of these results, we encourage researchers to keep developing lower bounds, the two technique being complementary. We also encourage practitioners to use our algorithm. We make the latter easy by releasing the Tempo library [6], providing our C++ implementations with Python/Numpy bindings under the permissive BSD-3 license.

Our next step will be to implement our algorithm for the multivariate case. We also plan to fit some ensemble classifiers such as Proximity Forest or TSChief with our distances, expecting significant speed up.

References

[1] Anthony Bagnall, Michael Flynn, James Large, Jason Lines, and Matthew Middlehurst. A tale of two toolkits, report the third: On the usage and performance of HIVE-COTE v1.0. *arXiv:2004.06069 [cs, stat]*, April 2020.

[2] Seif-Eddine Benkabou, Khalid Benabdeslem, and Bruno Canitia. Unsupervised outlier detection for time series by entropy and dynamic time warping. *Knowledge and Information Systems*, 54(2):463–486, 2018.

[3] Lei Chen and Raymond Ng. On the marriage of lp-norms and edit distance. In *Proceedings 2004 VLDB Conference*, pages 792 – 803, 2004.

[4] Hoang Anh Dau, Anthony Bagnall, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, and Eamonn Keogh. The UCR Time Series Archive. *arXiv:1810.07758 [cs, stat]*, September 2019.

[5] Matthieu Herrmann. Experimentation source code and ressources. `https://github.com/HerrmannM/paper-2021-EAPElasticDist`, 2021.

[6] Matthieu Herrmann. Tempo, the Monash time series classification library. `https://github.com/MonashTS/tempo`, 2021.

[7] F. Itakura. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 23(1):67–72, 1975.

[8] Young-Seon Jeong, Myong K. Jeong, and Olufemi A. Omitaomu. Weighted dynamic time warping for time series classification. *Pattern Recognition*, 44(9):2231–2240, September 2011.

[9] Eamonn Keogh and Chotirat Ann Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and Information Systems*, 7(3):358–386, 2005.

[10] Eamonn J. Keogh and Michael J. Pazzani. Derivative Dynamic Time Warping. In *Proceedings of the 2001 SIAM International Conference on Data Mining*, pages 1–11. Society for Industrial and Applied Mathematics, April 2001.

[11] Daniel Lemire. Faster retrieval with a two-pass dynamic-time-warping lower bound. *Pattern Recognition*, 42(9):2169–2180, September 2009.

[12] Jason Lines and Anthony Bagnall. Time series classification with ensembles of elastic distance measures. *Data Mining and Knowledge Discovery*, 29(3):565–592, May 2015.

[13] Jason Lines, Sarah Taylor, and Anthony Bagnall. Time Series Classification with HIVE-COTE: The Hierarchical Vote Collective of Transformation-Based Ensembles. *ACM Transactions on Knowledge Discovery from Data*, 12(5):1–35, July 2018.

[14] Benjamin Lucas, Ahmed Shifaz, Charlotte Pelletier, Lachlan O'Neill, Nayyar Zaidi, Bart Goethals, François Petitjean, and Geoffrey I. Webb. Proximity Forest: An effective and scalable distance-based classifier for time series. *Data Mining and Knowledge Discovery*, 33(3):607–635, May 2019.

[15] P.-F. Marteau. Time Warp Edit Distance with Stiffness Adjustment for Time Series Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(2):306–318, February 2009.

[16] Abdullah Mueen and Eamonn Keogh. Extracting optimal performance from dynamic time warping. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, pages 2129–2130. ACM Press, 2016.

[17] François Petitjean, Alain Ketterlin, and Pierre Gançarski. A global averaging method for dynamic time warping, with applications to clustering. *Pattern Recognition*, 44(3):678–693, March 2011.

[18] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. Searching and mining trillions of time series subsequences under dynamic

time warping. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '12*, page 262, Beijing, China, 2012. ACM Press.

[19] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):43–49, February 1978.

[20] Hiroaki Sakoe and Seibi Chiba. A dynamic programming approach to continuous speech recognition. In *Proceedings of the Seventh International Congress on Acoustics, Budapest*, volume 3, pages 65–69, Budapest, 1971. Akadémiai Kiadó.

[21] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007.

[22] Sang-Wook Kim, Sanghyun Park, and W.W. Chu. An index-based approach for similarity search supporting time warping in large sequence databases. In *Proceedings 17th International Conference on Data Engineering*, pages 607–614. IEEE Comput. Soc, 2001.

[23] Ahmed Shifaz, Charlotte Pelletier, Francois Petitjean, and Geoffrey I. Webb. TS-CHIEF: A Scalable and Accurate Forest Algorithm for Time Series Classification. *arXiv:1906.10329 [cs, stat]*, February 2020.

[24] Diego F. Silva and Gustavo E. A. P. A. Batista. Speeding Up All-Pairwise Dynamic Time Warping Matrix Calculation. In *Proceedings of the 2016 SIAM International Conference on Data Mining*, pages 837–845. Society for Industrial and Applied Mathematics, June 2016.

[25] Diego F. Silva, Rafael Giusti, Eamonn Keogh, and Gustavo E. A. P. A. Batista. Speeding up similarity search under dynamic time warping by pruning unpromising alignments. *Data Mining and Knowledge Discovery*, 32(4):988–1016, July 2018.

[26] Alexandra Stefan, Vassilis Athitsos, and Gautam Das. The Move-Split-Merge Metric for Time Series. *IEEE Transactions on Knowledge and Data Engineering*, 25(6):1425–1438, June 2013.

[27] Chang Wei Tan, Christoph Bergmeir, Francois Petitjean, and Geoffrey I. Webb. Time series extrinsic regression. *Data Mining and Knowledge Discovery*, in press.

[28] Chang Wei Tan, François Petitjean, and Geoffrey I. Webb. FastEE: Fast ensembles of elastic distances for time series classification. *Data Mining and Knowledge Discovery*, 34(1):231–272, 2020.

[29] Geoffrey I. Webb and François Petitjean. Tight lower bounds for dynamic time warping. *Pattern Recognition*, 115, 2021.

[30] Renjie Wu and Eamonn J. Keogh. FastDTW is approximate and generally slower than the algorithm it approximates. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.

[31] Qiang Zhu, Gustavo Batista, Thanawin Rakthanmanon, and Eamonn Keogh. A Novel Approximation to Dynamic Time Warping allows Anytime Clustering of Massive Time Series Datasets. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 999–1010. Society for Industrial and Applied Mathematics, April 2012.