Approximate Embedding-Based Subsequence Matching of Time Series

Vassilis Athitsos [1], Panagiotis Papapetrou [2], Michalis Potamias [2],
George Kollios [2], and Dimitrios Gunopulos [3]

[1] Computer Science and Engineering Department, University of Texas at Arlington
[2] Computer Science Department, Boston University
[3] Department of Informatics and Telecommunications, University of Athens

ABSTRACT

A method for approximate subsequence matching is introduced, that significantly improves the efficiency of subsequence matching in large time series data sets under the dynamic time warping (DTW) distance measure. Our method is called EBSM, shorthand for Embedding-Based Subsequence Matching. The key idea is to convert subsequence matching to vector matching using an embedding. This embedding maps each database time series into a sequence of vectors, so that every step of every time series in the database is mapped to a vector. The embedding is computed by applying full dynamic time warping between reference objects and each database time series. At runtime, given a query object, an embedding of that object is computed in the same manner, by running dynamic time warping between the reference objects and the query. Comparing the embedding of the query with the database vectors is used to efficiently identify relatively few areas of interest in the database sequences. Those areas of interest are then fully explored using the exact DTW-based subsequence matching algorithm. Experiments on a large, public time series data set produce speedups of over one order of magnitude compared to brute-force search, with very small losses ($< 1\%$) in retrieval accuracy.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing methods; H.2.8 [**Database Applications**]: Data Mining; H.2.4 [**Systems**]: Multimedia Databases

General Terms

Algorithms

1. INTRODUCTION

Time series data naturally appear in a wide variety of domains, including scientific measurements, financial data, sensor networks, audio, video, and human activity. Subsequence matching is the problem of identifying, given a query time series and a database of time series, the database *subsequence* (i.e., some part of some time series in the database) that is the most similar to the query sequence. Achieving efficient subsequence matching is an important problem in domains where the database sequences are much longer than the queries, and where the best subsequence match for a query can start and end at any position of any database sequence. Improved algorithms for subsequence matching can make a big difference in real-world applications such as query by humming [44], word spotting in handwritten documents, and content-based retrieval in large video databases and motion capture databases.

Naturally, identifying optimal subsequence matches assumes the existence of a similarity measure between sequences, that can be used to evaluate each match. A key requirement for such a measure is that it should be robust to misalignments between sequences, so as to allow for time warps (such as stretching or shrinking a portion of a sequence along the time axis) and changes in sequence length. This requirement effectively rules out Euclidean and more general L_p measures. Typically, similarity between time series is measured using dynamic time warping (DTW) [20], which is indeed robust to misalignments and time warps, and has given very good experimental results for applications such as time series mining and classification [16].

The classical DTW algorithm can be applied for full sequence matching, so as to compute the distance between two time series. With small modifications, the DTW algorithm can also be used for subsequence matching, so as to find, for one time series, the best matching subsequence in another time series [1, 21, 25, 26, 31]. The complexity of the DTW algorithm scales linearly with the length of the query and also scales linearly with the size of the database (i.e., the sum of the lengths of all time series in the database). While this complexity is definitely attractive compared to exhaustively matching the query with every possible database subsequence, in practice subsequence matching is still a computationally expensive operation in many real-world applications, especially in the presence of large database sizes.

1.1 Our Contributions

In this paper we present EBSM (shorthand for Embedding-Based Subsequence Matching) a general method for speeding up subsequence matching in time series databases. Our method is the first to explore the usage of embeddings for subsequence matching for unconstrained DTW. The key differentiating features of our method are the following:

- EBSM converts, at least partially, subsequence matching under DTW into a much easier vector matching problem. Vector matching is used to identify very fast a relatively small

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

number of candidate matches. The computationally expensive DTW algorithm is only applied to evaluate those candidate matches.

- EBSM is the first indexing method, in the context of subsequence matching, that focuses on unconstrained DTW, where optimal matches do not have to have the same length as the query. The only alternative method for this setting, PDTW, which uses piecewise aggregate approximation (PAA) [17], is a generic method for speeding up DTW.
- Our implementation of PDTW (for the purpose of comparing it to EBSM) is also a contribution, as it differs from the way PDTW has been described by its creators [17]: we add a refine step that significantly boosts the accuracy vs efficiency trade-offs achieved by PDTW.
- In our experiments, EBSM provides the best performance in terms of accuracy versus efficiency, compared to the current state-of-the-art methods for subsequence matching under unconstrained DTW: the exact SPRING method [31] that uses the standard DTW algorithm, and the approximate PDTW method [17].

The key idea behind our method is that the subsequence matching problem can be partially converted to the much more manageable problem of nearest neighbor retrieval in a real vector space. This conversion is achieved by defining an embedding that maps each database sequence into a sequence of vectors. There is a one-to-one correspondence between each such vector and a position in the database sequence. The embedding also maps each query series into a vector, in such a way that if the query is very similar to a subsequence, the embedding of the query is likely to be similar to the vector corresponding to the endpoint of that subsequence.

Embeddings are defined by matching queries and database sequences with so-called *reference sequences*, i.e., a relatively small number of preselected sequences. The expensive operation of matching database and reference sequences is performed offline. At runtime, the embedding of the query is computed by matching the query with the reference sequences, which is typically orders of magnitude faster than matching the query with all database sequences. Then, the nearest neighbors of the embedded query are identified among the database vectors. An additional refinement step is performed, where subsequences corresponding to the top vector-based matches are evaluated using the DTW algorithm. Figure 1 illustrates the flowchart of the offline and the online stages of the proposed method.

Converting subsequence matching to vector retrieval is computationally advantageous for the following reasons:

- Sampling and dimensionality reduction methods can easily be applied to reduce the amount of storage required for the database vectors, and the amount of time per query required for vector matching.
- Numerous internal-memory and external-memory indexing methods exist for speeding up nearest neighbor retrieval in vector and metric spaces [4, 13, 41]. Converting subsequence matching to a vector retrieval problem allows us to use such methods for additional computational savings.

EBSM is an approximate method, that does not guarantee retrieving the correct subsequence match for every query. Performance can be easily tuned to provide different trade-offs between accuracy and efficiency. In the experiments, EBSM provides very good trade-offs, by significantly speeding up subsequence match retrieval, even when only small losses in retrieval accuracy (incorrect results for less than 1% of the queries) are allowed.

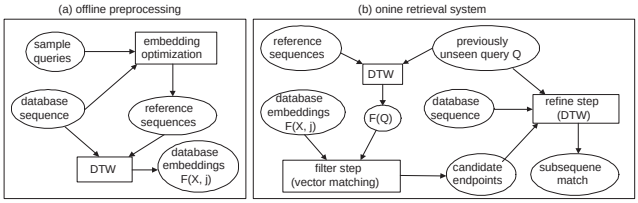

Figure 1: Flowchart of the offline and the online stages of the proposed method. System modules are shown as rectangles, and input/output arguments are shown as ellipses. The goal of the online stage is to identify, given a query time series Q, its optimal subsequence match in the database.

In Section 2 we discuss related work and we emphasize the key differences between our method and existing methods. In Section 3 we provide necessary background information by defining what an optimal subsequence match is, and describing how to use the DTW algorithm to find that match. In Section 4 we define the proposed novel type of embeddings that can be used to speed up subsequence matching. Section 5 describes how the proposed embeddings can be integrated within a filter-and-refine retrieval framework. In Section 6 we describe how to optimize embedding quality using training data. Section 7 discusses the issue of how to handle domains where there is a large difference in length between the smaller and the larger queries that the system may have to handle. Finally, in Section 8 we quantitatively evaluate our method on a large public benchmark dataset, and we illustrate that our method can significantly speed up subsequence matching compared to existing state-of-the-art methods.

2. RELATED WORK

The topic of efficient sequence matching has received significant attention in the database community. However, several methods assume that sequence similarity is measured using the Euclidean distance [8, 23, 24] or variants [2, 10, 29, 42]. Naturally, such methods cannot handle even the smallest misalignment caused by time warps. In the remaining discussion we restrict our attention to methods that are robust to such misalignments.

Dynamic time warping (DTW) [20] is a distance measure that is robust to misalignments and time warps, and it is widely used for time series matching. Time series matching methods can be divided into two categories: 1). methods for full sequence matching, where the best matches for a query are constrained to be entire database sequences, and 2). methods for subsequence matching, where the best matches for a query can be arbitrary subsequences of database sequences. Several well-known methods only address full sequence matching [16, 32, 34, 37, 43], and cannot be used for efficient retrieval of subsequences.

The query-by-humming system described in [44] addresses the problem of matching short melodies hummed by users to entire songs stored in the database. That method cuts each song into smaller, disjoint pieces, and performs full sequence matching between query melodies and the song pieces stored in the database. A similar approach is taken in [30] for searching words in handwritten documents: as preprocessing, the documents are segmented automatically into words, and full sequence matching is performed between query words and database words. Such approaches can only retrieve pieces that the original database sequences have been

segmented to. In contrast, subsequence matching can retrieve any database subsequence matching the query.

In [27] an indexing structure is proposed for unconstrained DTW-based subsequence matching, but retrieval complexity is still linear to the product of the lengths of the query and the database sequence. Furthermore, as database sequences get longer, the time complexity becomes similar to that of unoptimized DTW-based matching.

A method for efficient exact ranked subsequence matching is proposed in [11]. In that method, queries and database sequences are broken into segments, and lower bounds are established using LB_Keogh [16], so as to prune the majority of candidate matches. There are two key differences between the method in [11] and the proposed EBSM method: First, the method in [11] can only find subsequence matches that have the exact same length as the query. Our method has no such limitation. In practice, for efficiency, we make the assumption that the optimal subsequence match has length between zero and twice the length of the query. This is a much milder assumption than requiring the subsequence match to have the exact same length as the query. Second, the method in [11] is only applicable to constrained DTW [16], where the warping path has to stay close to the diagonal. Our method can also be applied to unconstrained DTW.

An efficient method for retrieving subsequences under DTW is presented in [28]. The key idea in that method is to speed up DTW by reducing the length of both query and database sequences. The length is reduced by representing sequences as ordered lists of monotonically increasing or decreasing segments. By using monotonicity, that method is only applicable to 1D time series. A related method that can also be used for multidimensional timeseries is PDTW [17]. In PDTW, time series are approximated by shorter sequences, obtained by replacing each constant-length part of the original sequence with the average value over that part in the new sequence. We compare our method with a modified, improved version of PDTW in the experiments.

The SPRING method for subsequence matching is proposed in [31]. In SPRING, optimal subsequence matches are identified by running the DTW algorithm between the query and each database sequence. Subsequences are identified by prepending to the shorter sequence a "null" symbol that matches any sequence prefix with zero cost (similar ideas are also used in [1, 21, 25, 26]). The complexity of SPRING is still linear to both database size and query size. In EBSM, we use SPRING for matching the query and database sequences with the reference sequences, and for refining the embedding-based retrieval results.

Compared to SPRING, the key source of computational savings in EBSM is that expensive DTW-based matching is only performed between the query and a small fraction of the database, whereas in SPRING the query is matched to the entire database using DTW. The price for this improved efficiency is that EBSM cannot guarantee correct results for all queries, whereas SPRING is an exact method. Still, it is often desirable in database applications to trade accuracy for efficiency, and our method, in contrast to SPRING, provides the capability to achieve such trade-offs.

The method proposed in this paper is embedding-based. Several embedding methods exist in the literature for speeding up distance computations and nearest neighbor retrieval. Examples of such methods include Lipschitz embeddings [12], FastMap [7], MetricMap [38], SparseMap [14], and query-sensitive embeddings [3]. Such embeddings can be used for speeding up sequence matching, as done for example in [3, 14]. However , existing embedding methods are only applicable in the context of full sequence matching, not subsequence matching.The method proposed in this paper is applicable for subsequence matching.

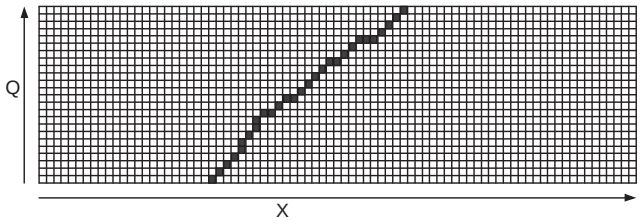

Figure 2: Example of a warping path between a query sequence Q and a database sequence X. Each black square indicates a correspondence between an element of Q and an element of X.

In particular, the above-mentioned embedding methods map each sequence into a single vector, in such a way that if two sequences are similar to each other then the embeddings of those two sequences are also expected to be similar to each other. However, a query sequence can be very similar to a *subsequence* of a database sequence, while being very dissimilar to the entire database sequence. For that reason, existing embedding methods are not useful for efficiently identifying subsequence matches. In contrast, the method proposed in this paper maps each database sequence not to a single vector, but to a *sequence* of vectors, so that there is a one-to-one correspondence between each such vector and a position in the database sequence. If the query is very similar to a subsequence, we expect the embedding of the query to be similar to the vector corresponding to the endpoint of that subsequence.

Another way to illustrate the difference between the embedding methods in [3, 7, 12, 14, 38] and EBSM (our method) is by considering the case where the database contains just a single very long sequence. Existing embedding methods would simply map that sequence into a single vector. Comparing the embedding of the query with that vector would not provide any useful information. Instead, EBSM maps the database sequence into a sequence of vectors. Comparing the embedding of the query with those vectors is used to efficiently identify relatively few areas of interest in the database sequence. Those areas of interest are then fully explored using the exact DTW-based subsequence matching algorithm.

3. BACKGROUND: DTW

In this section we define dynamic time warping (DTW), both as a distance measure between time series, and as an algorithm for evaluating similarity between time series. We follow to a large extent the descriptions in [16] and [31]. We use the following notation:

- Q, X, R, and S are sequences (i.e., time series). Q is typically a query sequence, X is typically a database sequence, R is typically a reference sequence, and S can be any sequence whatsoever.
- $|S|$ denotes the length of any sequence S.
- S_t denotes the t-th step of sequence S. In other words, $S = (S_1, \ldots, S_{|S|})$.
- $S^{i:j}$ denotes the subsequence of S starting at position i and ending at position j. In other words, $S^{i:j} = (S_i, \ldots, S_j)$, $S_t^{i:j}$ is the $t-th$ step of $S^{i:j}$, and $S_t^{i:j} = S_{i+t-1}$.
- $D_{\text{full}}(Q, X)$ denotes the full sequence matching cost between Q and X. In full matching, Q_1 is constrained to match with X_1, and $Q_{|Q|}$ is constrained to match with $X_{|X|}$.

- $D(Q, X)$ denotes the subsequence matching cost between sequences Q and X. This cost is **a**symmetric: we find the subsequence $X^{i:j}$ of X (where X is typically a large database sequence) that minimizes $D_{\text{full}}(Q, X^{i:j})$ (where Q is typically a query).

- $D_{i,j}(Q, X)$ denotes the smallest possible cost of matching $(Q_1, \ldots, Q_i)$ to any suffix of $(X_1, \ldots, X_j)$ (i.e., Q_1 does not have to match X_1, but Q_i has to match with X_j). $D_{i,j}(Q, X)$ is also defined for $i = 0$ and $j = 0$, as specified below.

- $D_j(Q, X)$ denotes the smallest possible cost of matching Q to any suffix of $(X_1, \ldots, X_j)$ (i.e., Q_1 does not have to match X_1, but $Q_{|Q|}$ has to match with X_j). Obviously, $D_j(Q, X) = D_{|Q|,j}(Q, X)$.

- $\|X_i - Y_j\|$ denotes the distance between X_i and Y_j.

Given a query sequence Q and a database sequence X, the subsequence matching problem is the problem of finding the subsequence $X^{i:j}$ of X that is the best match for the entire Q, i.e., that minimizes $D_{\text{full}}(Q, X^{i:j})$. In the next paragraphs we formally define what the best match is, and we specify how it can be computed.

3.1 Legal Warping Paths

A warping path $W = ((w_{1,1}, w_{1,2}), \ldots, (w_{|W|,1}, w_{|W|,2}))$ defines an alignment between two sequences Q and X. The i-th element of W is a pair $(w_{i,1}, w_{i,2})$ that specifies a correspondence between element $Q_{w_{i,1}}$ of Q and element $X_{w_{i,2}}$ of X. The cost $C(Q, X, W)$ of warping path W for Q and X is the L_p distance (for any choice of p) between vectors $(Q_{w_{1,1}}, \ldots, Q_{w_{|W|,1}})$ and $(X_{w_{1,2}}, \ldots, X_{w_{|W|,2}})$:

$$C(Q, X, W) = \sqrt[p]{\sum_{i=1}^{|W|} \|Q_{w_{i,1}} - X_{w_{i,2}}\|^p}. \quad (1)$$

In the remainder of this paper, to simplify the notation, we will assume that $p = 1$. However, the formulation we propose can be similarly applied to any choice of p.

For W to be a legal warping path, in the context of subsequence matching under DTW, W must satisfy the following constraints:

- **Boundary conditions:** $w_{1,1} = 1$ and $w_{|W|,1} = |Q|$. This requires the warping path to start by matching the first element of the query with some element of X, and end by matching the last element of the query with some element of X.

- **Monotonicity:** $w_{i+1,1} - w_{i,1} \geq 0, w_{i+1,2} - w_{i,2} \geq 0$. This forces the warping path indices $w_{i,1}$ and $w_{i,2}$ to increase monotonically with i.

- **Continuity:** $w_{i+1,1} - w_{i,1} \leq 1, w_{i+1,2} - w_{i,2} \leq 1$. This restricts the warping path indices $w_{i,1}$ and $w_{i,2}$ to never increase by more than 1, so that the warping path does not skip any elements of Q, and also does not skip any elements of X between positions $X_{w_{1,2}}$ and $X_{w_{|W|,2}}$.

- **(Optional) Diagonality:** $w_{|W|,2} - w_{1,2} = |Q| - 1, w_{i,2} - w_{1,2} \in [w_{i,1} - \Theta(Q, w_{i,1}), w_{i,1} + \Theta(Q, w_{i,1})]$, where $\Theta(Q, t)$ is some suitably chosen function (e.g., $\Theta(Q, t) = \rho|Q|$, for some constant ρ such that $\rho|Q|$ is relatively small compared to $|Q|$) . This is an optional constraint, employed by some methods, e.g., [11, 16], and not employed by other methods, e.g., [31]. The diagonality constraint imposes that the subsequence $X^{w_{1,2}:w_{|W|,2}}$ be of the same length as Q. Furthermore, the diagonality constraint severely restricts the number of possible positions $w_{i,2}$ of X that can match position $w_{i,1}$ of Q, given the initial match match $(w_{1,1}, w_{1,2})$. In the rest of the paper, we will not consider this constraint, and in the experiments this constraint is not employed.

3.2 Optimal Warping Paths and Distances

The optimal warping path $W^*(Q, X)$ between Q and X is the warping path that minimizes the cost $C(Q, X, W)$:

$$W^*(Q, X) = \text{argmin}_W C(Q, X, W). \quad (2)$$

We define the optimal subsequence match $M(Q, X)$ of Q in X to be the subsequence of X specified by the optimal warping path $W^*(Q, X)$. In other words, if $W^*(Q, X) = ((w^*_{1,1}, w^*_{1,2}), \ldots, (w^*_{m,1}, w^*_{m,2}))$, then $M(Q, X)$ is the subsequence $X^{w^*_{1,2}:w^*_{m,2}}$. We define the partial dynamic time warping (DTW) distance $D(Q, X)$ to be the cost of the optimal warping path between Q and X:

$$D(Q, X) = C(Q, X, W^*(Q, X)). \quad (3)$$

Clearly, partial DTW is an asymmetric distance measure.

To facilitate the description of our method, we will define two additional types of optimal warping paths and associated distance measures. First, we define $W^*_{\text{full}}(Q, X)$ to be the optimal *full warping path*, i.e., the path $W = ((w_{1,1}, w_{1,2}), \ldots, (w_{|W|,1}, w_{|W|,2}))$ minimizing $C(Q, X, W)$ under the additional boundary constraints that $w_{1,2} = 1$ and $w_{|W|,2} = |X|$. Then, we can define the full DTW distance measure $D_{\text{full}}(Q, X)$ as:

$$D_{\text{full}}(Q, X) = C(Q, X, W^*_{\text{full}}(Q, X)). \quad (4)$$

Distance $D_{\text{full}}(Q, X)$ measures the cost of full sequence matching, i.e., the cost of matching the entire Q with the entire X. In contrast, $D(Q, X)$ from Equation 3 corresponds to matching the entire Q with a *subsequence* of X.

We define $W^*(Q, X, j)$ to be the optimal warping path matching Q to a subsequence of X ending at X_j, i.e., the path $W = ((w_{1,1}, w_{1,2}), \ldots, (w_{|W|,1}, w_{|W|,2}))$ minimizing $C(Q, X, W)$ under the additional boundary constraint that $w_{|W|,2} = j$. Then, we can define $D_j(Q, X)$ as:

$$D_j(Q, X) = C(Q, X, W^*(Q, X, j)). \quad (5)$$

We define $M(R, X, j)$ to be the optimal subsequence match for R in X under the constraint that the last element of this match is X_j:

$$M(R, X, j) = \text{argmin}_{X^{i:j}} D_{\text{full}}(R, X^{i:j}). \quad (6)$$

Essentially, to identify $M(R, X, j)$ we simply need to identify the start point i that minimizes the full distance D_{full} between R and $X^{i:j}$.

3.3 The DTW Algorithm

Dynamic time warping (DTW) is a term that refers both to the distance measures that we have just defined, and to the standard algorithm for computing these distance measure and the corresponding optimal warping paths.

We define an operation $\oplus$ that takes as inputs a warping path $W = ((w_{1,1}, w_{1,2}), \ldots, (w_{|W|,1}, w_{|W|,2}))$ and a pair (w', w'') and returns a new warping path that is the result of appending (w', w'') to the end of W:

$$W \oplus (w', w'') = ((w_{1,1}, w_{1,2}), \ldots, (w_{|W|,1}, w_{|W|,2}), (w', w'')). \quad (7)$$

The DTW algorithm uses the following recursive definitions:

$$D_{0,0}(Q,X) = 0, D_{i,0}(Q,X) = \infty, D_{0,j}(Q,X) = 0 \quad (8)$$

$$W_{0,0}(Q,X) = (), W_{0,j}(Q,X) = () \quad (9)$$

$$\mathrm{A}(i,j) = \{(i,j-1),(i-1,j),(i-1,j-1)\} \quad (10)$$

$$(\mathrm{pi}(Q,X),\mathrm{pj}(Q,X)) = \mathrm{argmin}_{(s,t)\in \mathrm{A}(i,j)} D_{s,t}(Q,X) \quad (11)$$

$$D_{i,j}(Q,X) = \|Q_i - X_j\| + D_{\mathrm{pi}(Q,X),\mathrm{pj}(Q,X)}(Q,X) \quad (12)$$

$$W_{i,j}(Q,X) = W_{\mathrm{pi}(Q,X),\mathrm{pj}(Q,X)} \oplus (i,j) \quad (13)$$

$$D(Q,X) = \min_{j=1,\ldots,|X|} \{D_{|Q|,j}(Q,X)\} \quad (14)$$

The DTW algorithm proceeds by employing the above equations at each step, as follows:

- **Inputs.** A short sequence Q, and a long sequence X.
- **Initialization.** Compute $D_{0,0}(Q,X), D_{i,0}(Q,X), D_{0,j}(Q,X)$.
- **Main loop.** For $i = 1,\ldots,|Q|, j = 1,\ldots,|X|$:
  1. Compute $(\mathrm{pi}(Q,X),\mathrm{pj}(Q,X))$.
  2. Compute $D_{i,j}(Q,X)$.
  3. Compute $W_{i,j}(Q,X)$.
- **Output.** Compute and return $D(Q,X)$.

The DTW algorithm takes time $O(|Q||X|)$. By defining $D_{0,j} = 0$ we essentially allow arbitrary prefixes of X to be skipped (i.e., matched with zero cost) before matching Q with the optimal subsequence in X [31]. By defining $D(Q,X)$ to be the minimum $D_{|Q|,j}(Q,X)$, where $j = 1,\ldots,|X|$, we allow the best matching subsequence of X to end at any position j. Overall, this definition matches the entire Q with an optimal subsequence of X.

For each position j of sequence X, the optimal warping path $W^*(Q,X,j)$ is computed as value $W_{|Q|,j}(Q,X)$ by the DTW algorithm (step 3 of the main loop) . The globally optimal warping path $W^*(Q,X)$ is simply $W^*(Q,X,j_{\mathrm{opt}})$, where j_{opt} is the endpoint of the optimal match: $j_{\mathrm{opt}} = \mathrm{argmin}_{j=1,\ldots,|X|}\{D_{|Q|,j}(Q,X)\}$.

4. EBSM: AN EMBEDDING FOR SUBSEQUENCE MATCHING

Let $X = (X_1,\ldots,X_{|X|})$ be a database sequence that is relatively long, containing for example millions of elements. Without loss of generality, we can assume that the database only contains this one sequence X (if the database contains multiple sequences, we can concatenate them to generate a single sequence). Given a query sequence Q, we want to find the subsequence of X that optimally matches Q under DTW. We can do that using brute-force search, i.e., using the DTW algorithm described in the previous section. This paper proposes a more efficient method. Our method is based on defining a novel type of embedding function F, which maps every query Q into a d-dimensional vector and every element X_j of the database sequence also into a d-dimensional vector. In this section we describe how to define such an embedding, and then we provide some examples and intuition as to why we expect such an embedding to be useful.

Let R be a sequence, of relatively short length, that we shall call a *reference object* or *reference sequence*. We will use R to create a 1D embedding F^R, mapping each query sequence into a real number $F(Q)$, and also mapping each step j of sequence X into a real number $F(X,j)$:

$$F^R(Q) = D_{|R|,|Q|}(R,Q)\,. \quad (15)$$

$$F^R(X,j) = D_{|R|,j}(R,X)\,. \quad (16)$$

Naturally, instead of picking a single reference sequence R, we can pick multiple reference sequences to create a multidimensional embedding. For example, let $R_1,\ldots,R_d$ be d reference sequences. Then, we can define a d-dimensional embedding F as follows:

$$F(Q) = (F^{R_1}(Q),\ldots,F^{R_d}(Q))\,. \quad (17)$$

$$F(X,j) = (F^{R_1}(X,j),\ldots,F^{R_d}(X,j))\,. \quad (18)$$

Computing the set of all embeddings $F(X,j)$, for $j = 1,\ldots,|X|$ is an off-line preprocessing step that takes time $O(|X|\sum_{i=1}^{d}|R_i|)$. In particular, computing the i-th dimension F^{R_i} can be done simultaneously for all positions (X,j), with a single application of the DTW algorithm with inputs R_i (as the short sequence) and X (as the long sequence). We note that the DTW algorithm computes each $F^{R_i}(X,j)$, for $j = 1,\ldots,|X|$, as value $D_{|R_i|,j}(R_i,X)$ (see Section 3.3 for more details).

Given a query Q, its embedding $F(Q)$ is computed online, by applying the DTW algorithm d times, with inputs R_i (in the role of the short sequence) and Q (in the role of the long sequence). In total, these applications of DTW take time $O(|Q|\sum_{i=1}^{d}|R_i|)$. This time is typically negligible compared to running the DTW algorithm between Q and X, which takes $O(|Q||X|)$ time. We assume that the sum of lengths of the reference objects is orders of magnitude smaller than the length $|X|$ of the database sequence.

Consequently, a very simple way to speed up brute force search for the best subsequence match of Q is to:

- Compare $F(Q)$ to $F(X,j)$ for $j = 1,\ldots,|X|$.
- Choose some j's such that $F(Q)$ is very similar to $F(X,j)$.
- For each such j, and for some length parameter L, run dynamic time warping between Q and $(X^{j-L+1:j})$ to compute the best subsequence match for Q in $(X^{j-L+1:j})$.

As long as we can choose a small number of such promising areas $(X^{j-L+1:j})$, evaluating only those areas will be much faster than running DTW between Q and X. Retrieving the most similar vectors $F(X,j)$ for $F(Q)$ can be done efficiently by applying a multidimensional vector indexing method to these embeddings [9, 40, 33, 5, 22, 6, 15, 39, 19, 35].

We claim that, under certain circumstances, if Q is similar to a subsequence of X ending at X_j, and if R is some reference sequence, then $F^R(Q)$ is likely to be similar to $F^R(X,j)$. Here we provide some intuitive arguments for supporting this claim.

Let's consider a very simple case, illustrated in Figure 3. In this case, the query Q is *identical* to a subsequence $X^{i':j}$. Consider a reference sequence R, and suppose that $M(R,X,j)$ (defined as in Equation 6) is $X^{i:j}$, and that $i \geq i'$. In other words, $M(R,X,j)$ is a suffix of $X^{i':j}$ and thus a suffix of Q (since $X^{i':j} = Q$). Note that the following holds:

$$F^R(Q) = D_{|R|,|Q|}(R,Q) = D_{|R|,j}(R,X) = F^R(X,j). \quad (19)$$

In other words, if Q appears exactly as a subsequence $X^{i':j}$ of X, it holds that $F^R(Q) = F^R(X,j)$, *under the condition* that the optimal warping path aligning R with $X^{1:j}$ does not start before position i', which is where the appearance of Q starts.

This simple example illustrates an ideal case, where the query Q has an exact match $X^{i':j}$ in the database. The next case to consider is when $X^{i':j}$ is a slightly perturbed version of Q, obtained, for example, by adding noise from the interval $[-\epsilon,\epsilon]$ to each Q_t. In that case, assuming always that $M(R,X,j) = X^{i:j}$ and $i \geq i'$, we can show that $|F^R(Q) - F^R(X,j)| \leq (2|Q|-1)\epsilon$. This

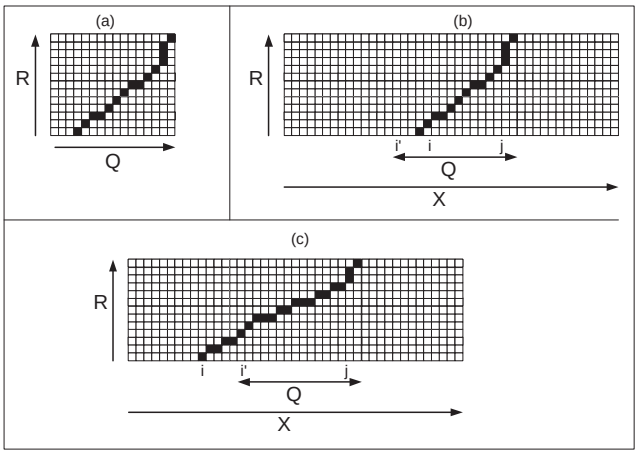

Figure 3: (a) Example of an optimal warping path $W^*(R, Q, |Q|)$ aligning a reference object R to a suffix of Q. $F^R(Q)$ is the cost of $W^*(R, Q, |Q|)$. (b) Example of a warping path $W^*(R, X, j)$, aligning a reference object R to a subsequence $X^{i:j}$ of sequence X. $F^R(X, j)$ is the cost of $W^*(R, X, j)$. The query Q from (a) appears exactly in X, as subsequence $X^{i':j}$, and $i' < i$. Under these conditions, $F^R(Q) = F^R(X, j)$. (c) Similar to (b), except that $i' > i$. In this case, typically $F^R(Q) \neq F^R(X, j)$.

is obtained by taking into account that warping path $W^*(R, X, j)$ cannot be longer than $2|Q| - 1$ (as long as $i \geq i'$).

There are two cases we have not covered:

- Perturbations along the *temporal* axis, such as repetitions, insertions, or deletions. Unfortunately, for unconstrained DTW, due to the non-metric nature of the DTW distance measure, no existing approximation method can make any strong mathematical guarantees in the presence of such perturbations.

- The case where $i < i'$, i.e., the optimal path matching the reference sequence to a suffix of $X^{1:j}$ starts before the beginning of $M(Q, X, j)$. We address this issue in Section 7.

Given the lack of mathematical guarantees, in order for the proposed embeddings to be useful in practice, the following *statistical* property has to hold empirically: given position $j_{\text{opt}}(Q)$, such that the optimal subsequence match of Q in X ends at $j_{\text{opt}}(Q)$, and given some random position $j \neq j_{\text{opt}}(Q)$, it should be statistically very likely that $F(Q)$ is closer to $F(X, j_{\text{opt}}(Q))$ than to $F(X, j)$. If we have access to query samples during embedding construction, we can actually optimize embeddings so that $F(Q)$ is closer to $F(X, j_{\text{opt}}(Q))$ than to $F(X, j)$ as often as possible, over many random choices of Q and j. We do exactly that in Section 6.

5. FILTER-AND-REFINED RETRIEVAL

Our goal in this paper is to design a method for efficiently retrieving, given a query, its best matching subsequence from the database. In the previous sections we have defined embeddings that map each query object and each database position to a d-dimensional vector space. In this section we describe how to use such embeddings in an actual system.

5.1 General Framework

The retrieval framework that we use is filter-and-refine retrieval, where, given a query, the retrieval process consists of a filter step and a refine step [12]. The filter step typically provides a quick way to identify a relatively small number of candidate matches. The refine step evaluates each of those candidates using the original matching algorithm (DTW in our case), in order to identify the candidate that best matches the query.

The goal in filter-and-refine retrieval is to improve retrieval efficiency with small, or zero loss in retrieval accuracy. Retrieval efficiency depends on the cost of the filter step (which is typically small) and the cost of evaluating candidates at the refine step. Evaluating a small number of candidates leads to significant savings compared to brute-force search (where brute-force search, in our setting, corresponds to running SPRING [31], i.e., running DTW between Q and X). Retrieval accuracy, given a query, depends on whether the best match is included among the candidates evaluated during the refine step. If the best match is among the candidates, the refine step will identify it and return the correct result.

Within this framework, embeddings can be used at the filter step, and provide a way to quickly select a relatively small number of candidates. Indeed, here lies the key contribution of this paper, in the fact that we provide a novel method for quick filtering, that can be applied in the context of subsequence matching. Our method relies on computationally cheap vector matching operations, as opposed to requiring computationally expensive applications of DTW. To be concrete, given a d-dimensional embedding F, defined as in the previous sections, F can be used in a filter-and-refine framework as follows:

Offline preprocessing step: Compute and store vector $F(X, j)$ for every position j of the database sequence X.

Online retrieval system: Given a previously unseen query object Q, we perform the following three steps:

- **Embedding step:** compute $F(Q)$, by measuring the distances between Q and the chosen reference sequences.

- **Filter step:** Select database positions (X, j) according to the distance between each $F(X, j)$ and $F(Q)$. These database positions are candidate *endpoints* of the best subsequence match for Q.

- **Refine step:** Evaluate selected candidate positions (X, j) by applying the DTW algorithm.

In the next subsections we specify the precise implementation of the filter step and the refine step.

5.2 Speeding Up the Filter Step

The simplest way to implement the filter step is by simply comparing $F(Q)$ to every single $F(X, j)$ stored in our database. The problem with doing that is that it may take too much time, especially with relatively high-dimensional embeddings (for example, 40-dimensional embeddings are used in our experiments). In order to speed up the filtering step, we can apply well-known techniques, such as sampling, PCA, and vector indexing methods. We should note that these three techniques are all orthogonal to each other.

In our implementation we use sampling, so as to avoid comparing $F(Q)$ to the embedding of every single database position. The way the embeddings are constructed, embeddings of nearby positions, such as $F(X, j)$ and $F(X, j + 1)$, tend to be very similar. A simple way to apply sampling is to choose a parameter δ, and sample uniformly one out of every δ vectors $F(X, j)$. That is, we only store vectors $F(X, 1), F(X, 1 + \delta), F(X, 1 + 2\delta), \ldots$.

Given $F(Q)$, we only compare it with the vectors that we have sampled. If, for a database position (X, j), its vector $F(X, j)$ was not sampled, we simply assign to that position the distance between $F(Q)$ and the vector that was actually sampled among $\{F(X, j - \lfloor \delta/2 \rfloor), \ldots, F(X, j + \lfloor \delta/2 \rfloor)\}$.

PCA can also be used, in principle, to speed up the filter step, by reducing the dimensionality of the embedding. Finally, vector indexing methods [9, 40, 33, 5, 22, 6, 15, 39, 19, 35] can be applied to speed up retrieval of the nearest database vectors. Such indexing methods may be particularly useful in cases where the embedding of the database does not fit in main memory; in such cases, external memory indexing methods can play a significant role in optimizing disk usage and overall retrieval runtime.

Our implementation at this point is a main-memory implementation, where the entire database embedding is stored in memory. In our experiments, using sampling parameter $\delta = 9$, and without any further dimensionality reduction or indexing methods, we get a very fast filter step: the average running time per query for the filter step is about 0.5% of the average running time of brute-force search. For that reason, at this point we have not yet incorporated more sophisticated methods, that might yield faster filtering.

5.3 The Refine Step for Unconstrained DTW

The filter step ranks all database positions (X, j) in increasing order of the distance (or estimated distance, when we use approximations such as PCA, or sampling) between $F(X, j)$ and $F(Q)$. The task of the refine step is to evaluate the top p candidates, where p is a system parameter that provides a trade-off between retrieval accuracy and retrieval efficiency.

Algorithm 1 describes how this evaluation is performed. Since candidate positions (X, j) actually represent candidate *endpoints* of a subsequence match, we can evaluate each such candidate endpoint by starting the DTW algorithm from that endpoint and going backwards. In other words, the end of the query is aligned with the candidate endpoint, and DTW is used to find the optimal start (and corresponding matching cost) for that endpoint.

If we do not put any constraints, the DTW algorithm will go all the way back to the beginning of the database sequence. However, subsequences of X that are much longer than Q are very unlikely to be optimal matches for Q. In our experiments, 99.7% out of the 1000 queries used in performance evaluation have an optimal match no longer than twice the length of the query. Consequently, we consider that twice the length of the query is a pretty reasonable cut-off point, and we do not allow DTW to consider longer matches.

One complication is a case where, as the DTW algorithm moves backwards along the database sequence, the algorithm gets to another candidate endpoint that has not been evaluated yet. That endpoint will need to be evaluated at some point anyway, so we can save time by evaluating it now. In other words, while evaluating one endpoint, DTW can simultaneously evaluate all other endpoints that it finds along the way. The two adjustments that we make to allow for that are that:

- The "sink state" $Q_{|Q|+1}$ matches candidate endpoints (that have not already been checked) with cost 0 and all other database positions with cost ∞.
- If in the process of evaluating a candidate endpoint j we find another candidate endpoint j', we allow the DTW algorithm to look back further, up to position $j' - 2|Q| + 1$.

The $\mathrm{endpoint}$ array in Algorithm 1 keeps track, for every pair (i, j), of the endpoint that corresponds to the cost stored in $\mathrm{cost}[i][j]$.

input : Q: query.
X: database sequence.
sorted: an array of candidate endpoints j, sorted in decreasing order of j.
p: number of candidates to evaluate.

output : $(X, j_{\mathrm{start}}), (X, j_{\mathrm{end}})$: start and end point of estimated best subsequence match.
distance: distance between query and estimated best subsequence match.
columns: number of database positions evaluated by DTW (this is a key measure of retrieval efficiency).

for $i = 1$ **to** $|X|$ **do**
  $\mathrm{unchecked}[i] = 0$;
end
for $i = 1$ **to** p **do**
  $\mathrm{unchecked}[\mathrm{sorted}[i]] = 1$;
end
$\mathrm{distance} = \infty$;
$\mathrm{columns} = 0$;
// main loop, check all candidates sorted[1], ..., sorted[p].
for $k = 1$ **to** p **do**
  $\mathrm{candidate} = \mathrm{sorted}[k]$;
  if $(\mathrm{unchecked}[\mathrm{candidate}] == 0)$ **then continue**;
  $j = \mathrm{candidate} + 1$;
  for $i = |Q| + 1$ **to** 1 **do**
    $\mathrm{cost}[i][j] = \infty$;
  end
  while *(***true***)* **do**
    $j = j - 1$;
    if $(\mathrm{candidate} - j \geq 2 * |Q|)$ **then break**;
    if *(*$\mathrm{unchecked}[j] == 1$*)* **then**
      $\mathrm{unchecked}[j] = 0$;
      $\mathrm{candidate} = j$; // found another candidate endpoint.
      $\mathrm{cost}[|Q| + 1][j] = 0$;
      $\mathrm{endpoint}[|Q| + 1][j] = j$;
    else
      $\mathrm{cost}[|Q| + 1][j] = \infty$; // j is not a candidate endpoint.
    end
    for $i = |Q|$ **to** 1 **do**
      $\mathrm{previous} = \{(i + 1, j), (i, j + 1), (i + 1, j + 1)\}$;
      $(p_i, p_j) = \mathrm{argmin}_{(a,b) \in \mathrm{previous}} \mathrm{cost}[a][b]$;
      $\mathrm{cost}[i][j] = |Q_i - X_j| + \mathrm{cost}[p_i][p_j]$;
      $\mathrm{endpoint}[i][j] = \mathrm{endpoint}[p_i][p_j]$;
    end
    if *(*$\mathrm{cost}[1][\mathrm{j}] < \mathrm{distance}$*)* **then**
      $\mathrm{distance} = \mathrm{cost}[1][j]$;
      $\mathrm{j_{start}} = j$;
      $j_{\mathrm{end}} = \mathrm{endpoint}[1][j]$;
    end
    $\mathrm{columns} = \mathrm{columns} + 1$;
    if $(\min\{\mathrm{cost[i][j]} | i = 1, \ldots, |Q|\} \geq \mathrm{distance})$ **then break**;
  end
end
//final alignment step
$\mathrm{start} = j_{\mathrm{end}} - 3|Q|$;
$\mathrm{end} = j_{\mathrm{end}} + |Q|$;
Adjust j_{start} and j_{start} by running the DTW algorithm between Q and $X^{\mathrm{start:end}}$;

Algorithm 1. The refine step for unconstrained DTW.

This is useful in the case where multiple candidate endpoints are encountered, so that when the optimal matching score is found (stored in variable $\mathrm{distance}$), we know what endpoint that matching score corresponds to.

input : X: database sequence.
Q_S: training query set.
d: embedding dimensionality.
RSK: initial set of k reference subsequences.

output : R: set of d reference subsequences.

// select d reference sequences with highest variance from RSK
$R = \{R_1, .., R_d \,|R_i \in RSK \quad with \quad maximum \quad variance\}$
CreateEmbedding(R, X);
oldSEE $= 0$;
for $i = 1$ **to** $|Q_S|$ **do**
  oldSEE$+ = EE(Q_S[i])$;
end
j $= 1$;
while *(***true***)* **do**
  // consider replacing R_j with another reference object
  $CandR = RSK - R$;
  for $i = 0$ **to** $|CandR|$ **do**
    CreateEmbedding$(R - \{R_j\} + \{CandR[i]\}, X)$;
    newSEE $= 0$;
    for $i = 1$ **to** $|Q_S|$ **do**
      newSEE$+ = EE(Q_S[i])$;
    end
    if *(*newSEE $<$ oldSEE*)* **then**
      $R_j = CandR[i]$;
      oldSEE $=$ newSEE;
    end
  end
  $j = (j \mod d) + 1$;
end

Algorithm 2. The training algorithm for selection of reference objects.

The columns variable, which is an output of Algorithm 1, measures the number of database positions on which DTW is applied. These database positions include both each candidate endpoint and all other positions j for which $\text{cost}[i][j]$ is computed. The columns output is a very good measure of how much time the refine step takes, compared to the time it would take for brute-force search, i.e., for applying the original DTW algorithm as described in Section 3. In the experiments, one of the main measures of EBSM efficiency (the DTW cell cost) is simply defined as the ratio between columns and the length $|X|$ of the database.

We note that each application of DTW in Algorithm 1 stops when the minimum $\text{cost}[i][j]$ over all $i = 1, \ldots, |Q|$ is higher than the minimum distance found so far. We do that because any $\text{cost}[i][j-1]$ will be at least as high as the minimum (over all i's) of $\text{cost}[i][j]$, except if $j-1$ is also a candidate endpoint (in which case, it will also be evaluated during the refine step).

The refine step concludes with a final alignment/verification operation, that evaluates, using the original DTW algorithm, the area around the estimated optimal subsequence match. In particular, if j_{end} is the estimated endpoint of the optimal match, we run the DTW algorithm between Q and $X^{(j_{\text{end}}-3|Q|):(j_{\text{end}}+|Q|)}$. The purpose of this final alignment operation is to correctly handle cases where j_{start} and j_{end} are off by a small amount (a fraction of the size of Q) from the correct positions. This may arise when the optimal endpoint was not included in the original set of candidates obtained from the filter step, or when the length of the optimal match was longer than $2|Q|$.

6. EMBEDDING OPTIMIZATION

In this section, we present an approach for selecting reference objects in order to improve the quality of the embedding. The goal is to create an embedding where the rankings of different subsequences with respect to a query in the embedding space resemble the rankings of these subsequences in the original space. Our approach is largely an adaptation of the method proposed in [36].

The first step is based on the max variance heuristic, i.e., the idea that we should select subsequences that cover the domain space (as much as possible) and have distances to other subsequences with high variance. In particular, we select uniformly at random l subsequences with sizes between *(minimum query size)/2* and *maximum query size* from different locations in the database sequence. Then, we compute the DTW distances for each pair of them ($O(l^2)$ values) and we select the k subsequences with the highest variance in their distances to the other $l-1$ subsequences. Thus we select an initial set of k reference objects.

The next step is to use a learning approach to select the final set of reference objects assuming that we have a set of samples that is representative of the query distribution. The input to this algorithm is a set of k reference objects RSK selected from the previous step, the number of final reference objects d (where $d < k$) and a set of sample queries $\mathbb{Q}_s$. The main idea is to select d out of the k reference objects so as to minimize the embedding error on the sample query set. The embedding error $\text{EE}(Q)$ of a query Q is defined as the number of vectors $F(X, j)$ in the embedding space that the embedding of the query $F(Q)$ is closer to than it is to the embedding of $F(X, j_Q)$, where j_Q is the endpoint of the optimal subsequence match of Q in the database.

Initially, we select d initial reference objects $R_1, \ldots, R_d$ and we create the embedding of the database and the query set $\mathbb{Q}_s$ using the selected R_i's. Then, for each query, we compute the embedding error and we compute the sum of these errors over all queries, i.e., $\text{SEE} = \sum_{Q \in \mathbb{Q}_s} \text{EE}(Q)$. The nest step, is to consider a replacement of the i-th reference object with an object in $\text{RSK} - \{R_1, \ldots, R_d\}$, and re-estimate the SEE. If SEE is reduced, we make the replacement and we continue with the next $(i+1)$-th reference object. This process starts from $i = 1$ until $i = d$. After we replace the d-th reference object we continue again with the first reference object. The loop continues until the improvement of the SEE over all reference objects falls below a threshold. The pseudo-code of the algorithm is shown in Algorithm 2. To reduce the computation overhead of the technique we use a sample of the possible replacements in each step. Thus, instead of considering all objects in $\text{RSK} - \{R_1, \ldots, R_d\}$ for replacement, we consider only a sample of them. Furthermore, we use a sample of the database entries to estimate the SEE.

Note that the embedding optimization method described here largely follows the method described in [36]. However, the approach in [36] was based on the Edit distance, which is a metric, and therefore a different optimization criterion was used. In particular, in [36], reference objects are selected based on the pruning power of each reference object. Since DTW is not a metric, reference objects in our setting do not have pruning power, unless we allow some incorrect results. That is why we use the sum of errors as our optimization criterion.

7. HANDLING VERY LARGE RANGES OF QUERY LENGTHS

In Section 4 and in Figure 3 we have illustrated that, intuitively, when the query Q has a very close match $X^{i:j}$ in the database, we expect $F^R(Q)$ and $F^R(X, j)$ to be similar, as long as $M(R, X, j)$

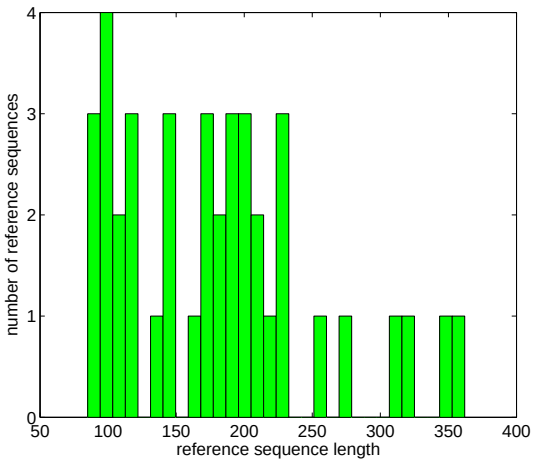

Figure 4: Distribution of lengths of the 40 reference objects chosen by the embedding optimization algorithm in our experiments.

is a suffix of $M(Q, X, j)$. If we fix the length $|Q|$ of the query, as the length $|R|$ of the reference object increases, it becomes more and more likely that $M(R, X, j)$ will start before the beginning of $M(Q, X, j)$. In those cases, $F^R(Q)$ and $F^R(X, j)$ can be very different, even in the ideal case where Q is identical to $X^{i:j}$.

In our experiments, the minimum query length is 152 and the maximum query length is 426. Figure 4 shows a histogram of the lengths of the 40 reference objects that were chosen by the embedding optimization algorithm in our experiments. We note that smaller lengths have higher frequencies in that histogram. We interpret that as empirical evidence for the argument that long reference objects tend to be harmful when applied to short queries, and it is better to have short reference objects applied to long queries. Overall, as we shall see in the experiments section, this 40-dimensional embedding provides very good performance.

At the same time, in any situation where there is a large difference in scale between the shortest query length and the longest query length, we are presented with a dilemma. While long reference objects may hurt performance for short queries, using only short reference objects gives us very little information about the really long queries. To be exact, given a reference object R and a database position (X, j), $F^R(X, j)$ only gives us information about subsequence $M(R, X, j)$. If Q is a really long query and R is a really short reference object, proximity between $F(Q)$ and $F(X, j)$ cannot be interpreted as strong evidence of a good subsequence match for the entire Q ending at position j; it is simply strong evidence of a good subsequence match ending at position j for some small *suffix* of Q defined by $M(R, Q, |Q|)$.

The simple solution in such cases is to use, for each query, only embedding dimensions corresponding to a subset of the chosen reference objects. This subset of reference objects should have lengths that are not larger than the query length, and are not too much smaller than the query length either (e.g., no smaller than half the query length). To ensure that for any query length there is a sufficient number of reference objects, reference object lengths can be split into d ranges $[r, rs), [rs, rs^2), [rs^2, rs^3), \ldots [rs^{d-1}, rs^d)$, where r is the minimum desired reference object length, rs^d is the highest desired reference object length, and s is determined given r, d and rs^d. Then, we can constrain the d-dimensional embedding so that for each range $[rs^i, rs^{i+1})$ there is only one reference object with length in that range.

We do not use this approach in our experiments, because the simple scheme of using all reference objects for all queries works well enough. However, it is important to have in mind the limitations of this simple scheme, and we believe that the remedy we have outlined here is a good starting point for addressing these limitations.

8. EXPERIMENTS

We evaluate the proposed method on time series data obtained from the UCR Time Series Data Mining Archive [18]. We compare our method to the two state-of-the-art methods for subsequence matching under unconstrained DTW:

- **SPRING:** the exact method proposed by Sakurai et al. [31], which applies the DTW algorithm as described in Section 3.3.
- **Modified PDTW:** a modification of the approximate method based on piecewise aggregate approximation that was proposed by Keogh et al. [17].

Actually, as formulated in [17], PDTW (given a sampling rate) yields a specific accuracy and efficiency, by applying DTW to smaller, subsampled versions of query Q and database sequence X. Even with the smallest possible sampling rate of 2, for which the original PDTW cost is 25% of the cost of brute-force search, the original PDTW method has an accuracy rate of less than 50%. We modify the original PDTW so as to significantly improve those results, as follows: in our modified PDTW, the original PDTW of [17] is used as a filtering step, that quickly identifies candidate endpoint positions, exactly as the proposed embeddings do for EBSM. We then apply the refine step on top of the original PDTW rankings, using the exact same algorithm (Algorithm 1) for the refine step that we use in EBSM. We will see in the results that the modified PDTW works very well, but still not as well as EBSM.

We do not make comparisons to the subsequence matching method of [11], because the method in [11] is designed for indexing constrained DTW (whereas in the experiments we use unconstrained DTW), and thus would fail to identify any matches whose length is not equal to the query length. As we will see in Section 8.3, the method in [11] would fail to identify optimal matches for the majority of the queries.

8.1 Datasets

To create a large and diverse enough dataset, we combined three of the datasets from UCR Time Series Data Mining Archive [18]. The three UCR datasets that we used are shown on Table 1.

Each of the three UCR datasets contains a test set and a training set. As can be seen on Table 1, the original split into training and test sets created test sets that were significantly larger than the corresponding training sets, for two of the three datasets. In order to evaluate indexing performance, we wanted to create a sufficiently large database, and thus we generated our database using the large test sets, and we used as queries the time series in the training sets.

More specifically, our database is a single time series X, that was generated by concatenating all time series in the original test sets: 455 time series of length 270 from the 50Words dataset, 6164 time series of length 152 from the Wafer dataset, and 3000 time series of length 426 from the Yoga dataset. The length $|X|$ of the database is obviously the sum of lengths of all these time series, which adds up to 2,337,778.

Our set of queries was the set of time series in the original training sets of the three UCR datasets. In total, this set includes 1750

Name	50Words	Wafer	Yoga
Length of each time series	270	152	426
Size of "training set" (used by us as set of queries)	450	1000	300
Number of time series used for validation (subset of set of queries)	192	428	130
Number of time series used for measuring performance (subset of set of queries)	258	572	170
Size of "test set" (used by us to generate the database)	455	6164	3000

Table 1: Description of the three UCR datasets we combined to generate our dataset. For each original UCR dataset we show the sizes of the original training and test sets. We note that, in our experiments, we use the original training sets to obtain queries for embedding optimization and for performance evaluation, and we use the original test sets to generate the long database sequence (of length 2,337,778).

time series. We randomly chose 750 of those time series as a validation set of queries, that was used for embedding optimization using Algorithm 2. The remaining 1000 queries were used to evaluate indexing performance. Naturally, the set of 1000 queries used for performance evaluation was completely disjoint from the set of queries used during embedding optimization.

8.2 Performance Measures

Our method is approximate, meaning that it does not guarantee finding the optimal subsequence match for each query. The two key measures of performance in this context are accuracy and efficiency. Accuracy is simply the percentage of queries in our evaluation set for which the optimal subsequence match was successfully retrieved. Efficiency can be evaluated using two measures:

- **DTW cell cost:** For each query Q, the DTW cell cost is the ratio of number of cells $[i][j]$ visited by Algorithm 1 over number of cells $[i][j]$ using the SPRING method (for the SPRING method, this number is the product of query length and database length). For PDTW with sampling rate s, we add $\frac{1}{s^2}$ to this ratio, to reflect the cost of running the DTW algorithm between the subsampled query and the subsampled database. For the entire test set of 1000 queries, we report the average DTW cell cost over all queries.

- **Retrieval runtime cost:** For each query Q, given an indexing method, the retrieval runtime cost is the ratio of total retrieval time for that query using that indexing method over the total retrieval time attained for that query using the SPRING method. For the entire test set, we report the average retrieval runtime cost over all 1000 queries. While runtime is harder to analyze, as it depends on diverse things such as cache size, memory bus bandwidth, etc., runtime is also a more fair measure for comparing EBSM to PDTW, as it includes the costs of both the filter step and the refine step. The DTW cell cost ignores the cost of the filter step for EBSM.

We remind the reader that the SPRING method simply uses the standard DTW algorithm of Section 3.3. Consequently, by definition, the DTW cell cost of SPRING is always 1, and the retrieval runtime cost of SPRING is always 1. The actual average running time of the SPRING method over all queries we used for performance evaluation was: 4.43 sec/query for queries of length 152, 7.23 sec/query for queries of length 270, and 11.30 sec/query for

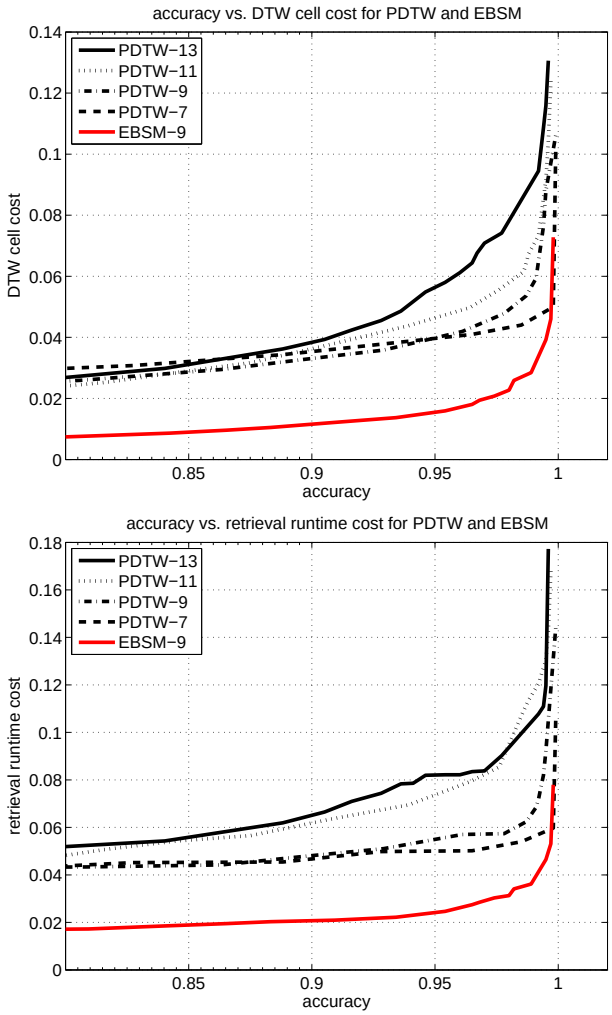

Figure 5: Comparing the accuracy versus efficiency trade-offs achieved by EBSM with sampling rate 9 and by modified PDTW with sampling rates 7, 9, 11, and 13. The top figure measures efficiency using the DTW cell cost, and the bottom figure measures efficiency using the retrieval runtime cost. The costs shown are average costs over our test set of 1000 queries. Note that SPRING, being an exact method, corresponds to a single point (not shown on these figures), with perfect accuracy 1 and maximal DTW cell cost 1 and retrieval runtime cost 1.

queries of length 426. The system was implemented in C++, and run on an AMD Opteron 8220 SE processor running at 2.8GHz.

Trade-offs between accuracy and efficiency can be obtained very easily, for both EBSM and the modified PDTW, by changing parameter p of the refine step (see Algorithm 1). Increasing the value of p increases accuracy, but decreases efficiency, by increasing both the DTW cell cost and the running time.

We should emphasize the runtime retrieval cost depends on the retrieval method, the data set, the implementation, and the system platform. On the other hand, the DTW cell cost only depends on the retrieval method and the data set; different implementations of the same method should produce the same results (or very similar, when random choices are involved) on the same data set regardless of the system platform or any implementation details.

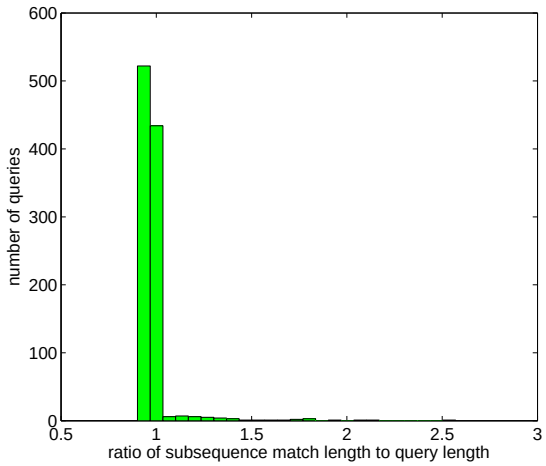

Figure 6: Distribution of lengths of optimal subsequence matches (as fractions of the query length) for the 1000 queries used for performance evaluation. We note that a significant fraction of the optimal matches have lengths that are not identical to the query length.

8.3 Results

We compare EBSM to modified PDTW and SPRING. We note that the SPRING method guarantees finding the optimal subsequence match, whereas modified PDTW (like EBSM) is an approximate method. For EBSM, unless otherwise indicated, we used a 40-dimensional embedding, with a sampling rate of 9. For the embedding optimization procedure of Section 6, we used parameters $l = 1755$ (l was the number of candidate reference objects before selection using the maximum variance criterion) and $k = 1000$ (k was the number of candidate reference objects selected based on the maximum variance criterion).

Figure 5 shows the trade-offs of accuracy versus efficiency achieved. We note that EBSM provides very good trade-offs between accuracy and retrieval cost. Also, EBSM significantly outperforms the modified PDTW, in terms of both DTW cell cost and retrieval runtime cost. For many accuracy settings, EBSM attains costs smaller by a factor of 2 or more compared to PDTW. As highlights, for 99.5% retrieval accuracy our method is about 21 times faster than SPRING (retrieval runtime cost = 0.046), and for 90% retrieval accuracy our method is about 47 times faster than SPRING (retrieval runtime cost = 0.021).

Figure 6 shows a histogram of the length of the optimal subsequence match for each query, as a fraction of the length of that query. The statistics for this histogram were collected from all 1000 queries used for performance evaluation. We see that, although for the majority of cases the match length is fairly close to the query length, it is only for a minority of queries that the match length is exactly equal to the query length. We should note that the subsequence matching method of [11] would fail to identify any matches whose length is not equal to the query length. As a result, it would not be meaningful to compare the performance of our method versus the method in [11] for this dataset.

Figure 7 shows how the performance of EBSM varies with different sampling rates. For all results in that figure, 40-dimensional embeddings were used, optimized using Algorithm 2. Sampling rates between 1 and 15 all produced pretty similar DTW cell costs

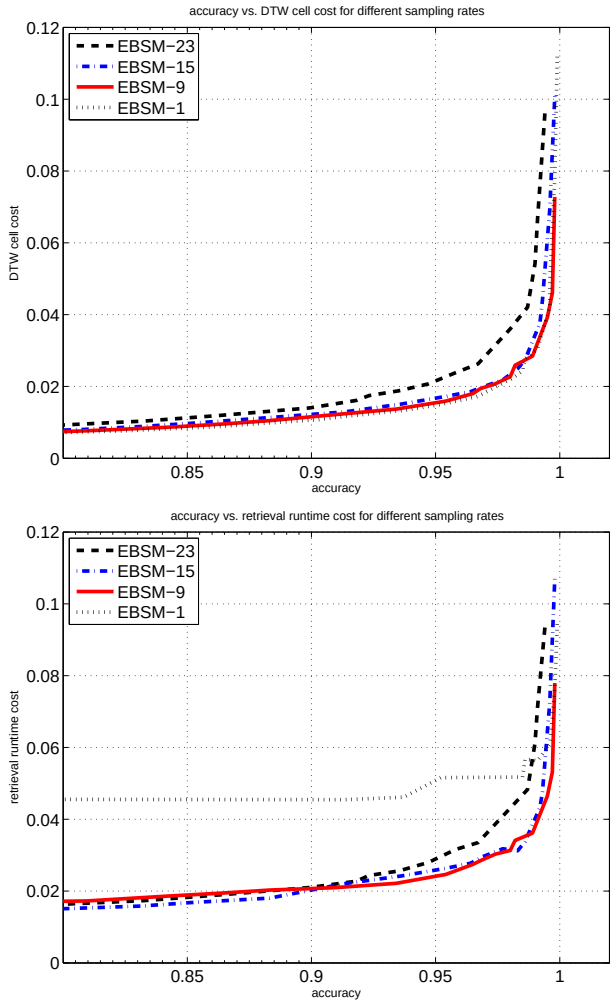

Figure 7: Accuracy vs. efficiency for EBSM with sampling rates 1, 9, 15, and 23. The top figure measures efficiency using the DTW cell cost, and the bottom figure measures efficiency using the retrieval runtime cost. The costs shown are average costs over our test set of 1000 queries.

for EBSM, but a sampling rate of 23 produced noticeably worse DTW cell costs. In terms of retrieval runtime, a sampling rate of 1 performed much worse compared to sampling rates of 9 and 15, because the cost of the filter step is much higher for sampling rate 1: the number of vector comparisons is equal to the length of the database divided by the sampling rate.

Figure 8 compares different methods for embedding construction. For all results in that figure, 40-dimensional embeddings and a sampling rate of 9 were used. We notice that selecting reference objects using the max variance heuristic (i.e., using only the first two lines of Algorithm 2) improves performance significantly compared to random selection. Using the full Algorithm 2 for embedding construction improves performance even more.

Figure 9 shows how the performance of EBSM varies with different embedding dimensionality, for optimized (using Algorithm 2) and unoptimized embeddings. For all results in that figure, a sampling rate of 9 was used. For optimized embeddings, in terms of DTW cell cost, performance clearly improves with increased di-

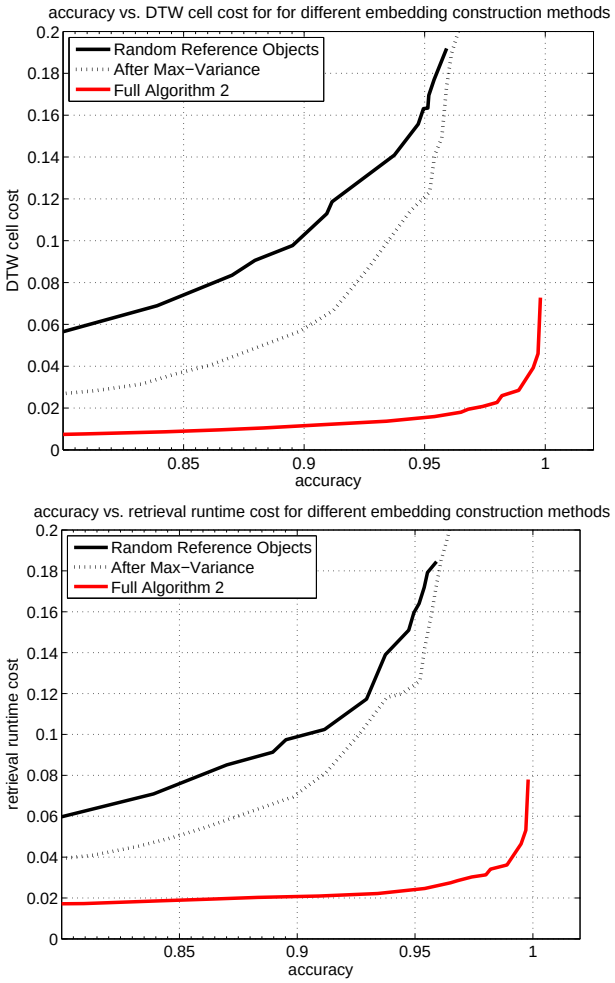

Figure 8: Accuracy vs. efficiency for EBSM, using embeddings constructed randomly, optimized with the max variance heuristic, and optimized using Algorithm 2 for embedding optimization. The top figure measures efficiency using the DTW cell cost, and the bottom figure measures efficiency using the retrieval runtime cost. The costs shown are average costs over our test set of 1000 queries.

mensionality up to about 40 dimensions, and does not change much between 40 and 160. Actually, 160 dimensions give a somewhat worse DTW cell cost compared to 40 dimensions, providing evidence that our embedding optimization method suffers from a mild effect of overfitting as the number of dimensions increases. When reference objects are selected randomly, overfitting is not an issue. As we see in Figure 9, a 160-dimensional unoptimized embedding yields a significantly lower DTW cell cost than lower-dimensional unoptimized embeddings.

In terms of offline preprocessing costs, selecting 40 reference sequences using Algorithm 2 took about 3 hours, and computing the 40-dimensional embedding of the database took about 240 seconds.

Code and datasets for duplicating the experiments described here are publicly available on our project website, at two mirror sites:

- `http://cs-people.bu.edu/panagpap/ebsm/`
- `http://crystal.uta.edu/~athitsos/ebsm/`

9. DISCUSSION AND FUTURE WORK

EBSM, the method proposed in this paper, was shown to significantly outperform the current state-of-the-art methods for subsequence matching under unconstrained DTW. At the same time, the idea of using embeddings to speed up subsequence matching opens up several directions for additional investigation, both for improving performance under unconstrained DTW, and for extending the current formulation to additional settings.

The proposed embeddings treat every position of every database sequence as a candidate *endpoint* for the optimal subsequence match. It is fairly straightforward to change our formulation so that it treats every database position as a candidate *startpoint*. The open question is how to combine both approaches, by simultaneously using embeddings of endpoints and embeddings of startpoints.

It is worth noting that the PDTW method of [17] is not a direct competitor of our method, but rather a complimentary method, that can possibly be combined with our method to provide even better results. For example, PDTW can be used to speed up computing the embedding of the query, or to introduce a PDTW-based additional filter step after our current filter step and before the final refinement. Alternatively, our method could be used to quickly identify candidate database areas which would then be explored using PDTW. Identifying the best way to combine EBSM with PDTW is an interesting topic for future work.

The discussion in this paper has focused on finding the optimal subsequence match for each query. It is pretty straightforward to also apply our method for retrieving top-k subsequence matches: we simply modify the refine step to return the k-best startpoint-endpoint pairs. It will be interesting to evaluate how accuracy and efficiency vary with k.

Another interesting direction is applying our method in different settings, such as subsequence matching under constrained DTW and the edit distance. The key idea of embedding database positions, as opposed to existing approaches that embed entire database sequences, can readily be extended to both constrained DTW and the edit distance. Perhaps by exploiting known lower bounds of constrained DTW [16], or by using the metric properties of the edit distance, we can obtain an exact indexing scheme for embedding-based subsequence matching under those distance measures.

In conclusion, the proposed EBSM method is the first subsequence matching method for unconstrained DTW that converts, at least partially, the subsequence matching problem into a much easier vector matching problem. As a result, a relatively small number of database areas of interest can be identified very fast, over two orders of magnitude faster compared to brute-force search in our experiments. The computationally expensive DTW algorithm is still employed within EBSM, but only to refine results by evaluating the identified database areas of interest. The resulting end-to-end retrieval system is one to two orders of magnitude faster than brute-force search,with relatively small losses in accuracy, and provides state-of-the-art performance in the experiments.

Acklownedgements

This work was supported in part by grants NSF HCC-0705749 and NSF CAREER IIS-0133825. V. Athitsos' research was also supported by his UT Arlington startup grant. D. Gunopulos' research was supported by the NSF IIS-0534781, Aware and Health-e-Child projects.

10. REFERENCES

[1] J. Alon, V. Athitsos, and S. Sclaroff. Accurate and efficient gesture spotting via pruning and subgesture reasoning. In

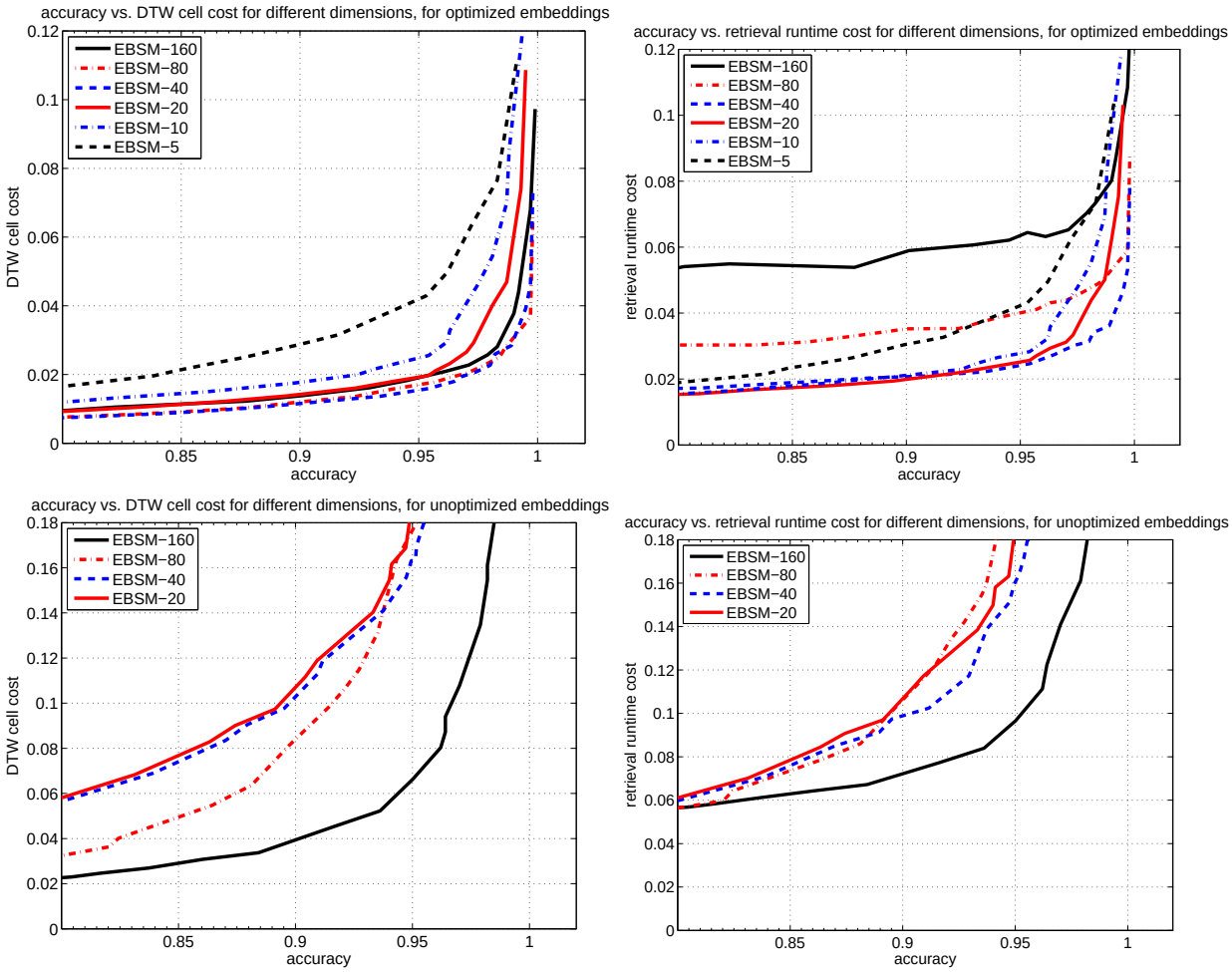

Figure 9: Accuracy vs. efficiency for EBSM, using embeddings with different dimensionality. The plots on the left measure efficiency using the DTW cell cost, and the plots on the right measure efficiency using the retrieval runtime cost. The costs shown are average costs over our test set of 1000 queries. The top plots show results for embeddings optimized using Algorithm 2. The bottom plots show results for embeddings with randomly selected reference objects.

IEEE Workshop on Human Computer Interaction, pages 189–198, 2005.

[2] T. Argyros and C. Ermopoulos. Efficient subsequence matching in time series databases under time and amplitude transformations. In *International Conference on Data Mining*, pages 481–484, 2003.

[3] V. Athitsos, M. Hadjieleftheriou, G. Kollios, and S. Sclaroff. Query-sensitive embeddings. In *ACM International Conference on Management of Data (SIGMOD)*, pages 706–717, 2005.

[4] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[5] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *International Conference on Very Large Data Bases*, pages 89–100, 2000.

[6] Ö. Egecioglu and H. Ferhatosmanoglu. Dimensionality reduction and similarity distance computation by inner product approximations. In *International Conference on Information and Knowledge Management*, pages 219–226, 2000.

[7] C. Faloutsos and K. I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *ACM International Conference on Management of Data (SIGMOD)*, pages 163–174, 1995.

[8] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *ACM International Conference on Management of Data (SIGMOD)*, pages 419–429, 1994.

[9] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *International Conference on Very Large Databases*, pages 518–529, 1999.

[10] D. Q. Goldin and P. C. Kanellakis. On similarity queries for time-series data: Constraint specification and implementation. In *International Conference on Principles and Practice of Constraint Programming*, pages 137–153, 1995.

[11] W.-S. Han, J. Lee, Y.-S. Moon, and H. Jiang. Ranked

subsequence matching in time-series databases. In *International Conference on Very Large Data Bases (VLDB)*, pages 423–434, 2007.

[12] G. Hjaltason and H. Samet. Properties of embedding methods for similarity searching in metric spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5):530–549, 2003.

[13] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, 2003.

[14] G. Hristescu and M. Farach-Colton. Cluster-preserving embedding of proteins. Technical Report 99-50, CS Department, Rutgers University, 1999.

[15] K. V. R. Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *ACM International Conference on Management of Data (SIGMOD)*, pages 166–176, 1998.

[16] E. Keogh. Exact indexing of dynamic time warping. In *International Conference on Very Large Data Bases*, pages 406–417, 2002.

[17] E. Keogh and M. Pazzani. Scaling up dynamic time warping for data mining applications. In *Proc. of SIGKDD*, 2000.

[18] E. Keogh, X. Xi, L. Wei, and C. A. Ratanamahatana. The UCR time series classification/clustering homepage: www.cs.ucr.edu/~eamonn/time_series_data/, 2006.

[19] N. Koudas, B. C. Ooi, H. T. Shen, and A. K. H. Tung. LDC: Enabling search by partial distance in a hyper-dimensional space. In *IEEE International Conference on Data Engineearing*, pages 6–17, 2004.

[20] J. B. Kruskall and M. Liberman. The symmetric time warping algorithm: From continuous to discrete. In *Time Warps*. Addison-Wesley, 1983.

[21] H. Lee and J. Kim. An HMM-based threshold model approach for gesture recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(10):961–973, October 1999.

[22] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):792–808, 2002.

[23] Y. Moon, K. Whang, and W. Han. General match: a subsequence matching method in time-series databases based on generalized windows. In *ACM International Conference on Management of Data (SIGMOD)*, pages 382–393, 2002.

[24] Y. Moon, K. Whang, and W. Loh. Duality-based subsequence matching in time-series databases. In *IEEE International Conference on Data Engineering (ICDE)*, pages 263–272, 2001.

[25] P. Morguet and M. Lang. Spotting dynamic hand gestures in video image sequences using hidden Markov models. In *IEEE International Conference on Image Processing*, pages 193–197, 1998.

[26] R. Oka. Spotting method for classification of real world data. *The Computer Journal*, 41(8):559–565, July 1998.

[27] S. Park, W. W. Chu, J. Yoon, and J. Won. Similarity search of time-warped subsequences via a suffix tree. *Information Systems*, 28(7), 2003.

[28] S. Park, S. Kim, and W. W. Chu. Segment-based approach for subsequence searches in sequence databases. In *Symposium on Applied Computing*, pages 248–252, 2001.

[29] D. Rafiei and A. O. Mendelzon. Similarity-based queries for time series data. In *ACM International Conference on Management of Data (SIGMOD)*, pages 13–25, 1997.

[30] T. M. Rath and R. Manmatha. Word image matching using dynamic time warping. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages 521–527, 2003.

[31] Y. Sakurai, C. Faloutsos, and M. Yamamuro. Stream monitoring under the time warping distance. In *IEEE International Conference on Data Engineering (ICDE)*, 2007.

[32] Y. Sakurai, M. Yoshikawa, and C. Faloutsos. FTW: fast similarity search under the time warping distance. In *Principles of Database Systems (PODS)*, pages 326–337, 2005.

[33] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *International Conference on Very Large Data Bases*, pages 516–526, 2000.

[34] Y. Shou, N. Mamoulis, and D. W. Cheung. Fast and exact warping of time series using adaptive segmental approximations. *Machine Learning*, 58(2-3):231–267, 2005.

[35] E. Tuncel, H. Ferhatosmanoglu, and K. Rose. VQ-index: An index structure for similarity searching in multimedia databases. In *Proc. of ACM Multimedia*, pages 543–552, 2002.

[36] J. Venkateswaran, D. Lachwani, T. Kahveci, and C. Jermaine. Reference-based indexing of sequence databases. In *International Conference on Very Large Databases (VLDB)*, pages 906–917, 2006.

[37] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 216–225, 2003.

[38] X. Wang, J. T. L. Wang, K. I. Lin, D. Shasha, B. A. Shapiro, and K. Zhang. An index structure for data mining and clustering. *Knowledge and Information Systems*, 2(2):161–184, 2000.

[39] R. Weber and K. Böhm. Trading quality for time with nearest-neighbor search. In *International Conference on Extending Database Technology: Advances in Database Technology*, pages 21–35, 2000.

[40] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *International Conference on Very Large Data Bases*, pages 194–205, 1998.

[41] D. A. White and R. Jain. Similarity indexing: Algorithms and performance. In *Storage and Retrieval for Image and Video Databases (SPIE)*, pages 62–73, 1996.

[42] H. Wu, B. Salzberg, G. C. Sharp, S. B. Jiang, H. Shirato, and D. R. Kaeli. Subsequence matching on structured time series data. In *ACM International Conference on Management of Data (SIGMOD)*, pages 682–693, 2005.

[43] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *IEEE International Conference on Data Engineering*, pages 201–208, 1998.

[44] Y. Zhu and D. Shasha. Warping indexes with envelope transforms for query by humming. In *ACM International Conference on Management of Data (SIGMOD)*, pages 181–192, 2003.