2010 IEEE International Conference on Data Mining

Fast and Flexible Multivariate Time Series Subsequence Search

Kanishka Bhaduri
MCT Inc., NASA ARC
Kanishka.Bhaduri-1@nasa.gov

Qiang Zhu
CSE Dept, UCR
qzhu@cs.ucr.edu

Nikunj C. Oza, Ashok N. Srivastava
NASA Ames Research Center
{Nikunj.C.Oza, Ashok.N.Srivastava}@nasa.gov

***Abstract*—Multivariate Time-Series (MTS) are ubiquitous, and are generated in areas as disparate as sensor recordings in aerospace systems, music and video streams, medical monitoring, and financial systems. Domain experts are often interested in searching for *interesting* multivariate patterns from these MTS databases which can contain up to several gigabytes of data. Surprisingly, research on MTS search is very limited. Most existing work only supports queries with the same length of data, or queries on a fixed set of variables. In this paper, we propose an efficient and flexible subsequence search framework for massive MTS databases, that, for the first time, enables querying on any subset of variables with arbitrary time delays between them. We propose two provably correct algorithms to solve this problem — (1) an R^*-tree *B*ased *S*earch (*RBS*) which uses Minimum Bounding Rectangles (MBR) to organize the subsequences, and (2) a *L*ist *B*ased *S*earch (*LBS*) algorithm which uses sorted lists for indexing. We demonstrate the performance of these algorithms using two large MTS databases from the aviation domain, each containing several millions of observations. Both these tests show that our algorithms have very high prune rates (>95%) thus needing actual disk access for only less than 5% of the observations. To the best of our knowledge, this is the first flexible MTS search algorithm capable of subsequence search on any subset of variables. Moreover, MTS subsequence search has never been attempted on datasets of the size we have used in this paper.**

I. Introduction

Many data mining application domains generate large multivariate time series (MTS) databases. Examples of such domains include earth sciences, music, video, medical monitoring, aerospace systems, and financial systems. Domain experts are often interested in searching for particular patterns—waveforms over subsets of variables with some delays between them.

The motivation for this research comes from applications in any domain where an entity can be described as a multivariate sequence and one needs to search for entities having specific characteristics defined by a particular combination of some or all of those features. Suppose that an airline has a large database of one million flights of multivariate time series that show the settings of the control surfaces (usually discrete signals), the pilot inputs (discrete), as well as the heading, speed, and readings from the propulsion systems (all usually continuous). In many such databases, the number of recorded parameters from a modern aircraft is nearly 1000. The safety analyst may want to find all situations in the database that correspond to a "go-around" situation in which a landing has been aborted and the aircraft has been directed to circle back for another landing.

One can find such situations using a subset of the fields in the time series database where the event "Landing Gear Retracted" occurs just after altitude descends below 2000 feet. Another search for indicators of an "unstable approach" may include searching on parameters such as speed, descent rate, vertical flight path, and several cockpit configuration parameters. Again, this search would be done on about a dozen parameters out of the 1000 parameters that may be recorded on the aircraft. The events would be separated in time and may or may not occur on a particular flight.

Fig. 1 shows an MTS from a real aviation dataset of CarrierX [1]. Each MTS contains the data collected from multiple sensors of an aircraft during a flight. We plot only six variables for clarity. In the figure, the x-axis refers to the different parameters while the y-axis refers to time. Typically, queries by the analyst may look like:

1. Return all flights where the altitude monotonically changes from 10000 ft to 5000 ft, speed decreases from 300 knots to 200 knots, and landing gear is down. Such a combination of parameter values may be precursors to unstable approaches while landing.

2. Return all small-cap stocks whose daily price drops by 10% over 3 days just before a strong sell-off (30% over 10 days) in at least m out of K stocks and then increases by at least 15% over the remaining 30 days. This could be a signature indicative of insider-trading in an attempt to unfairly control the share prices in the specific sector.

None of the current research in MTS search [1][2][3][4] support the types of queries described here. Current algorithms in this area require that the query be of the same length as that of the entire MTS and that all queries be on a fixed set of variables (usually all the variables). Additionally, current algorithms do not allow for any time lag between the variables in the query.

In this paper we address the following problem: given a large database of multivariate time series data representing entities, we wish to provide a search technology that allows analysts to *rapidly* identify entities with particular character-

[1]We cannot release the name of the carrier due to the data sharing agreement.

1550-4786/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDM.2010.36

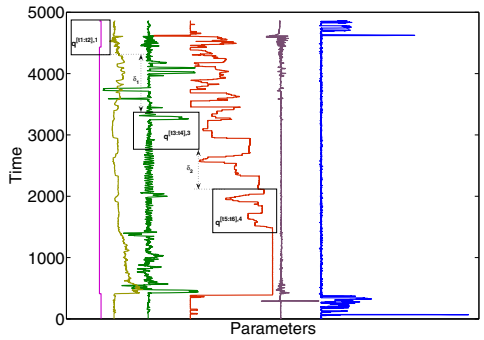

Figure 1. Sample MTS dataset and query Q. x-axis refers to different parameters and y-axis refers to time. Components of query and time delays are also shown.

istics such as the scenarios described above. We assume that the user supplies a query consisting of waveforms over several variables — typically substantially fewer than the total number of variables present in the database. Additionally, the user may choose (at search time) how many and which variables to query, *i.e.*, this need not be fixed in advance (during index-building time). This requires tremendous *flexibility* of the search algorithm. Also the query may cover any desired length of time up to the maximum length of the available time series. The waveforms may have some time-shifts between them. The user also supplies a threshold for each variable describing the maximum allowable difference between the query variable and the corresponding variable in any matches that are returned. The MTS search algorithm must return all matches with no false dismissals or false positives. The specific contributions of this paper are as follows:

(1) We propose two algorithms — an R^*-tree based search algorithm (RBS), and a list based search algorithm (LBS) for efficient searching of massive MTS subsequences defined on an arbitrary subset of variables with arbitrary time delays.
(2) We have demonstrated the usefulness of our algorithm by searching for this "go around" pattern in a real commercial aviation dataset.
(3) To the best of our knowledge, the datasets that we have used for testing the performance of our algorithms are much larger than those reported in the literature.

The rest of the paper is organized as follows. In Section II, we discuss work related to this area of research. In Section III, we describe the notation and give a precise definition of the MTS search problem. In Section IV we describe a fast UTS subsequence search algorithm leading to the MTS search algorithm in Section V. We analyze the algorithms in Section VI. In Section VII we demonstrate the performance of our algorithm experimentally. We provide conclusions and descriptions of future work in Section VIII.

II. Related work

In general, prior research on MTS is limited. Yang and Shahabi [1] present a PCA-based similarity technique for comparing two MTS. Given a database of MTS this technique first computes the covariance matrix between two MTS. Then eigenvectors and eigenvalues of the covariance matrix are used as a measure of similarity between the MTS. This work is extended in [5] in which the authors propose the use of kernel PCA instead of traditional PCA. Distance-based index structure for MTS has been discussed by Yang and Shahabi [6]. The work by Lee *et al.* [4] addresses the problem of searching in multi-dimensional sequences. The multi-dimensional sequence is partitioned into subsequences, packed into MBR and then indexed using the R^*-tree scheme. Vlachos *et al.* [3] proposes an index structure for multi-dimensional time series which can handle multiple distance functions such as LCSS and DTW.

There exist a plethora of work on subsequence search for univariate datasets (UTS). Popular techniques for performing entire length time series search include the ones proposed by Keogh and Ratanamahatana [7] and the references therein. One of the early works of subsequence matching is by Faloutsos *et al.* (FRM) [8] in which the authors have proposed a Discrete Fourier Transform (DFT)/R^*-tree based indexing scheme. In this algorithm, input time series is first broken into overlapping window sequences of fixed length and then 6 DFT coefficients are extracted from each sequence. These 6-dimensional representations are then packed into a minimum bounding rectangle (MBR) and indexed using an R^*-tree data structure. On receiving a query, the same process is applied (extracting DFT coefficients) and then searched in the R^*-tree. Candidate MBRs are then checked with the actual database to remove false alarms. We compare this algorithm with our algorithms in the experimental section. A *dual* approach to this one, proposed by Moon *et al.* [9], is to decompose the input time series into disjoint sequences and the query sequence into sliding windows. However, as the size of the time series increases to millions of points, storing all the points in the index may become challenging. To alleviate this problem, Traina *et al.* [10] recently proposed a technique of using multiple reference points to speed up the search. Our algorithm is different than theirs in the following sense: (1) [10] only talks about range queries whereas we can perform arbitrary subsequence matching and nearest neighbor search, and (2) unlike [10] which only works for univariate time series, we can perform multivariate subsequence search on an arbitrary number of variables and arbitrary time delays among those variables. Several other techniques exist for subsequence matching [11][12][13].

At this point, we would like to mention that none of the existing algorithms for multivariate search is applicable in our problem setting. This is primarily because most of them

require that all the variables be used for the query. In our problem, we query over an arbitrary subset of variables and thus, to apply the existing algorithms, we need build and store a separate index for all possible combinations of input features. For example, the real **CarrierX** dataset that we have used in our experiments has 16 variables, and therefore to allow any subset of variables in the query, we need to build and store $2^{16} = 65536$ indices which is impractical for storage and computational reasons. This motivates us to provide a different solution to this problem which alleviates these issues by building a much smaller number of indices (linear in the number of features).

III. Background

In this section we define the notations that we have used in the rest of this paper and also present a formal problem definition.

A. Notations

First, we define a UTS database. A UTS database U_DB consists of $|D|$ UTS. For ease of explanation, we assume that each UTS is stored in a separate file; multiple UTS can also be stored in the same file in other applications. The i-th file stores a time series $y^{(i)} = \{y_1^{(i)}, y_2^{(i)}, \dots\}$, where each $y_k^{(i)} \in \mathbb{R}$ or $\{0, 1\}$. The superscript refers to the file id while the subscript refers to the sample point in that file. Let $y^{(i)}$ and $y^{(j)}$ be two UTS sequences in two different files of U_DB. Then, (1) $L\left(y^{(i)}\right)$ denotes the length (number of points) of $y^{(i)}$, (2) $y^{(i)}_{[a:b]}$ denotes the subsequence that includes entries in positions a through b for UTS in the i-th file, and (3) $dist(y^{(i)[a:b]}, y^{(j)}_{[}a : b])$ denotes the Euclidean distance between two univariate subsequences.

It is natural to extend this definition to a multivariate database M_DB in which each file contains a set of vectors. Let d be the number of features or attributes across all the files in M_DB. Denoting vectors of dimension d in bold, we can similarly write the MTS stored in the i-th file as $\mathbf{y}^{(i)} = \{\mathbf{y}_1^{(i)}, \mathbf{y}_2^{(i)}, \dots\}$, where $\mathbf{y}_k^{(i)} \in \mathbb{R}^d$ or $\{0, 1\}^d$. Let w denote the size of a sliding window containing w consecutive samples of a UTS.

B. Problem definition

We first define ϵ nearest neighbors ϵ-NN of UTS.

Definition 3.1 (ϵ-NN UTS search): Given a user defined threshold ϵ, U_DB, and a UTS subsequence Q of length w, (which we call the query), UTS ϵ-NN returns all the subsequences S_i of length w from U_DB, such that, $dist(S_i, Q) < \epsilon$.

Our next definition deals with multivariate query Q.

Definition 3.2 (Multivariate Query Q): A multivariate query Q consists of the following components:

- any (sub)set of variables $Q.var \subset \{1, \dots, d\}$
- a set of UTS subsequences $\{Q.seq_i\}$ for each variable $i \in Q.var$, and
- time delays $\delta_1, \delta_2, \dots$ between the sequences in $Q.var$

We are now in a position to define ϵ-NN for MTS search.

Definition 3.3 (ϵ-NN MTS search): Given M_DB, a multivariate query Q, and user defined thresholds $\boldsymbol{\epsilon} = \{\epsilon_1, \epsilon_2, \dots\}$ for each variable in Q, MTS ϵ-NN returns a table $\{MTS\_i, Begin\_offset_1, Begin\_offset_2, \dots,\}$ such that (1) UTS ϵ-NN is satisfied by every feature in Q, (2) the subsequences are found in the same MTS file, and (3) the Begin_offset's are delayed by $\delta_1, \delta_2, \dots$ in which $Begin\_offset_j$ denotes the starting time point for Q.seq$_j$.

IV. Fast UTS subsequence search

When a query Q defined in Section III-B contains only one variable, it becomes a univariate time series search. For clarity and ease of exposition, we will start with solving this problem. We assume there is a minimal length for all queries and it is set to w. Smaller choice of w provides better granularity of search while increasing both the indexing and the search time. We first discuss the RBS algorithm in detail and then discuss the salient differences with our LBS algorithm.

A. Overview of algorithm

For a univariate query Q on the v-th variable, the brute-force method to find all its ϵ-NN is to compare it with all subsequences of length $L(Q)$ for every offset of time series $y^{(i)}$ $(\forall i = 1, 2, \dots, |D|)$, which is time consuming and impractical.

A classic data mining solution to speed up this process is to find a lower bound of the distance measure and use this bound to prune irrelevant candidates. This lower bound should be: (1) computationally more efficient than computing the distances between all subsequences, and (2) tight (very close) with respect to the original distance, so that we can prune sufficiently.

One such technique for deriving a lower bound, also used in the literature [10][14], is using a reference subsequence based on the triangle inequality. Fig. 2 illustrates the basic idea of pruning. First, we randomly pick a subsequence R (of the same length w), and calculate its distance to all the remaining subsequences. Then, we order them by their distance to R. Only S_1 and S_2 are shown for clarity in the figure. Note that these two steps are done before the query Q arrives and only need to be done once. When a query Q is applied, we calculate the distance $dist(Q, R)$. All candidates whose distances are not in the range $[dist(Q, R) - \epsilon, dist(Q, R) + \epsilon]$ (*e.g.* S_2 in Fig. 2) can be pruned. This is due to the triangle inequality:

$$dist(Q, S_2) \geq |dist(Q, R) - dist(S_2, R)| > \epsilon.$$

Finally, for all candidates in this range (*e.g.* S_1 in Fig. 2), we do an exact calculation to remove the false positives. In order to reduce the number of such false positives, we use multiple reference points to build several indices and then

join the candidates from these indices to get the final set of candidates. We discuss this in detail in the next section.

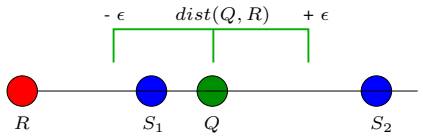

Figure 2. Candidate subsequences (S_1, S_2) ordered by their distance to a reference subsequence R. When a query Q is applied, a range based on $dist(Q, R)$ can be used to prune candidates.

B. RBS algorithm details

R^*-tree based algorithm (RBS) uses the concept of spatial indexing to store and retrieve time series subsequences. In order to make this indexing more efficient, we devise a novel technique of incorporating the triangular inequality directly into this R^*-tree scheme. We can control the amount of pruning and the corresponding search time by using multiple reference points against which the triangular inequality is applied. To the best of our knowledge, using spatial indexing along with multiple global reference points for time series subsequence search has never been explored before.

We first discuss the index building algorithm followed by the search algorithm. Alg. 1 presents the pseudo-code of RBS build index. The inputs are U_DB and length of the sliding window w. The output is a set of spatial indices $Index_1, \ldots, Index_r$. In the first step, we select r subsequences randomly $R_1, \ldots, R_r$ of size w from U_DB which we call *reference points*. Then, for each subsequence S of length w from the i-th UTS ($y^{(i)}$) in U_DB, we find the Euclidean distance of S from the k-th reference point R_k. Therefore, each subsequence of length w gets mapped to a 1-D point (its distance to R_k). Next, we arrange several such 1-D points into a minimum bounding rectangle or MBR as follows. Each entry of the MBR consists of the uts_id, min, max, $Begin\_Offset$, End_Offset, where min and max are the minimum and maximum values (here distances to R_k) of all points included in that MBR. $Begin\_Offset$ and End_Offset are the beginning and end time points of all the elements in this MBR. For any UTS, the first point included in the MBR is trivially $\{uts\_i, Dist, Dist, 1, 1\}$, where $Dist$ is the distance of the first sequence to R_k. For all other subsequences, we first compute $Dist$, and then check if adding this point to the existing MBR will increase its *marginal cost*, a heuristic proposed by Faloutsos *et al.* [8]. Due to shortage of space we do not describe it here. If the new marginal cost (after adding the new point) is greater than the old cost (without the point), a new MBR is started with this new point as the sole entry, else the old MBR is updated. The **CheckMC** routine in the pseudo code performs this task. Once all the subsequences of uts_i are processed, all the MBR's are appended to file mbr_k and the next UTS is processed. Finally, each of these mbr_k files are indexed using an **RTreeBuild** routine and the spatial indices are saved on disk.

We would like to point out that while Faloutsos *et al.* [8] also use MBR to combine subsequences to reduce the index space, they map each subsequence into 6 DFT coefficients while we map each subsequence into a single value *viz.* distance to the reference point. So in our case, each MBR is a two dimensional point, leading to better scalability.

Algorithm 1: Build Index for RBS

Input: U_DB, w
Output: Indices $Index_1, \ldots, Index_r$
Initialization: Select r reference points $R_1, \ldots, R_r$;
begin
  for k = *1 to r* **do**
    for *uts_i in UTS_Database* **do**
      $nMBR \leftarrow 1$;
      $Dist \leftarrow dist(R_k, y^{uts\_i}_{[1:w]})$;
      $mbr(nMBR) \leftarrow \{uts\_i, Dist, Dist, 1, 1\}$;
      $MaxOffset \leftarrow (L(uts\_i) - w + 1)$;
      for *j = 2 to MaxOffset* **do**
        $Dist \leftarrow dist(R_k, y^{uts\_i}_{[j:j+w-1]})$;
        $[ud, newMBR] \leftarrow$ **CheckMC**$(mbr, Dist)$;
        if $ud == 0$ **then** $nMBR = nMBR + 1$;
        $mbr(nMBR) \leftarrow newMBR$;
      Append mbr to file mbr_k;
    $Index_k \leftarrow$ **RTreeBuild**(mbr_k);
    Save $Index_k, R_k$;

When a query Q of length w is provided, we use the search code shown in Alg. 2. The inputs in this case are the UTS query Q, U_DB, the set of indices, the set of reference points, w, and ϵ. The output is ϵ-NN of Q. First, for each reference point R_k, we find the distance D_k of the query from it. Then we perform a range query search $\{D_k - \epsilon, D_k + \epsilon\}$ using the **RTreeSearch** routine. We call this step the *first level* of pruning. The output of the search code are a set of candidate MBR's which intersect the query MBR. In the *second level* of pruning, we intersect the candidate MBRs found using different reference points. This reduces the number of false alarms dramatically as we show in our experiments, leading to very high prune rate and very low search time. Once a compact candidate set is found, we do disk access to retrieve those candidates and remove false alarms.

We now discuss how RBS handles queries longer than w in the following two cases:

- $L(Q) = nw \quad (n > 1)$: We first divide Q into n disjoint subsequences of length w, and search the indices set for each of them with the threshold $\epsilon/\sqrt{(n)}$. Finally, we do an exact calculation of full length candidates (over all n parts) to remove false alarms. The correctness of this approach relies on the following Theorem [8].

Algorithm 2: RBS ϵ-NN Search on UTS

Input: U_DB, Q, $Index_1, \ldots, Index_r$, $R_1, \ldots, R_r$, w, ϵ
Output: ϵ-NN of Q
begin
  ϵ-NN $\leftarrow \emptyset$;
  for k = *1 to* r **do**
    $D_k = dist(Q.seq_1, R_k)$;
    $Cand_k$ = **RTreeSearch**$(Index_k, \{D_k - \epsilon, D_k + \epsilon\})$;
  $CandAll \leftarrow \{\bigcap_{k=1}^{r} Cand_k\}$;
  forall the $\{uts\_i, b, e\} \in CandAll$ **do**
    Fetch $y_{[b:e]}^{(uts\_i)}$ from uts_i file on disk;
    $Dist = dist(y_{[b:e]}^{(uts\_i)}, Q.seq_1)$;
    if $Dist \leq \epsilon$ **then** ϵ-NN $\leftarrow$ ϵ-NN $\bigcup\{uts\_i, b, e\}$;

Theorem 4.1: If $dist(Q, S) < \epsilon$, then for at least one pair of disjoint sequences Q_i and S_i of length w, we have $dist(Q_i, S_i) < \epsilon/\sqrt{(}n)$.

- $L(Q) = nw + v \quad (0 < v < w)$: We can ignore the last subsequence of length v and perform search on the nw disjoint subsequences as described before. We only consider the last subsequence when we perform the exact calculation.

C. LBS algorithm details

In ***RBS***, the smallest unit of search is an MBR. Now, for one reference point, ***RBS*** has a prune rate directly proportional to the number of MBR's searched times the number of points in that MBR. Although the search time for ***RBS*** can be very low, large sizes of candidate set increase the overall search time to fetch all the potential candidates from the disk. To alleviate this problem, we present another novel algorithm ***LBS***, in which the search unit is a subsequence in the input space. This algorithm directly exploits the triangular inequality to effectively prune bad candidates by choosing a random subsequence as a reference subsequence. Moreover, to increase the prune rate further, we have used multiple reference points.

As before, the inputs to ***LBS*** are U_DB and length of the sliding window w. The output is a set of sorted lists as indices. In the first step, similar to ***RBS***, we compute the distances of all the subsequences from a few reference points $R_1, \ldots, R_r$. We store these distances (as the key) along with the offset and $UTS{-}id$ into a list called $Index_k$, for reference point R_k. In the next step we simply sort these k lists and store them along with the reference points.

During searching, when a query Q of length w is provided, for each reference point R_k, we find the distance $Dist_k$ of the query from R_k. Then we collect those candidates from $Index_k$ whose key (distance) lies in the range $Dist_k \pm \epsilon$. This is a direct application of the triangle inequality. As before, we intersect the candidate sets for all the reference points finally do a disk access to remove false alarms. We do not present the pseudo-code here due to shortage of space.

V. Flexible MTS subsequence search

We now describe our algorithm for MTS query search. In our problem setting, we have substantially more variables to index compared to the number of variables given in a typical query. Moreover, the query variables are not known apriori which severely restricts the use of existing MTS search algorithms. The algorithm we propose here has excellent performance for the multivariate queries that we want to execute.

As before, we split the discussion into two parts. The index building algorithm is very similar to the one presented for UTS search. Alg. 3 presents the pseudo code. The first step is to decompose the MTS database M_DB into a series of univariate time series databases $U\_DB^{(1)}, \ldots U\_DB^{(d)}$, one for each feature in the MTS. Then we select r reference points for each UTS independently, and use Alg. 1 to build indices for each of the d UTS's. Thus for d features, we will have $d \times r$ number of sorted lists for LBS algorithm and $d \times r$ number of R^*-trees for RBS. We store these indices along with the reference points on disk.

Algorithm 3: MTS Build Index using RBS

Input: M_DB, w
Output: $Index$ for MTS search
begin
  Convert M_DB into $U\_DB^{(1)}, \ldots, U\_DB^{(d)}$;
  for f = *1 to* d **do** `// each feature`
    Select $R_1^{(f)}, \ldots, R_r^{(f)}$ for $U\_DB^{(f)}$;
    Index each $U\_DB^{(f)}$ using Alg. 1.

Given a search query Q having v sequences for v variables and $v-1$ time delays between them, the goal of MTS search algorithm (Alg. 4) is to return all matching multivariate patterns from M_DB. To solve this, we first take the first variable (call it $Q.var(1)$) of Q and do a search on the index corresponding to feature $Q.var(1)$. The **FindCandidates** function in Alg. 4 performs this search by first finding a candidate set from each index file of $Q.var(1)$ and then joining them over multiple reference points. This routine is similar to Alg. 2 (except the disk access part). This generates an MTS table as: $\{MTS\_id, Begin\_offset_1\}$. Similarly, the next variable $Q.var(2)$ is searched on the relevant index. These two searches on the indices correspond to the *first level* of pruning. At this point we prune the candidates further by joining these candidate sets $(Cand_{12})$ and noting that (1) all candidates in candidate 1 and candidate 2 must have the same MTS_id, and (2) the begin offsets between any two candidates from the two sets must be delayed by an amount δ_1. The **JoinCandidate** routine performs this join. By this *second level* of pruning, we add another column to the table for the second variable $\{MTS\_id, Begin\_offset_1, Begin\_offset_2\}$. Note that until this point, we have not performed any actual disk

access, and searched only on the indices. We could continue joining the candidate sets and create a compact set for all the variables in Q. However, in our experiments (not reported here), we notice that the size of the candidate set after the first two joins is very small and does not reduce further on joining other candidate sets. We validated this for several variables in the candidate sets; in most cases, the size of the candidate set was less than 5% of the total number of subsequences. Thus, heuristically it becomes redundant to search for the remaining variables in the index. Instead, we do a disk access to retrieve all candidates from $Cand_{12}$ to remove the false alarms. The resulting subsequences *Cand* are the true nearest neighbors of Q considering the first two variables. We continue to search the remaining variables $Q.var(3:v)$ by retrieving them directly from the disk after noting that they must come from the same MTS and satisfy the specified time delays.

Algorithm 4: MTS ϵ-NN Search using RBS

Input: M_DB, Q, $Index$, $R_1, \ldots, R_r$, w, $\boldsymbol{\epsilon}$
Output: ϵ-NN of Q
begin
  ϵ-NN $\leftarrow \emptyset$;
  for $i = 1$ *to* $Q.var$ **do** `// each feature`
    $Cand_i \leftarrow$ **FindCandidates**$(Q.var(i))$;
  $Cand \leftarrow \bigcap_{i=1}^{Q.var-1}$ **JoinCandidates**$(Cand_i, Cand_{i+1}, \delta_i)$
  for $c \in Cand$ **do** `// remove false positives`
    Fetch c from i-th MTS $Dist_1 = dist(c.seq_1, Q.seq_1)$;
    $Dist_2 = dist(c.seq_2, Q.seq_2)$;
    $\vdots$
    if $Dist_1 \leq \epsilon_1$ *and* $Dist_2 \leq \epsilon_2$ *and* $\ldots$ **then**
      ϵ-NN $\leftarrow \epsilon$-NN $\bigcup\{c, i, j\}$;

VI. Analysis of algorithms

In this section analyze the properties of the algorithms.

A. Correctness of LBS and RBS

Theorem 6.1: Both LBS and RBS algorithms are correct *i.e.* they guarantee no false dismissals.

Proof: The proof is based on the triangle inequality. For a reference point R, query Q and any arbitrary subsequence S, we can write by virtue of triangle inequality:

$$|dist(Q,R) - dist(S,R)| < dist(Q,S).$$

Now for any query Q which belongs to ϵ-NN of S, $dist(Q,S) < \epsilon$. Combining, we get

$$|dist(Q,R) - dist(S,R)| < dist(Q,S) < \epsilon$$

i.e. $dist(Q,S) < \epsilon \Rightarrow |dist(Q,R) - dist(S,R)| < \epsilon \Rightarrow dist(Q,R) - \epsilon < dist(S,R) < dist(Q,R) + \epsilon$. Since in both LBS and RBS, we retrieve all sequences from the index in the range $dist(Q,R) \pm \epsilon$, both these algorithms guarantee no false dismissals. ■

B. Storage complexity of LBS and RBS

For LBS, we need to insert every subsequence in the sorted list for every UTS. Let T_i be the length (number of time points) of any MTS in the i-th file. The number of subsequences for the i-th MTS is, therefore, $T_i - w + 1$. Given there are d variables in each of the MTS files, the number of subsequences to process for the i-th MTS file is $d(T_i - w + 1)$. For $|D|$ total MTS files, we get the total number of subsequences as, $d\sum_{i=1}^{|D|}(T_i - w + 1)$. For r reference points, the overall storage complexity is $O(rd\sum_{i=1}^{|D|}(T_i - w + 1)) = O(rd\sum_{i=1}^{|D|} T_i)$. For RBS, the index storage complexity is $O(rd\sum_{i=1}^{|D|} M_i)$, where M_i are the number of MBR's created from the i-th MTS. Since in general, $M_i \ll T_i$, RBS has a much lower index storage complexity.

C. Running time of LBS and RBS

For LBS, the index building time is proportional to the number of distances computed for each subsequence: $w(T_i - w + 1)$. For d variables, r reference points and $|D|$ MTS files, the overall running time for inserting all the elements in the index is $O(wrd\sum_{i=1}^{|D|}(T_i - w + 1)) = O(wrd\sum_{i=1}^{|D|} T_i)$. Moreover, since $rd\sum_{i=1}^{|D|}(T_i - w + 1)$ elements need to be sorted, the overall running time is the maximum of the sorting time and the insertion time. For RBS, we need to do some extra computation for checking the marginal cost of each point. Let the time required for it be λ. Therefore, the overall time complexity is, $O((w+\lambda)rd\sum_{i=1}^{|D|}(T_i - w + 1))$, where we have ignored the time to insert M_i MBRs in the R^*-tree.

The query time for both the algorithms is bounded by: $O(\max_i |Cand_i|) + O(w|Cand|)$, where the max is taken over all the candidate sets and the second term reflects the time for actual disk access and exact computation.

D. Choice of reference points

The choice of the reference points is crucial to the performance of our algorithms. From Th. 6.1, a point S is not a potential candidate to be the nearest neighbor of Q if $|dist(Q,R) - dist(S,R)| > \epsilon$, where R is an arbitrarily chosen reference point. This is because, by triangular inequality, $dist(Q,S) \geq |dist(Q,R) - dist(S,R)| > \epsilon$ too. Therefore, such an S cannot belong to the set of nearest neighbors of Q. If, on the other hand, $|dist(Q,R) - dist(S,R)| < \epsilon$, then we cannot prune S since $dist(Q,S)$ can be greater or less than ϵ. Therefore, the *goodness* of R can be evaluated based on the size of the following set: $\mathcal{S} = \{S : |dist(Q,R) - dist(S,R)| < \epsilon\}$. Minimizing the size of $\mathcal{S}$ gives a good R. However, in the above formulation, Q is typically unknown until query time, making the optimization problem unsolvable. Our heuristic is to choose multiple reference points randomly from the database with the hope that each such point will prune many candidates and we can only work with the intersection of these sets. Our extensive

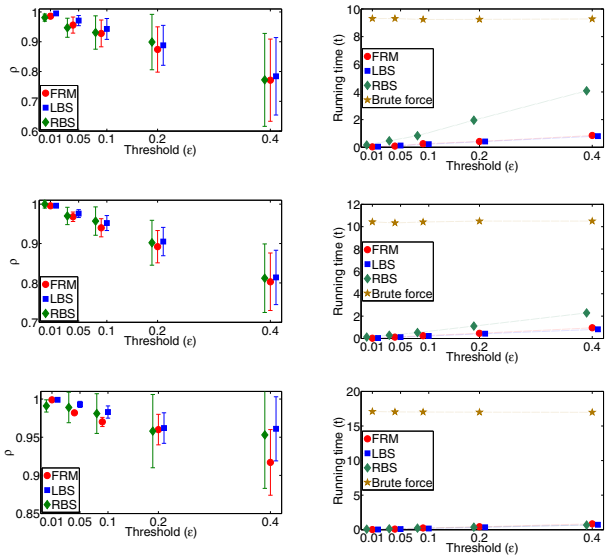

Figure 3. Variation of ρ and t (mean and std dev) for different w, averaged over ten queries for random walk dataset. Left column shows ϵ vs. ρ and right column shows t vs. ρ for $w = 128, 256, 1024$ from top to bottom respectively. In most cases, LBS shows higher prune rate while prune rates of RBS are comparable to FRM. Also the running time of all the algorithms are comparable; in most cases, LBS has the least search time.

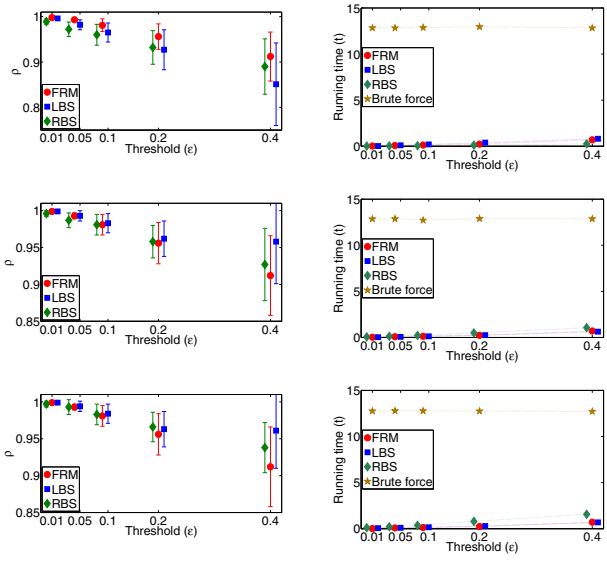

Figure 4. Variation of ρ and t (both mean and std dev) with the number of reference points, averaged over ten queries for random walk dataset. Left column shows ϵ vs. ρ and right column shows t vs. ρ for $|r| = 1, 2, 3$ from top to bottom respectively. In most cases, LBS shows higher prune rate while prune rates of RBS are comparable to FRM. Also the running time of all the algorithms are comparable; in most cases, LBS has the least search time.

experimental results show the effectiveness of this simple heuristic by choosing 3-5 reference points (see Fig. 6 and Fig. 4).

VII. Experiments

To validate the performance of the LBS and RBS algorithms, we have run a variety of tests using both univariate and multivariate datasets. All algorithms have been implemented in Matlab and run on a 64-bit 2.33 GHz quad core dell precision 690 desktop running Red Hat Enterprise Linux version 5.4 having 2GB of physical memory. We have measured the following quantities:

- ρ – the prune rate (=1 − $|C|/T$), where C and T are sizes of the candidate set and the number of sliding windows
- t – running time

A. Univariate dataset experiments

1) Dataset description and experimental setup: We have used 2 univariate datasets for testing our algorithms which have been used in the literature [8][9] for UTS subsequence search. The first dataset is a random walk dataset generated synthetically (500,000 points). The second dataset is a stock market dataset having 329,112 entries. We have tested 3 algorithms on these datasets: (1) the FRM algorithm using the adaptive MBR approach [8], (2) *LBS*, and (3) *RBS*.

We have measured ρ and t at varying window sizes w (128, 256, 512, 1024) and the number of reference points (1∼5). The default values of these parameters are fixed at 512 and 3 respectively. For each choice of w and t, we have experimented with five different ϵ. The choice of each ϵ is such that the selectivity (*i.e.* actual number of nearest neighbors/T) ranges between $10^{-6} \sim 10^{-1}$ [8]. ρ and t at each measurement point is an average over ten randomly generated queries. We present the results in the next section.

2) Results: We summarize the results of FRM, LBS and RBS in Figures 3 – 6. Fig. 3 shows the average and standard deviation of ρ and t for each ϵ, over ten queries for the random walk dataset for different values of w. For most of the thresholds, we see that the prune rate of LBS is the highest. Also, the prune rates of RBS tend to be very close to the FRM algorithm for smaller number of reference points. One significant advantage of both LBS and RBS over FRM is that the prune rates for the former two algorithms can easily be controlled by increasing the number of reference points; however this increases the running time as well. Also, the prune rates for all these algorithms increase with increasing w, due to lesser number of windows to index. Fig. 4 demonstrates the performance of the algorithms for varying number of reference points. As expected, the prune rate increases with increasing number of reference points. We have similar results for the random walk dataset shown in the Figures 5 and 6. In this case, RBS has a higher prune rate compared to LBS or FRM.

To sum up, both the LBS and the RBS algorithms offer an excellent prune rate for UTS search. LBS offers the best prune rate of all the 3 algorithms compared here, but as

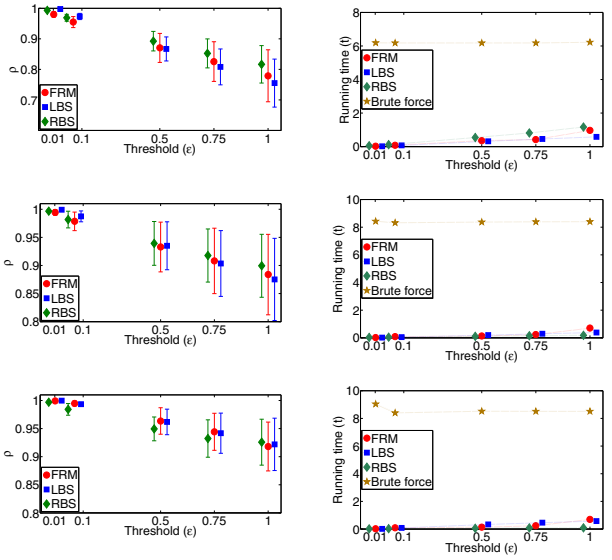

Figure 5. Variation of ρ and t (both mean and std dev) for different w, averaged over ten queries for stock market dataset. Left column shows ϵ vs. ρ and right column shows t vs. ρ for $w = 128, 256, 1024$ from top to bottom respectively. For this dataset, RBS shows higher prune rate than FRM or LBS. Also the running time of all the algorithms are comparable; in most cases, LBS has the least search time.

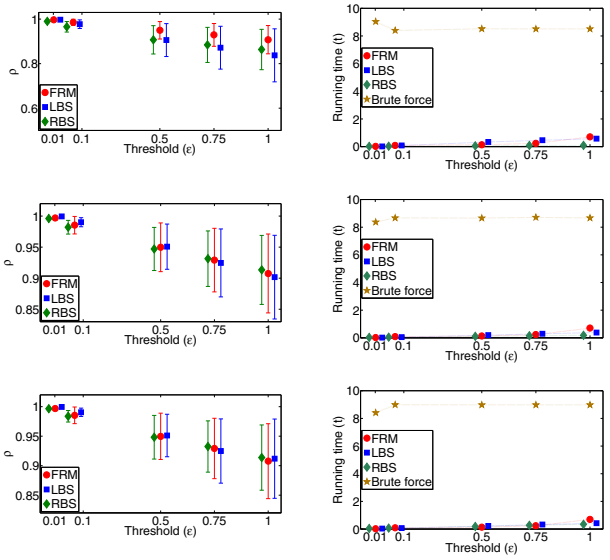

Figure 6. Variation of ρ and t (both mean and std dev) with the number of reference points, averaged over ten queries for stock market dataset. Left column shows ϵ vs. ρ and right column shows t vs. ρ for $|r| = 1, 2, 3$ from top to bottom respectively. In most cases, LBS shows higher prune rate while prune rates of RBS are comparable to FRM. Also the running time of all the algorithms are comparable; in most cases, LBS has the least search time.

discussed before, suffers from large storage cost. On the other hand, RBS uses MBRs to group similar points and hence can reduce the storage cost dramatically. In many cases, this reduces the search time as well. However, since the unit of search is an MBR (containing several points) and not individual points (as in LBS), the prune rate of RBS is lower than LBS. It also needs to be mentioned that if the variables are not normalized, the MBR creation heuristic (I-adaptive in [8]) decides on the density of each MBR based on ϵ. Too high a value of ϵ packs more points per MBR, reducing the number of MBRs. This, in turn, reduces the prune rate. Lower values of ϵ fragments the MBRs to only a few points in each. This increases the prune rate but increases the index search time. We test with different values of ϵ during building indices and always choose an ϵ in the middle range of those reported here.

B. Multivariate dataset experiments

1) Dataset description: We have used two large multivariate datasets for demonstrating the search capabilities of LBS and RBS in the multivariate domain. To the best of our knowledge, these multivariate datasets are much larger than the datasets used in the literature for multi-dimensional time series search. The datasets are described next.

C-MAPSS dataset: The first dataset is simulated commercial aircraft engine data. The dataset contains 6,875 (=$|D|$) full flight recordings sampled at 1 Hz with 29 engine and flight condition parameters. This dataset has 32,640,967 tuples. We have tested our algorithm with 16 variables only.

US Regional carrier dataset (CarrierX): The second dataset is a real life commercial aviation dataset of a US regional carrier consisting of 3,573 (=$|D|$) flights. Each flight contains 46 variables. Domain experts identified a subset of 9 variables which are important. There are 22,207,852 tuples.

For all the multivariate experiments, we have used $w = L(Q) = 256$ and 3 reference points for both LBS and RBS.

2) Results: We have tested 5 randomly chosen queries, each with three different thresholds. For each query and threshold combination, the selectivities of each ranges from $10^{-7} \sim 10^{-6}$. We do not present the thresholds for each variable here due to shortage of space.

The performance results of LBS and RBS on CMAPSS and CarrierX are presented in Table I. The second column refers to the five different queries we have run along with the variables for each query. The next three columns show the number of candidates generated for the first variable ($Cand_1$), the second variable ($Cand_2$), and after joining these two candidate sets $Cand_{12}$ both for LBS and RBS. Column C_{exact} is the actual number of these candidates which are found to be less than the threshold after doing the exact calculation. The smaller the size of $Cand_{12}$, the fewer the number of actual disk accesses necessary. ϵ-NN column refers to the actual number of nearest neighbors of the query after taking all the variables and time delays into consideration. The last two columns show the prune rate $\rho = Cand_{12}/T$ and the query time for LBS. Since the query times for RBS are very similar, we do not report

them here. For this experimental setup, the index building time for LBS and RBS on the **CarrierX** dataset are 7 hrs and 9 hrs respectively.

These results show that for the two large multivariate datasets, for different queries and thresholds, the prune rates are very high ($\sim$ 95%). Also, we notice that the sizes of the candidate sets are smaller for LBS than RBS for all the queries thereby generating fewer false positives. However, the storage requirement of LBS is non-trivial. For example, for CarrierX, we need to index approximately 22 million distances using each reference point per UTS. The total storage requirement for the index will be (22,000,000$\times$(4+4+4)/(1024$\times$1024)) $\approx$ 250 MBytes, for each UTS, assuming we store $\{Dist, MTS\_id, Begin\_offset\}$ for each window sequence as a float of (4+4+4) bytes. For RBS, let's assume that (1) we have M MBRs on average for each reference point, and (2) we store $\{min\_MBR, max\_MBR, MTS\_id, Begin\_offset, End\_offset\}$ for each MBR. In our experiments we have $M = 5,174,619$. Then the total storage requirements (assuming 4 bytes for each) will be (5,174,619$\times$(4+4+4+4+4)/(1024$\times$1024)) $\approx$ 98 MBytes, lower than that of LBS. Also note that the query time for most of the queries are extremely small considering the large sizes of the datasets.

From these results we conclude that: (1) query execution time of LBS is expected to be much lower than RBS due to higher prune rate, (2) RBS has relatively higher rate of false positives compared to LBS, and (3) the index storage requirements of LBS may be significantly higher compared to RBS. However, the choice of RBS vs. LBS is application dependent.

C. Application: finding anomalous flights

We have used the MTS search algorithm to find flight landing patterns which result in go around/aborted landing. In many cases, an aircraft on approach to landing needs to abort the landing, climb back on full throttle and try the landing again. This can happen due to improper landing configuration. Currently, most safety analysts study these events based on only one variable at a time which generates a large number of false positives. These so-called exceedences or anomalies can be indicators of safety issues. The frequency of such events are tracked as a measure of safety of operations. These events can aid significantly in understanding the underlying causal factors.

We have searched for such incidents in the **CarrierX** dataset using two variables: airspeed (in knots) and altitude (in feet). A domain expert (a retired commercial pilot) has helped us sketch a typical go around pattern as shown in Fig. 7. The left figure shows the variation in airspeed while the right one shows the variation in altitude. Using such a query as the input and thresholds 100, 4000 for the two variables, we have searched the **CarrierX** dataset. The algorithm

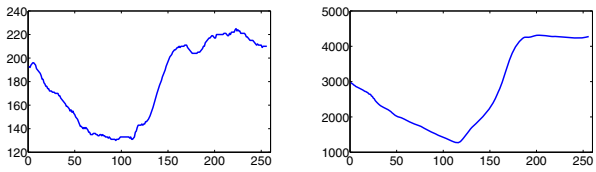

Figure 7. Typical pattern for "go around" in **CarrierX** dataset. Left plot shows airspeed (knots) vs time while right plot shows altitude (feet) vs. time.

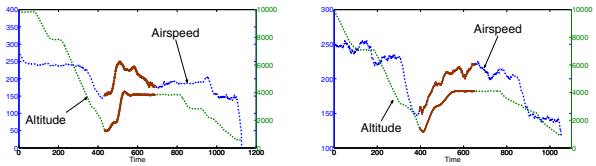

Figure 8. Examples of "go arounds" detected by our multi-variate search algorithm on **CarrierX** dataset. The matching regions are highlighted.

returned 10 hits. Fig. 8 shows 2 such flight profiles. We have plotted the altitude and airspeed on the same graph with the left axis as the airspeed and the right axis as the altitude. A visual inspection of each of these flights demonstrates the usefulness of the algorithm in finding all the "go around" patterns (no false positives). The highlighted portion shows the matched time series for each of these plots which shows that the algorithm is accurate at finding *similar*, not exact, motifs, *i.e.*, it has good noise tolerance. The average time taken for running the query is approx. 12 secs.

VIII. Conclusion

In this paper we present two algorithms LBS and RBS for finding multivariate subsequences from large MTS datasets. Both these algorithms guarantee no false dismissals. RBS algorithm is novel in the sense that it organizes subsequences into MBRs and uses multiple reference points to reduce false positives. To the best of our knowledge, using spatial indexing along with multiple global reference points for time series subsequence search has never been explored before. Experiments on two massive commercial aviation related MTS datasets show that both these algorithms offer excellent prune rates (greater than 0.95). The CMAPSS and CarrierX datasets that we have tested are much bigger than any of the MTS datasets used in the literature for multivariate subsequence search. As an application of the proposed method, we have shown how it can be used for finding a critical safety pattern from real aviation dataset, that of aborted landings. For future work, we plan to implement this algorithm on Map-Reduce and explore other distance measures such as time warping.

Acknowledgements

This work was supported by the NASA Integrated Vehicle Health Management Project and a NASA-Google Annex.

Table I

RESULTS OF LBS AND RBS CMAPSS AND CARRIERX DATASET FOR FIVE DIFFERENT QUERIES AND THREE DIFFERENT THRESHOLDS PER QUERY. FOR BOTH LBS AND RBS, THE PRUNE RATES ARE ALWAYS GREATER THAN 0.95, SIGNIFYING THAT LESS THAN 5% OF THE CANDIDATES NEED TO BE RETRIEVED FROM THE MTS DATABASE FOR EXACT CALCULATIONS.

Queryid	$\|Cand_1\|$		$\|Cand_2\|$		$\|Cand_{12}\|$		C_{exact}	$\|\epsilon$-NN$\|$	Prune rate ρ		Time (secs)
	LBS	RBS	LBS	RBS	LBS	RBS			LBS	RBS	
CMAPSS											
1: (25, 27, 4)	18409	3007594	738	2477549	52	801400	6	6	**0.9999**	**0.9741**	2.63
	81409	3263815	7567	2565309	2668	1003839	17	10	**0.9999**	**0.9675**	102.91
	251981	3841664	81330	2702600	23694	1454776	540	297	**0.9992**	**0.9529**	291.8
2: (20, 29, 5)	53585	870835	14969	2390063	1411	266022	252	6	**0.9999**	**0.9914**	6.91
	179850	1295644	50502	2454707	13862	481096	1187	17	**0.9995**	**0.9844**	130.91
	317793	1587719	141444	2633060	58905	633137	20124	259	**0.9981**	**0.9795**	710.12
3: (5, 15, 28)	528470	4753958	14725	306706	6171	290593	453	8	**0.9998**	**0.9906**	201.13
	1137522	4861533	87236	425813	63690	399972	16289	121	**0.9979**	**0.9871**	770.18
	2115994	5101127	177992	550198	174391	536022	79332	1445	**0.9944**	**0.9826**	945.1
4: (26, 5, 27)	1311	2013861	57144	3655449	344	86193	5	3	**0.9999**	**0.9972**	23.1
	34492	2143905	193974	3894274	8034	194616	2060	337	**0.9997**	**0.9937**	41.1
	115350	2317163	501207	4634240	38648	609697	22034	6471	**0.9987**	**0.9803**	99.13
5: (5, 23, 2)	101344	4010042	74609	878140	12945	114419	18	9	**0.9996**	**0.9963**	141.98
	316085	4101886	164881	1160134	49908	203004	332	49	**0.9983**	**0.9934**	121.9
	771259	4356479	337201	1521911	150020	375037	4925	479	**0.9951**	**0.9879**	821.1
CarrierX											
1: (29, 23, 28)	26235	469928	55610	530788	96	10226	3	3	**0.9999**	**0.9995**	3.69
	79606	523225	204310	716418	952	14391	15	15	**0.9999**	**0.9993**	9.41
	133451	583050	374437	896063	2640	20771	27	27	**0.9998**	**0.999**	15.58
2: (8, 28, 27)	17338	1120516	16541	74930	450	26361	3	1	**0.9999**	**0.9987**	28.56
	48149	1174920	62316	267710	3595	92246	7	3	**0.9998**	**0.9957**	119.32
	83177	1218440	1577348	3028623	54214	754404	885	9	**0.9974**	**0.9645**	694.94
3: (38, 8, 29)	935844	870535	223138	391564	71342	94594	12318	7	**0.9966**	**0.9955**	69.4
	1500995	1369274	379346	555599	175800	213822	48395	64	**0.9917**	**0.9899**	147.69
	1760160	1564834	527712	705614	277017	313020	102401	269	**0.9869**	**0.9853**	197.97
4: (6, 27, 30)	22039	2164753	13866	901583	71	402047	10	10	**0.9999**	**0.9811**	3.01
	103096	2289089	156448	1033504	2204	477704	30	30	**0.9998**	**0.9775**	17.7
	213954	2429383	351061	1196446	9408	568003	48	48	**0.9995**	**0.9733**	44.01
5: (28, 8, 29)	1298247	2671533	184660	1649628	76445	476399	47559	2	**0.9964**	**0.9776**	64.63
	1947774	3368141	205164	129643	105286	29617	78467	125	**0.9951**	**0.9986**	92.95
	5161965	6417365	227501	1735525	168155	972349	136137	882	**0.9921**	**0.9543**	197.27

The authors would also like to thank Dr. Matthew E. Otey and Bryan Matthews for their valuable suggestions.

REFERENCES

[1] K. Yang and C. Shahabi, "A PCA-based Similarity Measure for Multivariate Time Series," in *Proceedings of MMDB'04*, 2004, pp. 65–74.

[2] ——, "An Efficient k Nearest Neighbor Search for Multivariate Time Series," *Inf. Comput.*, vol. 205, no. 1, pp. 65–98, 2007.

[3] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh, "Indexing Multi-Dimensional Time-Series with Support for Multiple Distance Measures," in *Proceedings of KDD'03*, New York, NY, USA, 2003, pp. 216–225.

[4] S. Lee, S. Chun, D. Kim, J. Lee, and C. Chung, "Similarity Search for Multidimensional Data Sequences," in *Proceedings of ICDE'00*, 2000, pp. 599–608.

[5] K. Yang and C. Shahabi, "A PCA-based Kernel for Kernel PCA on Multivariate Time Series," in *Proceedings of ICDM'05 Workshops*, 2005, pp. 149–156.

[6] ——, "A Multilevel Distance-Based Index Structure for Multivariate Time Series," in *Proceedings of TIME'05*, Washington, DC, USA, 2005, pp. 65–73.

[7] E. Keogh and C. Ratanamahatana, "Exact Indexing of Dynamic Time Warping," *KAIS*, vol. 7, no. 3, pp. 358–386, 2005.

[8] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast Subsequence Matching in Time-series Databases," *SIGMOD Rec.*, vol. 23, no. 2, pp. 419–429, 1994.

[9] Y. Moon, K. Whang, and W. Loh, "Duality-Based Subsequence Matching in Time-Series Databases," in *Proceedings of ICDE'01*, Washington, DC, USA, 2001, pp. 263–272.

[10] C. Traina, R. Filho, A. Traina, M. Vieira, and C. Faloutsos, "The Omni Family of All-purpose Access Methods: A Simple and Effective Way to Make Similarity Search More Efficient," *The VLDB Journal*, vol. 16, pp. 483–505, 2007.

[11] W. Han, J. Lee, Y. Moon, and H. Jiang, "Ranked Subsequence Matching in Time-Series Databases," in *Proceedings of VLDB'07*, 2007, pp. 423–434.

[12] A. Mueen, E. Keogh, and N. Bigdely-Shamlo, "Finding Time Series Motifs in Disk-Resident Data," in *Proceedings of ICDM'09*, Miami, 2009, pp. 367–376.

[13] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces," in *Proceedings of VLDB'97*, 1997, pp. 426–435.

[14] A. Mueen, E. Keogh, Q. Zhu, S. Cash, and M. Westover, "Exact Discovery of Time Series Motifs," in *Proceedings of SDM'09*, 2009, pp. 473–484.