

LB_Keogh Supports Exact Indexing of Shapes under Rotation Invariance with Arbitrary Representations and Distance Measures

Eamonn Keogh

Li Wei

Xiaopeng Xi

Sang-Hee Lee¹

Michail Vlachos

Department of Computer Science & Engineering

¹Department of Anthropology

University of California - Riverside

Riverside, CA 92521

{eamonn, wli, xxi}@cs.ucr.edu, shlee@ucr.edu, vlachos@us.ibm.com

ABSTRACT

The matching of two-dimensional shapes is an important problem with applications in domains as diverse as biometrics, industry, medicine and anthropology. The distance measure used must be invariant to many distortions, including scale, offset, noise, partial occlusion, etc. Most of these distortions are relatively easy to handle, either in the representation of the data or in the similarity measure used. However rotation invariance seems to be uniquely difficult. Current approaches typically try to achieve rotation invariance in the representation of the data, at the expense of discrimination ability, or in the distance measure, at the expense of efficiency. In this work we show that we can take the slow but accurate approaches and dramatically speed them up. On real world problems our technique can take current approaches and make them four orders of magnitude faster, without false dismissals. Moreover, our technique can be used with any of the dozens of existing shape representations and with all the most popular distance measures including Euclidean distance, Dynamic Time Warping and Longest Common Subsequence.

1. INTRODUCTION

The matching of two-dimensional shapes is an important problem with applications in domains as diverse as biometrics, industry, medicine and anthropology. The distance measure used must be invariant to many distortions, including scale, offset, noise, partial occlusion, etc. Most of these distortions are relatively easy to handle, particularly if we use the well-known technique of converting the shapes into time series as in Figure 1. However, no matter what representation is used, rotation invariance seems to be uniquely difficult to handle. For example [14] notes “*rotation is always something hard to handle compared with translation and scaling*”, and the literature abounds with similar statements. Many current approaches try to achieve rotation invariance in the representation of the data, at the expense of discrimination ability [19], or in the distance measure, at the expense of efficiency [1][2][3][7].

As an example of the former, the very efficient rotation invariant

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12–15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

technique of [19] cannot differentiate between the shapes of the lowercase letters “d” and “b”. As an example of the latter, the work of Adamek and Connor [1], which is state of the art in terms of accuracy or precision/recall takes an untenable $O(n^3)$ for each shape comparison.

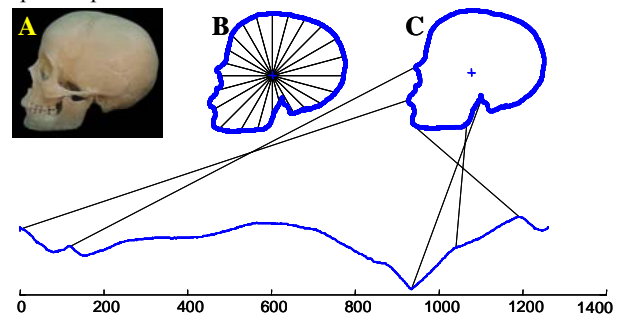


Figure 1: Shapes can be converted to time series. A) A bitmap of a human skull. B) The distance from every point on the profile to the center is measured and treated as the Y-axis of a time series of length n (C)

In this work we show that we can take the slow but accurate approaches and dramatically speed them up. This dramatic improvement in efficiency does not come at the expense of accuracy; we can prove that we will always return the same answer set as the slower methods.

We achieve speedup over the existing methods in two ways, dramatically decreasing the CPU requirements, and allowing indexing. Our technique works by grouping together similar rotations, and defining an admissible lower bound to that group. Given such a lower bound, we can utilize the many search and indexing techniques known in the database community.

Our technique has the following advantages:

- There are dozens of techniques in the literature for converting shapes to time series [1][3][6][24][25][28], including some that are domain specific [4][21]. Our approach works for *any* of these representations.
- While there are many distance measures for shapes in the literature, Euclidean distance, Dynamic Time Warping [2][4][20][21] and Longest Common Subsequence [23] accounts for the majority of the literature. Our approach works for *any* of these distance measures.
- Our approach uses the idea of LB_Keogh lower bounding as its cornerstone. Since the introduction of this idea a few years ago [11], dozens of researchers world wide have adopted and extended this framework for applications as diverse as motion capture indexing [13], P2P searching [9], handwriting retrieval [21], dance indexing, and query by humming and monitoring streams [26]. This widespread

adoption of LB_Keogh lower bounding has insured that it has become a mature and widely supported technology, and suggests that any contributions made here can be rapidly adopted and expanded.

- In some domains it may be useful to express *rotation-limited* queries. For example, in order to robustly retrieve examples of the number “8”, without retrieving infinity symbols “∞”, we can issue a query such as: “Find the best match to this shape allowing a maximum rotation of ± 15 degrees”. Our framework supports such rotation-limited queries.

The rest of this paper is organized as follows. In Section 2 we discuss background material and related work. In Section 3 we formally introduce the problem and in Section 4 we offer our solution. Section 5 offers a comprehensive empirical evaluation of our technique. Finally Section 6 offers some conclusions and directions for future work.

2. BACKGROUND AND RELATED WORK

The literature on shape matching is vast; we refer the interested reader to [6][22] and [28] for excellent surveys. While not all work on shape matching uses a 1D representation of the 2D shapes, an increasingly large majority of the literature does. We therefore only consider such approaches here. Note that we lose little by this omission. The two most popular measures that operate directly in the image space, the Chamfer [5] and Hausdorff [18] distance measures, require $O(n^2 \log n)$ time¹ and recent experiments (including some in this work) suggest that 1D representations can achieve comparable or superior accuracy.

In essence there are three major techniques for dealing with rotation invariance, landmarking, rotation invariant features and brute force rotation alignment. We consider each below.

2.1 Landmarking

The idea of “landmarking” is to find the one “true” rotation and only use that particular alignment as the input to the distance measure. The idea comes in two flavors, domain dependent and domain independent.

In domain dependent landmarking, we attempt to find a single (or very few) fixed feature to use as a starting point for conversion of the shape to a time series. For example, in face profile recognition the most commonly used landmarks (fiducial points) are the chin or nose [4]. In limited domains this may be useful, but it requires building special purpose feature extractors. For example, even in a domain as intuitively well understood as human profiles, accurately locating the nose is a non-trivial problem, even if we discount the possibility of mustaches and glasses. Probably the only reason any progress has been made in this area is that most work reasonably assumes that faces presented in an image are likely to be upright. For shape matching in skulls, the canonical landmark is called the Frankfurt Horizontal [27], which is defined by the right and left porion (the highest point on the margin of the external auditory meatus) and the left orbitale (the lowest point on the orbital margin). However, a skull can be missing the relevant bones to determine this orientation and still have enough global

information to match its shape to similar examples. Indeed the famous Skhul V skull shown in Figure 12 is such an example.

In domain independent landmarking, we align all the shapes to some cardinal orientation, typically the major axis. This approach may be useful for the limited domains in which there is a well-defined major axis, perhaps the indexing of hand tools. However there is increasing recognition that the “...major axis is sensitive to noise and unreliable” [28]. For example a recent paper shows that under some circumstances, a single extra pixel can change the rotation by ± 90 degrees [29].

To show how brittle landmarking can be we performed a simple clustering experiment where we clustered three primate skulls using Euclidean distance with both the major axis technique, and the minimum distance of all possible rotations (as found by brute force). Figure 2 shows the result.

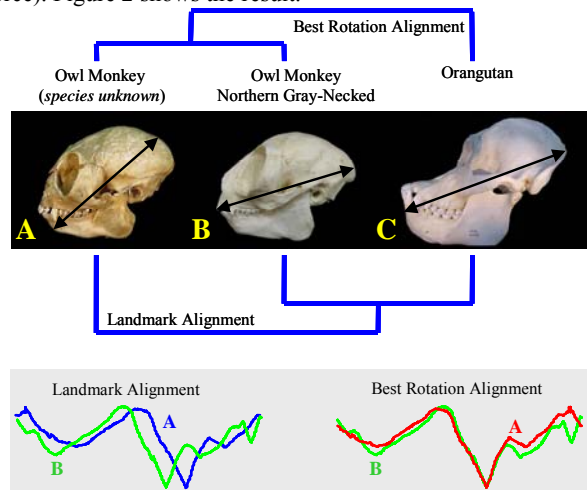


Figure 2: *Top*) Three primate skulls, two of them from the same genus, are clustered using both the landmark rotation beginning at the major axis, and the best rotation. *Bottom*) The landmark-based alignment of A and B explains why the landmark-based clustering is incorrect: a small amount of rotation error results in a large difference in the distance measure

The most important lesson we learned from this experiment (and dozens of other similar experiments on diverse domains [10]) is that rotation (mis)alignment is the most important invariance for shape matching, unless we have the best rotation then nothing else matters.

2.2 Rotation Invariant Features

A large number of papers achieve fast rotation invariant matching by extracting only rotation invariant features and indexing them with a feature vector [6]. This feature vector is often called the shapes “signature”. There are literally dozens of rotation invariant features including ratio of perimeter to area, fractal measures, elongatedness, circularity, min/max/mean curvature, entropy, perimeter of convex hull etc. In addition many researchers have attempted to frame the shape-matching problem as a more familiar histogram-matching problem. For example in [19] the authors build a histogram containing the distances between two randomly chosen points on the perimeter of the shapes in question. The approach seems to be attractive, for example it can trivially also handle 3D shapes, however it suffers from extremely poor precision. For example, it cannot differentiate between the shapes of the lowercase letters “d” and “b”, or “p” and “q”, since these pairs of shapes have identical histograms. In general, all

¹ More precisely the time complexity is $O(Rp \log p)$, where p is the number of pixels in the perimeter and R is the number of rotations that need to be executed. Here $p = n$, and while R is a user defined parameter, it should be approximately equal n to guarantee all rotations (up to the limit of rasterization) are considered.

these methods suffer from very poor discrimination ability [6]. In retrospect this is hardly surprising. In order to achieve rotation invariance, all information that contains rotation information must be discarded; inevitably, some useful information will also be discarded in this process. Our experience with these methods suggests that they can be useful for making quick coarse discriminations, for example differentiating between skulls and vertebrae. However we could not get these methods to distinguish between the skulls of humans and orangutan, a trivial problem for human or the brute force algorithm discussed in the next section.

2.3 Brute Force Rotation Alignment

There are a handful of papers that recognize that the above attempts at approximating rotation invariance are unsatisfactory for most domains, and they achieve true rotation invariance by exhaustive brute force search over all possible rotations, but only at the expense of computational efficiency and indexability [1][2][3][7][25]. For example, paper [1] uses DTW to handle nonrigid shapes in the time series domain, while they note that most invariances are trivial to handle in this representation, they state “rotation invariance can (only) be obtained by checking all possible circular shifts for the optimal diagonal path.” This step makes the comparison of two shapes $O(n^3)$ and forces them to abandon hope of indexing. Similarly paper [25] notes “In order to find the best matching result, we have to shift one curve n times, where n is the number of possible start points.” All the techniques introduced thus far to mitigate this untenable computational complexity do so at the expense of introducing false dismissals. Typically they offer some implicit or explicit trick to find a one (or a small number of) of starting point(s) [2][3][7]. For example paper [2] suggests “In order to avoid evaluation of the dissimilarity measure for every possible pair of starting contour points ...we propose to extract a small set of the most likely starting points for each shape.” Furthermore, both the heuristic used and the number of starting points must “be adjusted to a given application”, and it is not obvious how to best achieve this. In forceful experiments on publicly available datasets it has been demonstrated that brute force rotation alignment produces the best precision/recall and accuracy in diverse domains [1][2]. In retrospect this is not too surprising. The rival techniques with rotation invariant features are all using some lossy transformation of the data. In contrast the brute force rotation alignment techniques are using a (potentially) lossless transformation of the data. With more high quality information to use, any distance measures will have an easier time reflecting the true similarity of the original images.

The contribution of this work is to speed up these accurate but slow methods by many orders of magnitude while producing identical results.

3. ROTATION INVARIANT MATCHING

We begin by formally defining the rotation invariant matching problem. We begin by assuming the Euclidean distance, and generalize to other distance measures later. For clarity of presentation we will generally refer to “time series”, which the reader will note can be mapped back to the original shapes.

Suppose we have two time series, Q and C of length n , which were extracted from shapes by an arbitrary method.

$$Q = q_1, q_2, \dots, q_i, \dots, q_n$$

$$C = c_1, c_2, \dots, c_j, \dots, c_n$$

As we are interested in large data collections we denote a database of m such time series as \bar{Q} .

$$\bar{Q} = \{Q_1, Q_2, \dots, Q_m\}$$

If we wish to compare two time series, and therefore shapes, we can use the ubiquitous Euclidean distance:

$$ED(Q, C) \equiv \sqrt{\sum_{i=1}^n (q_i - c_i)^2}$$

When using Euclidean distance as a subroutine in a classification or indexing algorithm, we may be interested in knowing the exact distance only when it is eventually going to be less than some threshold r . For example, this threshold can be the “range” in range search or the “best-so-far” in nearest neighbor search. If this is the case, we can potentially speed up the calculation by doing early abandoning [12].

Definition 1. Early Abandon: During the computation of the Euclidean distance, if we note that the current sum of the squared differences between each pair of corresponding data points exceeds r^2 , then we can stop the calculation, secure in the knowledge that the exact Euclidean distance had we calculated it, would exceed r .

While the idea of early abandoning is fairly obvious and intuitive, it is so important to our work we illustrate it in Figure 3 and provide pseudocode in Table 1.

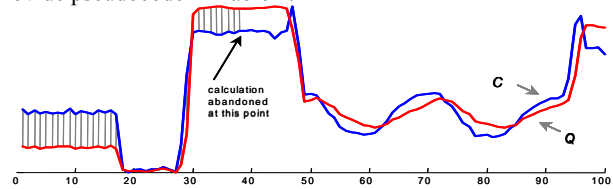


Figure 3: A visual intuition of early abandoning. Once the squared sum of the accumulated gray hatch lines exceeds r^2 , we can be sure the full Euclidean distance exceeds r

Note that the “num_steps” value returned by the optimized Euclidean distance in Table 1 is used only to tell us how useful the optimization was. If its value is significantly less than n this suggests dramatic speedup.

Table 1: Euclidean distance optimized with early abandonment

```

algorithm [dist, num_steps] = EA_Euclidean_Dist(Q, C, r)
accumulator = 0
for  $i = 1$  to length(Q) // Loop over time series
  accumulator +=  $(q_i - c_i)^2$  // Accumulate error contribution
  if accumulator >  $r^2$  // Can we abandon?
    disp('doing an early abandon')
    num_steps =  $i$ 
  return [infinity, num_steps] // Terminate and return an
end // infinite error to signal the
end // early abandonment.
return [sqrt(accumulator), length(Q)] // Terminate with true dist

```

While the Euclidean distance is a simple distance measure it produces surprisingly good results for clustering, classification and query by content of shapes, if the time series in question happen to be rotation aligned. For example, in an experiment in [20] we manually performed rotation alignment of the time series extracted from face profiles by explicitly showing the algorithm the beginning and endpoint of a face (the nape and Adams apple respectively).

However if the shapes are not rotation aligned, this method can produce extremely poor results. To overcome this problem we need to hold one shape fixed, rotate the other, and record the minimum distance of all possible rotations.

For reasons that will become apparent later, we achieve this by expanding one time series into a matrix C of size n by n .

$$C = \begin{Bmatrix} c_1, c_2, \dots, c_{n-1}, c_n \\ c_2, \dots, c_{n-1}, c_n, c_1 \\ \vdots \\ c_n, c_1, c_2, \dots, c_{n-1} \end{Bmatrix}$$

Note that each row of the matrix is simply a time series, shifted (rotated) by one from its neighbors. It will be useful below to address the time series in each row individually, so we will denote the i^{th} row as C_i , which allows us to denote the matrix above in the more compact form of $C = \{C_1, C_2, \dots, C_n\}$.

We can now define the Rotation invariant Euclidean Distance (RED) as:

$$RED(Q, C) = \min_{1 \leq j \leq n} \left\{ ED(Q, C_j) \equiv \sqrt{\sum_{i=1}^n (q_i - c_i)^2} \right\}$$

Table 2 shows the pseudocode to calculate this.

Table 2: An algorithm to find the rotated match between two time series

```

algorithm: [bestSoFar] = Test_All_Rotations(Q, C, r)
bestSoFar = r
for j = 1 to n
    distance = EA_Euclidean_Dist(Q, Cj, bestSoFar) // As in Table 1
    if distance < bestSoFar
        bestSoFar = distance;
    end;
end;
return[bestSoFar]

```

Note that the algorithm tries to take advantage of early abandoning by passing EA_Euclidean_Dist the value of r , the best rotation alignment discovered thus far.

If we are simply measuring the distance between two time series then the algorithm is invoked with r set to infinity, however, as we shall see below, if the algorithm is being used as a subroutine in a linear scan of a large dataset \bar{Q} , the calling routine can set the value of r to achieve speedup. In particular the calling function sets r to the value of the best match (under any rotation) discovered thus far. Table 3 shows the pseudocode. Note that the time complexity for this algorithm is $O(mn^2)$. This is simply untenable for large datasets.

Table 3: An algorithm to find the best rotated match to query from a database of possible matches

```

algorithm: [best_match_loc, bestSoFar] = Search_Database_for_Rotated_Match(C,  $\bar{Q}$ )
best_match_loc = null
bestSoFar = inf
for i = 1 to number_of_time_series_in_database( $\bar{Q}$ )
    distance = Test_All_Rotations( $\bar{Q}_i$ , C, bestSoFar); // As in Table 2
    if distance < bestSoFar
        best_match_loc = i
        bestSoFar = distance
    end;
end;
return[best_match_loc, bestSoFar]

```

Before continuing we will review the notation introduced thus far in Table 4.

Table 4: Notation Table

C	A time series $c_1, c_2, \dots, c_j, \dots, c_n$
\mathbf{C}	A n by n matrix containing every rotation of C
C_i	The i^{th} row of the above
Q	Another time series $q_1, q_2, \dots, q_i, \dots, q_n$
\bar{Q}	A database containing many time series $\bar{Q} = \{Q_1, \dots, Q_m\}$

Note that our notation seems somewhat space inefficient in that it expands time series C , of length n , to a matrix of size n by n . However the rest of the database uses the original (arbitrary rotation) time series, and since the size of the database is assumed to be large, this overhead is asymptotically irrelevant.

There are two simple and useful generalizations of definitions thus far.

Mirror Image Invariance: Depending on the application we may wish to retrieve shapes that are enantiomorphic (mirror images) to the query. For example, in matching skulls, the best match may simply be facing the opposite direction. In contrast when matching letters we *don't* want to match a "d" to a "b". If enantiomorphic invariance is required we can trivially achieve this by augmenting matrix C to contain C_i and $\text{reverse}(C_i)$ for $1 \leq i \leq n$.

Rotation-Limited Invariance: In some domains it may be useful to express *rotation-limited* queries. For example, in order to robustly retrieve examples of the number "6", without retrieving examples of the number "9", we can issue a query such as: "Find the best match to this shape allowing a maximum rotation of ± 15 degrees". Our framework trivially supports such rotation-limited queries, by removing from the matrix C all time series that correspond to the unwanted rotations.

Thus far we have shown a brute force search algorithm that can support rotation invariance, rotation-limited invariance and/or mirror image invariance. We simply put the appropriate time series into matrix C and invoke the algorithm in Table 3. This algorithm, even though speeded up by the early abandoning optimization, is too slow for large datasets. In the next section we introduce our novel search mechanism.

4. WEDGE BASED ROTATION MATCHING

We will begin by showing how we can efficiently search for the best match in main memory. Since large datasets may not fit on disk we will further show how we can index the data.

4.1 Fast and Exact Main Memory Search

We begin by defining time series *wedges*. Imagine that we take several time series, C_1, \dots, C_k , from our matrix C . We can use these sequences to form two new sequences U and L :

$$U_i = \max(C_{1i}, \dots, C_{ki})$$

$$L_i = \min(C_{1i}, \dots, C_{ki})$$

U and L stand for Upper and Lower respectively. We can see why in Figure 4. They form the smallest possible bounding envelope that encloses all members of the set C_1, \dots, C_k from above and below. More formally:

$$\forall_i \quad U_i \geq C_{1i}, \dots, C_{ki} \geq L_i$$

For notational convenience, we will call the combination of U and L a *wedge*, and denote a wedge as W :

$$W = \{U, L\}$$

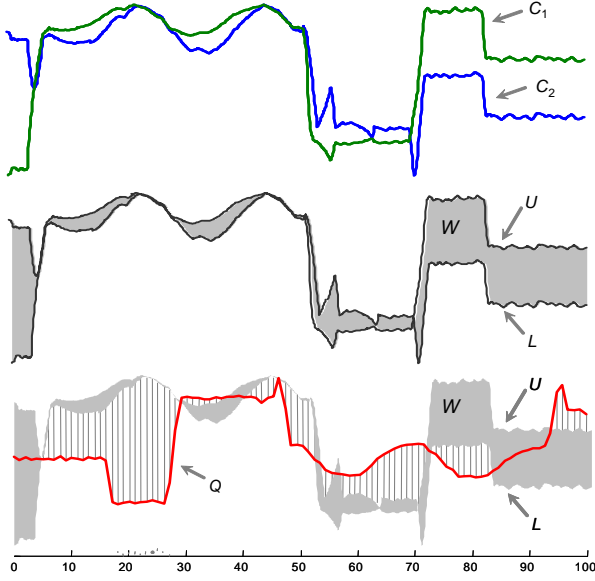


Figure 4: *Top*) Two time series C_1 and C_2 . *Middle*) A time series wedge W , created from C_1 and C_2 . *Bottom*) An illustration of LB_Keogh

We can now define a lower bounding measure between an arbitrary time series Q and the entire set of candidate sequences contained in a wedge W :

$$LB_Keogh(Q, W) = \sqrt{\sum_{i=1}^n \begin{cases} (q_i - U_i)^2 & \text{if } q_i > U_i \\ (q_i - L_i)^2 & \text{if } q_i < L_i \\ 0 & \text{otherwise} \end{cases}}$$

For brevity we do not show a proof of this lower bounding property. A proof appears in [10] and also in [15], where the authors use this representation for different problem.

Note that the LB_Keogh function has been used before to support DTW [11][20][21][23], uniform scaling [13], and query filtering [26]. For these tasks the lower bounding distance function is the same, but the definition of U and L are different.

There are two important observations about LB_Keogh. First, in the special case where W is created from a single candidate sequence, it degenerates to the Euclidean distance. Second, not only does LB_Keogh lower bound all the candidate sequences C_1, \dots, C_k , but we can also do *early abandon* with LB_Keogh. While the latter fact might be obvious, for clarity we make it explicit in Table 5.

Table 5: LB_Keogh optimized with early abandonment

```

algorithm [dist, num_steps] = EA_LB_Keogh(Q, W, r)
  accumulator = 0
  for  $i = 1$  to length(Q)
    // Loop over time series
    if  $q_i > W.U_i$ 
      // Accumulate error contribution
      accumulator +=  $(c_i - W.U_i)^2$ 
    elseif  $q_i < W.L_i$ 
      accumulator +=  $(c_i - W.L_i)^2$ 
    end
    if accumulator >  $r^2$ 
      // Can we abandon?
      return [infinity,  $i$ ]
      // Terminate and return an infinite error
    end
    // to signal the early abandonment.
  end
  return [sqrt(accumulator), length(Q)] // Terminate with true dist

```

Note once again that the value returned in “num_steps” is merely a bookkeeping device to allow a post mortem evaluation of efficiency.

Suppose we have just two time series C_1 and C_2 of length n , and we know that in future we will be given a time series query Q and asked if one (or both) of C_1 and C_2 are within r of the query. We naturally wish to minimize the number of steps we must perform (“steps” are measured by “num_steps”). We are now in a position to outline two possible approaches to this problem.

- We can simply compare the two sequences, C_1 and C_2 (in either order) to the query using the early abandon algorithm introduced in Table 1. We will call this algorithm, *classic*.
- We can combine the two candidate sequences into a wedge, and compare Q to the wedge using LB_Keogh. If the LB_Keogh function early abandons, we are done. We can say with absolute certainty that neither of the two candidate sequences is within r of the query. If we cannot early abandon on the wedge, we need to individually compare the two candidate sequences, C_1 and C_2 (in either order) to the query. We will call this algorithm, *Merge*.

Let us consider the best and worst cases for each approach. For *classic* the worst case is if both candidate sequences are within r of the query, which will require $2n$ steps. In the best case, the first point in the query may be radically different to the first point in either of the candidates, allowing immediate early abandonment and giving a total cost of 2 steps.

For *Merge*, the worst case is also if both candidate sequences are within r of the query, because we will waste n steps in the lower bounding test between the query and the wedge, and then n steps for each individual candidate, for a total of $3n$. However the best case, also if the first point in the query is radically different, would allow us to abandon with a total cost of 1 step.

Which of the two approaches is better depends on:

- The shapes of C_1 and C_2 . If they are similar, this greatly favors *Merge*.
- The shape of Q . If Q is truly similar to one (or both) of the candidate sequences, this would greatly favor *classic*.
- The matching distance r . Here the effect is non monotonic and dependent on the two factors above.

We can generalize the notion of wedges by hierarchically nesting them. Let us begin by augmenting the notation of a wedge to include information about the sequences used to form it. For example, if a wedge is built from C_1 and C_2 , we will denote it as $W_{(1,2)}$. Note that a single sequence is a special case of a wedge, for example the sequence C_1 can also be denoted as W_1 . We can combine $W_{(1,2)}$ and W_3 into a single wedge by finding maximum and minimum values for each i^{th} location, from *either* wedge. More concretely:

$$\begin{aligned}
 U_i &= \max(W_{(1,2)i}, W_{3i}) \\
 L_i &= \min(W_{(1,2)i}, W_{3i}) \\
 W_{((1,2),3)} &= \{U, L\}
 \end{aligned}$$

In Figure 5 we illustrate this notation. We call $W_{(1,2)}$ and W_3 *children* of wedge $W_{((1,2),3)}$. Since individual sequences are special cases of wedges, we can also call C_1 and C_2 children of $W_{(1,2)}$.

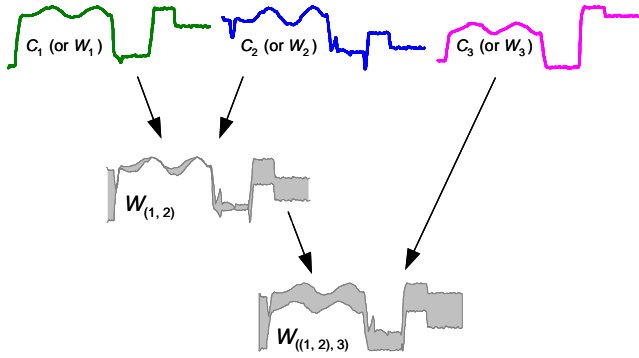


Figure 5: An illustration of hierarchically nested wedges

Given the generalization to hierarchal wedges, we can now also generalize the *Merge* approach. Suppose we have a time series Q and a wedge $W_{(1,2,3)}$. We can compare the query to the wedge using LB_Keogh . If the LB_Keogh function early abandons, we are done. We know with certainty that none of the three candidate sequences is within r of Q . If we cannot early abandon on the wedge, we need to compare the two child wedges, $W_{(1,2)}$ and W_3 to the query. Again, if we cannot early abandon on the wedge $W_{(1,2)}$, we need to individually compare the two candidate sequences, C_1 and C_2 (in either order) to the query. We call this algorithm *H-Merge* (Hierarchal Merge).

The utility of a wedge is strongly correlated to its area. We can get some intuition as to why by visually comparing $LB_Keogh(Q, W_{(1,2)})$ with $LB_Keogh(Q, W_{(1,2,3)})$ as shown in Figure 6. Note that the area of $W_{(1,2,3)}$ is much greater than that of $W_{(1,2)}$, and that this reduces the value returned by the lower bound function and thus the possibility to early abandon.

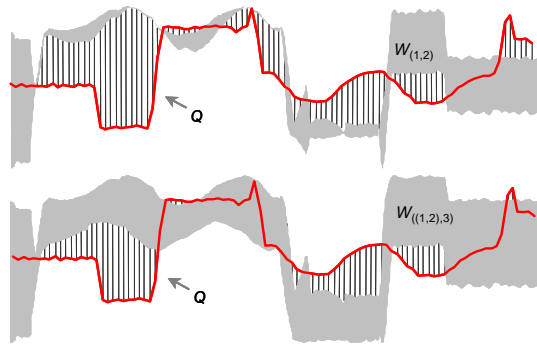


Figure 6: *Top*) An illustration of $LB_Keogh(Q, W_{(1,2)})$. *Bottom*) An illustration of $LB_Keogh(Q, W_{(1,2,3)})$. Note that the tightness of the lower bound is proportion to the number and (squared) length of vertical lines

For some problems, the *H-Merge* algorithm can give exceptionally poor performance. If the wedge $W_{(1,2)}$, created from C_1 and C_2 has an exceptional large area (i.e. C_1 and C_2 are very dissimilar), it is very unlikely to be able to prune off any steps.

At this point we can see that the efficiency of *H-Merge* is dependent on the candidate sequences and Q itself. In general, merging similar sequences into a hierarchal wedge is a good idea, but merging dissimilar sequences is a bad idea.

The observations above motivate a final generalization of *H-Merge*. Recall that to achieve rotation invariance we expanded our time series C into a matrix with n time series. Given these n sequences,

we can merge them into K hierarchal wedges, where $1 \leq K \leq n$. This merging forms a partitioning of the data, with each sequence belonging to exactly one wedge. We will use W to denote a set of hierarchal wedges:

$$W = \{W_{set(1)}, W_{set(2)}, \dots, W_{set(K)}\}, \quad 1 \leq K \leq n$$

where $W_{set(i)}$ is a (hierarchically nested) subset of the n candidate sequences. Note that we have

$$W_{set(i)} \cap W_{set(j)} = \emptyset \text{ if } i \neq j, \text{ and}$$

$$|W_{set(1)} \cup W_{set(2)} \cup \dots \cup W_{set(K)}| = n$$

We will attempt to merge together only similar sequences. We can then compare this set of wedges against our query. Table 6 formalizes the algorithm.

Table 6: Algorithm *H-Merge*

```

algorithm [dist] = H-Merge( $Q, W, K, r$ )
 $S = \{\text{empty}\}$  // Initialize a stack.
for  $i = 1$  to  $K$  // Place all the wedges into the stack.
    enqueue( $W_{set(i)}, S$ )
end
while not empty( $S$ )
     $T = \text{dequeue}(S)$ 
     $\text{dist} = EA\_LB\_Keogh(Q, T, r)$  // Note that is early abandon version.
    if  $\text{isfinite}(\text{dist})$  // We did not early abandon.
        if  $\text{cardinality}(T) = 1$  // T was an individual sequence.
            disp('The sequence ',  $T$ , ' is ',  $\text{dist}$ , ' units from the query')
            return[ $\text{dist}$ ]
        else // T was a wedge, find its children
            enqueue(children( $T$ ),  $S$ ) // and push them onto the stack.
        end
    end
end

```

Note that this algorithm is designed to replace the **Test_All_Rotations** algorithm that is invoked as a subroutine in the **Search_Database_for_Rotated_Match** algorithm shown in Table 3.

As we shall see in our empirical evaluations, *H-Merge* can produce very impressive speedup if we make judicious choices in the set of hierarchal wedges that make up W . However, the number of possible ways to arrange the hierarchal wedges is greater than K^K , and the vast majority of these arrangements will be very poor, so specifying a good arrangement of W is critical.

A simple observation alleviates the need to invent a new algorithm to find a good arrangement of W . Note that hierarchal clustering algorithms have very similar goals to an ideal wedge-producing algorithm. In particular, hierarchal clustering algorithms can be seen as attempting to minimize the distances between objects in each subtree. A wedge-producing algorithm should attempt to minimize the area of each wedge. However the area of a wedge is simply the maximum Euclidean distance between any sequences contained therein (i.e Newton-Cotes rule from elementary calculus). This motivates us to derive wedge sets based on the result of a hierarchal clustering algorithm. Figure 8 shows wedge sets W , of every size from 1 to 5, derived from the dendrogram shown in Figure 7.

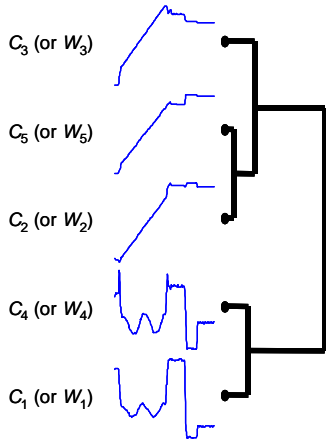


Figure 7: A dendrogram of five sequences C_1, C_2, \dots, C_5 , clustered using group average linkage

Given that the clustering algorithm produces the tentative wedge sets, all we need to do is to choose the best one. We could attempt to do this by eye, for example in Figure 8 it is clear that any sequence that early abandons on W_3 , will almost certainly also early abandon on both W_2 and W_5 ; similar remarks apply to W_1 and W_4 . At the other extreme, the wedge at $K = 1$ is so “fat” that it is likely have poor pruning power. The set $W = \{W_{((2,5),3)}, W_{(1,4)}\}$ is probably the best compromise. However because the set of time series might be very large, such visual inspection is not scalable.

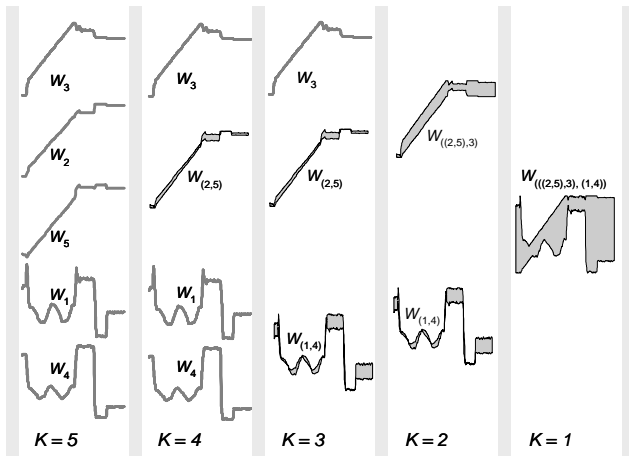


Figure 8: Wedge sets W , of size 1 to 5, derived from the dendrogram shown in Figure 7

The problem is actually even more complex, in that the best value for K also depends on the current value of r (Recall r is the “best-so-far” in nearest neighbor search.). If r is large then very little early abandoning is possible and this favors a large value for K . In contrast, if r is small we can do a lot of early abandoning, and we are better off having many sequences in a single wedge so we can early abandon all of them with a single calculation. Note however that for nearest neighbor search the value of r will get smaller as we search through the database.

With this in mind, we dynamically choose the wedge set based on a fast empirical test. We start with the wedge set where $K = 2$. Each time the **bestSoFar** value changes, we test a subset of the possible values of K and choose the most efficient one (as measured by **num_steps**) as the next K to use. Which subset to test

is decided on-the-fly based on the current K value. They are the values which evenly divide the ranges $[1, \text{current_K}]$ and $[\text{current_K}, \text{max_K}]$ into 5 intervals. Note that on average the **bestSoFar** value only changes $\log(m)$ during a linear search, so this slight overhead in adjusting the parameter is not too burdensome, however we do include this cost in all experiments in Section 5.

4.2 Lower Bounding in Index Space

True rotation invariance has traditionally been so demanding in terms of CPU time that little or no effort was made to index it (or it was indexed with the possibility of false dismissals). As we shall see in the experiments in Section 5.2, the ideas presented in the last section produce such dramatic reductions in CPU time that it is worth considering indexing the data.

There are several possible techniques we could consider for indexing. Recent years have seen dozens of papers on indexing time series envelopes that we could attempt to leverage off [11][15][20][21][23]. The only non-trivial adaptation to be made is that instead of the query being a single envelope, it would be necessary to search for the best match to K envelopes in the wedge set W .

Note however that we do not necessarily have to use the enveloping idea in the indexing phase. So long as we can lower bound in the index space we can use an arbitrary technique to get (hopefully a small subset of) the data from disk to main memory, where our *H-Merge* can very efficiently find the distance to the best rotation. One possible method to achieve this indexable lower bound is to use Fourier methods. Many authors have independently noted that transforming the signal to the Fourier space and calculating the Euclidean distance between the *magnitude* of the coefficients produces a lower bounds to any rotation [24]. We can leverage of this lower bound to use a VP-tree to index our time series as shown in Table 7.

Table 7: A Vantage Point Tree for Indexing Shapes

```

Algorithm [BSF] = NNSearch(C)
  BSF.ID = null; // BSF is the Best-So-Far variable
  BSF.distance = infinity;
  W = convert_time_series_to_wedge_set(C);
  Search(Qroot, W, BSF); // Invoke subroutine on the root of index
Subroutine Search(NODE, W, BSF)
if NODE.isLeaf // we are at a leaf node.
  for each compressed time-series cT in node
    LB = computeLowerBound(cT, W);
    queue.push(cT, LB); // sorted by lower bound.
  end
  while (not (queue.empty()) and (queue.top().LB < BSF.distance))
    if (BSF.distance > queue.top().LB)
      retrieve full time series Q of queue.top() from disk;
      distance = H-Merge(Q, W, BSF.distance) // calculate full distance.
      if distance < BSF.distance // update the best-so-far
        BSF.distance = distance; // distance and location.
        BSF.ID = Q;
      end
    end
  end
end
else // we are at a vantage point.
  LB = computeLowerBound(VP, W);
  queue.push(VP, LB);
  if LB < (node.median + BSF.distance)
    search(NODE.left, W, BSF); // recursive search left.
  else
    search(NODE.right, W, BSF); // recursive search right.
  end
end
end

```

This technique is adapted from [24], and we refer the reader to this work for a more complete treatment.

