EMBEDDING-BASED SUBSEQUENCE MATCHING IN LARGE SEQUENCE DATABASES

PANAGIOTIS PAPAPETROU

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

BOSTON

UNIVERSITY

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

EMBEDDING-BASED SUBSEQUENCE MATCHING IN LARGE SEQUENCE DATABASES

by

PANAGIOTIS PAPAPETROU

B.S., University of Ioannina, Greece, 2003,
M.A., Boston University, 2006

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2010

Approved by

First Reader

George Kollios, PhD
Associate Professor of Computer Science

Second Reader

Stan Sclaroff, PhD
Professor of Computer Science

Third Reader

Margrit Betke, PhD
Associate Professor of Computer Science

Acknowledgments

First of all, I thank my advisor, George Kollios, for his guidance, patience, and friendship throughout my PhD studies. He has been an excellent advisor for all these years and has influenced me significantly in discovering my own research interests.

A huge thanks to Vassilis Athitsos for all his help, discussions, collaboration, and support during the last years of my PhD studies. The core contributions of this thesis stemmed from our numerous discussions; his help was priceless. I also thank Professors Stan Sclaroff, Margrit Betke, Dimitrios Gunopulos, and Steve Homer for participating in my committee and for providing many useful comments that helped me improve the quality of the document.

I would also like to thank all students, professors, instructors, and staff I have interacted with in the Computer Science Department, for all the times they have helped and supported me. A special thanks to my officemate Michalis Potamias with whom we had great times inside and outside the lab.

All my relatives in the United States and Greece have played a major role in supporting me throughout this period. I am extremely thankful to all, especially my mother Vasiliki and my cousins Katerina, Thodoris, Perry, Leo, and Ioanna. Finally, I would like to commemorate my father Petros. Even though I lost him 18 years ago, his memories served as inspiration and motivation to pursue and complete my PhD studies.

I am very grateful to my friends in and outside Boston who have made a huge difference in my life throughout my PhD studies. The space and time are really constrained to list everyone who deserves to be here. However, I should especially thank my Boston friends: Georgios Smaragdakis, Niky Riga, Yola Katsargyri, Georgios Zervas, Christos Thomidis, Vijay Erramilli, and Kathleen Maniataki, who have been precious parts of my life in Boston. A special thanks to my best childhood friends: Vassilis Panoulas, Stathis Ioannou, Angeliki Koloka, Chrysa Kotsoni, Vasiliki and

Stavroula Liaska, Katerina Kostoula, Maria Gouveli, and Maria Konstantakopoulou. Despite the 4,731-mile distance, we have managed to keep our friendship intact. Their support throughout these years was priceless.

I would like to thank the “Georgios Stavros Scholarship Foundation” for the partial financial support of my PhD studies for the years 2004-2007, as well as the Mitropolis of Ioannina and Bishop Theoklitos.

This work was supported in part by the National Science Foundation under grants IIS-0329009 and IIS-0812309.

Finally, I would like to dedicate this thesis to my mother Vasiliki. She devoted the greatest part of her life to me and my education. Her inspiration and support during my PhD studies was precious.

EMBEDDING-BASED SUBSEQUENCE MATCHING

IN LARGE SEQUENCE DATABASES

(Order No.)

PANAGIOTIS PAPAPETROU

Boston University, Graduate School of Arts and Sciences, 2010

Major Professor: George Kollios, Associate Professor of Computer Science

ABSTRACT

Sequential data, such as time series and categorical sequences, naturally appear in a wide variety of domains including financial and scientific data, human activity, biological sequences, etc. In such domains large databases of sequences are used as knowledge repositories. Information retrieval from such repositories is challenging, due to the large amount of data that needs to be searched.

Our attention is focused on subsequence matching methods that employ dynamic programming-based distance measures. Such approaches are robust to misalignments and time warps and are widely used for time series and DNA matching. Three methods are proposed for efficient subsequence matching in large sequence databases.

The first method works for time series databases and the *Dynamic Time Warping (DTW)* distance. It converts subsequence matching to vector matching using an embedding that maps each database time series into a sequence of vectors. The embedding is computed by applying the full DTW distance between each reference and database time series. At query time, the embedding of the query time series is computed in a similar manner. Relatively few areas of interest are identified by performing vector comparisons, and are then fully explored using the exact DTW distance. The second method defines a similar type of embedding that stores additional information into the vector representation and significantly improves the efficiency of subsequence matching under the

constrained Dynamic Time Warping (cDTW) distance.

The third method speeds up retrieval of optimal subsequence matches in string databases, under the Edit Distance and the Smith-Waterman similarity measure. Filtering of candidate matches is performed using precomputed alignment scores between the database sequence and a set of fixed-length reference sequences. At query time, the query sequence is partitioned into segments of the same length as the reference sequences. For each of those segments, the alignment scores between the segment and the reference sequences are used to efficiently identify a relatively small number of candidate matches. Experiments show that the proposed method outperforms BLAST by over an order of magnitude in retrieval runtime for large queries (up to $10,000$ bases) and similarity levels of up to 15%.

Contents

List of Tables

List of Figures

List of Abbreviations

1D	. . . . .	One-dimensional
2D	. . . . .	Two-dimensional
EBSM	. . . . .	Embedding-Based Subsequence Matching
BSE	. . . . .	Bidirectional Subsequence Embeddings
RBSA	. . . . .	Reference-Based Sequence Alignment
DP	. . . . .	Dynamic Programming
ED	. . . . .	Edit Distance
SW	. . . . .	Smith Waterman
DTW	. . . . .	Dynamic Time Warping
cDTW	. . . . .	constrained Dynamic Time Warping
PAA	. . . . .	Piecewise Aggregate Approximation
LB	. . . . .	Lower Bound

To my mother Vasiliki, the most important person in my life.

Chapter 1

Introduction

Subsequence matching is the problem of identifying, given a query sequence and a database of sequences, the database *subsequence* that best matches 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. Motivating applications include keyword-based search in handwritten documents, DNA and protein matching, query-by-humming, etc.

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 small misalignments between sequences, so as to allow for time warps in time series data and insertions/deletions in strings. Typically, similarity between sequences is measured using algorithms based on dynamic programming (DP). In particular, dynamic time warping (DTW) [42] is widely used for time series data, and the edit distance [45] is used for strings and biological sequences.

This thesis is focused on two types of sequences: time series and biological sequences.

1.1 Searching Time Series Databases

Time series data naturally appear in a wide variety of domains, including financial data (e.g. stock values), scientific measurements (e.g. temperature, humidity, earthquakes), medical data (e.g. electrocardiograms), audio, video and human activity. Improved algorithms for time series subsequence matching can make a big difference in real-world applications such as query by humming [95], word spotting in handwritten documents, and content-based retrieval in large video databases and motion capture databases.

For the case of time series sequences, one commonly used similarity measure is the Euclidean Distance and generally the L_p measures. However, these measures fail to identify misalignments and warps in the time axis. Typically, similarity between time series is measured using dynamic time warping (DTW) [42], 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 [34].

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 [44, 58, 64, 75]. Constrained Dynamic Time Warping (cDTW) is a modification of DTW that places constraints on the possible alignment between two sequences [34, 76]. In cDTW each position in one sequence can only be matched to a relatively short range of positions in the other sequence. These constraints have been shown to improve the meaningfulness of the results in many applications, as measured for example based on nearest neighbor classification accuracy [72]. Constraints can improve accuracy by eliminating from consideration pathological cases, i.e., accidental alignments that are legal (in the absence of constraints) and produce optimal scores, but do not capture a meaningful correspondence between the two time series.

The aforementioned DP-based algorithms can be used both for full sequence and subsequence matching, and identify the globally optimal subsequence match for a query in time linear to the length of the database [44, 58, 64, 75]. 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.

The first contribution of this thesis is an approximate method for Embedding-Based Subsequence Matching (EBSM) [6]. 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.

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 [7, 25, 90]. 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 experiments with real time series data, 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 1·1 illustrates the flowchart of the offline and the online stages of the proposed method. The key idea behind EBSM is that the subsequence matching problem can be partially converted to the much more manageable problem of nearest neighbor retrieval in a real-valued 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.

The second topic studied in this thesis is efficient subsequence matching under cDTW. An approximate method is proposed, that introduces a new embedding, called Bidirectional Subse-

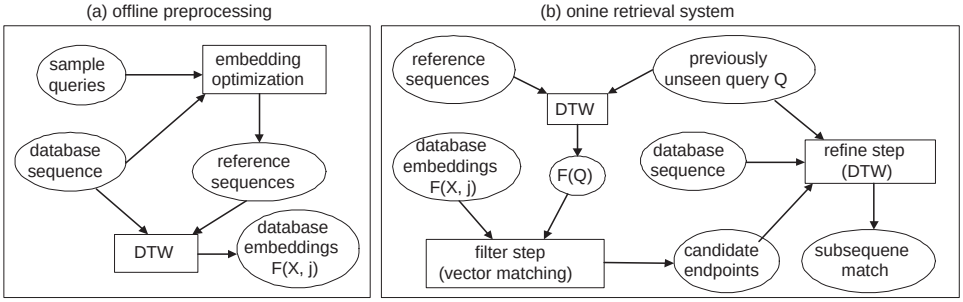

Figure 1·1: System modules are shown as rectangles, and input/output arguments are shown as ellipses. The goal of the online stage is to identify, given a query time series Q, its optimal subsequence match in the database.

quence Embedding (BSE), that manages to trade accuracy for efficiency and can yield significant performance gains in practice. Each query is mapped into a vector, and each database sequence is mapped into an equally long sequence of vectors. Given a query, there is a one-to-one correspondence between each database vector and a possible subsequence match for the query. If the query is very similar to a subsequence, we expect the embedding of the query to be similar to the vector corresponding to that subsequence. This approach allows a relatively short number of candidate matches to be identified using efficient vector comparisons. The main differentiation from EBSM is that BSE exploits some properties of cDTW and uses both end-point and start-point embeddings to define the embedding index. Experiments on real time series data have shown significant speedups (of at least one order of magnitude) in contrast with standard methods for time series matching under cDTW.

As mentioned earlier, in many domains cDTW has been shown to identify more meaningful matches than unconstrained DTW [72]. A method that is explicitly designed for efficient subsequence matching under cDTW can help obtain these meaningful results more efficiently, and can make the use of cDTW a realistic option for larger datasets than it was possible with existing methods. While EBSM is an embedding method for subsequence matching under unconstrained DTW [6], BSE focuses on embedding-based subsequence matching under cDTW.

1.2 Searching String and Biological Sequence Databases

There are many applications that require fast searching in sequence databases that consist of collections of strings. Given a query string, the goal is to find the most similar substrings in the database using a distance/similarity measure such as the edit distance (ED) or Smith-Waterman (SW). Applications in this area include 1) spell-checking: given some input text the spell-checker consults its dictionary to find words of high similarity to the text, so as to identify potential typos, 2) data cleaning: data obtained from different sources might contain inconsistencies which can be eliminated by looking for similar entities (strings) in the data, 3) homology search in biological sequences: given different genomes we want to find regions of high similarity that were the result of a mutation, etc. Being able to efficiently answer such queries is crucial, especially for online string search applications.

In order to generate and interpret complete genomes of different organisms, various searches need to be performed that 1) involve queries of large length, and 2) only target *near exact* matches [21, 29]. We focus on these two major requirements: we want to be able to retrieve *near-exact matches* of *long query sequences* efficiently. As a motivating example for large query lengths, consider large EST (Expressed Sequence Tag) databases, that contain portions of genes expressed as mature mRNA. In such databases, large scale searches need to be performed against other genomic databases to determine locations of genes [29]. In practice, genes can vary in size from hundreds to millions of nucleotides. Searches can also target whole chromosomes, where the goal is to find chromosome similarities across different organisms. Since chromosomes can be relatively large (e.g. Human Chromosome 1 is approximately 272 million bases), such searches require algorithms that can handle large queries efficiently.

In many applications, database matches are of interest only if their deviation from the query does not exceed a certain, relatively small, fraction of the query length [9, 21, 33]. We denote that fraction as δ and focus on values of δ up to 15%, which is typical in applications, such as shotgun sequencing [59] and mutation analysis [74]. Notice that our focus is on DNA sequences, where the alphabet size is small (4) and the query size can be large (up to $10,000$ bases). In this setting,

only near homology search is biologically significant, whereas *remote homology search* is more meaningful and mostly used not for DNA, but for protein sequences.

In this thesis, we propose a novel method, called reference-based string alignment (RBSA) [65] for efficient subsequence matching in large databases of strings under the edit distance or the Smith-Waterman similarity measure. In RBSA, we decompose the subsequence matching problem into two distinct problems:

- **The fixed query length problem:** achieve efficient retrieval assuming that all queries have the same length.
- **The variable query length problem:** using a solution to the fixed query length problem, achieve efficient retrieval for queries of arbitrary length.

To solve the fixed query length problem, RBSA precomputes, for each position of every database string, alignment scores corresponding to different reference sequences. These alignment scores are based on the edit distance. Given a query, alignment scores between the query and all reference sequences are computed online. These alignment scores are used to prune away large portions of the database, so as to leave a relatively small number of candidate matches. We can guarantee that the optimal subsequence match will be included among the candidates. Exact alignment scores (using the edit distance or Smith-Waterman) are then computed to identify the optimal match among the remaining candidates. One of the main contributions in this thesis is in showing how to use alignment scores with reference sequences to achieve efficient subsequence matching for fixed query lengths.

To solve the variable query length problem, the RBSA first breaks up that problem into multiple fixed query length problems, by partitioning the query sequence into segments of fixed length. In the exact version of RBSA, all query segments are considered, and subsequence matches found for those segments are used to identify candidate subsequence matches for the entire query. In the approximate version of RBSA, only a subset of query segments is considered. Another contribution in this thesis consists in showing that the probability of failing to find the optimal match drops very fast (exponentially) as we increase the number of query segments that we consider, and thus

we can achieve both significantly improved efficiency and very high accuracy rates by considering only a relatively small number of segments.

An important advantage of RBSA compared to BLAST and its variants is that RBSA has an exact version that provably achieves 100% accuracy, and as shown in the experimental evaluation outperforms BLAST (which is the most widely used method for near-exact homology search in DNA sequences) by more than an order of magnitude even for large query lengths. Compared to other exact methods, such as OASIS[54] and BWT-SW[43], that are used for near-exact homology search of short queries in biological sequences, RBSA achieves a retrieval runtime of more than one order of magnitude. Moreover, RBSA differs from (MV and MP) [85] in that it is developed for substring matching and not for full string matching.

1.3 Contributions

The main contributions of this thesis are summarized below.

EBSM (Embedding-Based Subsequence Matching) is a method for speeding up subsequence matching in time series databases. It is the first to explore the usage of embeddings for subsequence matching for unconstrained DTW. The key differentiating features of EBSM are the following:

- EBSM converts, at least partially, subsequence matching under DTW into a much easier vector matching problem. Vector similarity retrieval is used to identify very fast a relatively small 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) [36], is a generic method for speeding up DTW.
- In experiments with real time series data, 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 [75] that uses the standard DTW algorithm, and the approximate PDTW method.

BSE (Bidirectional Subsequence Embeddings) is a bi-directional embedding-based method for subsequence matching under cDTW. The main features of BSE are described below:

- On the algorithmic side, we exploit the constraints of cDTW to define a new embedding, BSE, that includes more information than previously proposed embeddings for this problem. This information includes startpoint embeddings that are defined in a manner similar to that of endpoint embeddings. An interesting feature of BSE embeddings is that they are customized, online, to the length of each query.
- On the practical side, we provide experimental results on real time series data where our method produces speedups of one to two orders of magnitude compared to the state-of-the-art methods of LB_Keogh [34] and DTK [56], at the cost of 1% to 20% loss in retrieval accuracy. We believe that this trade-off between accuracy and efficiency is highly desirable for many real world settings. Furthermore, our method has shown speedups that significantly outperform competing methods by over an order of magnitude. It is also able to tolerate large warping widths and large query lengths as shown in the experiments.
- Another important practical result is that, unlike results previously published on unconstrained DTW [6], our embedding method has very good perfornamce without using training data for embedding optimization. Not requiring training data makes it much easier to implement and deploy our method in real-world systems.
- In the experiments we apply our method to real-world datasets of time series as well as random walk synthetic datasets. The proposed BSE embedding significantly speeds up subsequence retrieval with relatively small losses in accuracy (by at most 5%), and performance compares favorably to that of existing state-of-the-art methods (LB_Keogh [34] and DTK [56]) for constrained subsequence matching. Our method also out-

performs the embedding method of [6], thanks to the novel embeddings, explicitly designed for cDTW, that we introduce in this thesis.

RBSA (Reference-Based Sequence Alignment) is the first reference based method for subsequent matching in string databases that both guarantees no false dismissals and performs well for large queries. The main characteristics of RBSA are described below:

- RBSA produces lower bounds of the edit distance and upper bounds of the Smith-Waterman similarity between the query and database subsequences using precomputed alignment scores with reference sequences. In prior work, such bounds have only been derived for full sequence matching [85].
- An exact method is presented for decomposing the variable-length query problem into multiple fixed-length queries, so that we can achieve significant retrieval runtimes (one to two orders of magnitude faster than the Edit Distance [45], Smith-Waterman [79] and BLAST [2]) for long queries, while still guaranteeing correct results.
- An approximate method is presented for decomposing the variable-length query problem into multiple fixed-length queries. At the same time, the probability of missing the correct result in approximate RBSA drops exponentially with the number of query segments that we consider, and thus can easily be reduced to a negligible quantity. The experimental evaluation shows that, for query lengths ≥ 200, RBSA outperforms current state-of-the-art sequence alignment methods: BLAST2[2], BWT-SW[43] and q-grams. Speedups of one to two orders of magnitude over the current state of the art are demonstrated for query sizes $\geq 2,000$.

1.4 Roadmap

The remainder of this thesis is organized as follows:

Chapter 2 This chapter describes the related work on subsequence matching in time series and biological sequence databases and places our contribution in the current literature.

Chapter 3 The first contribution of this thesis includes an analysis of the background on time series subsequence matching, the description of the proposed method, EBSM, and an experimental evaluation against state-of-the-art time series subsequence matching methods.

Chapter 4 The second contribution of this thesis includes the definition of bidirectional subsequence embeddings and the method that uses them to perform efficient subsequence matching under cDTW. Experiments show the superiority of the proposed method against state-of-the-art constrained subsequence matching methods.

Chapter 5 The third contribution of this thesis is a reference based method for subsequence matching (RBSA) in string databases.

Chapter 6 This last chapter provides a discussion of the proposed methods underlying their major contributions and pointing out their limitations. Finally, directions for future work are discussed.

1.5 List of related papers

Parts of this thesis are based on the material from the following published papers:

Chapter 3 is mostly based on:

V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios and D. Gunopulos. "Approximate embedding-based subsequence matching of time series," in ACM SIGMOD, pages 365–378, 2008.

Chapter 5 is mostly based on:

P. Papapetrou, V. Athitsos, G. Kollios and D. Gunopulos. "Reference-Based Alignment of Large Sequence Databases," in VLDB, 2009 (To Appear).

Chapter 2

Related Work

2.1 Literature on Time Series Subsequence Matching

A large body of literature addresses the problem of efficient sequence matching. Several methods assume that sequence similarity is measured using the Euclidean distance [19, 14, 55, 57] or variants [3, 71, 91]. However, such methods cannot handle even the smallest misalignment caused by time warps, insertions, or deletions. Robustness to misalignments is achieved using distance measures based on dynamic programming (DP), such as DTW [42]. In the remaining discussion we restrict our attention to the DTW distance measure, which is the most popular measure for time series.

Sequence 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 [34, 76, 86, 93], and cannot be easily used for efficient retrieval of subsequences.

Some methods reduce subsequence matching to full sequence matching, by cutting database sequences into small pieces, and requiring each query to correspond to an entire such piece. One example is the query-by-humming system described by Zhu et al. [95], where each database song is cut into smaller, disjoint pieces. Another example is the method for word search in handwritten documents described by Rath et al. [73], where, as preprocessing, the documents are segmented automatically into words, and full sequence matching is performed between query words and database words. Such approaches fail when the query corresponds to a database subsequence that is not stored as a single piece.

An indexing structure for unconstrained DP-based subsequence matching is proposed by Park et al. [66]. However, as database sequences get longer, the time complexity for that method becomes similar to that of unoptimized DP-based matching. The method by Park et al. [67] can handle such long database sequences; the key idea 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 be used for multidimensional time series is proposed by Keogh et al. [36]. In that method, 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.

The SPRING method [75] has been developed for efficient subsequence matching under unconstrained DTW. In that method, optimal subsequence matches are identified by performing full sequence matching between the query and each database sequence. Subsequences are identified by prepending to each query a "null" symbol that matches any sequence prefix with zero cost. The complexity of that method is linear to both database size and query size. 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 DTK method [56] is a method for subsequence matching under cDTW. DTK breaks the database into small non-overlapping sequences and further employs the piece-wise approximation method (PAA) [36] for efficient indexing. This approach however, does not scale well as the query size increases, as shown in the experiments of chapter 4. A similar approach is used to index time series for sequence and subsequence matching under scaling and dynamic time warping [20]. Actually, when the scaling factor is 1 (no scaling at all), the indexing and query algorithm of Moon et al. [56] are the same as the ones proposed by Fu et al. [20]. Therefore, since here we do

not consider scaling, we just use the DTK as a competitor BSE.

The more powerful lower-bounding method LB_Keogh for efficient time series matching under cDTW is described in [34]. The main idea is to use the warping constraint to create an envelope around the query sequence. Then, using a sliding window of size equal to the query, we can estimate a lower bound of the matching cost between the query and each possible subsequence. Since LB_Keogh gives a lower bound on the actual distance, this approach can be used to prune a large number of subsequences. For the subsequences that cannot be pruned, the exact dynamic programming algorithm is used to compute the distances and ultimately find the best match. However, as shown in our experiments, performance of LB_Keogh is highly dependent on the warping width parameter w and the query size; performance deteriorates as warping width and query size increase. The second method proposed in this thesis, BSE, achieves significant speedups even for high warping widths and long query sizes (1000). Furthermore, computing the LB_Keogh for each possible subsequence can be time consuming for large databases. Note that, although some improvements to the LB_Keogh have been proposed (e.g. Shou et al. [78]), these improvements achieve not more than a small constant factor in terms of both the tightness of the lower bound and the query time performance. Therefore, we can use LB_Keogh as a good yardstick to evaluate the performance of BSE. Compared to EBSM, BSE is explicitly designed to take advantage of the constraints of cDTW. This novel embedding stores additional information at the same amount of space, and thus leads to better performance, as shown in the experiments. Furthermore, EBSM embeddings are fixed regardless of the length of the query, whereas BSE embeddings are customized online to the length of each query, so as to contain information highly relevant for that query length.

The two time series subsequence matching methods proposed in this thesis are 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 [26], FastMap [18], MetricMap [87], SparseMap [27], and BoostMap [4, 5]. Such embeddings can be used for speeding up full sequence matching [4, 5, 27]. However, the above-mentioned embedding methods can only be used for full sequence matching, not subsequence matching.

2.2 Literature on String Subsequence Matching

A preeminent group of methods for string subsequence matching are based on dynamic programming [45]. Needleman et al. [62], describes a global alignment method, where both query and database sequences are aligned along their entire lengths, using *match*, *mismatch* and *gap* scores. A similar, but generalized algorithm [28] for global alignment, handles sequences of intermittent similarities. Smith and Waterman [79] developed a dynamic programming approach for local alignment, where a subsequence of the query is matched to a subsequence of the database. Ukkonen et al. [83] exploits the fact that in approximate string searching we are looking for patterns that match with substrings of the text with at most k errors. Thus, it speeds up the dynamic programming (DP) computation by pruning cells in the DP matrix with values larger than k.

Several q-gram-based methods [9, 10, 11, 37, 47, 48, 52, 60, 92] have been developed to solve the problem of exact and approximate string matching in large sequence databases. Their main characteristic is that they build a dictionary of words on a given database of sequences. At query time the query is broken into a set of overlapping q-grams and the index is searched for exact matches of those q-grams. These matches provide candidate hits that are later refined to remove false positives.

QUASAR [9] is a subsequence matching method that performs q-gram based filtering on a sequence database. QUASAR is limited to relatively short queries (the maximum query length on which the performance of QUASAR was evaluated was 393 characters) of high similarity to the database. A generalization of QUASAR, which uses gapped instead of contiguous q-grams is described in [10]. Similar q-gram based methods for approximate full string matching are described in [47, 48, 92].

VGRAM [48] employs a q-gram dictionary where the words are of variable length and more representative of the dataset. Again, the limitations to small queries persist (the experimental evaluation reports queries of average size ranging from 8 to 62 characters) and the performance seriously deteriorates as k (the number of edit operations applied to the queries) increases (> 4). An improved vgram-based method is described by Yang et al. [92], but is again limited to small

query sizes (varying between 4 and 249 characters). Several methods [37, 11] employ a two level q-gram index to speed up the database search. A q-gram based approximate string matching method is described by Navarro et al. [60], where disjoint text substrings of length q are collected by the index at fixed intervals. Finally, Li et al. [47] introduces several strategies for improving the join cost of the gram lists found during a query search in an inverted q-gram index and shows how to incorporate these strategies into existing filtering methods to improve string matching.

A key property of q-gram based methods, such as the ones mentioned above, is the following: if the query size is $|Q|$ and we are searching for matches with edit distance within k, q can be at most $\lceil |Q|/(k+1) \rceil$ to guarantee no false dismissals. It can be seen that as k increases, q decreases, and thus, the index size becomes larger. Consequently, and also as shown in the experiments, q-gram based approaches can only handle short queries of relatively high similarity to the database. However, the biologically interesting types of queries (e.g. mutated genes) can be significantly long (up to $10,000$ nucleotides or more [43]) and thus, q-gram based methods are not able to handle them efficiently.

Another group of methods has been proposed for *exact string matching*, targeting exact occurrences of the query sequence in a database [8, 15, 16, 32, 39, 52, 53, 70, 84]. However, exact string matching is quite different from the main focus of this paper and thus, these methods are not discussed any further.

Several methods have been developed for aligning biological sequences. FASTA [51, 68] detects locally similar regions between two sequences using only identities and no gaps, and then based on some measure of similarity it re-scores them accordingly. Additional heuristics are proposed in BLAST [1]. Given a query (DNA or protein), BLAST performs a linear scan on the sequence database searching for a set of seeds belonging to the neighborhood of some substrings of the query. Having identified a set of candidate hits, it then extends them both ways, until the accumulated similarity score begins to decrease. Finally, BLAST reports as matches those regions with high statistical significance.

A new version of BLAST, known as BLAST2 [2], improves accuracy by allowing a limited number of insertions and deletions during the alignment formation and improves search speed

by imposing more stringent criteria when performing a local alignment. Further improvements of BLAST include MegaBLAST [94], MPBLAST [40] and miBLAST [38]. MegaBLAST is a greedy algorithm for detecting sequences that differ slightly as a result of sequencing. MPBLAST and miBLAST are different versions of BLAST used for parallel queries.

BLAT [1] builds an index of the database and then given a query, it linearly scans the query searching for matches in the index. Apart from using an inverse index, BLAT differs from BLAST and BLAST2 in that it triggers extensions on any number of perfect hits whereas in BLAST extensions are triggered when one or two hits occur in proximity to each other. Several hash-based approaches [30, 63] have been developed for further speed up. A key limitation of all the above-mentioned variants of BLAST is that their accuracy and retrieval cost deteriorates as the query size increases. As the volume of biological sequence databases increases, all the aforementioned exhaustive systems become prohibitively expensive.

Another key limitation of BLAST-like approaches is that there is no guarantee that the optimal local alignment will be reported. Several methods have been developed to handle this weakness. OASIS [54] employs a best first search technique over a suffix tree for string alignment. The algorithm outperforms BLAST by an order of magnitude, but only for small query sizes (5 to 60); this is one of its major limitations. Another indexing method that uses suffix trees is discussed by Navarro et al. [61], whereas Phoophakdee et al. [69] discusses an efficient method for suffix tree construction in external memory. Finally, BWT-SW[43] employs a suffix array to speedup local alignment search in biological sequences. It outperforms BLAST for queries of size up to 1000; for larger queries its performance deteriorates. Both OASIS and BWT-SW always find the best local alignment according to Smith-Waterman.

Two reference-based indexing methods for full sequence matching are proposed by Venkateswaran et al. [85] that use reference sequences to represent the database. At query time, the edit distance of the query against each reference sequence is computed. Lower and upper bounds are applied to efficiently filter candidate matches. DSIM[12] uses a set of selected reference words formed from high-frequency data sub- strings. SST [21] is used for subsequence matching in biological sequences and maps the biological sequence database to a d-dimensional vector space; this mapping

is used to filter a significant portion of the database from consideration during the query process. This method outperforms BLAST by an order of magnitude but only for applications where there exists an extremely high similarity (95% and over) between the query sequence and its match in the database.

The RBSA method proposed in this thesis is also related to EBSM [6], which uses precomputed alignments between database sequences and reference sequences for efficient subsequence matching in time series databases. The key differences between RBSA and EBSM stem from the fact that RBSA addresses near-exact string matching under the edit distance or Smith-Waterman, whereas EBSM addresses general time series matching under DTW. RBSA exploits the metric properties of the edit distance, and the additional near-exact matching constraint, to provide either guaranteed correct results (for exact RBSA) or guaranteed high probability of correct results (for approximate RBSA). No equivalent guarantees are present in EBSM. Furthermore, RBSA can handle queries of arbitrary size (query lengths range from 40 to 10,000 in our experiments) by breaking up queries into fixed-size segments, whereas EBSM requires that query lengths be within a relatively narrow range (query lengths range from 152 to 426 in the experiments of EBSM), and provides no mechanism for handling queries of arbitrary size.

Chapter 3

Embedding-based Subsequence Matching in Time Series Databases

In this chapter we describe Embedding-Based Subsequence Matching (EBSM), an embedding-based method for subsequence matching under Dynamic Time Warping (DTW). EBSM is an approximate method that uses an embedding index to filter candidate endpoint positions of a possible match and then performs the expensive dynamic programming computation only for those candidates. In the remainder of this chapter, we first provide some background on time series matching under DTW, then we describe EBSM in detail, and finally we present an extensive experimental evaluation on real time series data.

3.1 Background

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 [34] and [75]. 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.

3.1.1 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 $(X_{w_{1,2}}, \ldots, X_{w_{|W|,2}})$:

$$C(Q, X, W) = \sqrt[p]{\sum_{i=1}^{|W|} \|Q_{w_{i,1}} - X_{w_{i,2}}\|^p}\,. \tag{3.1}$$

In the remainder of this section, 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., [24, 34], and not employed by other methods, e.g., [75]. 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 this thesis, we will not consider this constraint, and in the experiments this constraint is not employed.

3.1.2 Optimal Warping Paths and Distances

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

$$W^*(Q, X) = \text{argmin}_W C(Q, X, W). \tag{3.2}$$

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:

$$D(Q, X) = C(Q, X, W^*(Q, X)). \tag{3.3}$$

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:

$$D_{\text{full}}(Q, X) = C(Q, X, W^*_{\text{full}}(Q, X)). \tag{3.4}$$

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 3.3 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:

$$D_j(Q, X) = C(Q, X, W^*(Q, X, j)). \tag{3.5}$$

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:

$$M(R, X, j) = \text{argmin}_{X^{i:j}} D_{\text{full}}(R, X^{i:j}). \tag{3.6}$$

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 $X^{i:j}$.

3.1.3 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:

$$W \oplus (w', w'') = ((w_{1,1}, w_{1,2}), \ldots, (w_{|W|,1}, w_{|W|,2}), (w', w'')). \tag{3.7}$$

The DTW algorithm uses the following recursive definitions:

$$D_{0,0}(Q, X) = 0, D_{i,0}(Q, X) = \infty, D_{0,j}(Q, X) = 0 \tag{3.8}$$

$$W_{0,0}(Q, X) = (), W_{0,j}(Q, X) = () \tag{3.9}$$

$$\mathrm{A}(i, j) = \{(i, j-1), (i-1, j), (i-1, j-1)\} \tag{3.10}$$

$$(\mathrm{pi}(Q, X), \mathrm{pj}(Q, X)) = \mathrm{argmin}_{(s,t)\in \mathrm{A}(i,j)} D_{s,t}(Q, X) \tag{3.11}$$

$$D_{i,j}(Q, X) = \|Q_i - X_j\| + D_{\mathrm{pi}(Q,X),\mathrm{pj}(Q,X)}(Q, X) \tag{3.12}$$

$$W_{i,j}(Q, X) = W_{\mathrm{pi}(Q,X),\mathrm{pj}(Q,X)} \oplus (i, j) \tag{3.13}$$

$$D(Q, X) = \min_{j=1,\ldots,|X|} \{D_{|Q|,j}(Q, X)\} \tag{3.14}$$

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 [75]. 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 $W^*(Q, X, j_{\text{opt}})$, where j_{opt} is the endpoint of the optimal match: $j_{\text{opt}} = \text{argmin}_{j=1,\ldots,|X|}\{D_{|Q|,j}(Q, X)\}$.

3.2 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 thesis 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 $F(X, j)$:

$$F^R(Q) = D_{|R|,|Q|}(R, Q) . \quad (3.15)$$

$$F^R(X, j) = D_{|R|,j}(R, X) . \quad (3.16)$$

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:

$$F(Q) = (F^{R_1}(Q), \ldots, F^{R_d}(Q)) . \quad (3.17)$$

$$F(X, j) = (F^{R_1}(X, j), \ldots, F^{R_d}(X, j)) . \quad (3.18)$$

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 3.1.3 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 $F(X, j)$.

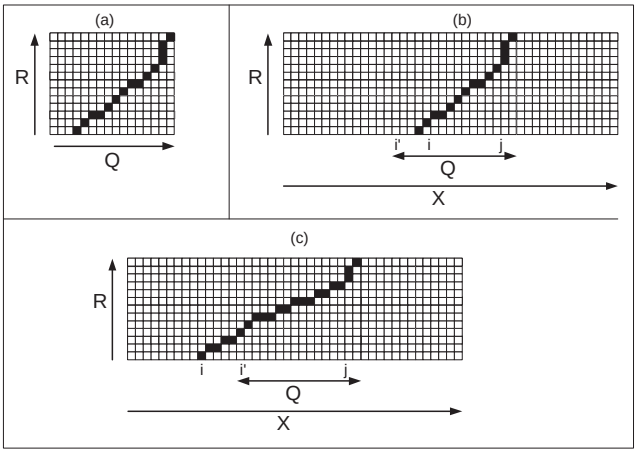

Figure 3·1: (b) Example of a warping path $W^*(R, X, j)$, aligning a reference object R to a subsequence $X^{i:j}$ of sequence X. $F^R(X, j)$ is the cost of $W^*(R, X, j)$. The query Q from (a) appears exactly in X, as subsequence $X^{i':j}$, and $i' < i$. Under these conditions, $F^R(Q) = F^R(X, j)$. (c) Similar to (b), except that $i' > i$. In this case, typically $F^R(Q) \neq F^R(X, j)$.

- 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 [22, 89, 77, 13, 46, 17, 31, 88, 41, 82].

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 3·1. 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 3.6) 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:

$$F^R(Q) = D_{|R|,|Q|}(R, Q) = D_{|R|,j}(R, X) = F^R(X, j). \tag{3.19}$$

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 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 3.5.

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 4.5.

3.3 Filter-and-Refine Retrieval

Our goal in this thesis 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.

3.3.1 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 [26]. 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 [75], 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 thesis, 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.

3.3.2 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. Moreover, vector indexing methods [22, 89, 77, 13, 46, 17, 31, 88, 41, 82] 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. Finally, a recent method [80] for nearest neighbor search in high dimensional spaces could be embedded in the filter step for fast vector search.

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.

3.3.3 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 4.1 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 4.1 keeps track, for every pair (i, j), of the endpoint that corresponds to the cost stored in $\mathrm{cost}[i][j]$. 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.

The $\mathrm{columns}$ variable, which is an output of Algorithm 4.1, 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 $\mathrm{cost}[i][j]$ is computed. The $\mathrm{columns}$ 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 5.1. In the experiments, one of the main measures of EBSM efficiency (the DTW cell cost) is simply defined as the ratio between $\mathrm{columns}$ and the length $|X|$ of the database.

We note that each application of DTW in Algorithm 4.1 stops when the minimum $\mathrm{cost}[i][j]$ over all $i = 1, \ldots, |Q|$ is higher than the minimum distance found so far. We do that because any $\mathrm{cost}[i][j-1]$ will be at least as high as the minimum (over all i's) of $\mathrm{cost}[i][j]$, 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 $X^{(j_{\mathrm{end}}-3|Q|):(j_{\mathrm{end}}+|Q|)}$. 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 $2|Q|$.

3.4 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 [85].

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 $\mathrm{EE}(Q)$ 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., $\mathrm{SEE} = \sum_{Q \in \mathbb{Q}_s} \mathrm{EE}(Q)$. The nest step, is to consider a replacement of the i-th reference object with an object in $\mathrm{RSK} - \{R_1, \ldots, R_d\}$, 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 4.2. 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 $\mathrm{RSK} - \{R_1, \ldots, R_d\}$ 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 [85]. However, the approach in [85] was based on the Edit distance, which is a metric, and therefore a different optimization criterion was used. In particular, in [85], 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.

3.5 Handling Large Ranges of Query Lengths

In Section 3.2 and in Figure 3·1 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 $M(R, X, j)$ 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 3·2 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

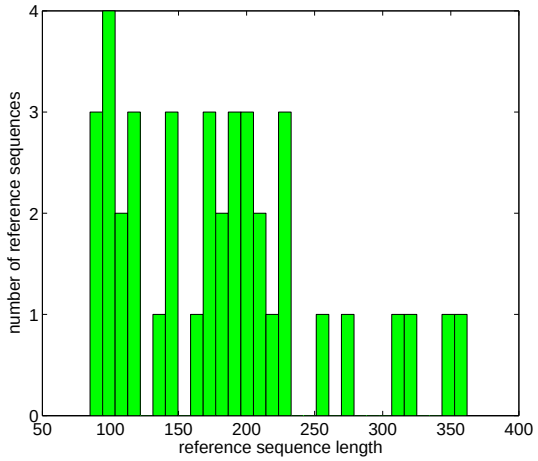

Figure 3·2: Distribution of lengths of the 40 reference objects chosen by the embedding optimization algorithm in our experiments.

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.

3.6 Experiments

We evaluate the proposed method on time series data obtained from the UCR Time Series Data Mining Archive [35]. 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. [75], which applies the DTW algorithm as described in Section 3.1.3.
- **Modified PDTW:** a modification of the approximate method based on piecewise aggregate approximation that was proposed by Keogh et al. [36].

Actually, as formulated in [36], 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 [36] 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

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 3.1: 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).

4.1) 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 [24], because the method in [24] 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 3.6.3, the method in [24] would fail to identify optimal matches for the majority of the queries.

3.6.1 Datasets

To create a large and diverse enough dataset, we combined three of the datasets from UCR Time Series Data Mining Archive [35]. The three UCR datasets that we used are shown on Table 4.1.

Each of the three UCR datasets contains a test set and a training set. As can be seen on Table 4.1, 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.

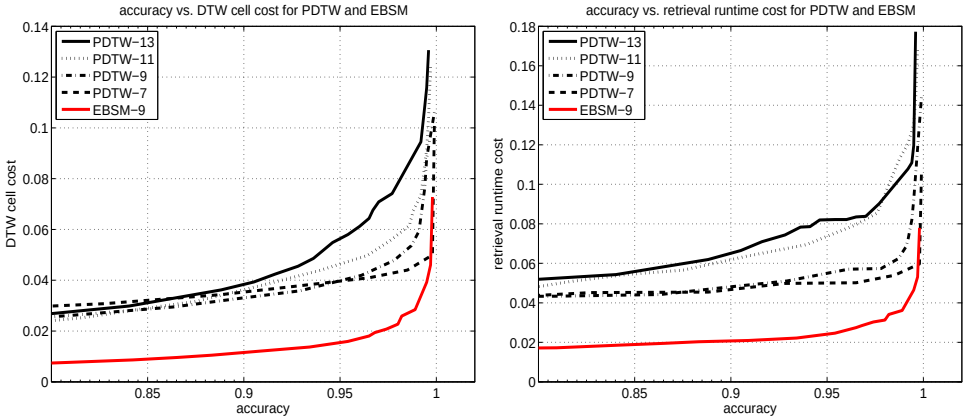

Figure 3·3: The top figure measures efficiency using the DTW cell cost, and the bottom figure measures efficiency using the retrieval runtime cost. The costs shown are average costs over our test set of 1000 queries. Note that SPRING, being an exact method, corresponds to a single point (not shown on these figures), with perfect accuracy 1 and maximal DTW cell cost 1 and retrieval runtime cost 1.

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 time series. We randomly chose 750 of those time series as a validation set of queries, that was used for embedding optimization using Algorithm 4.2. 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.

3.6.2 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

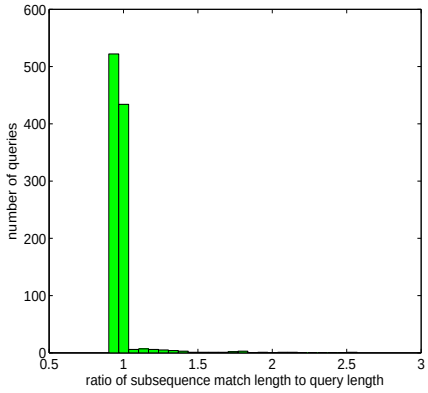

Figure 3·4: We note that a significant fraction of the optimal matches have lengths that are not identical to the query length.

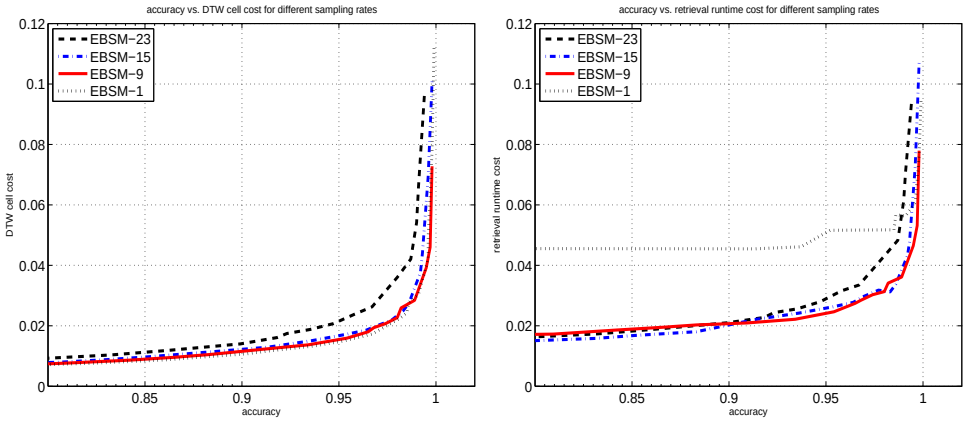

Figure 3·5: The top figure measures efficiency using the DTW cell cost, and the bottom figure measures efficiency using the retrieval runtime cost. The costs shown are average costs over our test set of 1000 queries.

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 4.1 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

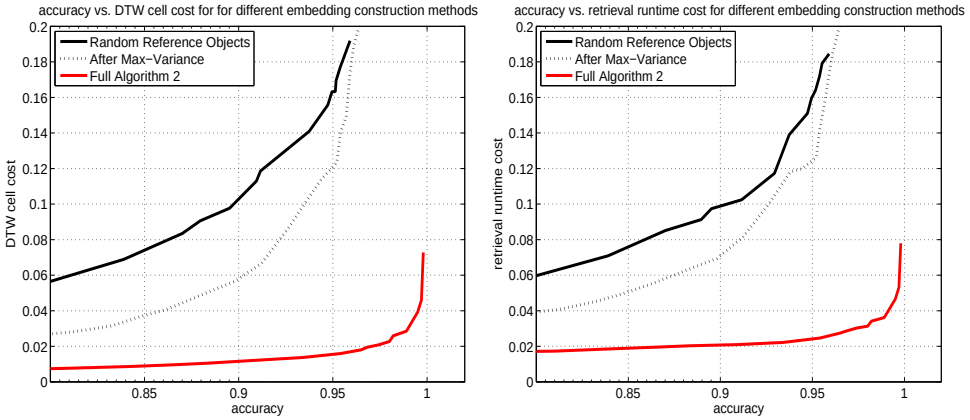

Figure 3·6: The top figure measures efficiency using the DTW cell cost, and the bottom figure measures efficiency using the retrieval runtime cost. The costs shown are average costs over our test set of 1000 queries.

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 3.1.3. 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 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 4.1). 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.

3.6.3 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 4.5, 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). The training time for the above settings was approximately 3.5 hours.

Figure 3·3 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 3·4 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 [24] 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 [24] for this dataset.

Figure 3·5 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 4.2. Sampling rates between 1 and 15 all produced pretty similar DTW cell costs 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 3·6 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 4.2) improves performance significantly compared to random selection. Using the full Algorithm 4.2 for embedding construction improves performance even more.

Figure 3·7 shows how the performance of EBSM varies with different embedding dimensionality, for optimized (using Algorithm 4.2) 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 dimensionality 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 3·7, 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 4.2 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/`

3.6.4 Replication of EBSM on other datasets

In order to run EBSM on a time series dataset, a few parameters need to be set. These parameters are: *k* (number of time series sequences used in the initial stage of the training phase), *p* (number of database candidates to be evaluated), and *d* (embedding dimensionality). For the datasets used in our experiments, we have tested different values of these parameters and determined the ones with the best retrieval runtime. These parameters however, are dataset-dependent, meaning that EBSM might require a different setting of these parameters for each given dataset, to guarantee best retrieval runtime.

Suppose that an individual wants to use EBSM for a given time series dataset. EBSM should be tuned so as to achieve best performance in terms of retrieval runtime. For the training phase, one approach is to ask the user to provide a sample of queries similar to those expected when EBSM will be running online. Half of those queries will be used for training and the other half for validation purposes. After setting *k* to be a large number, e.g. $2,000$ reference sequences, Algorithm 3.2 will be applied to determine the optimal reference sequences for different dimensionality values d. The d and p values with the best retrieval runtime on the validation set of queries will be chosen for the online phase of EBSM.

Another approach is to set k, d, and p to an initial value empirically chosen based on datasets

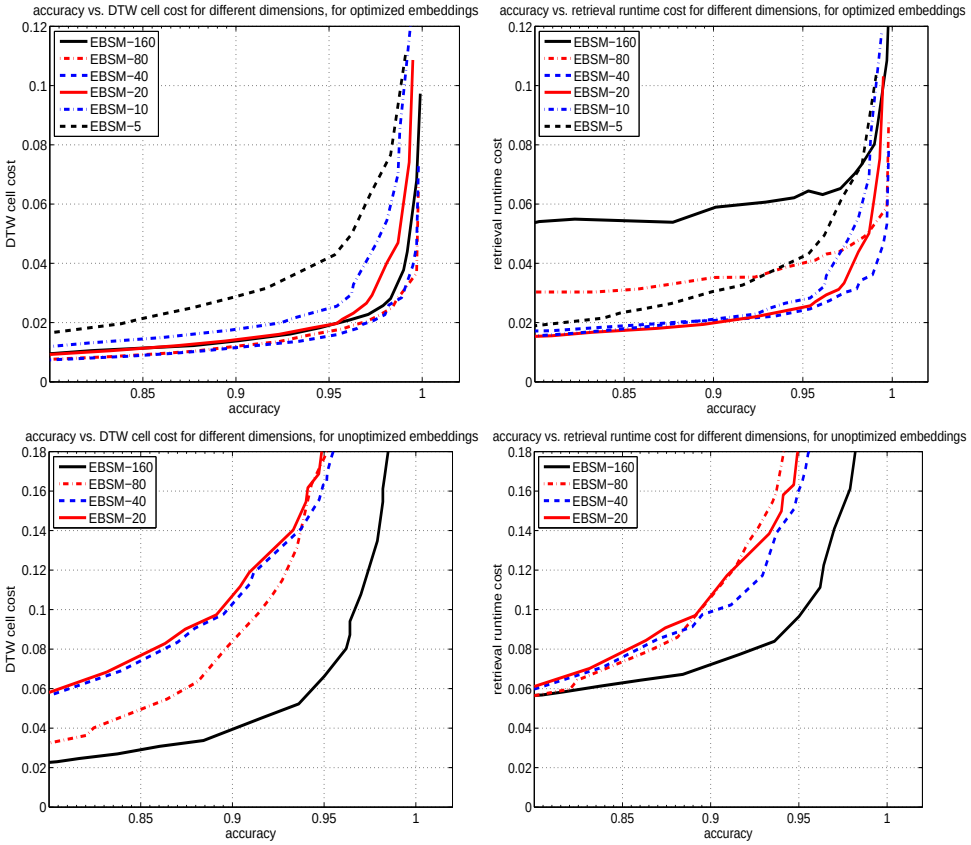

Figure 3·7: The plots on the left measure efficiency using the DTW cell cost, and the plots on the right measure efficiency using the retrieval runtime cost. The costs shown are average costs over our test set of 1000 queries. The top plots show results for embeddings optimized using Algorithm 4.2. The bottom plots show results for embeddings with randomly selected reference objects.

seen so far. Then, these parameters can be updated in an online manner. For the set of queries seen so far we can keep track of the distribution of their pairwise distances. When their distances start increasing in variance, this means that possibly a new type of queries has entered our system and thus it needs to be trained accordingly. Algorithm 3.2 can be invoked to determine new values for the parameters that yield the best retrieval runtime for the new query sample.

3.7 Summary

EBSM 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 discussion in this thesis 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.

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 : $(X, j_{\text{start}}), (X, j_{\text{end}})$: 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.

```
for i = 1 to |X| do
    unchecked[i] = 0;
end
for i = 1 to p do
    unchecked[sorted[i]] = 1;
end
distance = ∞; columns = 0;
for k = 1 to p do
    candidate = sorted[k];
    if (unchecked[candidate] == 0) then continue;
    j = candidate + 1;
    for i = |Q| + 1 to 1 do
        cost[i][j] = ∞;
    end
    while (true) do
        j = j − 1;
        if (candidate − j ≥ 2 ∗ |Q|) then break;
        if (unchecked[j] == 1) then
            unchecked[j] = 0; candidate = j; cost[|Q| + 1][j] = 0; endpoint[|Q| + 1][j] = j;
        else
            cost[|Q| + 1][j] = ∞; // j is not a candidate endpoint.
        end
        for i = |Q| to 1 do
            previous = {(i+1, j), (i, j+1), (i+1, j+1)};(p_i, p_j) = argmin_{(a,b)∈previous} cost[a][b];
            cost[i][j] = |Q_i − X_j| + cost[p_i][p_j]; endpoint[i][j] = endpoint[p_i][p_j];
        end
        if (cost[1][j] < distance) then
            distance = cost[1][j]; j_start = j;j_end = endpoint[1][j];
        end
        columns = columns + 1;
        if (min{cost[i][j]|i = 1, . . . , |Q|} ≥ distance) then break;
    end
end
start = j_end − 3|Q|; end = j_end + |Q|;
Adjust j_start and j_start by running the DTW algorithm between Q and X^{start:end};
```

Algorithm 3.1. The refine step for unconstrained DTW.

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 ∈ RSK with maximum 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 3.2. The training algorithm for selection of reference objects.

Chapter 4

Bidirectional Embedding-based Subsequence Matching in Time Series Databases

The main focus of this chapter is time series similarity search under the constrained dynamic time warping (cDTW) similarity measure. A new embedding (BSE) is defined that differs from the one used in EBSM in that it is constructed using both start and endpoint matches of the reference sequences. Exploiting the additional constraints of cDTW, it is powerful enough to achieve very high efficiency even without training, as opposed to EBSM. The remainder of this chapter is organized as follows: first, some background is provided on time series similarity search under cDTW, then BSE is described in more detail, and finally an experimental analysis on the proposed methods is presented.

4.1 Background: The cDTW Algorithm

Constrained DTW (cDTW) is obtained from DTW simply by placing an additional constraint, which narrows down the set of positions in one sequence that can be matched with a specific position in the other sequence.

Consider the definition of DTW given in section 3.1.3. Given a warping width w, this constraint is defined as follows:

$$D_{i,j}(Q, X) = \infty \text{ if } |i - j| > w \; . \tag{4.1}$$

The term "Sakoe-Chiba band" is often used to characterize the set of (i, j) positions for which $D_{i,j}$ is not infinite. Notice that if $w = 0$, cDTW becomes the L_p distance. While a simple modification of DTW, cDTW has been shown to be significantly more efficient than DTW for full

sequence matching [34], and to also produce more meaningful matching scores [72].

Given the above definitions, the subsequence match of Q in a database X is the subsequence $X_{\text{opt}} = (X_j, \ldots, X_{j+|Q|-1})$ that minimizes $D(Q, X_{\text{opt}})$. Similarly to other approaches for subsequence matching under cDTW, namely LB_Keogh [34] and DTK [56], we require that the subsequence match have the same length as the query. A simple approach for finding the subsequence match of Q is the sliding-window approach: we simply compute the matching cost between Q and every subsequence of X that has length $|Q|$.

The LB_Keogh [34] method speeds up the sliding window approach, often by orders of magnitude, by computing an efficient lower bound of the matching cost, that can be used to reject many subsequences without computing the exact cDTW cost between Q and those subsequences. With respect to LB_Keogh, which is an exact method, the method proposed in this thesis can be seen as an approximate alternative for quickly rejecting many candidate subsequences; in our method, accuracy can be easily traded for efficiency, so as to achieve significantly larger speedups than LB_Keogh.

4.2 Bidirectional Subsequence Embeddings

In this section we introduce Bidirectional Subsequence Embeddings (BSE), a new embedding-based method for subsequence matching under cDTW. Following [34] and [56], we require that the length of the subsequence match has to be equal to the query length.

Our starting point is similar to that of EBSM: we use reference sequences to define 1D embeddings. Given a reference sequence R, and given the definition in Section 4.1 of the matching cost D for cDTW, we define a 1D embedding H^R as follows:

$$H^R(Q) = \begin{cases} D(R, (Q_{|Q|-|R|+1}, \ldots, Q_{|Q|})) & \text{if } |R| \leq |Q| \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

$$H^R_Q(X, j) = \begin{cases} D(R, (X_{j-|R|+1}, \ldots, X_j)) & \text{if } |R| \leq |Q| \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

If we compare the above two equations with the corresponding equations **??** and **??** for EBSM,

we notice two important differences: first, if the reference sequence is longer than the query, then the query is mapped to zero. Second, the embedding $H_Q^R(X, j)$ depends not only on X and j, but also on the length of the query: $H_Q^R(X, j) = 0$ when the reference sequence R is longer than Q. These changes have a simple interpretation: they effectively force us to ignore, given a query Q, any reference sequence longer than Q.

If $R_1, \ldots, R_d$ are d reference sequences, then a d-dimensional embedding H is defined as follows:

$$H(Q) = (H^{R_1}(Q), \ldots, H^{R_d}(Q)) . \tag{4.4}$$

$$H_Q(X, j) = (H_Q^{R_1}(X, j), \ldots, H_Q^{R_d}(X, j)) . \tag{4.5}$$

Again, we note that the embedding of the database position (X, j) also depends on the length of the query.

If Q is exactly identical to a database subsequence ending at position (X, j^*), then $H(Q) = H_Q(X, j^*)$. If we perturb that subsequence match $(X_{j^*-|Q|+1}, \ldots, X_{j^*})$ so that it is not identical to Q anymore, we expect that small perturbations will lead to small changes in $H_Q(X, j^*)$, so that $H(Q)$ will still be fairly similar to $H_Q(X, j^*)$. Therefore, embeddings H are useful for identifying candidate endpoints of subsequence matches. Because of that, we refer to embeddings H as *endpoint embeddings*.

We can easily adapt the definition of endpoint embeddings H to also define startpoint embeddings G, that can be used to identify candidate start points of subsequence matches. We define 1D startpoint embeddings G^R and multidimensional startpoint embeddings G as follows:

$$G^R(Q) = \begin{cases} D(R, (Q_1, \ldots, Q_{|R|})) & \text{if } |R| \leq |Q| \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

$$G_Q^R(X, j) = \begin{cases} D(R, (X_j, \ldots, X_{j+|R|-1})) & \text{if } |R| \leq |Q| \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

$$G(Q) = (G^{R_1}(Q), \ldots, G^{R_d}(Q)) . \quad (4.8)$$

$$G_Q(X, j) = (G_Q^{R_1}(X, j), \ldots, G_Q^{R_d}(X, j)) . \quad (4.9)$$

Suppose that we have chosen reference sequences and using those sequences we have defined startpoint embeddings G and endpoint embeddings H. Given a query Q, every possible subsequence match $(X_{j-|Q|+1|}, \ldots, X_j)$ corresponds to a startpoint embedding $G_Q(X, j - |Q| + 1)$ and an endpoint embedding $H_Q(X, j)$. If Q is similar to $(X_{j-|Q|+1|}, \ldots, X_j)$ we expect both $G(Q)$ to be similar to $G_Q(X, j - |Q| + 1)$, and $H(Q)$ to be similar to $H_Q(X, j)$. To capture the correspondence, given Q, between startpoint embedding $G_Q(X, j - |Q| + 1)$ and endpoint embedding $H_Q(X, j)$, we define a unified embedding F, which we call a *bidirectional subsequence embedding* (BSE), that combines startpoint and endpoint embeddings. The BSE embedding F is simply a concatenation of the startpoint and endpoint embeddings:

$$F(Q) = (G(Q), H(Q)) . \quad (4.10)$$

$$F_Q(X, j) = (G_Q(X, j - |Q| + 1), H_Q(X, j)) . \quad (4.11)$$

Figure 4·1 illustrates the construction of a BSE embedding given a query Q and a reference object R.

A key difference between the EBSM method of [6] and the BSE method we have described (in addition to the fact that EBSM is formulated for unconstrained DTW, and BSE is formulated for cDTW) is that EBSM uses only the equivalent of endpoint embeddings. The question of how to combine startpoint embeddings and endpoint embeddings in unconstrained DTW is nontrivial. On the other hand, using the constraints available in cDTW we can easily combine startpoint and endpoint embeddings, online, based on the length of the query. As we shall see in the experiments, this combination leads to improved performance over using only endpoint embeddings.

4.3 Computing Database Embeddings

Suppose that we have chosen d reference sequences $R_1, \ldots, R_d$. We note that applying Equations 4.7 and 4.3 to compute embeddings $G_Q(X, j)$ and $H_Q(X, j)$ requires knowing the query, or, at

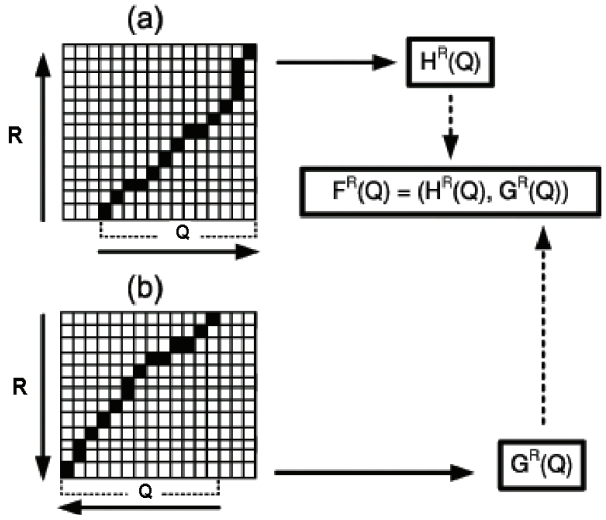

Figure 4·1: An example that illustrates the construction of the bidirectional embedding given a query Q and a reference object R.

least, the length of the query. At the same time, computing $G_Q(X, j)$ and $H_Q(X, j)$ online, given a query, is too expensive (actually, even more expensive than using brute force to find the subsequence match of the query), unless we can make use of some precomputed information.

This precomputed information is in the form of query-independent embeddings $G(X, j)$ and $H(X, j)$, that are simply defined by dropping the dependency on the query length. Given reference sequences $R_1, \ldots, R^d$, we define:

$$G^R(X, j) = D(R, (X_j, \ldots, X_{j+|R|-1})) \quad (4.12)$$

$$G(X, j) = (G^{R_1}(X, j), \ldots, G^{R_d}(X, j)) \quad (4.13)$$

$$H^R(X, j) = D(R, (X_{j-|R|+1}, \ldots, X_j)) \quad (4.14)$$

$$H(X, j) = (H^{R_1}(X, j), \ldots, H^{R_d}(X, j)) \quad (4.15)$$

Embeddings $G(X, j)$ and $H(X, j)$ do not depend on the query, and so they can be precomputed off-line and stored. Given a query Q, $G_Q(X, j)$ and $H_Q(X, j)$ can be obtained by putting a 0 to all embedding dimensions corresponding to reference sequences longer than Q. Even more simply, those embedding dimensions can be ignored when computing Euclidean distances.

It is also important to note that G^R and H^R are related as follows:

$$G^R(X, j) = H^R(X, j + |R| - 1) \; . \tag{4.16}$$

This means that, in practice, only startpoint embeddings $G(X, j)$ need to be precomputed and stored. Embeddings $H(X, j)$, and the query-sensitive embeddings $F_Q(X, j)$, can be easily obtained, online, from the precomputed embeddings $G(X, j)$. As we will see in the experiments, the total retrieval time, that includes these online computations, is still much faster than the retrieval time obtained using brute force or alternative exact methods such as LB_Keogh [34] and DTK [56].

4.4 Filter-and-Refined Retrieval

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

The retrieval framework that we use is filter-and-refine retrieval. Given a query, the retrieval process consists of a filter step and a refine step [26]. A set of candidate matches is identified during the filter step and it is forwarded to the refine step which evaluates each of those candidates using the original matching algorithm (cDTW in our case). The candidate that best matches the query is identified and reported during the refine step.

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 and the cost of evaluating candidates at the refine step. Retrieval accuracy, given a query, depends on whether that 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. Apparently, reducing the number of candidates can significantly speedup the method, assuming that the best match is included in the set of candidates.

Given this framework, embeddings can be used at the filter step, and provide a computation-

ally efficient way to quickly select a relatively small number of candidates, as vector matching operations are computationally cheaper than cDTW. In particular, given embeddings G, H, and $F = (G, H)$, 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 $G(X, j)$ for every position j of the database sequence X. Computing embeddings $G(X, j)$, for $j = 1, \ldots, |X|$, is an off-line preprocessing step that takes time $O(|X| \sum_{i=1}^{d} |R_i|^2)$.

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

- **Embedding step:** compute $G(Q)$ and $H(Q)$, by measuring the cDTW matching cost between Q and the reference sequences. Concatenate $G(Q)$ and $H(Q)$ to form vector $F(Q)$. Also, given Q and the precomputed $G(X, j)$, form vectors $G_Q(X, j)$, $H_Q(X, j)$, and $F_Q(X, j)$.

- **Filter step:** For some user-defined parameter p, select p database positions (X, j) according to the Euclidean distance between each $F_Q(X, j)$ and $F(Q)$. These database positions define candidate subsequence matches $(X_{j-|Q|+1}, \ldots, X_j)$ for Q.

- **Refine step:** Evaluate the selected candidate subsequence matches. Evaluation proceeds by first applying LB_Keogh [34] to establish a lower bound of the matching cost, and then evaluating the exact cDTW matching cost for enough candidates to assure that the best matching candidate has been found, as described in [34].

We note that the refine step, instead of simply measuring the cDTW matching cost between the query and all candidate subsequence matches, uses LB_Keogh to speed up computations. LB_Keogh is an exact method, so it guarantees that, if the correct subsequence match has been included in the candidates, the refine step will identify that match. At the same time, the correct subsequence match will not be retrieved unless it has been identified as a candidate during the filter step. Our method is approximate, and it is possible that, for some queries, the correct subsequence match will be rejected during the filter step. At the same time, the user can easily trade accuracy

for efficiency by adjusting parameter p, which specifies how many candidates to select during the filter step. Larger values of p lead to higher accuracy and lower efficiency.

4.4.1 Faster Filtering Using Sampling

The cost of the filter step can be a significant part of the overall retrieval cost, as filtering involves comparisons between high-dimensional vectors. 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 $F_Q(X, j)$ and $F_Q(X, j + 1)$, 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 $F_Q(X, j)$. Given $F(Q)$, we only compare it with vectors $F_Q(X, 1), F_Q(X, 1+\delta), F_Q(X, 1+2\delta), \ldots$. If, for a database position (X, j), its vector $F_Q(X, j)$ was not sampled, we simply assign to that position the distance between $F(Q)$ and the vector that was actually sampled among $\{F_Q(X, j - \lfloor \delta/2 \rfloor), \ldots, F_Q(X, j + \lfloor \delta/2 \rfloor)\}$.

4.5 Embedding Optimization

In this section, we present two approaches for selecting reference objects in order to improve the quality of the embedding. These approaches have already been described in [85] and [6]; in this section we provide a short summary for easy reference.

The first approach is based on the max variance heuristic, i.e., the idea that we should select reference sequences that cover the domain space (as much as possible) and have distances to other reference sequences with high variance. To define our reference sequences, we select randomly 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). Then, if we want d reference sequences, we can simply choose the d subsequences with the highest variance in their distances to the other $l - 1$ subsequences.

The second approach is a learning approach that minimizes the embedding error, defined as follows: the embedding error $\mathrm{EE}(Q)$ of a query Q is defined as the number of vectors $F_Q(X, j)$ in the embedding space that the embedding of the query $F(Q)$ is closer to than it is to the embedding

of $F_Q(X, j_Q)$, where j_Q is the endpoint of the optimal subsequence match of Q in the database. The embedding error on a set $\mathbb{Q}_s$ of sample queries is simply the sum of the individual embedding errors: $\text{SEE} = \sum_{Q \in \mathbb{Q}_s} \text{EE}(Q)$.

Given a sample set $\mathbb{Q}_s$ of queries, the embedding error is minimized via a greedy training algorithm. First, we select k candidate reference sequences using the max variance heuristic. Second, we select d reference sequences randomly out of the k candidates. Then, in the main loop, we evaluate a large number of substitutions: each substitution involves replacing one of the selected d reference sequences with one of the remaining $k - d$ candidate reference sequences. If that substitution reduces the embedding error it is kept, otherwise it is reverted. This process stops when the embedding error stops decreasing.

The advantage of using just the max variance heuristic is that no training set of queries needs to be available during embedding construction. Obtaining a representative training set of queries may not always be feasible. Furthermore, the distribution of queries can vary widely over time, and a training set that used to be representative during embedding construction may not be representative after a while. The max variance heuristic, by not requiring a training set of queries, does not suffer from these drawbacks.

On the other hand, if the distribution of queries is static over time, and if a representative set of queries is available for training, then the learning method outlined above can be used to optimize embedding performance. For the datasets used in our experiments, the max variance heuristic is sufficient for constructing embeddings that give state-of-the-art results. The learning method improves performance even further.

4.6 Experiments

The proposed method is evaluated on time series data obtained from the UCR Time Series Data Mining Archive [35] and also on an additional random walk synthetic dataset. Our method is compared to two state-of-the-art methods for subsequence matching under constrained DTW:

- **LB_Keogh with sliding window:** Given a query of length $|Q|$, a sliding window of size $|Q|$ scans the time series database, performing the LB_Keogh lower bounding technique at each

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 embedding optimization (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)	450	1000	300

Table 4.1: 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 2337778).

step.

- **DTK:** the exact subsequence matching method proposed in [56]. We note that this method has been designed to work for external memory, but here we evaluate it on main memory datasets. Therefore, first we buffer the complete index in main memory and then we run the queries. Thus, all the operations are executed in main memory.

Both BSE and LB_Keogh were implemented in C++. The code for DTK has been obtained from the authors [56]. All experiments were run on an AMD Opteron 8220 SE processor running at 2.8GHz.

The main focus of the experimental evaluation is to demonstrate the main contributions of our work and the robustness of the proposed method with respect to query size and warping width. In particular, our experiments demonstrate:

- significant speedups, at the cost of modest loss in retrieval accuracy, compared to the exact methods LB_Keogh [34] and DTK [56].

- the performance gains of bidirectional embeddings, compared to using only endpoint em-

beddings, as done in [6].

- the effect of training in the new embedding scheme, and the fact that competitive results are obtained even when not using training.

- the robustness of our method with respect to query size and warping width.

4.6.1 Datasets

To create a large and diverse enough dataset, we combined three of the datasets from UCR Time Series Data Mining Archive [35]. The three UCR datasets that we used are shown on Table 4.1.

Each of the three UCR datasets contains a test set and a training set. As can be seen on Table 4.1, 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 450 from the 50Words dataset, 6164 time series of length 152 from the Wafer dataset, and 3000 time series from the Yoga dataset. The length $|X|$ of the database is obviously the sum of length of all these time series, which adds up to 2337778.

Our set of queries was the set of time series in the original training set of the three UCR datasets. In total, this set includes 1750 time series. 750 of those queries were set aside and used as a sample set of queries for the learning method discussed in Section 4.5, that performs embedding optimization. We should emphasize that embedding construction using the max variance heuristic did not use that sample set at all; only the learning method used the sample set. The remaining 1000 queries were used for performance evaluation of all methods.

To further evaluate the robustness of BSE we created a random walk synthetic dataset. In this dataset the database time series X was generated as follows: for each value X_i we produce a random real number r and if r is positive, $X_i = X_{i-1} + 0.005$, else $X_i = X_{i-1} - 0.005$. X_0

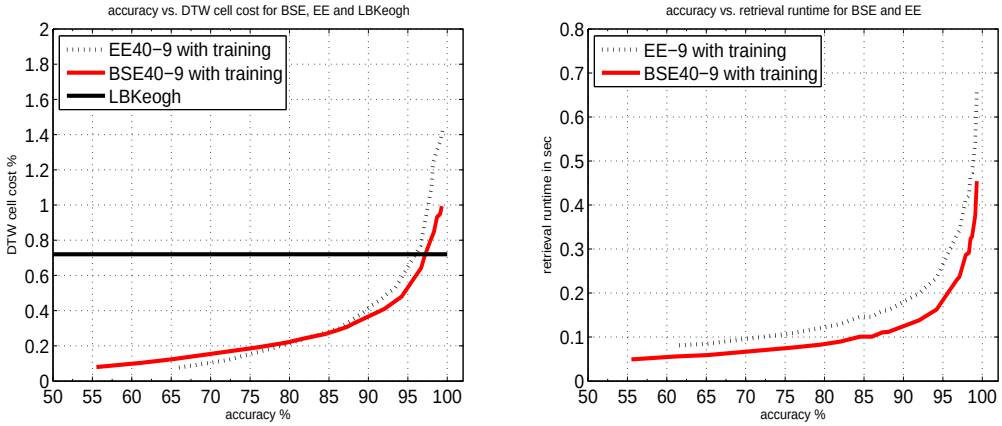

Figure 4·2: Warping width is 5% of the query size. The cell cost is also shown for LB_Keogh (corresponding to 100% accuracy). The cell cost for DTK is 18.73%. The retrieval runtime is 8.21 sec for LB_Keogh, and 17.93 sec for DTK.

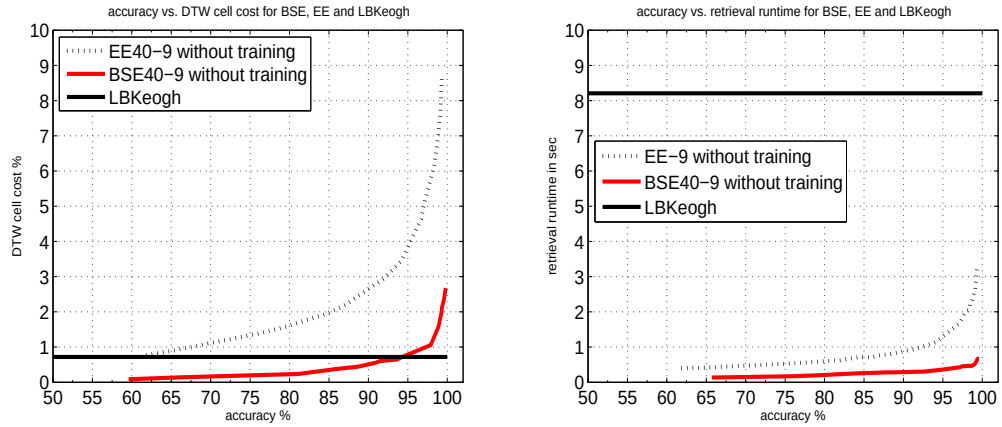

Figure 4·3: Dimensionality = 40 and sampling rate = 9. Warping width is 5% of the query size. Results are also shown for LB_Keogh, as horizontal bars corresponding to the costs for 100% retrieval accuracy. The cell cost for DTK is 18.73%, and the retrieval runtime for DTK is 17.93 sec.

is set to 1.5. Queries were generated in the same way. The query size varied from 100 to 1000 in increments of 100. We used 100 queries per query size.

4.6.2 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

Accuracy	BSE40-9	EE40-9	DTK	LB_Keogh
100%			18.73%	0.72%
99%	1.58%	6.77%		
98%	1.24%	5.79%		
95%	1.02%	3.66%		
90%	0.61%	2.44%		
85%	0.45%	2.08%		
80%	0.38%	1.59%		

Table 4.2: Note that DTK and LB_Keogh are exact and thus have 100% retrieval accuracy.

Accuracy	BSE40-9	EE40-9	DTK	LB_Keogh
100%			17.93	8.21
99%	0.52	2.40		
98%	0.46	2.02		
95%	0.37	1.31		
90%	0.29	0.82		
85%	0.26	0.71		
80%	0.21	0.60		

Table 4.3: Note that DTK and LB_Keogh are exact and thus have 100% retrieval accuracy.

efficiency. Accuracy is simply the percentage of queries in our evaluation set for which the optimal subsequence match was successfully retrieved. Efficiency can be measured based on the **runtime cost** in seconds for each query. We report the average runtime cost for each group of queries. To compute the runtime cost for a query, we measure the total runtime of the entire retrieval algorithm for that query, including both the filter and the refine steps. Efficiency can also be measured based on the **cell cost** for each query, which is the percentage of the number of database positions visited during the refine step divided by the database size.

Query Size	BSE40-9 (sec)	LB_Keogh (sec)
152	0.04	4.36
270	0.04	7.87
426	0.04	13.51

Table 4.4: Runtime (in seconds) for the filter step of BSE with sampling rate 9 and dimensionality 40 and for the filter step of LB_Keogh for the UCR dataset.

Trade-offs between accuracy and efficiency can be obtained very easily, for BSE, by changing parameter p of the refine step (see Section 4.4). Increasing the value of p increases accuracy, but decreases efficiency, by increasing both cell cost and retrieval runtime.

4.6.3 Parameter Settings

One parameter that we need to set is the dimensionality of the BSE embedding. Unless noted otherwise, we use a 40-dimensional embedding. In Section 4.6.5 we discuss the effect of changing the dimensionality of the embedding.

The other parameter that we need to set for our method is δ, the sampling rate discussed in Section 4.4.1. Unless noted otherwise, we use $\delta = 9$. In Section 4.6.5 we discuss the effect of different sampling rates.

4.6.4 Comparison to Other Methods

In this section, BSE is compared with the two aforementioned state-of-the-art methods, LB_Keogh and DTK.

Accuracy vs. Efficiency

Applying LB_Keogh with a sliding window on the UCR dataset yielded a cell cost of 0.72% with an average retrieval runtime of 8.21 seconds per query. On the other hand, the performance of DTK is poor in terms of both cell cost (18.73%) and retrieval runtime ($17.93\ sec$). In Figures 4·2 and 4·3 we see results with respect to cell cost and retrieval runtime; the results are also summarized in Tables **??**, **??**, 4.2, and 4.3. For an accuracy of 99% BSE embeddings (constructed via learning) are faster than LB_Keogh by a factor of 22.2 in terms of retrieval runtime. For an accuracy of 80%, BSE embeddings (constructed via learning) yield a speedup of two orders of magitude compared to LB_Keogh and DTK. As seen in Table 4.3, BSE embeddings constructed via max variance (and thus not requiring a training set of queries) also perform well, being faster by a factor of 15.8 and 39.1 over LB_Keogh, for retrieval accuracy 99% and 80% respectively.

In terms of cell cost LB_Keogh appears to have a better performance for accuracies above 95%. However, the cell cost does not consider the cost of the filter step. The filter step of LB_Keogh is

much more expensive than that of BSE. This can be seen in Table 4.4 for the UCR dataset.

Robustness

Here we present experimental results that demonstrate that the performance of BSE embeddings is more robust than that of LB_Keogh and DTK, with respect to changes in the warping width w and changes in the length of queries.

Table 4.5 shows the effect of the warping width w for both LB_Keogh and BSE, as measured on the random walk dataset. For BSE, we selected an accuracy of 95%, a dimensionality of 40 and a sampling rate of 9. It can be seen that as w increases, the pruning power of LB_Keogh deteriorates fast. A similar observation is also made in [78]. The runtime of BSE also deteriorates, but at a much smaller pace: increasing w from 0.5% of the query length to 20% of the query length makes LB_Keogh 32 times slower, and BSE about 13 times slower.

The effect of query size is studied next, by setting the warping width to 5% and varying the query size from 100 to 1000. Tables 4.6 and 4.7 summarize our findings regarding cell cost and retrieval runtime respectively, for the two competitor methods and BSE, as measured on the random walk dataset. For BSE, we selected an accuracy of 95%, a dimensionality of 40 and a sampling rate of 9. For query sizes up to 300 the performance of DTK is improved as the query sizes increases; after that point, DTK deteriorates rapidly as the query size keeps increasing. Overall, increasing the query length from 100 to 1000 makes LB_Keogh more than 250 times slower, DTK about 38 times slower, and BSE about 8.6 times slower; BSE clearly demonstrates the slowest deterioration with increasing query length.

4.6.5 Further Analysis of our Method

This section provides a further analysis of BSE. We compare BSE embeddings with endpoint embeddings (EE), we compare performance of BSE embeddings optimized using the max variance heuristic vs. BSE embeddings optimized using learning, and we analyze the effects of dimensionality and sampling rate on the performance of BSE.

	LB_Keogh		BSE	
Warping width	Cell Cost	Runtime	Cell Cost	Runtime
0.5%	0.52%	1.91	0.81%	0.23
1.0%	0.93%	2.87	0.82%	0.34
2.5%	1.61%	4.65	0.89%	0.55
5.0%	2.68%	7.89	0.97%	0.81
10.0%	4.68%	12.62	1.02%	1.36
15.0%	10.19%	25.33	1.16%	2.09
20.0%	25.22%	61.73	1.27%	2.86

Table 4.5: Query size is set to 400.

Query size	BSE40-9 (95%)	LB_Keogh	DTK
100	0.375%	0.0672%	12.14%
200	0.552%	0.1972%	10.53%
300	0.765%	0.9082%	9.55%
400	0.974%	2.6834%	13.63%
500	1.183%	3.8764%	17.34%
600	1.212%	6.8772%	28.35%
700	1.491%	7.8972%	36.86%
800	1.527%	13.7644%	52.88%
900	1.753%	32.0987%	77.71%
1000	1.849%	46.5289%	89.35%

Table 4.6: Warping width is set to 5% of the query size. For BSE we show the cell costs for 95% accuracy.

Query size	BSE40-9 (95%)	LB_Keogh	DTK
100	0.66	1.19	15.89
200	0.72	3.42	11.23
300	0.75	5.32	9.52
400	0.81	8.57	13.56
500	0.97	14.35	19.66
600	1.35	25.92	42.33
700	1.84	49.22	84.47
800	2.58	98.80	156.22
900	3.22	173.33	311.18
1000	5.65	302.57	609.56

Table 4.7: Warping width is set to 5% of the query size. For BSE we show the cell costs for 95% accuracy.

Bidirectional vs. Endpoint Embeddings

The performance of BSE is compared with that of using only endpoint embeddings (denoted as EE embeddings). In Figures 4·2 and 4·3 we can see the performance of BSE vs. EE with respect to cell cost and retrieval runtime; the results are also summarized in Tables 4.2, and 4.3. In terms of retrieval runtime, BSE embeddings outperform EE embeddings across the board. The difference is even more pronounced for embeddings optimized via max variance; as Table 4.3 shows, BSE embeddings lead to runtimes between 2.5 and 4.5 times smaller compared to the runtimes attained using EE embeddings.

Effect of Training

In Figures 4·2 and 4·3, and Tables 4.2, and 4.3 we see the results obtained using BSE embeddings constructed using each of the two methods described in Section 4.5: the max variance heuristic and the greedy learning algorithm that uses a training set of queries. We see that the greedy learning algorithm invariably produces better results.

It is also interesting to compare how using learning affects BSE embeddings and EE embeddings. Comparing Figures 4·2 and 4·3 it can be seen that the learning method affects the performance of BSE embeddings much less than it affects the performance of EE embeddings. For example, for 99% accuracy the retrieval runtime is decreased by a factor of 1.4 for BSE embeddings, and by a factor of 4.44 for EE embeddings. In these experiments, BSE embeddings are shown to be less reliant on learning than EE embeddings. This is an additional advantage of BSE embeddings, as the learning method is not always a realistic option, as discussed in Section 4.5.

Effect of Dimensionality

For this set of experiments, the sampling rate was set to 1 and the dimensionality of the embedding varied from 10 to 160. In Figure 4·4 we can see the performance of BSE (optimized using learning) with respect to accuracy vs. cell cost and retrieval runtime respectively. We note that an embedding of dimensionality 40 produces the best accuracy with respect to both cell cost and retrieval runtime. The fact that the cell cost (which excludes the cost of comparing high-dimensional

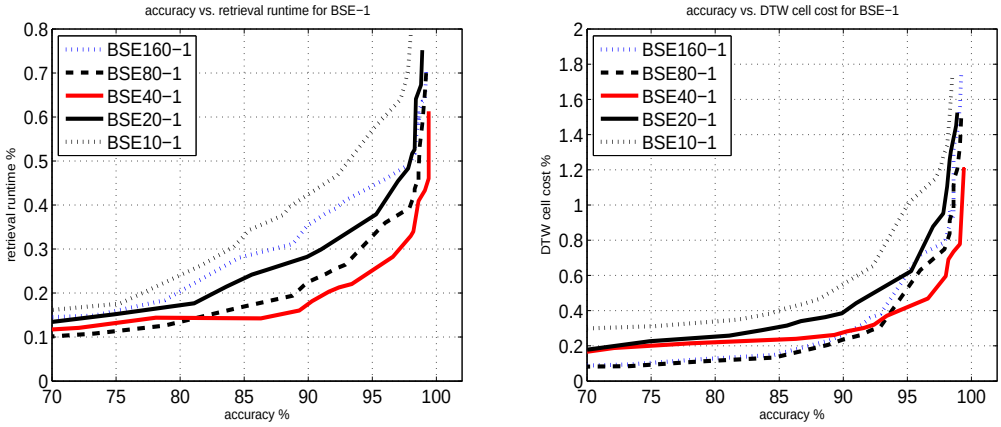

Figure 4·4: Sampling rate was set to 1 and the dimensionality of the embedding varies from 10 to 160. Warping width is 5% of the query size.

vectors) increases as the dimensionality goes from 40 to 80 and 160 is evidence that the learning algorithm suffers from overfitting, i.e., it tries to fit the training data too much. Using more training data is the standard way to avoid overfitting.

Effect of Sampling

Finally, the effect of sampling on both cell cost and retrieval runtime is studied. For this set of experiments, the dimensionality of the embedding was set to 40 and the sampling rate varied from 1 to 15. In Figure 4·5 we can see a comparison of accuracy vs. cell cost and retrieval runtime respectively for BSE. Based on the experimental evaluation on the UCR dataset, for the best dimensionality determined in the previous paragraph, the best sampling rate is 9. At the same time, we note that sampling rates of 5, 7, 9, and 11 give results fairly similar to each other, and thus the performance of BSE embeddings is not particularly sensitive to the choice of sampling rate.

4.6.6 Replication of BSE on other datasets

For the same reasons discussed in section 3.6.4 of chapter 3, we are now going to describe how to set the parameters for BSE, given a new time series dataset. Notice that as opposed to EBSM, BSE does not require any training.

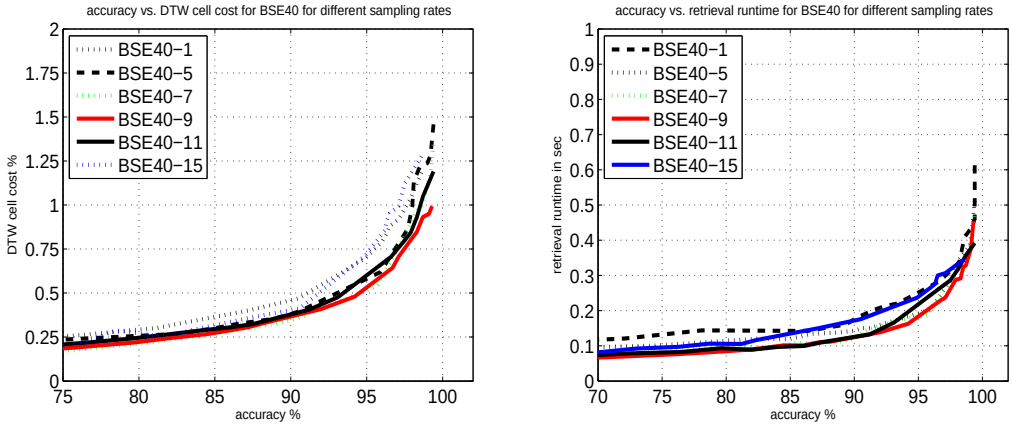

Figure 4·5: The dimensionality of the embedding is set to 40 and the sampling rate varies from 1 to 15. Warping width is 5% of the query size. Training has been performed on BSE.

BSE should be set up so that best performance can be achieved in terms of retrieval runtime. For the reference sequence selection phase, in a similar manner as EBSM, one approach is to ask the user to provide a sample of queries similar to those expected when BSE will be running online. Applying the maximum variance heuristic, out of $k = 2,000$ reference sequences taken randomly from the database, those d sequences with the maximum variance will be selected. Those d and p values with the best retrieval runtime on the query sample will be assigned for the online phase of BSE.

Another approach is to set k, d, and p to an initial value empirically chosen based on datasets seen so far. Then, during the online phase, for the set of queries seen so far, we can keep track of the distribution of their pairwise distances. When their distances start increasing in variance (i.e. possibly a new type of queries has entered the system), the maximum variance heuristic can be applied and determine new values for the parameters, if these values can achieve a better retrieval runtime on the new query sample.

4.7 Summary

We have described Bidirectional Subsequence Embeddings (BSE), a novel method for efficient subsequence matching of time series under constrained DTW. BSE embeddings take advantage of

the constraints of cDTW to associate, given a query, a vector to each possible subsequence match in the database. By comparing the embedding of the query to the embeddings of the possible subsequence matches, a relatively short number of candidate matches can be efficiently identified.

Experiments with real and synthetic datasets demonstrate the computational savings obtained using BSE, and a speedup of one to two orders of magnitude compared to the state-of-the-art exact methods for this problem, at the cost of some modest loss in retrieval accuracy. A speedup of over one order of magnitude is obtained while still maintaining 99% retrieval accuracy. A speedup of two orders of magnitude is achieved at the cost of getting correct results for only 80% of the queries. Furthermore, BSE embeddings are shown to be significantly more robust than competing methods in their ability to tolerate large warping widths and large query lengths.

Chapter 5

Reference-based Alignment of Sequence Databases

Two reference-based methods (one exact and one approximate) for similarity search in large sequence databases are described in this chapter. Both methods use a set of reference sequences to map each database point to a real value. The similarity measure used for this case is the edit distance (ED) and thus, due to its metric property, lower bounds can be defined and used to efficiently filter candidate matches. A letter collapsing technique is also used to improve performance.

In the remainder of this chapter, we provide some background on similarity search in string databases (mainly DNA sequence databases), then we describe the two methods in more detail and finally present the experimental evaluation of our methods against state-of-the-art biological sequence similarity search methods.

5.1 Background

In this section we define the edit distance and Smith-Waterman measures used to evaluate similarity between strings (e.g. DNA). We use the terms "string" and "sequence" interchangeably. Throughout this section, the following notation will be used:

- Q, X are sequences of length $|Q|$ and $|X|$ respectively. Q denotes a query sequence and X denotes a database sequence. Typically $|X| \gg |Q|$. Without loss of generality we assume that the database contains a single very long sequence, since we can always concatenate all the strings stored in the database into a single string.

- Subscripts denote elements of sequences. For example, $Q = (Q_1, \ldots, Q_{|Q|})$.

- For any sequence $X = (X_1, \ldots, X_{|X|})$, given start and end positions s and t respectively, we can define *subsequence* $X^{s:t}$ to be the sequence $(X_s, \ldots, X_t)$, i.e., the part of X that

starts at position s and ends at position t. Then, $X_i^{s:t}$ is the i-th element of $X^{s:t}$, and is equal to X_{s+i-1}.

5.1.1 The Edit Distance

The edit distance $\Delta(A, B)$ is a function measuring how *dissimilar* two strings A and B are. For a more general definition of the edit distance we need to specify a cost for each editing operation, i.e., for each insertion, deletion, and substitution. In this thesis we denote these costs as follows:

- C_{ins} denotes the cost of the edit operation that inserts a letter to string A.
- C_{del} denotes the cost of the edit operation that deletes a letter from string A.
- $C_{\text{sub}}(A_j, B_t)$ denotes the cost of the edit operation that replaces letter A_j with some letter $B_t \neq A_j$.

In the general case, $\Delta(A, B)$ is the smallest possible cost of converting A to B using insertions, deletions, and substitutions. In the most common version of ED, $C_{\text{ins}} = C_{\text{del}} = C_{\text{sub}} = 1$, and in that case $\Delta(A, B)$ is the smallest total number of insertions, deletions, and substitutions that can convert A to B. For simplicity, in the remainder of this thesis we assume that $C_{\text{ins}} = C_{\text{del}} = C_{\text{sub}} = 1$.

Given a query sequence Q and a database sequence X, the best (optimal) *subsequence match* of Q in X is the subsequence $X^{s:t}$ that minimizes $\Delta(Q, X^{s:t})$. We define the subsequence matching cost $D(Q, X)$ as:

$$D(Q, X) = \min\{\Delta(Q, X^{s:t}) | s \in \{1, \ldots, t\}, t \in \{1, \ldots, |X|\}\} \ . \tag{5.1}$$

In describing how to compute $D(Q, X)$ and the corresponding subsequence match $X^{s:t}$, it is useful to define an auxiliary distance $D^{j,t}$, as the smallest possible distance between $Q^{1:j}$ and a *suffix* $X^{s:t}$ of $X^{1:t}$:

$$D^{j,t}(Q, X) = \min\{\Delta(Q^{1:j}, X^{s:t}) | s \in \{1, \ldots, t\}\} \ . \tag{5.2}$$

We also define an auxiliary function $C(Q_j, X_t)$ that denotes the cost of matching letter Q_j with letter X_t:

$$C(Q_j, X_t) = \begin{cases} C_{\text{sub}} & \text{if } Q_j \neq X_t \\ 0 & \text{if } Q_j = X_t \end{cases} \tag{5.3}$$

Computing $D(Q, X)$ and the corresponding best subsequence match of Q in X can be performed using dynamic programming, by computing $D^{j,t}(Q, X)$ for $j = 1, \ldots, |Q|$ and $t = 1, \ldots, |Q|$, as follows:

initialization:

$$D^{0,0} = 0, D^{j,0} = \infty, D^{0,t} = 0 \; . \tag{5.4}$$

loop:

$$D^{j,t}(Q, X) = \min \begin{cases} D^{j,t-1}(Q, X) + C_{\text{ins}} \\ D^{j-1,t}(Q, X) + C_{\text{del}} \\ D^{j-1,t-1}(Q, X) + C(Q_j, X_t) \end{cases} \tag{5.5}$$

$$(j = 1, \ldots, |Q|; t = 1, \ldots, |X|) \; .$$

termination:

$$t^* = \text{argmin}_{t=1,\ldots,|X|}\{D^{|Q|,t}(Q, X)\} \; . \tag{5.6}$$

$$D(Q, X) = D^{|Q|,t^*}(Q, X) \; . \tag{5.7}$$

It should be clear that evaluating $D(Q, X)$ takes time $O(|Q||X|)$. We should also note that the optimal matching sequence can be found by keeping track, in each application of Equation 5.5, of the predecessor selected for each (j, t), and by backtracking, at termination, starting at position $(|Q|, t^*)$.

5.1.2 The Smith-Waterman Measure

A similarity measure $\Lambda(A, B)$, in contrast to a distance measure, measures how *similar* two strings A and B are. If $\Lambda(A, B) = 0$ then A and B are maximally different from each other. The Smith-Waterman measure [79] is a frequently used similarity measure for strings. In order to specify the Smith-Waterman measure, we need to choose values $P_{\text{match}}, P_{\text{sub}}$ and P_{gap}, that stand for the following terms:

- P_{match} is a positive number that denotes the reward for a letter of A being equal to the corresponding letter in B.
- P_{sub} is a negative number that denotes the penalty for a letter of A being substituted by another letter.
- P_{gap} is a negative number that denotes the penalty for deleting a letter of A, or inserting a letter to A.

In the remainder of the thesis, and in our experiments, we use $P_{\text{match}} = 2, P_{\text{sub}} = -1$, and $P_{\text{gap}} = -1$, which are commonly used choices for these parameters.

Given a query string Q and a database string X, finding the best (optimal) *local alignment* between Q and X is the task of finding subsequences $Q^{i:j}$ and $X^{s:t}$ that maximize $\Lambda(Q^{i:j}, X^{s:t})$. We define the Smith-Waterman similarity score $L(Q, X)$ as:

$$L(Q, X) = \max\{\Lambda(Q^{i,j}, X^{s:t}) | i \in \{1, \ldots, j\}, j \in \{1, \ldots, |Q|\}, \\ s \in \{1, \ldots, t\}, t \in \{1, \ldots, |X|\}\} \ . \tag{5.8}$$

In describing how to compute $L(Q, X)$ and the corresponding optimally matching subsequences $Q^{i:j}$ and $X^{s:t}$, it is useful to define an auxiliary score $S^{j,t}$, as the highest matching score between a suffix $Q^{i:j}$ of $Q^{1:j}$ and a *suffix* $X^{s:t}$ of $X^{1:t}$:

$$L^{j,t}(Q, X) = \max\{\Lambda(Q^{i:j}, X^{s:t}) | i \in \{1, \ldots, j\}, s \in \{1, \ldots, t\}\} \ . \tag{5.9}$$

We also define an auxiliary function $P(Q_j, X_t)$ that denotes the reward or penalty of matching letter Q_j with letter X_t:

$$P(Q_j, X_t) = \begin{cases} P_{\text{sub}} & \text{if } Q_j \neq X_t \\ P_{\text{match}} & \text{if } Q_j = X_t \end{cases} \tag{5.10}$$

Given Q and X, the Smith-Waterman algorithm identifies optimal subsequences $Q^{i:j}$ and $X^{s:j}$ and the corresponding similarity score $L(Q, X) = \Lambda(Q^{i:j}, X^{s:t})$. The Smith-Waterman algorithm is very similar to the algorithm computing the edit distance, and also proceeds using dynamic programming, by computing $L^{j,t}$ for $j = 1, \ldots, |Q|$ and $t = 1, \ldots, |Q|$, as follows:

initialization:

$$L^{j,0} = 0, L^{0,t} = 0 \; . \tag{5.11}$$

$$L^{j,t}(Q, X) = \max \begin{cases} L^{j,t-1}(Q, X) + P_{\text{gap}} \\ L^{j-1,t}(Q, X) + P_{\text{gap}} \\ L^{j-1,t-1}(Q, X) + P(Q_j, X_t) \\ 0 \end{cases} \tag{5.12}$$

$(j = 1, \ldots, |Q|; t = 1, \ldots, |X|) \; .$

termination:

$$L(Q, X) = \max_{j=1,\ldots,|Q|, t=1,\ldots,|X|} \{L^{j,t}(Q, X)\} \; . \tag{5.13}$$

Similar to the edit distance, Smith-Waterman takes time $O(|Q||X|)$, and finding the subsequences of Q and X that give the maximum similarity score can be easily done using backtracking.

5.2 RBSA for Fixed Query Length

In this section, we describe the proposed RBSA (Reference-Based String Alignment) method for queries of fixed length. We denote that fixed length as q. In Section 5.3, we will generalize RBSA to queries of arbitrary length.

RBSA follows a filter-and-refine approach. A set of random reference sequences is generated. For each database position, an alignment score with each reference sequence is computed, and an embedding-based index is constructed using those scores. The embedding is used for fast filtering of database positions that can lead to a potential match. Those positions are then passed to the refine step where the computationally expensive distance measure (edit distance or Smith-Waterman) is applied.

5.2.1 Embedding Queries and Database Positions

Let Q be a query sequence of fixed length $|Q| = q$, and X be the database sequence. At the core of our method is an embedding definition, that we use to produce one-dimensional $(1D)$ mappings, that map every query sequence Q to a number, and that map every database position (X, t) also to a number. We will use these $1D$ mappings to obtain bounds for the optimal subsequence matching or local alignment score *ending* at each database position (X, t), and then we will use those bounds to efficiently prune significant portions of the database.

Let R be a sequence of the same fixed length q as the queries. Using R we can define a 1D embedding F^R, mapping each query sequence into a real number $F^R(Q)$, and also mapping each database position (X, t) into a real number $F^R(X, t)$:

$$F^R(Q) = D^{|R|,|Q|}(R, Q) \; . \tag{5.14}$$

$$F^R(X, t) = D^{|R|,t}(R, X) \; . \tag{5.15}$$

The above equations can be interpreted intuitively as follows: the embedding $F^R(Q)$ of the query is the smallest edit distance matching R to a suffix of Q. The embedding $F^R(X, t)$ of database position (X, t) is the smallest edit distance matching R to a suffix of $X^{1:t}$. If a very close match to Q appears as $X^{s:t}$ in X, then we expect $F^R(Q)$ to be very similar to $F^R(X, t)$. Any sequence R used to define an embedding F^R is called a *reference sequence*.

5.2.2 Reference-based Bounds for the Edit Distance and Smith-Waterman

Let Q be a query string, X be the database sequence, and t be a position on X. As a reminder, $\Delta(A, B)$ is the edit distance between strings A and B, and $D^{|Q|,t}(Q, X)$ is the smallest edit distance between Q and any subsequence of X ending at position (X, t). To establish an exact reference-based filtering method for the subsequence matching problem, our first step is to establish a lower bound for $D^{|Q|,t}(Q, X)$ based on $F^{R_i}(Q)$ and $F^{R_i}(X, t)$, where R_i is any reference sequence.

Proposition 1 *For any query Q, database position (X, t), and reference sequence R_i, define $lb_{ED}^{i,t}(Q)$ as follows.*

$$lb_{ED}^{i,t}(Q) = F^{R_i}(X, t) - F^{R_i}(Q). \tag{5.16}$$

Then, it holds that:

$$lb_{ED}^{i,t}(Q) \leq D^{|Q|,t}(Q, X), \tag{5.17}$$

and thus $lb_{ED}^{i,t}(Q)$ is a lower bound for the smallest possible edit distance between Q and a subsequence of X ending at (X, t).

Proof: First, we need to make the following auxiliary definitions:

$$M(A, B, t) = \text{argmin}_{B^{s:t}|s=1,\ldots,t}\{\Delta(A, B^{s:t})\}, \tag{5.18}$$

$$Q' = M(R_i, Q, |Q|). \tag{5.19}$$

In words, $M(A, B, t)$ is the subsequence of B ending at position (B, t) that has the smallest edit distance from A, and Q' is the suffix of Q that has the smallest edit distance from R_i. Then, we can prove Proposition 1 as follows:

$$lb_{ED}^{i,t}(Q) = F^{R_i}(X, t) - F^{R_i}(Q) \tag{5.20}$$

$$= \Delta(R_i, M(R_i, X, t)) - \Delta(R_i, Q') \tag{5.21}$$

$$\leq \Delta(R_i, M(Q', X, t)) - \Delta(R_i, Q')) \tag{5.22}$$

$$\leq \Delta(M(Q', X, t), Q') \tag{5.23}$$

$$\leq \Delta(M(Q, X, t), Q). \tag{5.24}$$

To justify the above derivation, we note the following:

- $\Delta(R_i, M(R_i, X, t)) \leq \Delta(R_i, M(Q', X, t))$ since both $M(R_i, X, t)$ and $M(Q', X, t)$ are subsequences of X ending at (X, t), and $M(R_i, X, t)$ is defined as the subsequence of X ending at (X, t) that has the smallest distance with R_i.
- The edit distance is metric, so the triangle inequality holds, and $\Delta(R_i, M(Q', X, t)) - \Delta(R_i, Q') \leq \Delta(M(Q', X, t), Q')$.
- We can prove $\Delta(M(Q', X, t), Q') \leq \Delta(M(Q, X, t), Q)$ by considering that when we perform the minimal set of edit operations that convert Q to $M(Q, X, t)$, those same operations suffice to convert Q' (which is a suffix of Q) to a suffix of $M(Q, X, t)$. Therefore, the smallest possible edit distance between Q' and a subsequence of X ending at (X, t) cannot be greater than $\Delta(M(Q, X, t), Q)$.

□

If we are actually interested in retrieving optimal matches under the Smith-Waterman similarity measure, as opposed to the edit distance, we can easily convert the lower bound of the edit distance to an upper bound for Smith-Waterman. We can prove the following:

Proposition 2 *For any query Q and database position (X, t), define $ub_{SW}^{i,t}(Q)$ as follows:*

$$ub_{SW}^{i,t}(Q) = 2|Q| - lb_{ED}^{i,t}(Q). \tag{5.25}$$

Suppose that we define a Smith-Waterman similarity measure using $P_{\text{match}} = 2$, $P_{\text{gap}} = -1$, and $P_{\text{sub}} = -1$. Then, it holds that:

$$ub_{SW}^{i,t}(Q) \geq L^{|Q|,t}(Q, X), \tag{5.26}$$

where $L^{|Q|,t}(Q, X)$ is the highest Smith-Waterman score between Q and a subsequence of X ending at (X, t). Thus $ub_{SW}^{i,t}(Q)$ is an upper bound for the Smith-Waterman score between Q and any subsequence of X ending at (X, t).

Proof: First, we need to make the following auxiliary definition:

$$M_{SW}(Q, X, t) = \text{argmax}_{X^{s:t}|s=1,\ldots,t}\{\Lambda(Q, X^{s:t})\}. \tag{5.27}$$

In words, $M_{SW}(Q, X, t)$ is the subsequence of X ending at position (X, t) that has the highest Smith-Waterman score with Q. Then, we can prove Proposition 2 as follows:

$$
\begin{aligned}
ub_{SW}^{i,t}(Q) &= 2|Q| - lb_{ED}^{i,t}(Q) && (5.28)\\
&\geq 2|Q| - \Delta(Q, M(Q, X, t)) && (5.29)\\
&\geq 2|Q| - \Delta(Q, M_{SW}(Q, X, t)) && (5.30)\\
&\geq L^{|Q|,t}(Q, X) && (5.31)
\end{aligned}
$$

In justifying the above derivation, the most important step is showing that $2|Q| - \Delta(Q, M_{SW}(Q, X, t)) \geq L^{|Q|,t}(Q, X)$. The argument for that is as follows: Consider the optimal alignment (according to Smith-Waterman) between Q and $M_{SW}(Q, X, t)$. If Q perfectly matches $M_{SW}(Q, X, t)$, then the alignment score is $2|Q|$, since we get a reward of $P_{\text{match}} = 2$ for every letter of Q. Any mismatch and gap in the optimal alignment causes the alignment score to decrement by at least 1. Therefore, we know that the number of mismatches and gaps in the optimal alignment cannot be greater than $2|Q| - L^{|Q|,t}(Q, X)$. At the same time, the optimal alignment between Q and $M_{SW}(Q, X, t)$ defines a sequence of edit operations (substitutions for mismatches and insertions or deletions for gaps) that converts Q to $M_{SW}(Q, X, t)$. Consequently, the edit distance between Q and $M_{SW}(Q, X, t))$ cannot exceed $2|Q| - L^{|Q|,t}(Q, X)$.

□

Notice that since Smith-Waterman is a similarity score (and not a distance measure) upper bounds established efficiently during a filtering step can be used to prune away candidate database matches, while guaranteeing that the correct answer will not be pruned. This is quite analogous to the use of lower bounds for efficient filtering when looking for the best matches under a distance measure.

5.2.3 Offline Selection of Reference Sequences

We have shown how to use reference-based alignment scores computed for database positions and for the query in order to obtain lower bounds of the edit distance or upper bounds for the Smith-

Waterman similarity score between the query and subsequences ending at each database position. We say that, for a query Q, database position (X, j) is pruned using R_i, if $lb_{ED}^{i,j}(Q) > \delta q$, where $lb_{ED}^{i,j}(Q)$ is as defined in Eq. 5.16, and δ is the maximum amount (expressed as fraction of the query length) of difference between the query and its subsequence match that we are willing to tolerate. We note that, if the best match has an edit distance of more than δq from Q, we are not interested in retrieving that match.

The filter step of RBSA, which is described in Section 5.2.4, prunes database positions using information from reference sequences. However, given a query Q, it would take too much time to check for each database position if it can be pruned using every single reference sequence. Therefore, we perform an off-line preprocessing step, at which we identify, for every database position, the best reference sequences (out of thousands of available sequences) to use for that position. Intuitively, reference sequences R for which $F^R(X, j)$ is high (meaning that R is far from any subsequence of X ending at position j) tend to provide tighter lower bounds according to Eq. 5.16. Our reference selection method is inspired by that of [85], although that approach was proposed in the context of full sequence matching.

For the reference selection process, we use two sets: 1) a set $\mathcal{Q}_{\text{sample}} = \{Q_1, \ldots, Q_{|\mathcal{Q}_{\text{sample}}|}\}$ of randomly generated queries with $|Q_i| = q$, and 2) a set of randomly generated reference objects $\mathcal{R} = \{R_1, \ldots, R_{|\mathcal{R}|}\}$ with $|R_i| = q$. For each database position (X, j), the set of reference objects to use for that position are selected using a greedy approach. More specifically, for each position (X, j), we first choose reference object R_j^1 to be the reference sequence R that prunes position (X, j) for the largest number of queries in $\mathcal{Q}_{\text{sample}}$. Then, the queries for which (X, j) is pruned by R_j^1 are removed from $\mathcal{Q}_{\text{sample}}$. Similarly, we choose reference object R_j^i to be the reference sequence R that prunes position (X, j) for the largest number of queries in $\mathcal{Q}_{\text{sample}}$, where $\mathcal{Q}_{\text{sample}}$ has been modified to exclude queries for which (X, j) is pruned using the previously chosen reference objects $R_j^1, \ldots, R_j^{i-1}$.

The final outcome is the set $\mathcal{R}^K = \{\mathcal{R}_1^K, \ldots, \mathcal{R}_{|X|}^K\}$, where $\mathcal{R}_j^K$ contains the top K reference objects for position(X, j). For each position (X, j) we also store all values $F^{R_j^i}(X, j)$, for $i = 1, \ldots, K$. The pseudocode for selecting reference objects for each database position is given in

Algorithm 1. We should note that the selection of reference objects is an offline process and is executed only once.

5.2.4 Filter Step

Next we describe the online behavior of RBSA at query time, for queries of fixed size q. The retrieval process, given Q, consists of a filter step and a refine step. Given a query Q, its embeddings $F^{R_i}(Q)$ under all reference objects in $\mathcal{R}$ are computed. Then, for each database position (X, j), each $R_j^i \in \mathcal{R}_j^K$ is considered, until either an R_j^i is found that prunes (X, j), or all $R_j^i \in \mathcal{R}_j^K$ have been considered. In the latter case, position (X, j) is a candidate endpoint of a subsequence match, that will be considered by the refine step. The filter step is described in Algorithm 2.

5.2.5 Refine Step

The filter step produces set $\mathrm{candidates}$ that contains endpoints of possible database matches for the query. At the refine step, each of those candidates is evaluated. Naturally, depending on whether we want to retrieve the best matches according to the edit distance or Smith-Waterman, we use respectively the edit distance or Smith-Waterman to evaluate each candidate endpoint.

For the case of the edit distance, the refine step is shown in Algorithm 3. It is fairly straightforward to adapt that algorithm to work for the Smith-Waterman similarity measure.

5.2.6 Alphabet Collapsing

The filtering power of RBSA is improved by employing an alphabet collapsing technique. In particular, for the case of DNA sequences the alphabet is $\Sigma = \{A, C, G, T\}$. We can reduce the alphabet size to 2 by applying four possible collapsing schemes:

- Scheme 0: No collapsing (letters remain unchanged).
- Scheme 1: A and C map to X, G and T map to Y.
- Scheme 2: A and G map to X, C and T map to Y.
- Scheme 3: A and T map to X, C and G map to Y.

A combination of the four schemes is used to improve the filtering power of RBSA. Let T_i be a transformation function that converts an input string defined in alphabet Σ to its corresponding string defined in scheme i. In the offline selection of reference sequences for each database position (Section 5.2.3), each reference sequence $R \in \mathcal{R}$ eventually generates four different reference sequences: $T_0(R)$, $T_1(\mathcal{R})$, $T_2(\mathcal{R})$ and $T_3(\mathcal{R})$. The same transformations are also applied to the database thus producing $T_0(X)$, $T_1(X)$, $T_2(X)$ and $T_3(X)$.

Reference object $T_i(R)$ can be used to obtain bounds and prune database positions (X, j) by comparing $F^{T_i(R)}(T_i(Q))$ with
$F^{T_i(R)}(T_i(X), j))$. Bounds obtained using any of the transformations T_i are still true for the untransformed sequences, since we can easily show that, for any of the four T_i's, the edit distance $\Delta(A, B) \geq \Delta(T_i(A), T_i(B))$. The offline process for reference selection considers each of the $T_i(R)$'s as a separate candidate reference sequence and typically chooses, for each database position, reference sequences obtained from all letter collapsing schemes.

At query time, the query Q is also converted into each of the four representations, $T_0(Q)$, $T_1(Q)$, $T_2(Q)$ and $T_3(Q)$. Filtering is modified to include these transformations. For each database position (X, j), lower bounds are computed for each T_i.

We have found empirically that we get more pruning power by combining bounds from the untransformed sequences and bounds from the transformed sequences obtained using letter collapsing. Reference objects obtained via letter collapsing have a larger variance in their distances to database subsequences, thus leading to better pruning. We should underline that in [85] it is also noted (in the context of full sequence matching) that pruning power improves when using reference objects whose distances to database sequences have higher variance, but that approach did not use letter collapsing.

5.3 RBSA for Variable Query Length

The discussion in Section 5.2 addressed the problem of efficient retrieval of subsequence matches for query sequences of fixed length q. In this section we describe how to build upon the solutions proposed for the fixed query length problem to obtain solutions for the variable query length

problem. We assume that we have already prepared an index, as described in Section 5.2.3, for processing queries of fixed size q. In our experiments, $q = 40$.

Let Q be a query. In principle, Q can have arbitrary size, but for simplicity we assume that $|Q| = \alpha q$, for some $\alpha \in \mathbb{N}$. No constraints are placed on α, and α can be different for each query. At query time, the query is broken into non-overlapping segments $Q^1, \ldots, Q^\alpha$ of size q. We now proceed to describe two different methods, one exact, and one approximate, for using results obtained for the different segments Q^i in order to identify the subsequence match for the entire query.

5.3.1 Exact RBSA

The exact version of RBSA is based on a simple observation: if Q has a subsequence match with edit distance $\leq \delta|Q|$, then at least one of the query segments Q^i has a subsequence match with edit distance $\leq \delta q$. This can be seen by observing that each of the edit operations that transforms Q into its subsequence match is applied to one of the individual query segments. After all edit operations have been applied, each query segment Q^i has been transformed to a database subsequence. If each query segment Q^i needed more than δq edit operations to be converted to its optimal database match, then the entire query would need more than $\alpha\delta q = \delta|Q|$ operations to be converted to its optimal database match.

Let $X^{s:t}$ be a subsequence match for the entire query Q, with distance $\leq \delta|Q|$. Then, we can show that there exists at least one Q_i that has, within $X^{s:t}$, a subsequence match $X^{s':t'}$ with distance $\leq \delta q$, and such that $t' \in \{t - q(\alpha - i) - \delta|Q|, \ldots, t - q(\alpha - i) + \delta|Q|\}$. Conversely, if for some segment Q^i we have found a match $X^{s':t'}$ with distance $\leq \delta q$, this generates a set of candidate endpoints for a subsequence match of the entire query. This set of candidate endpoints is equal to $\{t' + q(\alpha - i) - \delta|Q|, \ldots, t + q(\alpha - i) + \delta|Q|\}$.

Let sorted be the union of the sets of candidate endpoints generated from all matches of all segments Q^i, and let's assume that sorted is sorted in descending order. Then, evaluating those candidate endpoints can be done by invoking Algorithm 3, i.e., the exact same algorithm that was used for the refine step of the fixed-query-length version. It should be clear from the preceding

paragraphs that this algorithm is guaranteed to identify the correct subsequence match, as long as that match is within edit distance $\delta|Q|$ from Q. As in the fixed-length case, Algorithm 3 can easily be adapted to use Smith-Waterman instead of the edit distance, so as to identify the optimal Smith-Waterman match for the query (but still assuming an edit distance $\leq \delta|Q|$ from Q).

5.3.2 Approximate RBSA

In the exact version of RBSA we try to find subsequence matches within δq edit distance of each of the α query segments Q^i. An important question, whose answer forms the foundation of the approximate version of RBSA, is the following: what if, instead of using all segments Q^i, we used a single random Q^i? What would be the probability of the endpoint of the subsequence match for the entire query being included in the set of candidate endpoints generated by that single Q^i? It turns out, as we prove next, that under some fairly reasonable assumptions, this probability is at least 50%.

In order to prove the above claim, we need to make some assumptions about the distribution of edit operations needed to convert Q into its optimal subsequence match. We denote the best subsequence match of Q in X as $M(Q, X)$. Since we assume that $\Delta(Q, M(Q, X)) \leq \delta|Q|$, at most $\delta|Q|$ edit operations are needed to convert Q to $M(Q, X)$. Each of these edit operations is applied to one and only one of the α segments Q^i that the query has been partitioned to. We denote by Q^{c_m} the query segment where the m-th edit operation is applied, and by $P(c_m = i)$ the probability that the m-th edit operation is applied to segment Q^i.

Proposition 3 *Let Q be a query, and $M(Q, X)$ be the optimal subsequence match of Q in X. We assume that $\Delta(Q, M(Q, X))$*
$= n \leq \delta|Q|$, $\alpha \geq 4$, $P(c_m = i)$ is uniform over all i, and the distributions $P(c_m = i)$ corresponding to all m are mutually independent. In other words, we assume that the distribution of c_m does not depend on any c_n, for $n \neq m$. Consider the optimal sequence of edit operations that convert Q to $M(Q, X)$. Given any Q^i, there is a probability of at least 50% that, out of those edit operations, at most δq edit operations are applied to Q^i.

Proof: The probability that exactly k out of the n edit operations are applied to Q^i follows a binomial distribution, where we have n trials, "success" is the case where an edit operation is

applied to Q^i, and the probability of success for an individual trial (i.e., a specific edit operation) is $\frac{1}{\alpha}$. The expected number of successes over n trials is $\frac{n}{\alpha}$ (as a reminder, α is defined as $|Q|/q$). If $\alpha \geq 4$, as we assume, the probability of success is ≤ 0.25, and for that case it has been shown [23] that there is at least a 50% probability that the number of successes will not exceed the expected value n/α. Since $n \leq \delta|Q|$, it follows that $\frac{n}{\alpha} \leq \delta\frac{|Q|}{\alpha} = \delta q$, and the probability that at most δq edit operations are applied to Q^i is at least 50%.

□

Based on Proposition 3, by choosing a single Q^i, and generating candidate endpoints for the subsequence match of the entire query based on subsequence matches retrieved for Q^i, we have a probability of at least 50% to include the correct endpoint (i.e., the endpoint of the optimal subsequence match for the entire query) in those candidates. If the correct point is not included in those candidates, it follows that more than δq edit operations were applied to Q^i. In that case, for any $j \neq i$, the probability that at most δq edit operations are applied to Q^j is still at least 50%, and it is actually higher now that we know that more than δq edit operations were applied to Q^i.

By extending that reasoning, if we generate candidate endpoints for the match of the entire Q using p segments $Q^{i_1}, \ldots, Q^{i_p}$, the probability of not including the correct endpoint in those candidates is at most $\frac{1}{2^p}$, and thus drops exponentially with respect to p. If the correct endpoint is indeed included in those candidates, then the optimal subsequence match is guaranteed to be identified using the same refine step as in exact RBSA, and as in Algorithm 3. In our experiments, we use $p = 10$, so that the probability of retrieving the correct result is at least 99.9%.

5.4 Experiments

The performance of RBSA is evaluated on biological data obtained from the NCBI repository. RBSA is compared with state-of-the-art methods for string matching under the edit distance and the Smith-Waterman similarity measure. With respect to the edit distance, we have compared with Q-grams. With respect to Smith-Waterman, we have compared with:

- BLAST2[2]: the expect value E has been adjusted to achieve retrieval accuracies of 95%,

98%, and 100%. In the tables and figures that follow, this adjustment is denoted as $BLASTX$, which means that the E values have been adjusted to guarantee $X\%$ retrieval accuracy compared to Smith-Waterman.

- BWT-SW[43]: a local alignment method that guarantees 100% retrieval accuracy.

For the purposes of the experimental evaluation, we denote the exact version of RBSA as E-RBSA, and the approximate version as A-RBSA. For notation purposes, the distance/similarity measure (edit distance (ED) or Smith-Waterman (SW)) used in the refine step of RBSA is added as a suffix at the end of each notation. For example, E-RBSA-ED is the exact version of RBSA using the edit distance at the refine step, whereas A-RBSA-SW denotes the approximate version of RBSA using Smith-Waterman at the refine step. In the following sections, we use the term $RBSA$ to refer to our method in general. The other notation is only used to distinguish within different versions of RBSA when needed.

5.4.1 Datasets

RBSA has been tested on Human Chromosome 22. The size of this chromosome is $35{,}059{,}634$ bases. For the experiments described in section 5.4.2, the database sequence consisted of the first $184{,}309$ bases of the chromosome. For the rest of the experiments, the database sequence consisted of the whole chromosome, and thus had a length of $35{,}059{,}634$ letters. Queries have been extracted from random chromosomes of the mouse genome. Their size varied from 40 to $10K$ nucleotides (i.e., 40 to $10K$ letters) and their similarity to the database varied within 5%, 10%, and 15% edit operations, which, as also discussed earlier, is a reasonable range of δ values needed for the applications targeted by this thesis. Several sets of queries have been created, one for each combination of the above parameters. Each set contains 200 queries.

Performance Measures

The two key measures of performance in this context are accuracy and efficiency. E-RBSA is **exact** meaning that it is always guaranteed to find the optimal match for each query. Hence its accuracy is always 100%. On the other hand, A-RBSA is **approximate**, therefore we use the term

Retrieval Accuracy (RA) to express the percentage of the correct nearest neighbors found over the total number of queries. Efficiency is measured based on the **Retrieval Runtime Percentage (RRP)** for each query. RRP is defined as follows:

$$RRP = \frac{RBSA\ in\ sec}{brute\ force\ in\ sec}\ 100\%. \tag{5.32}$$

For our experiments the brute-force case is the full dynamic programming algorithm. Efficiency is also measured based on the **cell cost** for each query, which is the percentage of database positions visited during the refine step.

Specifically, two sets of experiments have been performed: for the first set, the edit distance has been used at the refine step, whereas for the second we used the Smith-Waterman similarity measure. The system was implemented in C++, and run on an AMD Opteron 8220 SE processor running at 2.8 GHz. For all the experiments, parameter K of Algorithm 1 was set to 50. The runtime of Algorithm 1 to determine the best 50 reference sequences for a single database point was approximately 81 seconds, and for the whole Human Chromosome 22 it was approximately 5 days.

5.4.2 Experimental Results

First we show the experimental performance of RBSA when the edit distance is used at the refine step. In this case, the main competitors are the q-gram based methods. Then, we compare the performance of RBSA on Smith-Waterman against BLAST and BWT-SW. To provide a thorough experimental analysis we show the performance of RBSA considering the following factors: 1) the effect of letter collapsing, 2) the effect of query size and δ, and 3) the effect of the number of reference objects used for the filter step.

Edit Distance: Comparison with Q-grams

The major competitors in the case of edit distance are the q-gram based approaches. Their inefficiency for long queries with a relatively large deviation from the database has already been discussed earlier in this thesis. In Table 5.1 we show that their pruning power deteriorates for

queries of size larger than 100 and for values of δ that exceed 5%. For this experiment only, we used a small dataset that included the first $184{,}309$ bases of Human Chromosome 22. The queries had a match within $\delta = 5\%$, 10%, and 15%. The experiment was organized as follows: for each query size $|Q|$, we used a set of sliding windows $\mathcal{W}$ with size varying in $[|Q|(1-\delta),\ |Q|(1+\delta)]$. The database was scanned using $\mathcal{W}$, and all possible sequences were enumerated. For each query size and δ value, we show the cell cost for the optimal q value. Clearly, for query sizes larger than 100 or δ values greater than 10%, the pruning power of q-grams deteriorates significantly, rendering them inappropriate for such string searches in large string databases. Due to this observation, we did not perform any further experiments with q-gram based state-of-the-art methods for subsequence matching. Also, an application of a full sequence matching q-gram based algorithm like [47, 48] would not work either as these algorithms are designed for full sequence matching (as pointed out by one of the authors of [47, 48]).

For the experiments described in the remainder of this section the database sequence is the whole Human Chromosome 22. Next, we show the performance of E-RBSA-ED in terms of retrieval runtime percentage and cell cost for various query sizes and various δ values on Human Chromosome 22. E-RBSA-ED is not significantly affected by the query size as regards its retrieval runtime percentage. We also note that larger query sizes lead to smaller cell cost. This behavior is expected since the longer the query size the more segments will be used for pruning; thus the pruning power increases. With respect to δ, it is clear that, as the similarity of the query to the database increases, E-RBSA-ED improves both in terms of retrieval runtime percentage and cell cost. Table 5.2 summarizes the results.

Effect of Alphabet Collapsing

Table 5.3 shows how alphabet collapsing affects the performance of E-RBSA-ED and E-RBSA-SW. From the experiments we can see that applying alphabet collapsing can improve the performance of E-RBSA in most cases by factors of 1.3 and 1.55 or more, in terms of retrieval runtime percentage and cell cost respectively.

Cell cost of E-RBSA-ED vs. Q-grams				
Method	$\|Q\|$	δ=5%	δ=10%	δ=15%
Q-grams	20	2.1% (q=9)	8.2% (q=6)	28.4% (q=4)
RBSA	40	0.55%	1.02%	1.47%
Q-grams	40	3.2% (q=10)	9.3% (q=7)	31.9% (q=5)
Q-grams	100	15.3% (q=15)	27.4% (q=8)	58.8% (q=6)
RBSA	200	0.32%	0.89%	1.22%
Q-grams	200	32.9% (q=17)	45.5% (q=9)	73.7% (q=6)

Table 5.1: For E-RBSA-ED, alphabet collapsing has not been applied. For q-grams, the best q value for each case is shown. Notice that the database used in this experiment contains the first $184{,}309$ nucleotides of Human Chromosome 22, i.e. $|X| = 184{,}309$.

RRP of E-RBSA-ED				
δ	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
15%	3.49%	3.50%	3.52%	3.56%
10%	0.89%	0.91%	0.91%	0.94%
5%	0.27%	0.28%	0.28%	0.29%

Cell cost of E-RBSA-ED				
δ	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
15%	1.12%	1.01%	0.87%	0.76%
10%	0.11%	0.10%	0.088%	0.077%
5%	0.01%	0.011%	0.009%	0.008%

Table 5.2: The number of reference objects used at the filter step is 50.

RBSA-SW: Comparison with BLAST and BWT-SW

For the remaining part of our experimental analysis, we focus on the performance of RBSA on local alignment, i.e., when the Smith-Waterman similarity measure is used at the refine step. The performance of RBSA-SW is compared against two state-of-the-art local alignment methods, BLAST and BWT-SW, for various query sizes and δ values. We also show the significant improvement on both retrieval runtime percentage and cell cost for the approximate version of RBSA (i.e. A-RBSA-SW). For the following experiments, alphabet collapsing has been applied. Notice that A-RBSA-SW has not been studied for query sizes 40 and 200 since the number of possible chunks in both cases is extremely small to guarantee a high retrieval accuracy. Our findings are summarized in Tables 5.4 and 5.5. For clarity purposes, the same results are also shown in Figure 5·1. It can be seen that A-RBSA outperforms BLAST by more than an order of magnitude

RRP of E-RBSA-ED with Alphabet Collapsing				
RBSA	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
Coll.	2.342%	2.386%	2.400%	2.473%
Uncoll.	3.49%	3.50%	3.52%	3.56%

Cell cost of E-RBSA-ED with Alphabet Collapsing				
RBSA	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
Coll.	0.735%	0.663%	0.571%	0.499%
Uncoll.	1.12%	1.01%	0.87%	0.76%

RRP of E-RBSA-SW with Alphabet Collapsing				
RBSA	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
Coll.	2.630%	2.679%	2.695%	2.777%
Uncoll.	3.579%	3.638%	3.660%	3.754%

Cell cost of E-RBSA-SW with Alphabet Collapsing				
RBSA	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
Coll.	0.826%	0.745%	0.641%	0.560%
Uncoll.	1.358%	1.224%	1.055%	0.921%

Table 5.3: The first column describes whether alphabet collapsing has been used ($Coll.$) or not ($Uncoll.$). The number of reference objects used at the filter step is 50.

for large queries ($2,000$ and $10,000$). The retrieval accuracy of A-RBSA is $\geq 99.5\%$ for all the experiments described in this section. For $\delta = 15\%$ and 10%, A-RBSA has a retrieval accuracy of 99.5% when $|Q| = 2,000$, and 100% when $|Q| = 10,000$. For $\delta = 5\%$, A-RBSA achieves 100% accuracy for both query sizes. As regards BWT-SW, in terms of retrieval runtime percentage it outperforms BLAST and E-RBSA by over an order of magnitude for $|Q| = 40$ and is up to almost 3 times faster than BLAST for $|Q| = 200$. Its performance deteriorates, however, as $|Q|$ becomes larger.

RBSA-SW: Effect of the Number of Reference Objects used for Filtering

In the experiments we have seen so far, it is assume that 50 reference objects are assigned to each database position. In this section, we show the effect of the number of reference objects assigned per database point on the performance of RBSA-SW. We experiment on two query sizes, 200 and $2,000$ with $\delta = 10\%$. Also for these experiments alphabet collapsing has been applied. Table 5.6 summarizes our findings with respect to retrieval runtime percentage and cell cost. Clearly as the

RRP of RBSA-SW vs. BWT-SW and BLAST for $\delta = 15\%$				
Method	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
A-RBSA			0.476%	0.086%
E-RBSA	2.630%	2.679%	2.695%	2.777%
BWT-SW	0.34%	3.30%	8.63%	12.72%
BLAST95	11.17%	7.57%	7.46%	7.84%
BLAST98	16.34%	7.88%	7.60%	8.11%
BLAST100	19.35%	9.29%	8.20%	9.66%

RRP of RBSA-SW vs. BWT-SW and BLAST for $\delta = 10\%$				
Method	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
A-RBSA			0.087%	0.018%
E-RBSA	0.481%	0.490%	0.493%	0.508%
BWT-SW	0.204%	2.600%	6.889%	8.900%
BLAST95	4.623%	3.133%	3.086%	3.243%
BLAST98	6.783%	3.271%	3.155%	3.362%
BLAST100	8.251%	3.965%	3.498%	4.118%

RRP of RBSA-SW vs. BWT-SW and BLAST for $\delta = 5\%$				
Method	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
A-RBSA			0.019%	0.0053%
E-RBSA	0.106%	0.108%	0.109%	0.112%
BWT-SW	0.083%	0.688%	2.170%	5.460%
BLAST95	4.293%	2.910%	2.866%	3.011%
BLAST98	6.231%	3.005%	2.898%	3.089%
BLAST100	7.437%	3.573%	3.153%	3.711%

Table 5.4: The number of reference objects used at the filter step is 50. Results are shown for $\delta = 15\%, 10\%$, and 5%.

number of reference objects decreases, both retrieval runtime percentage and cell cost deteriorate. In particular, if less that 30 reference objects are used, RBSA-SW outperforms the brute-force Smith-Waterman by a factor smaller than 3.5, and for 10 reference objects this factor is less than 2.

RBSA-SW: Experiment on Queries with Various δ Values

Finally we created a set of queries where δ varies from 1% to 15% in increments of 2%. Two query sizes have been studied, 200 and $2,000$. We have created one query set per query size using different δ values. The total number of queries in each set is 400. Also, 50 reference objects have been used at the filter step. Results on retrieval runtime percentage and cell cost are summarized

Cell cost of RBSA-SW vs. BWT-SW and BLAST for $\delta = 15\%$				
Method	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
A-RBSA			0.126%	0.024%
E-RBSA	0.826%	0.745%	0.641%	0.560%
BWT-SW	0.017%	1.298%	6.107%	7.347%
BLAST95	6.032%	3.972%	3.751%	4.641%
BLAST98	8.98%	4.73%	4.55%	5.56%
BLAST100	9.35%	5.87%	5.44%	6.6%

Cell cost of RBSA-SW vs. BWT-SW and BLAST for $\delta = 10\%$				
Method	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
A-RBSA			0.016	0.003%
E-RBSA	0.103%	0.093%	0.080%	0.070%
BWT-SW	0.015%	1.166%	5.483%	6.596%
BLAST95	4.974%	3.175%	2.793%	3.127%
BLAST98	7.917%	4.170%	4.011%	4.902%
BLAST100	9.862%	4.936%	4.574%	5.550%

Cell cost of RBSA-SW vs. BWT-SW and BLAST for $\delta = 5\%$				
Method	$\|Q\|$=40	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=10,000
A-RBSA			0.001%	0.0002%
E-RBSA	0.010%	0.009%	0.008%	0.007%
BWT-SW	0.012%	0.911%	4.285%	5.155%
BLAST95	4.428%	2.397%	1.800%	2.512%
BLAST98	5.998%	3.216%	2.242%	3.123%
BLAST100	6.150%	4.583%	3.278%	3.479%

Table 5.5: The number of reference objects used at the filter step is 50. Results are shown for $\delta = 15\%, 10\%$, and 5%.

in Table 5.7. For the set of queries with size $2,000$ we show the approximate version of RBSA. At the refine step we have used the Smith-Waterman similarity measure. For both query sizes, RBSA is at least one order of magnitude faster than BLAST and BWT-SW. The retrieval accuracy of A-RBSA is 99.75%.

To summarize our findings, A-RBSA can support relatively large queries without significant loss in retrieval accuracy and outperforms current state-of-the-art local alignment methods (BLAST and BWT-SW) by over an order of magnitude in terms of retrieval runtime percentage. For completeness we should mention that the average retrieval runtime for the brute-force local alignment computation for queries of size 40, 200, $2,000$ and $10,000$ is 28.5, 132.4, 1317.8 and 6620.1 seconds respectively.

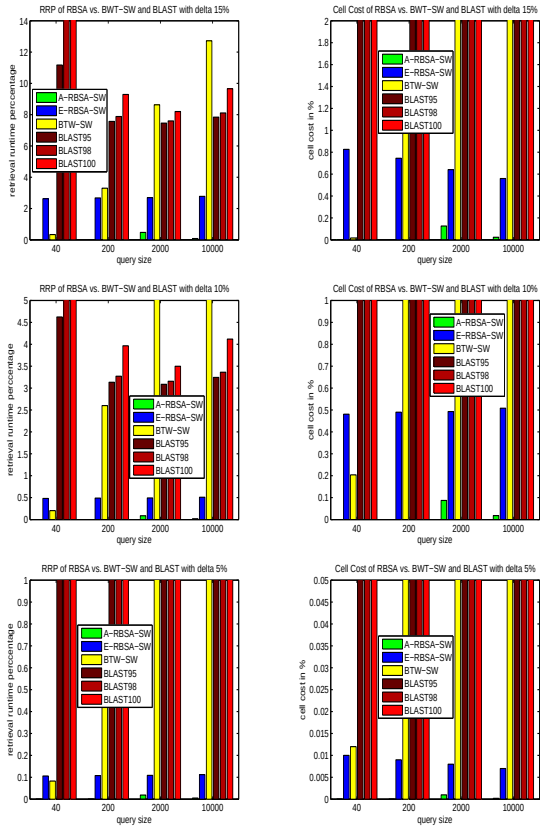

Figure 5·1: The number of reference objects used at the filter step is 50. Also, $\delta = 15\%$ on top row, $\delta = 10\%$ on middle row, and $\delta = 5\%$ on bottom row. Notice that A-RBSA has only been applied for query sizes of $2,000$ and $10,000$ and for the latter it can be barely seen due to its low cost.

5.4.3 Replication of RBSA on other datasets

Suppose that an individual wants to use RBSA for a given DNA dataset. RBSA should be tuned accordingly so as to provide best performance in terms of retrieval runtime and accuracy.

For the offline phase, as opposed to EBSM and BSE, there is no need for the user to provide any sample queries. This is due to the small DNA alphabet size (four bases). In fact, in the experiments described in this section, all reference sequences and queries used for the reference sequence selection process have been randomly generated.

Thus, given a new DNA database, we can generate a set of random queries and follow the reference sequence selection described in Algorithm 5.1 for a fixed reference sequence size (this

RRP and cell cost of RBSA-SW varying # of references				
	RRP		Cell Cost	
# of references	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=200	$\|Q\|$=2,000
50	0.490%	0.493%	0.093%	0.080%
40	1.143%	1.149%	0.217%	0.187%
30	7.873%	7.920%	1.498%	1.290%
20	28.440%	28.609%	5.411%	4.661%
10	64.743%	65.126%	9.012%	8.931%

Table 5.6: RRP and cell cost of E-RBSA-SW (exact RBSA using Smith-Waterman at the refine step) varying the number of reference objects assigned to each database point.

RRP and cell cost of RBSA-SW vs. BWT-SW and BLAST				
	RRP		Cell Cost	
Method	$\|Q\|$=200	$\|Q\|$=2,000	$\|Q\|$=200	$\|Q\|$=2,000
RBSA	0.530%	0.098%	0.088%	0.018%
BWT-SW	1.370%	2.958%	0.873%	4.233%
BLAST95	2.727%	2.406%	2.651%	2.640%
BLAST98	2.575%	2.483%	3.823%	3.815%
BLAST100	3.927%	3.304%	4.431%	4.454%

Table 5.7: For query size $2,000$ we have used A-RBSA (the approximate version of RBSA using Smith-Waterman at the refine step). The number of reference objects used at the filter step is 50.

can be initially set to 50) and fixed $K = 50$ (number of reference sequences for each database position). Notice that this reference selection process requires a significant amount of time (for the experimental settings described in this section, the time needed to assign reference sequences to one database position is approximately 81 seconds. Parameter δ (target dissimilarity percentage) should be given by the user. The above process should be repeated for different reference sequence sizes and different K's and determine those values with the best pruning power for a given dataset.

5.5 Summary

The proposed RBSA method uses precomputed alignment scores between reference sequences and database positions in order to efficiently identify, given a query Q, a relatively small number of candidate subsequence matches in the database. RBSA has an exact version, that is guaranteed to find the correct subsequence match, as long as that subsequence match has edit distance of at

most $\delta|Q|$ to the query. In our experiments, for query sizes ≥ 200, the exact version of RBSA outperforms state-of-the-art competitors such as BLAST, BWT, and q-grams.

Furthermore, an approximate version of RBSA has been developed that, for large queries, can efficiently identify candidate matches by considering only a relatively small number of fixed-size segments of the query. We show that, under some pretty realistic assumptions, the probability of failing to retrieve the correct match for approximate RBSA drops exponentially with the number of query segment that we consider. It is important to note that the number of query segments needed to guarantee a certain probability of success is independent of the actual length of the query, which makes the approximate version scale very well with large query lengths. The approximate version achieves significant speedups over the exact version of RBSA, and produces speedups of one to two orders of magnitude compared to the best results from existing competitors for $|Q| \geq 2,000$.

input : $\mathcal{Q}_{\text{sample}}$: a set of randomly generated queries.
$\mathcal{R}$: a set of reference objects.
$\{F^{R_i}(X, j)\}$: the embeddings of all positions (X, j) under all $R_i \in \mathcal{R}_j$.
X: database sequence.
δ: target dissimilarity percentage.
K: number of reference objects to be returned for each database position.

output : $\{\mathcal{R}_j^K\}$: for each database position (X, j), the set $\mathcal{R}_j^K$ of K reference objects to use for that position.

```
for j = 1 to |X| do
    // initialize R_j^K to the empty set.
    R_j^K = {};
    // insert all queries into a list Q.
    Q = list(Q_sample);
    for r = 1 to K do
        // initialize pruned to zero.
        pruned = uchar[|R|] = 0;
        for each R_i ∈ R do
            for k = 1 to |Q| do
                // compute lower bound for the k_th query.
                if (lb_ED^{i,j}(Q) > qδ) then pruned[i]++;
            end
        end
        BestRef = null; BestPrune = −1;
        for i = 1 to |R| do
            if pruned[i] > BestPrune then
                BestPrune = pruned[i];
                BestRef = R_i;
            end
        end
        R_j^K = R_j^K ∪ {BestRef};
        // remove pruned queries from Q using QPrune
        Q = EliminatePruned(Q, j, BestRef);
    end
end
```

Algorithm 1. Selecting reference sequences per database position.

input : Q: query.
X: database sequence.
δ: target dissimilarity percentage.
$F^{\mathcal{R}}(Q) = \{F^{R_i}(Q)$: embeddings of query Q.
$\{\mathcal{R}_j^K\}$: the set of reference sequences selected for position (X, j).
$\{F^{R_j^i}(X, j)\}$: embedding of each database position (X, j) under each reference object $R_j^i \in \mathcal{R}_j^K$.

output : $\mathrm{candidates}$: database positions to be passed to the refine step.

```
// insert all database positions into list candidates.
candidates = {1, ..., |X|};
// define lower bound cut-off threshold.
threshold = qδ;
for i = 1 to K do
    for j = 1 to |X| do
        x = F^{R_j^i}(X, j) − F^{R_j^i}(Q);
        if x > threshold then
            candidates = candidates − {j};
        end
    end
end
```

Algorithm 2. Filtering with maximum pruning.

input : Q: query.
X: database sequence.
δ: target dissimilarity percentage.
sorted: an array of candidate endpoints j, sorted in decreasing order of j.

output : $(X, j_{\text{start}}), (X, j_{\text{end}})$: start and end point of estimated best alignment.
distance: distance between Q and estimated best alignment.
columns: number of database positions evaluated by the edit distance DP.

for $i = 1$ **to** $|X|$ **do**
  unchecked$[i] = 0$;
end
for $i = 1$ **to** $|\text{sorted}|$ **do**
  unchecked[sorted$[i]] = 1$;
end
distance $= \delta \times |Q| + 1$; columns $= 0$; $n = |\text{sorted}|$;

for $k = 1$ **to** n **do**
  candidate $=$ sorted$[k]$;
  if (unchecked[candidate] $== 0$) **then continue**;
  $j =$ candidate + 1;
  for $i = |Q| + 1$ **to** 1 **do**
    cost$[i][j] = \infty$;
  end
  while *(***true***)* **do**
    $j = j - 1$;
    if (candidate $- j \geq |Q|\delta + 1$) **then break**;
    if *(*unchecked$[j] == 1$*)* **then**
      unchecked$[j] = 0$; candidate $= j$;
      cost$[|Q| + 1][j] = 0$; endpoint$[j + 1] = j$;
    else
      cost$[|Q| + 1][j] = \infty$; // j is not a candidate endpoint.
    end
    for $i = |Q|$ **to** 1 **do**
      previous $= \{(i + 1, j), (i, j + 1), (i + 1, j + 1)\}$;
      $(p_i, p_j) = \text{argmin}_{(a,b)\in\text{previous}}\text{cost}[a][b]$;
      cost$[i][j] = D(Q_i, X_j) +$ cost$[p_i][p_j]$; endpoint$[i][j] =$ endpoint$[p_i][p_j]$;
    end
    columns $=$ columns $+ 1$;
  end
end

Algorithm 3. The refine step for the edit distance.

Chapter 6

Conclusions and Future Work

In this final chapter of the thesis we summarize the main lessons learned from the described work, and point out open questions and interesting directions for future research.

6.1 Discussion of Contributions and Limitations

The proposed methods in this thesis are the first subsequence matching methods for unconstrained DTW, cDTW and edit distance that convert, 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 dynamic programming algorithm is still employed within EBSM, BSE and RBSA, 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 datasets used in our experiments.

A major limitation of both EBSM and BSE is the fact that both unconstrained DTW and cDTW are non-metric. This is one of the barriers that makes both methods approximate. Providing theoretical guarantees for both systems is a great challenge and we are planning to investigate it in the future. Another limitation of EBSM is that its performance depends on the training phase where the embedding index is constructed. This means that prior knowledge of the types of the expected queries is needed in order to achieve better performance. As opposed to EBSM and BSE, RBSA uses a metric distance measure (edit distance) to define the embedding index and thus it can provide 100% guarantees of finding the correct match. An important limitation of RBSA is the costly training phase, which has a much larger time requirement than that of the training phase

of EBSM. Another limitation is that it is designed for near-exact homology search, for which it manages to outperform standard methods by orders of magnitude. According to the experiments, as δ increases the performance of RBSA deteriorates.

6.2 Future Work and Other Interesting Directions

The topic of using embeddings for efficient subsequence matching is relatively new and there are many open problems in both time series and biological sequence databases.

6.2.1 Time Series Matching

One open problem is to adopt the idea of BSE to EBSM and combine startpoint and endpoint embeddings under the unconstrained DTW, where the subsequence match can have a different length than the query. Another related problem is to remove the constraint that the match must have the same length as the query for cDTW. This constraint is currently used not only by our method, but also by the other existing methods for subsequence matching under cDTW, i.e., LB_Keogh [34] and DTK [56].

Another interesting direction for future work for both EBSM and BSE is to compress the size of the embedding. One way of doing this, is to view each embedding dimension as a time series and apply standard time series segmentation techniques [81]. Each segment will be represented by an upper and a lower value. This type of compression may reduce the filter step, as it will not be required to perform Euclidean distance computations on the whole embedding.

Another open problem is using vector indexing methods to further speed up the embedding-based filter step for both EBSM and BSE. An additional challenge here is that BSE embeddings are query-sensitive: the reference sequences used depend on the query length, and the final combination of startpoint and endpoint embeddings also depends on the query length. Applying standard vector indexing methods [7, 25] in this setting is not a straightforward task, and developing appropriate indexing methods is an interesting topic for future work. An interesting work that can be used to improve similarity search over arbitrary subspaces under an L_p norm distance appeared recently [50] and is related to this problem.

Furthermore, it should be clear by the discussion in chapter 4 that LB_Keogh and similar bounds for cDTW cannot be applied directly for unconstrained subsequence matching as they require the match length to be equal to the query length (which is not the case in unconstrained subsequence matching). It would be challenging to define distance measures for unconstrained subsequence matching that are metric and/or allow the definition of tight bounds that can be used to efficiently speedup the filter step.

Finally, as mentioned earlier, a very challenging direction for future work would be to deeper investigate the non-metric property of both DTW and cDTW, and try to provide theoretical guarantees for both EBSM and BSE.

6.2.2 Sequence Alignment

An interesting direction for future work on RBSA is to compress the size of the embedding. As opposed to EBSM and BSE, the challenge here is the fact the each database point is represented by a set of reference sequences. One approach is to break the database sequence into small segments using a standard DNA segmentation technique [49] and then represent each segment with the K references with the best pruning power in that segment.

Another interesting topic for future work is the fact that DNA is not random, as it consists of *coding regions* (mainly genes) and *non-coding regions* (regions that do not contain any useful information). RBSA should be able to handle those regions effectively by not allowing reference sequences to be selected from non-coding regions. We should also note that there exist long-range dependencies in DNA sequences and it would be very interesting to see if such dependencies could be used to improve the performance of RBSA.

Another open question regarding RBSA, is whether it could be extended so that it does not require matches to be within edit distance $\delta|Q|$ from the query. Also, the reference sequence selection of RBSA is a costly step. An interesting future direction is to investigate other alternative ways of reference sequence selection with a lower cost. It will also be interesting to study more extensively the effects of letter collapsing, and to analyze theoretically the reasons that letter collapsing improves performance. Also, we believe that letter collapsing may turn out to lead to

more significant improvements in domains with larger alphabet sizes, such as proteins. We aim to explore these issues in future work.

Finally, the experimental evaluation of RBSA showed very promising results in terms of retrieval runtime and accuracy. However, we have only focused on performance in terms of database dissimilarity and database size, and did not further study the impact that RBSA might have in Biology. Indeed it would be challenging to see if this significant performance is also meaningful in the biological domain and can lead to the development of a widely used tool for homology search in DNA and protein sequences. Thus, one of our future plans is to study the potential impact of RBSA in Biology.

A very difficult and challenging problem is to provide algorithms for subsequence matching with provable sublinear retrieval running time. Developing index structures for non-Euclidean and non-metric spaces that allow approximate nearest neighbor retrieval in time sublinear to the database size will enable many important applications, like fast recognition and similarity-based matching in large databases of DNA and protein sequences, financial data, fingerprints, speech and audio data.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] S. F. Altschul, T. L. Madden, R. A. Schffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res*, 25:3389–3402, 1997.

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

[4] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. BoostMap: A method for efficient approximate similarity rankings. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 268–275, 2004.

[5] 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.

[6] V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios, and D. Gunopulos. Approximate embedding based subsequence matching of time series. In *ACM International Conference on Management of Data (SIGMOD)*, pages 365–378, 2008.

[7] 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.

[8] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[9] S. Burkhardt, A. Crauser, P. Ferragina, H-P. Lenhof, E. Rivals, and M. Vingron. Q-gram based database searching using a suffix array (quasar). In *International Conference on Computational Molecular Biology (RECOMB)*, pages 77–83, 1999.

[10] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundam. Inf.*, 56(1,2):51–70, 2002.

[11] C. Cao, L. C. Shuai, and A. K. H. Tung. Indexing DNA sequences using q-grams. *Database Systems for Advanced Applications*, 3453:4–16, 2005.

[12] X. Cao, B. C. Ooi, A. K. H. Tung, H. H. P., and K. L. Tan. DSIM: A distance-based indexing method for genomic sequences. In *IEEE Symposium on Bioinformatics and Bioengineering (BIBE)*, pages 97– 104, 2005.

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

[14] K. P. Chan and A. W. C. Fu. Efficient time series matching by wavelets. In *IEEE International Conference on Data Engineearing (ICDE)*, pages 126–133, 1999.

[15] W. I. Chang and E. L. Lawler. Sublinear expected time approximate string matching and biological applications. Technical Report CSD-91-654, University of California at Berkeley, 1991.

[16] M. Crochemore and T. Lecroq. Pattern matching and text compression algorithms. 28:39–41, 1996.

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

[18] 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.

[19] 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.

[20] A. W. C. Fu, E. Keogh, L. Y. H. Lau, C. Ratanamahatana, and R. C.-W. Wong. Scaling and time warping in time series querying. *The Very Large DataBases (VLDB) Journal*, 17(4):899–921, 2008.

[21] E. Giladi, M. G. Walker, J. Z. Wang, and W. Volkmuth. SST: an algorithm for finding near-exact sequence matches in time proportional to the logarithm of the database size. *Bioinformatics*, 18(6):873–877, 2002.

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

[23] K. Hamza. The smallest uniform upper bound on the distance between the mean and the median of the binomial and poisson distributions. *Statistics and Probability Letters*, 23(1):21–25, 1995.

[24] 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.

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

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

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

[28] X. Huang and K.-M. Chao. A generalized global alignment algorithm. *Bioinformatics*, 19(2):228–233, 2003.

[29] C. V. Jongeneel. Searching the expressed sequence tag (est) databases: Panning for genes. *Bioinformatics*, 1:76–92, 2000.

[30] K. J. Kalafus, A. R. Jackson, and A. Milosavljevic. Pash: Efficient genome-scale sequence anchoring by positional hashing. *Genome Resources*, 14(4):672678, 2004.

[31] 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.

[32] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[33] W. J. Kent. BLAT–the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

[34] E. Keogh. Exact indexing of dynamic time warping. In *International Conference on Very Large Databases (VLDB)*, pages 406–417, 2002.

[35] E. Keogh. The UCR time series data mining archive. http://www.cs.ucr.edu/ eamonn/tsdma, 2006.

[36] E. Keogh and M. Pazzani. Scaling up dynamic time warping for data mining applications. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 506–510, 2000.

[37] M. S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-Gram/2L: A space and time efficient two-level n-gram inverted index structure. In *International Conference on Very Large Data Bases (VLDB)*, pages 325–336, 2005.

[38] Y. J. Kim, A. Boyd, B. D. Athey, and J. M. Patel. miBLAST: scalable evaluation of a batch of nucleotide sequence queries with blast. *Nucleic Acids Res*, 33:4335–4344, 2005.

[39] D. E. Knuth. The art of computer programming. *Sorting and Searching*, 3, 1973.

[40] I. Korf and W. Gish. MPBLAST : improved BLAST performance with multiplexed queries. *Bioinformatics*, 16:1052–1053, 2000.

[41] 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 (ICDE)*, pages 6–17, 2004.

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

[43] T. W. Lam, W. K. Sung, S. L. Tam, C.K. Wong, and S.M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6), 2008.

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

[45] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics*, 10(8):707–710, 1966.

[46] 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 (TKDE)*, 14(4):792–808, 2002.

[47] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *IEEE International Conference on Data Engineering (ICDE)*, pages 257–266, 2008.

[48] C. Li, B. Wang, and X. Yang. VGRAM: improving performance of approximate queries on string collections using variable-length grams. In *International Conference on Very Large Data Bases (VLDB)*, pages 303–314, 2007.

[49] W. Li, P. Bernaola-Galvan, H. Fatameh, and I. Grosse. Applications of recursive segmentation to the analysis of DNA sequences. *Computes and Chemistry*, 26(2):491–510, 2002.

[50] X. Lian and L. Chen. Similarity search in arbitrary subspaces under lp-norm. In *IEEE International Conference on Data Engineearing (ICDE)*, pages 317–326, 2008.

[51] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, March 1985.

[52] W. Litwin, R. Mokadem, P. Rigaux, and T. Schwarz. Fast nGram-based string search over data encoded using algebraic signatures. In *International Conference on Very Large Data Bases (VLDB)*, pages 207–218, 2007.

[53] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *ACM SIAM Symposium On Discrete Algorithms (SODA)*, pages 319–327, 1990.

[54] C. Meek, J. M. Patel, and S. Kasetty. OASIS: An online and accurate technique for local-alignment searches on biological sequences. In *International Conference on Very Large Data Bases (VLDB)*, pages 910–921, 2003.

[55] Y. S. Moon, K. Y. Whang, and W. S. 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.

[56] Y. S. Moon, K. Y. Whang, and W. S. Han. Ranked subsequence matching in time-series databases. In *International Conference on Very Large Data Bases (VLDB)*, pages 423–434, 2007.

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

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

[59] G. Myers. Whole-genome DNA sequencing. *Computing in Science and Engg.*, pages 33–43, 1999.

[60] C. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Symposium on Combinatorial Pattern Matching (CPM)*, pages 350–363, 2000.

[61] G. Navarro and R. Baeza-yates. A new indexing method for approximate string matching. In *Symposium on Combinatorial Pattern Matching (CPM)*, pages 163–185, 1999.

[62] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal on Molecular Biology*, 48(3):443–53, 1970.

[63] Z. Ning, A. J. Cox, and J. C. Mullikin. SSAHA: A fast search method for large DNA databases. *Genome Resources*, 11(10):1725–1729, 2001.

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

[65] P. Papapetrou, V. Athitsos, G. Kollios, and D. Gunopulos. Reference-based alignment of large sequence databases. In *International Conference on Very Large Data Bases (VLDB)*, 2009 (To Appear).

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

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

[68] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the Natural Academy of Sciences (PNAS) U S A*, 85(8):2444–2448, April 1988.

[69] B. Phoophakdee and M. J. Zaki. TRELLIS+: an effective approach for indexing genome-scale sequences using suffix trees. In *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, pages 90–101, 2008.

[70] D. E. Knuth J. Pratt and R. Vaughan. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[71] 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.

[72] C. Ratanamahatana and E. J. Keogh. Three myths about dynamic time warping data mining. In *SIAM International Data Mining Conference (SDM)*, pages 506–510, 2005.

[73] 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.

[74] G. J. Russell and J. H. Subak-Sharpe. Similarity of the general designs of protochordates and invertebrates. *Nature*, 266:533–536, 1977.

[75] Y. Sakurai, C. Faloutsos, and M. Yamamuro. Stream monitoring under the time warping distance. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1046–1055, 2007.

[76] 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.

[77] 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 Databases (VLDB)*, pages 516–526, 2000.

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

[79] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[80] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 563–576, 2009.

[81] E. Terzi and P. Tsaparas. Efficient algorithms for sequence segmentation. In *Proceedings of the Sixth SIAM International Conference on Data Mining*, pages 316–327, 2006.

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

[83] E. Ukkonen. Algorithms for approximate string matching. *Information Control*, 64(1-3):100–118, 1985.

[84] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[85] 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.

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

[87] 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.

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

[89] 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 Databases (VLDB)*, pages 194–205, 1998.

[90] 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.

[91] 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.

[92] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *ACM International Conference on Management Of Data (SIGMOD)*, pages 353–364, 2008.

[93] 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 (ICDE)*, pages 201–208, 1998.

[94] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7:203–214, 2000.

[95] 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.

Curriculum Vitae

Panagiotis Papapetrou

Address:
14 Buswell Street, Apt 416
Boston, MA 02215

Email: panagpap@cs.bu.edu
Phone: 617-230-1050
Web page: http://cs-people.bu.edu/panagpap

Research Interests

Data Mining, Databases, Time Series Analysis, Bioinformatics, Multimedia Indexing, Machine Learning, Motion Mining, Sequential and Temporal Pattern Mining, Computational Biology.

Current Position

PhD Candidate
Boston University, Computer Science Department
Boston, MA
Dates: January 2004-present

Education

January 2004 - July 2009 Ph.D. Student, Computer Science Department, Boston University. Advisor: Prof. George Kollios, Co-Advisor: Prof. Vassilis Athitsos.

January 2004 - January 2007 M.A. in Computer Science, Computer Science Department, Boston University. Advisor: Prof. George Kollios.

September 1999 - June 2003 B.Sc. in Computer Science, Department of Computer Science, University of Ioannina, Greece. Advisor: Prof. Dimitrios Fotiadis.

Publications

Journals

1. Panagiotis Papapetrou, George Kollios, Stan Sclaroff, and Dimitrios Gunopulos, *Mining Frequent Arrangements of Temporal Intervals*, to appear in Knowledge and Information Systems (KAIS), 2009.

Conferences

1. Panagiotis Papapetrou, Vassilis Athitsos, George Kollios, and Dimitrios Gunopulos, *Reference-Based Alignment in Large Sequence Databases*, to appear in Very Large DataBases (VLDB), August 2009, Lyon, France.

2. Panagiotis Papapetrou, Paul Doliotis and Vassilis Athitsos, *Towards Faster Activity Search Using Embedding-based Subsequence Matching*, in proceedings of the PETRA Workshop on Multimedia Event Analysis for Assistive Environments (EventAnalysis), June 2009, Corfu, Greece.

3. Vassilis Athitsos, Panagiotis Papapetrou, Michalis Potamias, George Kollios, and Dimitrios Gunopulos, *Approximate Embedding-Based Subsequence Matching of Time Series*, in proceedings of ACM SIGMOD, June 2008, Vancouver, Canada.

4. Vassilis Athitsos, Michalis Potamias, Panagiotis Papapetrou, and George Kollios, *Nearest Neighbor Retrieval Using Distance-Based Hashing*, in proceedings of IEEE ICDE, April 2008, Cancun, Mexico.

5. Panagiotis Papapetrou, Gary Benson, and George Kollios, *Discovering Frequent Poly-Regions in DNA Sequences*, in proceedings of the IEEE ICDM Workshop on Data Mining in Bioinformatics (DMB), December 2006, Hong Kong.

6. Panagiotis Papapetrou, George Kollios, Stan Sclaroff, and Dimitrios Gunopulos, *Discovering Frequent Arrangements of Temporal Intervals*, in proceedings of IEEE ICDM, November 2005, Houston, Texas.

Technical Reports

1. Panagiotis Papapetrou, Gary Benson, and George Kollios, *Generalized Methods for Discovering Frequent Poly-Regions in DNA Sequences*, Department of Computer Science, Boston University, October 2008.

2. Ching Chang, Raymond Sweha, Panagiotis Papapetrou, *Extending snBench to Support a Graphical Programming Interface for a Sensor Network Tasking Language (STEP)*, Department of Computer Science, Boston University, July 2006.

Research Experience

- Computer Science Department, Boston University, Boston, MA. January 2004-present. Research Assistant for Prof. George Kollios

  – **Motion Mining** Project: Development and testing of methods for indexing, retrieval, and data mining of human motion trajectories in video databases. Implementation on ASL.

  – **Subsequence Matching** Project: Development and testing of methods for indexing and retrieval of time series and categorical sequence databases. Implementation on real time series data and DNA sequences.

Teaching Experience

- Computer Science Department, Boston University, Summer 2008.
  Instructor for the undergraduate course on introduction to computer science; responsible for teaching.
- Computer Science Department, Boston University, Spring 2004, Summer/Fall/Spring 2007, Spring/Fall 2008.
  Teaching assistant for the undergraduate course on introduction to computer science; responsible for teaching, grading and tutorials.
- Computer Science Department, Boston University, Fall 2005, Spring 2006.
  Teaching assistant for the undergraduate course on introduction to web computing; responsible for teaching, grading and tutorials.
- Computer Science Department, Boston University, Summer 2007.
  Teaching assistant for the undergraduate course on analysis of algorithms; responsible for teaching, grading and tutorials.
- Computer Science Department, Boston University, Spring 2007, Spring 2008.
  Teaching assistant for the graduate course on database systems; responsible for teaching, grading and tutorials.
- Computer Science Department, Boston University, Fall 2006.
  Teaching assistant for the undergraduate course on software engineering; responsible for teaching, grading and tutorials.

Professional Activities

- Student Member of IEEE.
- External reviewer for the following conferences: VLDB (2009), CIKM (2006, 2009), ICDM (2005, 2006), KDD (2006, 2008), DASFAA (2004).
- Reviewer for journals: TKDE.

Qualifications

- Programming skills: Matlab, C/C++, PHP and Web development tools. Familiar with WML, XML, Python, OpenGL, SQL and Visual Basic.
- Operating systems: Windows, Unix.
- Language skills: Greek (Native), English (Excellent), German (Very good), Spanish (Elementary).
- Other skills: Theory of Music (Excellent), Church Organ (Very Good).

References

Available upon request.