Set-based Similarity Search for Time Series

Jinglin Peng Hongzhi Wang Jianzhong Li Hong Gao
Harbin Institute of Technology
jlpengcs@gmail.com {wangzh,lijzh,honggao}@hit.edu.cn

ABSTRACT

A fundamental problem with time series involves k nearest neighbor (k-NN) query processing. However, existing methods are not fast enough for large datasets. In this paper, we propose a novel approach, STS3, for processing k-NN queries that transforms time series into sets and measures the similarity under the Jaccard metric. Our approach is more accurate than Dynamic Time Warping (DTW) in some suitable scenarios and is faster than most existing methods, due to the efficiency of similarity search for sets. In addition, we developed indexing, pruning and approximation techniques to improve the k-NN querying procedure. As shown in the experimental results, these techniques could accelerate the query processing effectively.

Keywords

time series; similarity search; k-NN; representation

1. INTRODUCTION

A time series is a sequence of data points measured in a fixed time interval. It is widely used in many applications such as financial data analysis [6], motion capture [10] and the tracking of moving objects [9]. In such applications, a k nearest neighbor (k-NN) similarity search of a time series is a basic operation that which aims at finding the time series in a set that are most k similar for a given query.

For example, the data in a electrocardiogram (ECG) could be modeled as a time series. In a computer-assisted diagnosis, a doctor may want to compare the ECG time series of a patient to the time series in a database and compare the k-NN time series to that of the patient to find candidates of diseases.

Due to its importance, some k-NN similarity search algorithms for time series have been proposed. Such algorithms are based on various similarity measures.

Similarity computation approaches aim to define and compute the similarity of time series. Some examples are Euclidean Distance (ED) [12], Longest Common Subsequence (LCSS) [30] and Dynamic Time Warping (DTW) [5].

Among these measures, the one with the most efficient search algorithm is ED. For two time series, S and Q, the computation complexity of ED is $O(n)$. However, the obvious drawbacks of ED are that it can only deal with time series of equal length and it is sensitive to noise. DTW and LCSS are more robust and can handle time series with various lengths, but their computations are expensive. The time complexity of DTW and LCSS restricted by the Sakoe-Chiba band [24] is $O(n\omega)$.

Although some indexing structures and pruning strategies have been proposed to accelerate DTW and LCSS, they do not solve the problem thoroughly. In LCSS, time series are indexed as MBRs (Minimum Boundary Rectangles) stored in an R-tree. When a query arrives, its Minimum Bounding Envelope (MBE) is constructed and split into MBRs. Then ”similarity estimates are computed and the exact LCSS (or DTW) is performed only on the qualified sequences” [32]. Thus, by excluding the series that cannot be in k-NN, LCSS is accelerated. Similarly, DTW is quickly evaluated by its lower-bound functions, which cost less, [35] and some indexing techniques [26]. These acceleration strategies mainly depend on the rapid estimation of accurate distance, which is related to the specific data. Hence the acceleration is limited, and it may not achieve satisfactory efficiency in some cases.

To avoid the drawbacks of existing approaches and perform k-NN similarity search efficiently and effectively, we propose STS3, a novel algorithm for this problem. In our approach, we model time series as points on a plane. We further divide the plane into cells and the points in the same cell are treated as one element. Thus, during the processing, a time series is converted to a set of cells. The motivation for this conversion is to change the comparison of time series in quadratic time complexity into a comparison between sets, which can be accomplished in linear time. Thus, even without additional indexing and acceleration strategies, the similarity search is faster than early approaches.

To achieve higher performance, we design three filter strategies. The first one uses an inverted index to select candidates. The second one divides the plane into a scale and tries to prune candidates in this scale. The third one divides the plane into different scales in various rounds and only considers time series having maximal similarity to the query in each scale for processing in the next step, which achieves substantial filtering in each step.

The main contributions of this paper are as follows.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882963

- We propose a novel approach that handles k-NN similarity search of time series by transforming time series to sets. To the best of our knowledge, this is the first work using set techniques to accelerate time series search.
- We propose three versions of STS3 to speed our approach up. The index-based STS3 uses an inverted list to index time series. The pruning-based STS3 prunes time series in a flexible scale, and the approximate STS3 filters time series in a coarse scale and then refines them. All three approaches can speed up the query processing effectively.
- To demonstrate the efficiency and effectiveness of the proposed approaches, we conduct extensive experiments on real data. The experimental results show that the proposed algorithm outperforms other approaches significantly in efficiency without significant loss of accuracy.

The rest of the paper is organized as follows. Background is discussed in Section 2. Section 3 describes our approach in detail, and the three speed-up techniques are demonstrated in Section 4. Section 5 extends our approach to handle situations outside of our assumptions. We discuss the pros and cons of our approach in Section 6. Our experimental results are presented in Section 7 and we discuss related work in Section 8. Section 9 concludes the paper.

2. BACKGROUND

In this paper, we investigate the problem of k-NN similarity search for time series. In this section, we introduce the definition of time series, the preprocessing approach and the assumptions used in this paper.

Definitions We define time series as follows.

Definition 1. *A time series S is described as a sequence of pairs, i.e., $S = [(t_1, s_1), (t_2, s_2), ..., (t_n, s_n)]$, in which t_i is the time stamp of s_i, satisfying $t_i < t_j$ if $i < j$. s_i is a d-dimension data point.*

Preprocessing Before we process the time series, it is necessary to normalize them. The commonly used approach is z-normalization [17], shown as follows:

$$Norm(S) = \frac{S - mean(S)}{std(S)}$$

Assumptions For the convenience of our discussions, we make the following assumptions.

- All time series are z-normalized before processing and we use S to denote $Norm(S)$ throughout this paper.
- All time series are one dimensional sequences. For example, S=(2,3,4) denotes the time series S=((1,2),(2,3),(3,4)). That is, the time stamp is the sequential labeling and is omitted. Our method can be extended into multi dimensions, as will be shown in Section 5.1.
- We aim at NN search without special emphasis. The extension of our algorithm to k-NN search will be discussed in Section 5.2.
- All time series are limited by a fixed bound. This bound is used to convert the time series into sets, a process we will discuss in Section 3. Some time series may have points outside the bound. We provide solutions for these occurrences in Section 5.3.

3. STS3: A SET-BASED TIME SERIES SIMILARITY SEARCH ALGORITHM

In this section, we propose our novel method, Set-based Time Series Similarity Search (STS3).

3.1 Overview

To achieve high performance and balance search accuracy and efficiency, we develop a novel method to measure time series. Since in the normalization step, the scaling and shifting factors in the comparison are reduced, if two time series are similar, most of their corresponding segments should be similar. Furthermore, when we treat a time series as a set of points in a plane with time as t-axis and value as x-axis, the point sets corresponding to a pair of similar time series should contain many near points. Thus, we could use the number of near points in the point sets to measure the similarity of the time series.

With the consideration that near points are located in similar areas, we divide the plane into small cells and use the small cells containing points instead of the points themselves to represent the time series for the similarity search. The more cells two time series share, the more near points they have, and, correspondingly, the greater their similarity. In this way, we convert similarity search on time series to search on sets, which is more efficient for computation.

3.2 Set Representation for Time Series

In this section, we discuss how to represent a time series as a set of cells. Before beginning our discussion, however, we first provide these relevant definitions.

Definition 2. *Given a set of time series D with a bound of D. bound(D)=[(t_{min},x_{min}),(t_{max},x_{max})] is the minimum bounding rectangle in t-x plane covering D, in which t_{min}, x_{min}, t_{max} and x_{max} denote the minimal t and x values and the maximal t and x values in all time series in D, respectively.*

Definition 3. *Given a time series S, a point p_i=(t_i,x_i) of S belongs to $cell_j$ or $cell_j$ is the corresponding cell of p_i, if $down_x < s_i \leq up_x$ and $left_t < i \leq right_t$, where the coordinates of the left-down and right-up of $cell_j$ are ($left_t$, $down_x$) and ($right_t$, up_x), respectively.*

Given a time series database D with bound(D), the zone in bound(D) is divided into cells, each of which are assigned an ID. For example, in Figure 1, the bound(D) is the minimal rectangle that contains the two time series. Then it is divided into cells with size $\sigma \times \varepsilon$, numbered from 1 to 6.

The generation of bound(D) is accomplished by scanning all points in D to find t_{min}, t_{max}, x_{min} and x_{max}.

With the generated bound, the cells are set to cover the bound with size $\sigma \times \varepsilon$. Since the goal of cell partition is measuring time series, it is necessary to ensure that similar time series have a greater number of common cells and to reduce the influence of time shift and value shift. While dividing the plane, we need two parameters, ε and σ, to tolerate time shift and value shift, respectively. These two parameters make our method robust.

After division, the cell IDs are assigned to the generated cells. i.e. the ID of a cell according to its row and column location in the grid is as follows:

$$ID = (row - 1) \times COLUMN\_NUM + column \quad (1)$$

where COLUMN_NUM is the number of columns in the grid. For example, in Figure 1, the number of columns is 3, so the ID of the cell in second row and first column of the grid is $(2-1) \times 3 + 1 = 4$.

Given a time series S, our algorithm generates its set representation by computing the cell IDs of all points. For a point (t, x) in S, the row and column of corresponding cell are $row = \frac{(x-x_{min})}{\sigma} + 1$ and $column = \frac{(t-t_{min})}{\varepsilon} + 1$, respectively.

Furthermore, the ID is computed using (1).

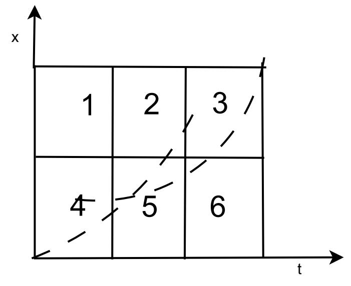


Figure 1: Grid Division

Algorithm 1 describes the procedure for transforming a time series S to the corresponding cell set S'. First, the columns of the grid are computed (Line 2). Then the cell ID of each point in S is computed (Lines 4-6). Finally all cell IDs are added to set S'(Line 7).

For a time series with n points, the time complexity of Algorithm 1 is O($nlogn$) with the set implemented by an order list for the convenience of linear-time intersection.

Algorithm 1: TimeSeriesTranstoSet

```
Input:
    $S$: time series
    $BD$: bound($D$)
    $\varepsilon$,$\sigma$: threshold
Output:
    $S'$: set presentation of $S$
1 $S' \leftarrow \Phi$;
2 $COLUMN\_NUM \leftarrow (BD.t_{max} - BD.t_{min}) \setminus \varepsilon$;
3 for every point $s_i$ of $S$ do
4     $row_i \leftarrow (x_i - BD.x_{min})/\sigma + 1$;
5     $column_i \leftarrow (t_i - BD.t_{min})/\varepsilon + 1$;
6     $number_i \leftarrow$
      $(row_i - 1) \times COLUMN\_NUM + column_i$;
7     $S' \leftarrow S' \cup \{number_i\}$;
8 end
9 return $S'$;
```

3.3 The STS3 Algorithm

When time series are converted into sets, their similarity can be computed with the Jaccard metric [19], which is one of the most popular similarities for sets. It is computed as follows: $Jaccard(S,Q) = \frac{|S \cap Q|}{|S \cup Q|}$.

In this section, we propose a naive version of the STS3 algorithm. As shown in Algorithm 2, this algorithm converts Q into cell set Q' (Line 2) and then scans the cell sets in D to find the one most similar to Q' (Lines 3-9).

The transformation cost from Q to set Q' is $O(nlogn)$. The Jaccard similarity computation cost of the sets is $O(|S|+|Q|)$ using a simple linear merge. Therefore, the time complexity of Algorithm 2 is $O(nlogn + \Sigma_{S \in D}|S| + |Q|)$.

Algorithm 2: Naive_STS3

```
Input:
    $Q$:set representation of query time series
    D:time series database with set representation
    $\varepsilon$,$\sigma$:threshold
Output:
    $ans$: NN results, $ans.TS$ denotes a time series and
    $ans.Jac$ denotes $Jaccard(Q, ans.TS)$
1 $ans \leftarrow null$;
2 $Q' \leftarrow TimeSeriesTranstoSet(S, bound(D), \varepsilon, \sigma)$;
3 for every time series S' in D' do
4     $Jac \leftarrow Jaccard(S', Q')$;
5     if $Jac > ans.Jac$ then
6         $ans.TS \leftarrow Q'$;
7         $ans.Jac \leftarrow Jac$;
8     end
9 end
10 return ans;
```

4. EFFICIENT STS3 ALGORITHMS

In this section, we propose three efficient versions of STS3 algorithms, which are suitable for different scenarios. These are, specifically, the index-based algorithm is suitable for long time series, the pruning-based algorithm for short time series and the approximate algorithm for very long time series. To facilitate concise discussions, we use "time series" to denote the set presentation of time series in this section.

4.1 The Index-based STS3 Algorithm

Consider the procedure of naive STS3. During query processing, the Jaccard similarities of all time series in database D and the query Q have to be computed. Since most time series in D have little intersection with Q, it is unnecessary to compute the similarities for all of them. A natural way to avoid this is to use indexing to select the candidates efficiently. In this section, we propose the index-based STS3 algorithm for this purpose.

we use the inverted list [16] as the index for STS3. In the index-based STS3 algorithm, we use an inverted list IL with each entry as a pair (c, IL_c), where c is a cell and IL_c is the set of time series containing c. Intuitively, $S = \bigcup_{e \in Q} IL_e$ contains the time series intersecting with Q. Thus, the time series of processing can be limited.

Since the size of the intersection of S and Q is the number of times that S appears in all IL_e $(e \in Q)$, we use a counter-array $intersection$ with each entry $intersection[S]$ counting the number of time series in S. The Jaccard similarities are computed according to $intersection$ and the selection is performed correspondingly.

The pseudo code of the index-based STS3 is shown in Algorithm 3. First, we visit every cell e in Q. IL_e is scanned and the corresponding counter in $intersection$ is refreshed for each element in it (Lines 1-5). After all counters are refreshed, the Jaccard similarity is computed directly from $intersection$, and the time series with the maximal similarity is returned as the answer (Lines 6-13).

```
Algorithm 3: Index_STS3
  Input:
    $Q$:query time series
    $D$:time series database
    $IL$:inverted list for database
  Output:
    $ans$: NN results, $ans.TS$ denotes a time series and
    $ans.Jac$ denotes $Jaccard(Q, ans.TS)$
1  for each element $e$ in $Q$ do
2      for each time series $t$ in $IL_e$ do
3          $intersection[t] \leftarrow intersection[t] + 1$;
4      end
5  end
6  $ans \leftarrow null$;
7  for each time series $t$ with non-zero $intersection[t]$ do
8      $nowJac \leftarrow intersection[T]/(T.length + Q.length - intersection[T]$;
9      if $nowJac > ans.Jac$ then
10         $ans.Jac \leftarrow nowJac$;
11         $ans.TS \leftarrow t$;
12     end
13 end
14 return $ans$;
```

4.2 The Pruning-based STS3 Algorithm

In order to accelerate processing, we use a pruning strategy to filter time series. Our goal is to find a time series S with a maximal Jaccard similarity to query Q. Intuitively, when the length of S and Q are fixed, the larger $|S \cap Q|$ is, the larger Jaccard(S,Q) will be. For effective pruning, we require a fast evaluation of a tight upper bound of $|S \cap Q|$.

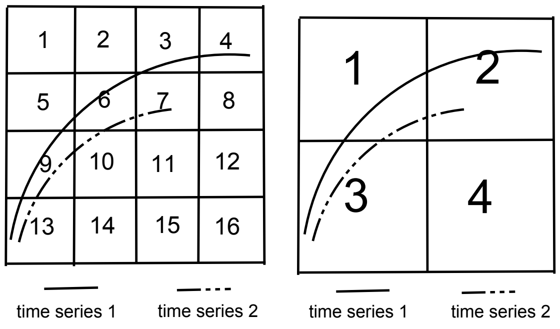
Unfortunately, it is hard to achieve both an accurate and efficient upper-bound evaluation. To balance the accuracy and efficiency, we set a parameter *scale* to make a flexible upper bound.

We divide the plane into zones, whose number is $scale \times scale$. An upper bound is computed for each zone, and the upper bounds of all zones are summed up. The trade-off between tightness and efficiency is tuned with the number of zones.

We use an example to illustrate this strategy. Figure 2(a) shows two time series. The plane is divided into 4 × 4 cells, and the set representations of time series 1 and 2 are (3,4,5,6,9,13) and (6,7,9,10,13), respectively. If we set *scale*=2, then we divide the plane into 2 × 2 zones, as shown in Figure 2(b). Cell 1,2,5,6 belongs to zone 1, cell 3,4,7,8 belongs to zone 2, cell 9,10,13,14 belongs to zone 3, and cell 11,12,15,16 belongs to zone 4. As shown in Figure 2(b), time series 1 has two cells in zone 1 whose cell IDs are 5 and 6 respectively, while time series 2 has one (cell ID 6). Hence the upper bound of their intersection in zone 1 is min(2,1)=1. Similarly, such upper bounds in zone 2, 3 and 4 are min(2,1)=1, min(2,3)=2 and min(0,0)=0, respectively. Hence an upper bound for intersection of the two time series is 1+1+2+0=4.

The whole algorithm is shown in Algorithm 4.

First, the plane is divided into zones. The number of cells in each zone of each time series in the database is recorded in Dzone, which is generated off line, and that of the query is recorded in Qzone (Lines 1-2). Each zone contains a subset of the time series S. We use S_i to denote the subset of S in $zone_i$. Since in $zone_i$, $min\{|S_i|, |Q_i|\}$ gives an upper



(a) 4 × 4 (b) 2 × 2

Figure 2: Two Time Series in the divided Plane

bound of $|S_i \cap Q_i|$, the upper bound of $|S \cap Q|$ is evaluated as the sum of all $min\{|S_i|, |Q_i|\}$ (Lines 6-8). After obtaining the upper bound, we use it for pruning (Lines 9-11). If the time series is not pruned, the Jaccard similarity is computed (Line 12). Finally, we find the NN answer (Lines 13-16).

```
Algorithm 4: Pruning_STS3
  Input:
    $Q$:query time series
    $D$:time series database
    $scale$:parameter
  Output:
    $ans$: NN results, $ans.TS$ denotes a time series and
    $ans.Jac$ denotes $Jaccard(Q, ans.TS)$
1  Dzone[N][i] ← number of elements of $N$ in zone i;
2  Qzone[i] ← number of elements of $Q$ in zone i;
3  $ans$ ← null;
4  for each time series $N$ in $D$ do
5      $LEN\_NQ \leftarrow Q.length + N.length$;
6      for $i$=1:$scale \times scale$ do
7          $upBound$ ← the sum of
           $min\{Qzone[i], Dzone[N][i]\}$;
8      end
9      if $upBound/(LEN\_NQ - upBound) < ans.Jac$
       then
10         prune $N$;
11     end
12     $Jac \leftarrow Jaccard(Q, N)$;
13     if $Jac > ans.Jac$ then
14         $ans.Jac \leftarrow Jac$;
15         $ans.TS \leftarrow N$;
16     end
17 end
18 return $ans$
```

4.3 The Approximate STS3 Algorithm

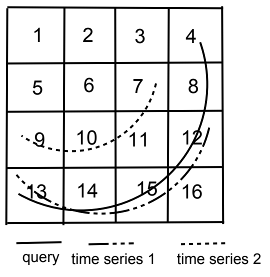
The two algorithms above can obtain accurate results. However, for a large database, their efficiencies may be not sufficient. For such scenarios, we develop the approximate STS3 algorithm. This algorithm is pretty fast but may miss the time series that are most similar.

The approximate algorithm is based on the observation that if two time series are similar in a refined scale, then they are probably similar in a coarse scale. For example, in Figure 2(a), when the plane is divided into 4 × 4 cells, the set representations of time series 1, and time series 2 are (3,4,5,6,9,13) and (6,7,9,10,13), respectively. They are similar, with a Jaccard similarity of $\frac{3}{8}$. When the plane is divided in a coarse scale 2 × 2, as shown in Figure 2(b), both of set representations are (1,2,3), are still similar and even the same with a Jaccard similarity of 1.

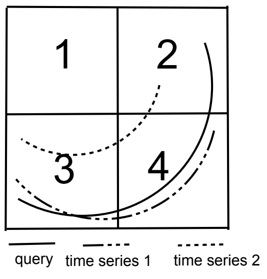
Based on this observation, we can also select candidates in the coarse scale and then refine them in a more refined scale. With this approach, a greater number of similarity computations are performed in the coarse scale. As a result, the query processing is accelerated.

Our method is shown in Algorithm 5. First, the plane is divided into cells in scale from 2 to *maxScale*, an input parameter, and the set representation of each time series in the database in each scale is computed (Lines 1-5; this step is performed offline). A set *searchSet* for the candidate time series is initialized by all time series in the database (Line 6). Then in each scale, we compute the set representation of the query (Line 8) and maintain all time series with maximal similarity to Q, while removing others from *searchSet* (Lines 9-18). If the size of *searchSet* is 1, then we exit the loop (Lines 19-21). Finally, we scan all time series remaining in *searchSet* and return the one with maximal similarity to Q (Lines 24-30).

As the cost of high efficiency, the computation in the coarse scale may miss the time series that are most similar. For example, in Figure 3(a), time series 1 is the time series most similar to query. It has five elements in common with the query, while time series 2 has no elements in common with the query. But when the plane is divided into a coarse scale, as shown in Figure 3(b), time series 2 is more similar to the query since time series 2 shares more cells with the query in this scale. In this case, the time series that is most similar is missed. Fortunately, this situation is rare, since two similar time series have similar cell or zone distributions in both the refined and coarse scales with high possibility.



(a) 2 × 2



(b) 4 × 4

Figure 3: Example of Missing the Most Similar Time Series when Using the Approximate STS3 Algorithm

Algorithm 5: Approximate_STS3

```
Input:
    Q:query time series
    D:time series database
    maxScale:parameter
Output:
    ans: NN results, ans.TS denotes a time series and
    ans.Jac denotes Jaccard(Q, ans.TS)
1  for scale=2:maxScale do
2      for each time series T of D do
3          Ddivision[scale][T] ← set representation of T in
           scale division;
4      end
5  end
6  searchSet ← all time series of D;
7  for scale=2:maxScale do
8      nowQuery ← set representation of Q in scale
       division;
9      for each time series T in searchSet do
10         maxSim ←
           max{Jaccard(Ddivision[scale][T], nowQuery)}
11     end
12     newSearchSet ← null;
13     for each time series T in searchSet do
14         if Jaccard(Ddivision[scale][T], nowQuery) =
           maxSim then
15             newSearchSet.add(T);
16         end
17     end
18     searchSet ← newSearchSet;
19     if newSearchSet.size == 1 then
20         break;
21     end
22 end
23 ans ← null;
24 for each time series T in searchSet do
25     if Jaccard(T, Q) > ans.Jac then
26         ans.Jac ← Jaccard(T, Q);
27         ans.TS ← T;
28     end
29 end
30 return ans;
```

5. EXTENSIONS

In this section, we extend our approach to multi-dimensional time series and k-NN similarity search.

5.1 Multi-dimensional Time Series

When a time series is multi-dimensional, we only need to divide the multi-dimensional space into cells and give a unique ID to each cell to apply our approach. For example, if a time series is two dimensional, then we need three parametersâĂŤα_t, α_x, α_y- to divide t-x-y space into cubes and give each cube a unique ID to transform the time series into set. Thus, the cell ID of the cubic (t, x, y) is $\alpha_x\alpha_y(t-1)+\alpha_x(y-1)+x$.

With such cell IDs, the algorithms in Section 3 and Section 4 can be applied directly.

When a time series is in multi-dimensional, the computation of similarity is still efficient. The reason is that after the transformation into a set, the number of elements is smaller than the original points and the similarity of the sets can be computed in linear time.

However, when a time series is in multiple dimensions, the accuracy may be worse. For the sake of conciseness in this discussion, we analyze how the accuracy may change when a time series is two dimensional. The analysis for multiple dimensions is similar. We attempt to discuss the difference in optimal accuracy that the parameter can bring between the cases in one and two dimensions.

In the case of two dimensions, the impact of α_t on accuracy may not change compared to the impact in one dimension. The reason is that if a point has a shift in the t-axis in the t-x-y space, it probably has the same shift in the t-axis in the t-x plane and the t-y plane. For example, as shown in the experimental results in Section 7.3.1, the accuracies of three dimensions of cricket datasets are similar when α_t changes. Thus the optimal α_t of the t-x plane and the t-y plane may be close or even identical, bringing a good accuracy in t-x-y space.

When one parameter α_{xy} is used to control the shift of the x-axis and y-axis by setting α_x=α_y=α_{xy}, then the accuracy depends on the data and noise distribution. If the x-axis and y-axis have same data and noise distribution, the accuracy may not change much. The reason is that if the axes have the same distribution, the impact of α_{xy} on accuracy in the x-axis and y-axis will be similar. Therefore the α_{xy} that brings good accuracy to one axis may also be a good choice in the other, making the accuracy good in two dimensions. For example, in experiments in Section 7.3.1, the FacesUCR and FaceAll datasets have similar distributions and the results show that the impacts of α_{xy} on them are similar.

However, if the distribution of data and noise in the x-axis is different from that in the y-axis, then the α_{xy} that brings good accuracy to one axis may be a worse choice for the other, affecting the accuracy negatively.

When two parameters are used to control the shift of the x-axis and y-axis, the accuracy may decrease. This is because for the same training dataset, having more parameters usually causes overfitting, which decreases the accuracy.

5.2 k-NN Similarity Search

The extension from an NN similarity search to k-NN is straightforward. During the processing of k-NN similarity search, we use a min heap to maintain the greatest number of k similar time series instead of one. Specifically, with the naive STS3, we find the NN answer by the time series and judging whether its similarity is larger than the current best one (Lines 5-8 in Algorithm 2). For a k-NN query, this is changed to judge whether the similarity of the current time series is larger than the top element of the min heap. If it is, then the top element of the min heap is deleted and the current time series is added to the min heap. In the index-based and pruning-based STS3 Algorithms, the same change is made in Lines 9-12 in Algorithm 3 and Lines 13-16 in Algorithm 4, respectively. For the approximate STS3 algorithm in Algorithm 5, this change is made in Lines 25-28, and Line 19 changes the limiting condition to ($newSearchSet.size \leq k$) since we need to maintain at least k time series for the k-NN search.

5.3 Handling Out-of-Bound Points

In our assumption, all time series are limited in a bound. This assumption is reasonable. For the t-axis, in the real world the length of the time series is limited. For the x-axis, after z-normalization, the value obeys Gaussian distribution in most cases [23]. Therefore, a range large enough, such as (-3,3) will make almost all points fall into the bound [22].

For points outside the bound, which we call "out-points", we propose an effective strategy to handle them in this section. Since the bound is generated when the database D is constructed, out-points appear only in the query or in some updated time series in D. We discuss the handling strategy for each of these instances below.

5.3.1 *For Out-points in the Query*

Since the similarity is mainly determined by the common cells of Q and the time series in D, the cell IDs for the points of Q in bound(D), denoted by Q_{in}, should be uniform with those of D for the computation of common cells. For the points out of bound(D), denoted by Q_{out}, we just need to generate cell IDs different from those in bound(D).

With this in mind, the cell generation is divided into two steps for the points in Q_{in} and Q_{out}, respectively. For Q_{in}, the cell IDs are computed with the formula 1 in Section 3.2. For Q_{out}, which has no intersection with the time series in D, we only need to make the cell IDs different from those in D. Therefore, we generate the bound for Q_{out}, get the corresponding cell IDs for each point by formula 1, and add maxNumber to each ID, in which maxNumber is the maximal cell ID in bound(D). In this way, the cell IDs in Q_{out} are different from those in D.

The pseudo code of the algorithm is shown in Algorithm 6. In this algorithm, first we divide Q into Q_{in} and Q_{out} (Line 2). The set representation of Q_{in} is directly computed (Line 3). Then we transform Q_{out} into its set representation Q'_{out} (Line 4-5) and add maxNumber to each element of Q'_{out} (Lines 6-8). Finally we combine Q'_{in} and Q'_{out} to get the set representation of Q (Line 10).

Algorithm 6: Trans_outQuery_to_Set

```
  Input:
     $Q$:query time series
     $BD$:bound($D$)
     $\varepsilon$,$\sigma$:threshold
  Output:
     $Q'$:set presentation of $Q$
 1 $Q' \leftarrow \Phi$;
 2 Divide $Q$ into $Q_{in}$ and $Q_{out}$;
 3 $Q'_{in} \leftarrow TimeSeriesTranstoSet(Q_{in}, BD, \varepsilon, \sigma)$;
 4 $BQ \leftarrow bound(Q_{out})$;
 5 $Q'_{out} \leftarrow TimeSeriesTranstoSet(Q_{out}, BQ, \varepsilon, \sigma)$;
 6 $maxNumber \leftarrow BD.rows \times BD.columns$;
 7 for every element $t$ of $Q'_{out}$ do
 8     $t \leftarrow t + maxNumber$;
 9 end
10 $Q' \leftarrow Q'_{out} \cup Q'_{in}$;
11 return $Q'$;
```

5.3.2 *For Out-points in the Updated Database*

When time series in the updated database contain out-points, which are called out-TSs, it is straightforward to generate the new bound and recompute the set representations of all time series in database D. Such a strategy makes the updating expensive, however.

Thus, we use a "lazy" updating strategy. To be specific, we use a buffer to store the out-TSs. The bound of the buffer can be changed anytime when adding out-TSs to it, and we make sure that it is larger than the bound(D). When the buffer is full, the time series in the buffer are updated to D

and bound(D) is refreshed from the buffer. Then the cell IDs of time series in D are recomputed.

To ensure the correctness of the query processing, we handle the k-NN search in D first and then use the time series in the buffer to try to refresh the k-NN answer. Given a query Q, we compute its set representation with bound(D) and then handle the k-NN search in D. After that, we recompute the set representation of Q with bound(buffer) and compute the Jaccard similarity for Q and the time series in the buffer to try to refresh the k-NN answer. This procedure provides the final k-NN answer.

How does one decide the buffer size? In general, a small buffer may cause frequent database refreshing, while a large one will cost substantial time when time series in a buffer are used to refresh the k-NN answer. We have the following propositions, which we provide proofs for in Appendix A.

PROPOSITION 1. *The average refreshing cost of adding one time series to the database is proportional to the probability that the time series is out-TS and inversely proportional to the buffer size.*

PROPOSITION 2. *The average query cost of a time series is proportional to the sum of the buffer size and database size.*

The above propositions indicate that increasing the buffer size will increase the query time and decrease the refresh time. Therefore, in order to make the query and refresh times efficient, we suggest setting a small buffer size and a large bound, thus the buffer size and the probability that out-TS appears are small, which saves both query and refresh time.

6. DISCUSSIONS OF STS3

6.1 Efficiency

STS3 uses similarity of sets to replace similarity of time series in order to achieve high efficiency. In this section, we analyze the improvement of efficiency over DTW and LCSS.

Suppose we have two time series S and Q. Consider the procedures for DTW and LCSS restricted by the Sakoe-Chiba band [24]. These can be regarded as finding an optimal way to match points and computing the score of matched points as an evaluation of distance/similarity. In DTW, point S_i and point Q_j can be matched if $|i - j| <= \omega$ while in LCSS, they can be matched if $|i - j| <= \omega$ and $|S_i - Q_j| <= \epsilon$. Hence the points that can match a fixed point are not unique in both DTW and LCSS. Therefore, we need a principle to match all points. This principle is implemented through expensive dynamic programming, making DTW and LCSS costly.

In comparison, in STS3, the element that can match the fixed one is unique (the only one that has the same cell ID to it), so all we need to do is match all elements that can be matched. This procedure is handled by the Jaccard similarity, which can be computed in linear time. Therefore, our approach is theoretically efficient than DTW and LCSS.

6.2 Effectiveness

In this section, we discuss the effectiveness of STS3 and show some suitable scenarios for it.

Consider the accuracy of our approach. The major factor that affects accuracy is cell size. A small cell will lose the ability to hold shift. A large cell may cause different points to fall into the same cell and to be regarded as one point, which would overestimate the similarity. Therefore, when the noise of the data is great, the accuracy of our approach decreases. As a comparison, DTW does not depend on data distribution and has no such trouble.

We note therefore that in some cases the effectiveness of STS3 can be affected. Still, in some scenarios our approach is highly accurate. We discuss some examples below.

As discussed above, cell size is important for accuracy. In order to maximize accuracy, the cell size needs to balance the ability to hold shift and distinguish different time series.

One scenario satisfying the above condition is when time series have a slight shift in axis. If the shift is slight, a small cell size can hold it. Since the cell size is small, STS3 can distinguish different time series properly.

Another suitable scenario is when time series have a large global shift in the t-axis, only a few points have different values, and the values of other points are equal. If there is a large global shift in the t-axis, the cell has to be long enough to hold the shift. Although a long cell may increase the similarity of different time series, only a few points (those with different values) can be used to distinguish the different time series. Therefore, if the width of a cell is small, these points will probably fall into different cells and so distinguish different time series correctly. One specific application of such a scenario is determining similarities in the electricity consumption of different devices [21].

6.3 Determining Values for the Parameters

From the algorithm descriptions and above discussions, it should be clear that our approaches can perform better when the proper parameters are used. In this section, we discuss how to determine the parameters according to the training set.

$\underline{\sigma \text{ and } \epsilon}$ For specific applications, we can use a small amount of data to train the parameter. The series in the training dataset is labeled according to specific requirements, and is usually done manually. As an alternative, time series clustering algorithms such as [2] can be used to label the data. After that, experiments for accuracy are conducted with various parameters σ and ϵ. The parameters with the best accuracy are selected.

$\underline{scale \text{ and } maxScale}$ *scale* (for the pruning-based STS3) and *maxScale* (for the approximate STS3) depends on the data. For *scale*, some queries are processed and the one returning maximal acceleration ratio is chosen. The range of *scale* is 2 to the square root of the time series length. For *maxScale*, it depends on the requirements and the one balancing the efficiency and accuracy in applications is chosen. A *maxScale* of 2 to 5 was usually enough in our tested cases.

7. EXPERIMENTS

To verify the effectiveness and efficiency of the proposed algorithms, we conduct extensive experiments.

All of our experiments are performed on a PC with quad core, 64-bit, 2.6 GHz CPU and 8 GB memory. The algorithms were implemented with JAVA. The operating system is Windows 7. We use runtime to measure the efficiency.

We use UCR datasets in [11] to test the accuracy of STS3. Without special emphasis, all other datasets are generated from an ECG dataset [23] that contains 20,140,000 points. The datasets are generated as follows. Given the number of time series in the database and queries, we first generate the database and then generate corresponding queries. All time series are generated by selecting l points in the order from the ECG dataset one by one and then z-normalizing them, where l is the given length of time series. The length of all time series in the database and queries is equal.

7.1 Preprocessing Time

Before using STS3, the time series need to be transformed into sets, whose cost is $O(nlogn)$. This experiment aims to test the preprocessing time on real datasets.

We use multiple representative data sets, including CBF, CinC_ECG_torso(CET) and ElectricDevices(ED). The information in the datasets and the parameters for STS3 are described in Table 1. The method used is the naive STS3 combined with an early-stopping strategy.

The transformation time for datasets and queries is shown in Table 2, as *offline time* and *online time*, respectively. For comparison, we also list query processing time as *query time*.

The results show that the transformation time of a query is very small compared to the query time. Thus, the major cost of querying is the computation of the similarity of the query and time series in the database. The experimental results show that transformation is valuable and costs little. In all the following experiments, the transformation time of the query is included in the runtime, that is, and we do not list it separately.

Table 1: Datasets Descriptions

Dataset	#query	#series	Length	(σ,ϵ)
CBF	30	900	128	(21,0.18)
CET	40	1380	1639	(76,0.82)
ED	7711	8926	96	(4,0.88)

Table 2: Series Transformation Time

Dataset	Offline Time	Online Time	Query Time
CBF	64	1	30
CET	121	3	35
ED	97	50	19,843

7.2 Comparisons

In this section, we compare our algorithm with ED, DTW, and LCSS in terms of efficiency and effectiveness.

7.2.1 *Efficiency*

To show the efficiency of STS3, we compare it to three representative approaches: ED, DTW and LCSS. We tested some optimization techniques based on these approaches. For DTW, we tested DTW with an LB_improved low boundary [18] and FastDTW [27]. For LCSS, we tested FTSE [22]. We used an early-stopping strategy in all techniques except for FastDTW, which uses a multi-layer filtering technique and cannot be stopped early.

The datasets used are the same as those in Section 7.1. The information on the datasets and the parameters for STS3 can be found in Table 1. The warping length for DTW used comes directly from UCR Archive [11]. FastDTW is implemented with Java-ML library [1]. The radius for FastDTW is set to 0, which gives it optimal speed. The warping length used for LCSS is 10% of the time series length and the ϵ is 0.5, as suggested in [31].

The runtime is shown in Table 3.

From the results, we observe that for these three datasets STS3 is faster than FTSE, FastDTW and LB_improved. For FTSE, before the computation of LCSS, it needs O(n+m) time to build the intersection list. Hence it is slower than STS3. FastDTW is slower, it cannot use the early-stopping strategy. In addition, its time cost is n(8r+14). When r=0, the constant cost is 14, larger than that of STS3 (which is 2). As for LB_improved, the cost of computing the envelope of the time series is $3n$, larger than STS3. Also, some time series cannot be pruned by LB_improved. Then the real and expensive DTW is computed for them. Hence it is slower than STS3 in these datasets.

As for ED, our approach is slightly faster than ED in two datasets. Theoretically our approach is slower than ED since we use linear intersection to compute the Jaccard similarity. However, when we transform the time series into sets, the length of the time series is usually reduced, which makes our approach faster sometimes than ED.

Table 3: Runtime of Different Approaches

Dataset	STS3	ED	FTSE	FastDTW	LB_improved
CBF	31	21	2262	2823	1155
CET	38	203	739753	110448	2106
ED	19893	28423	3506902	4023365	465266

7.2.2 *Effectiveness*

We use the datasets in the UCR Archive [11] to evaluate the accuracy of STS3. Each dataset has two parts, TRAIN and TEST, and each time series in the dataset is labeled. For each time series in TEST, an NN classification is performed on it and its class is computed as its nearest neighbor in TRAIN. The error rate is calculated as follows:

$$errorRate = \frac{number\ of\ wrongly\ classified\ time\ series}{total\ time\ series\ number\ of\ TEST}$$

We run experiments on the UCR Archive datasets with the different parameters ε and σ shown Table 5, where n is the length of the time series. The TRAIN dataset is divided into two parts in order to choose the optimal parameters, according to the principle that the number of time series belonging to same class is equal in each part.

In the interest of conserving, the error rate for all datasets is shown in Table 8 in Appendix B.2. For the convenience of our discussion, we extract some representative results to Table 4. The "tDTW" and "tSTS3" denote the train error rate for DTW and STS3, respectively.

From the results we observe that STS3 is as accurate as ED while DTW outperforms STS3 in 79.5% of the cases. We then examine why STS3 has better or worse accuracy, respectively.

In the suitable scenarios, described in Section 6.2, STS3 usually outperforms ED and sometimes outperforms DTW. For example, shapesAll(SA) and Herring(HE) are all image outline series. The shift in similar time series is slight, so STS3 can handle the shift and distinguish the different time series well. Therefore, STS3 outperforms DTW in HE. However, in SA, STS3 performs worse than DTW. The reason may be that SA has too many classes and this decreases the ability of grid to distinguish different classes.

Other examples include Computers (CP), RefrigerationDevices (RD) and ScreenType (ST). These datasets are the classifications for devices with similar electricity consumption usage patterns [21]. In the datasets, the time series have a global shift, but only a few points have different values. The value of others is 0. Hence, cells with long lengths and narrow widths can distinguish different time series well, achieving high accuracy.

From the results, we observe that in the above scenarios, the performance of STS3 in the Train dataset could be used to predict its performance in the TEST dataset. That is, if STS3 outperforms DTW in the TRAIN dataset, its probably outperform DTW in the TEST dataset.

Table 4: Error Rate in Part Datasets

Dataset	SA	HE	CP	RD	ST	PM	HO
ED	0.248	0.484	0.424	0.605	0.64	0.891	0.199
DTW	0.198	0.469	0.38	0.56	0.589	0.773	0.197
STS3	0.225	0.438	0.316	0.539	0.563	0.873	0.213
tDTW	0.188	0.484	0.248	0.331	0.456	0.785	0.100
tSTS3	0.263	0.161	0.185	0.29	0.425	0.758	0.065

We also find that most datasets in the UCR Archive do not fall within our suitable scenarios. For example, the phoneme(PM) dataset has a large amount of noise. The shift of points in similar time series is large, so the small cells cannot handle noise and the large cells cannot distinguish different time series. As a result, our approach does not perform as well as other approaches, as shown in Table 4.

Another example is the handoutlines(HO) dataset. It contains two classes and the time series belonging to the different classes are very similar. Since the cells cannot hold the shift, STS3 cannot distinguish different time series well in this situation. As a result, STS3 performs worse than ED and DTW, as shown in Table 4.

We also tested accuracy when the workload is relatively fixed. We use the TRAIN and TEST datasets to tune parameters and test the error rate of STS3 and DTW. The results are shown in Table 8 in Appendix B.2.

From these results, we conclude that STS3 outperforms DTW and ED when the workload is fixed. Hence it is also a good choice for that situation.

Table 5: Parameter Settings for STS3

Parameter	min Value	max Value	Step Size
σ	1	0.3*n	1
ε	0.02	1	0.02

7.3 The Impact of Parameters

In this section, we test the impact of parameters on our algorithms. The parameters include σ, ϵ, k, the number of time series in the database ($\#series$), the length of each time series ($length$), the number of queries ($\#query$), $scale$ for the pruning-based STS3 and $maxScale$ for the approximate STS3.

Without explicit explanation, the default settings of σ and ε are 3 and 0.58, which could achieve a large accuracy for the ECG200 dataset in [11]. Other default values are k=1 , $\#series$ =20,000, $length$=500. The parameters $scale$ for the pruning-based STS3 and $maxScale$ for the approximate STS3 vary from 2 to 50 and 2 to 10, respectively, and were chosen because they produce maximal speed-up rate.

7.3.1 *The Impact of σ and ϵ on Accuracy*

To explore the impact of σ and ϵ on accuracy, we fix the σ as the parameter bringing optimal accuracy and then vary ϵ to see how the accuracy changes. After that, we do the same with ϵ.

We observe that the trend depends on applications. As mentioned in Section 6.2, increasing the cell size may overestimate the similarity while decreasing the cell size will decrease the ability to handle shift. Hence, the impacts of σ and ϵ on accuracy mainly depend on the specific data.

For σ, if the data have a similar time shift, then the trends in accuracy are similar. This usually occurs in some multi-dimensional applications. In this case, if a point has time shift in one dimension, the time shift will also happen in other dimensions, so the trends are usually similar. For example, cricket_X, cricket_Y and cricket_Z are three dimensions of people mimicking the 12 gestures of a cricket umpire [7]. The figures for accuracy varying with σ are very similar, as shown in Figure 4(b)- 4(d).

As for ϵ, which controls the shift in the x-axis, it depends on the data and noise. For similar applications, the noise and data distributions are usually similar, so the figures are similar. For example, FacesUCR and FaceAll consist of time series transformed by the face images of 14 students in different situations [4]. Their figures for accuracy varying with ϵ are very similar, as shown in Figure 4(e)- 4(f).

7.4 The Impact of k on Efficiency

We test the efficiency of STS3 when parameter k changes. The results are shown in Figure 4(a). The time increases approximately logarithmically with k.This is because we use a heap to maintain the k-NN answer and the cost of updating heap is only $O(logk)$, which is pretty small.

7.4.1 *The Impact of the Query Number on Efficiency*

In this experiment, we test the impact of the query number on the efficiency by varying $\#query$ from 1000 to 8000.

The impacts on efficiency are shown in Figure 5(a). From the results we observe that the runtime of the pruning-based and the approximate STS3 increases linearly with $\#query$. The approximate STS3 is the fastest one. The runtime of the index-based STS3 is not stable. When the query number is small, it is slow, but when the query number increases, it becomes fast. This is because in the index-based STS3, the cost of accessing entries in the inverted list dominates the runtime. Thus, when $\#query$ is small, a single query with small selectivity may have a large impact on the whole runtime, while a large $\#query$ could reduce the impact of a single query.

The parameters with the maximal speed-up rate for the pruning-based and approximate algorithms are shown in Table 6. From it, we observe that the parameter leading to maximal efficiency is relatively stable and robust for $\#query$. That means when we choose parameters for these two algorithms, we could use some queries as a test and choose the parameters that produced maximal speed-up rate.

7.4.2 *The Impact of Time Series Length on Efficiency*

In this experiment, we test the impact of time series length on efficiency by varying $length$ from 100 to 800. The results are shown in Figure 5(b).

From Figure 5(b) we observe that the approximate STS3

Table 6: Parameter Settings for Pruning and Approximate STS3 with Maximal Speed-up Rate

#query	1K	2K	3K	4K	5K	6K	7K	8K
scale	25	27	25	27	27	25	25	27
maxScale	7	5	6	6	6	7	7	7

is significantly faster than other algorithms in most cases and that the length of time series has little impact on it. Hence it is suitable to handle time series with long lengths. As for the index-based STS3, it is suitable for relatively long time series. The reason is that the number of cells is large for long series. Thus, each entry in the inverted list contains a small series. Accessing them is cheap. As for the pruning-based STS3, its runtime is around linear to *length*. When the time series is short, it runs fast while when the time series is long, it is slower than the index-based STS3. For a long series, effective pruning requires a large *scale*, which leads to a high computation cost. Thus the efficiency is affected. As a result, it is suitable for handling short time series, while the index-based STS3 is more suitable for long time series.

7.4.3 Scalability

In this experiment, we test the efficiency of three algorithms with #*series* ranging from 5000 to 30000. The experimental results are shown in Figure 5(c).

From Figure 5(c), we observe that the runtime of the approximate STS3 is around linear to database size. Because the approximate STS3 usually returns the answer within a few steps, its major cost depends on database size. The runtime of the index-based and pruning-based STS3s does not increase significantly, and even decreases sometimes, as the database size increases. For the index-based STS3, even for large datasets, only a small share of entries in the inverted lists are accessed. For the pruning-based STS3, it is because the pruning strategy could filter a large share of time series for large dataset.

7.4.4 The Impact of *scale* on Efficiency

In this experiment, we test the speed-up rate and pruning rate with different *scale* for the pruning-based STS3.

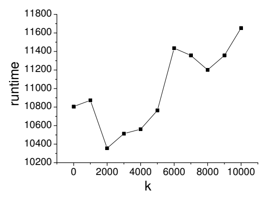
Results are shown in Figure 5(d).

As for the speed-up rate, we observe that it increases slowly at first, and then at some *scale* it reaches the peak, and then decreases slowly. This is because when *scale* is small, it cannot filter many time series and so the speed-up rate is relatively small. When it becomes too large, although the upper bound becomes tighter, the computation of upper bound becomes large, which leads to a slower speed-up rate. Only when *scale* is at some middle value can the pruning-based STS3 balance the efficiency and tightness of the upper bound, which leads to the maximal speed-up rate.

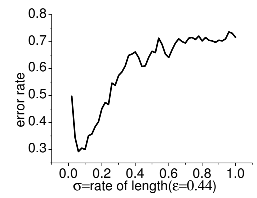
As for the pruning rate, we observe that it increases sharply with *scale* when *scale* is relatively small. After that, it increases slowly and is close to 1. Hence the *scale* bigger than the optimal *scale* cost much but does not bring large increase of pruning rate. As a consequence, the speed-up rate decreases.

7.4.5 The Impact of *maxScale*

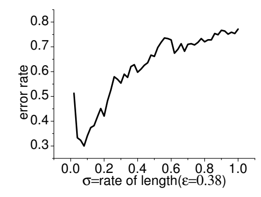
In this experiment, we test the speed-up rate, the compression rate and the error rate of various *maxScale*s for



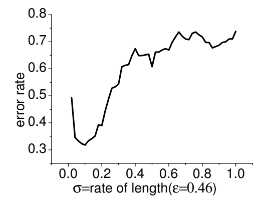
(a) Runtime VS. k



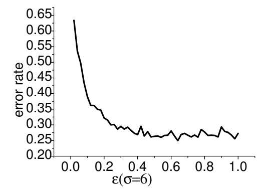
(b) Error rate-σ on cricket_X



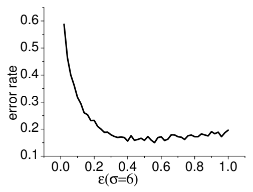
(c) Error rate-σ on cricket_Y



(d) Error rate-σ on cricket_Z



(e) Error rate-ϵ on FaceAll



(f) Error rate-ϵ on FacesUCR

Figure 4: Impact of Parameters on Accuracy-PartA

the approximate STS3. The *compression rate* is defined as the rate of set size after filtering and before filtering. The *Error rate* is defined as follows:

$$ErrorRate = \frac{approxDist - optimalDist}{optimalDist}[28]$$

.

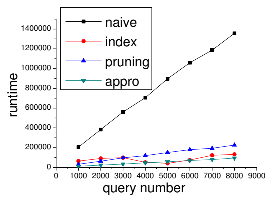
The impact on the speed-up rate is shown in Figure 5(e). From this figure, we observe that the speed-up rate increases very fast at first and reaches a peak and then it decreases slowly. The reason is that many time series are filtered with a small *maxScale*, saving much time. When *maxScale* gets some value, after filtering, the number of time series remaining reaches the threshold. In this case, the growth of *maxScale* cannot filter more time series but cost much. As a consequence, the speed-up rate decreases.

Figure 5(f) shows the compression and error rate. For the compression rate, we observe that it decreases quickly and after a few steps it becomes a constant, which verifies our analysis about the speed-up rate. As for the error rate, it grows fast and then becomes stable. Additionally, the larger speed-up rate and the smaller compression rate are, the larger the error rate is. Another observation is that the error rate is generally smaller than 20%.

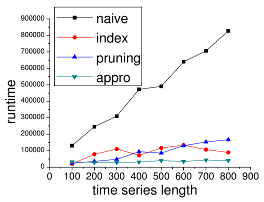
7.4.6 Comprehensive Comparison of Three STS3s

To make a comprehensive comparison of runtime and accuracy for the three versions of the STS3 algorithm, we conduct this experiment.

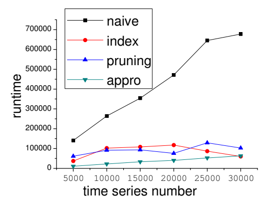
We use three datasets in the UCR Archive: ChlorineConcentration(CC), NonInvasiveFatalECG_Thorax1(NIFE) and ElectricDevices(ED). The information on the datasets and the settings of σ and ϵ are listed in Table 7. The param-



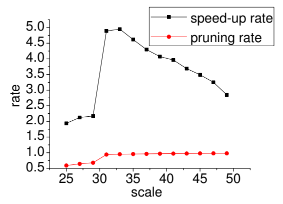
(a) Runtime VS. #*query*



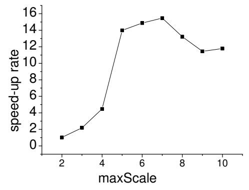
(b) Runtime VS. *length*



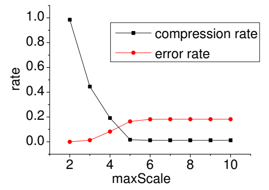
(c) Runtime VS. #*series*



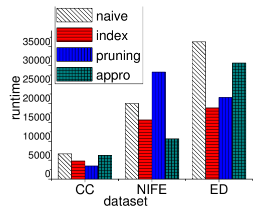
(d) The impact of *scale*



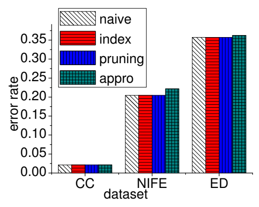
(e) Speed-up rate VS. *maxScale*



(f) Accuracy VS. *maxScale*



(g) Runtime of three STS3



(h) Error rate of three STS3

Figure 5: Experimental Results for the Impact of Parameters-PartB

eter *scale* is set to 6 and *maxScale* is set to 4. Note that each dataset has two sub-datasets and we chose the one containing fewer time series as the query and the other as the database.

Table 7: Datasets of Comprehensive Comparison of Three STS3s

Dataset	#query	#series	length	(σ,ϵ)
CC	467	3840	166	(1,0.28)
NIFE	1800	1965	750	(7,0.14)
ED	7711	8926	96	(4,0.88)

The runtime is shown in Figure 5(g). From the results, we observe that the pruning-based STS3 has a slight advantage in CC, whose time series is relatively short. The approximate STS3 performs better in NIFE, whose time series is long. The index-based STS3 has a slight advantage in ED, whose database is large (since ED has a large number of queries and the time cost in computing set representations in different scales is expensive, which makes the pruning-based STS3 and the approximate STS3 slow).

The results for accuracy are shown in Figure 5(h). From the results, we observe that the accuracy of the approximate STS3 is only a little smaller than the other approaches, which demonstrates that the approximate STS3 misses the best solution only for a few cases.

8. RELATED WORK

Time series similarity search is mainly studied in two respects, time series representations and time series similarity measures with corresponding optimization techniques.

8.1 Time Series Representations

A straightforward representation for a time series is to divide the time series into pieces and use the average values to represent each piece [13, 14]. Piecewise aggregate approximation(PAA) [13] directly divides time series into equal sizes, which may miss some information such as maximums or minimums. [14] proposes an adaptive division method. It transforms time series by wavelet transformation and then divides time series into adaptive lengths according to the principle that minimizes the fitting.

Other representations include Discrete Fourier Transformation (DFT) [12], Single Value Decomposition (SVD) [12], Discrete Wavelet Transform (DWT) [29] and so on.

8.2 Time Series Similarity Measures

L_p-norm [3] is the most straightforward measure for time series. However, it can only handle time series with equal lengths and it is sensitive to noise. DTW is a classic speech recognition approach. Berndt and Clifford [5] introduced it to time-series data mining. It can handle time series with different lengths by allowing comparison one to many points and it is more robust than ED. DTW is a hot study domain. Much work has done on it. Low bounds and index techniques have been proposed to speed up DTW [15, 35, 23, 26]. A linear and approximate DTW algorithm named FastDTW has been proposed [27]. [25] proposed a very efficient algorithm named SPRING to handle subsequence similarity searches based on DTW.

Another idea is inspired by string measures such as the distance based on Long Common Subsequence (LCSS) [30, 33], Edit Distance on Real sequence (EDR) [9] and Edit Distance with Real Penalty (ERP) [8]. [22] proposes an algorithm to evaluate these measures efficiently and accurately. [32] proposes an index structure based on MBR, which supports multiple distance measures.

We also refer readers to [34] and [20]. [34] makes a comparison of different time-series measures while [20] makes a survey about time-series measures and their applications to time-series data mining.

9. CONCLUSIONS

In this paper, we propose a method, STS3, for processing k-NN search of time series. STS3 is faster than most existing methods. In addition, our experiments show that STS3 is more accurate than DTW in some suitable scenarios.

Furthermore, we propose three methods to speed up the k-NN query procedure and test their efficiency by experiment. Future work will include scaling our approach on large datasets by adopting a parallelized mechanism and developing a compression strategy for time series.

Acknowledgments. This paper was supported by NGFR 973 grant 2012CB316200, NSFC grant U1509216,61472099, 61133002 and National Sci-Tech Support Plan 2015BAH10F01. Hongzhi Wang is the corresponding author of this work. Hongzhi Wang and Jinglin Peng contributed to the work equally and should be regarded as co-first authors.

10. REFERENCES

[1] T. Abeel, Y. Van de Peer, and Y. Saeys. Java-ml: A machine learning library. *The Journal of Machine Learning Research*, 10:931–934, 2009.

[2] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah. Time-series clustering–a decade review. *Information Systems*, 2015.

[3] R. Agrawal, C. Faloutsos, and A. Swami. *Efficient similarity search in sequence databases.* Springer, 1993.

[4] M. G. Baydogan. *Modeling Time Series Data for Supervised Learning.* PhD thesis, 2012.

[5] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, 1994.

[6] A. Chakraborti, M. Patriarca, and M. S. Santhanam. Financial time-series analysis: a brief overview. *New Economic Windows*, 48(2):51–67, 2007.

[7] G. S. Chambers, S. Venkatesh, G. A. West, and H. H. Bui. Segmentation of intentional human gestures for sports video annotation. In *MMM 2004.*

[8] L. Chen and R. Ng. On the marriage of lp-norms and edit distance. In *VLDB 2004.*

[9] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD 2005.*

[10] Y. Chen, G. Chen, K. Chen, and B. C. Ooi. Efficient processing of warping time series join of motion capture data. In *ICDE 2009.*

[11] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, and G. Batista. The ucr time series classification archive, July 2015. www.cs.ucr.edu/~eamonn/time_series_data/.

[12] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. *Fast subsequence matching in time-series databases*, volume 23. ACM, 1994.

[13] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases, 2000.

[14] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM SIGMOD Record*, 30(2), 2001.

[15] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and information systems*, 7(3), 2005.

[16] D. Knuth. Retrieval on secondary keys. *The art of computer programming: Sorting and Searching*, 3.

[17] E. Kreyszig. *Advanced Engineering Mathematics (Fourth ed.).* Wiley, 1979.

[18] D. Lemire. Faster retrieval with a two-pass dynamic-time-warping lower bound. *Pattern recognition*, 42(9), 2009.

[19] M. Levandowsky and D. Winter. Distance between sets. *Nature*, 234, 1971.

[20] T. W. Liao. Clustering of time series data-a survey. *Pattern recognition*, 38(11), 2005.

[21] J. Lines, A. Bagnall, P. Caiger-Smith, and S. Anderson. Classification of household devices by electricity usage profiles. In *IDEAL 2011.*

[22] M. D. Morse and J. M. Patel. An efficient and accurate method for evaluating time series similarity. In *SIGMOD 2007.*

[23] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Addressing big data time series: Mining trillions of time series subsequences under dynamic time warping. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 7(3), 2013.

[24] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics Speech & Signal Processing*, 26(1), 1978.

[25] Y. Sakurai, C. Faloutsos, and M. Yamamuro. Stream monitoring under the time warping distance. In *ICDE 2007.*

[26] Y. Sakurai, M. Yoshikawa, and C. Faloutsos. FTW: fast similarity search under the time warping distance. In *PODS 2005.*

[27] S. Salvador and P. Chan. Fastdtw: Toward accurate dynamic time warping in linear time and space. In *KDD workshop on mining temporal and sequential data.* Citeseer, 2004.

[28] S. W. Salvador. *Learning states for detecting anomalies in time series.* PhD thesis, Citeseer, 2004.

[29] Z. R. Struzik and A. Siebes. *Wavelet transform in similarity paradigm.* Springer, 1998.

[30] M. Vlachos, D. Gunopoulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE 2002.*

[31] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *SIGKDD 2003.*

[32] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. J. Keogh. Indexing multidimensional time-series. *VLDB J.*, 15(1), 2006.

[33] M. Vlachos, G. Kollios, and D. Gunopulos. Elastic translation invariant matching of trajectories. *Machine Learning*, 58(2-3), 2005.

[34] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery*, 26(2), 2013.

[35] M. Zhou and M. H. Wong. Boundary-based lower-bound functions for dynamic time warping and their indexing. *Inf. Sci.*, 181(19), 2011.

APPENDIX

A. PROOF OF PROPOSITIONS IN SECTION 5.3.2

First we make some assumptions: (1) time series length is n, (2) the database contains M time series, (3) the buffer size is B (that is, it contains B time series) and (4) the probability of a time series added to the database breaking the bound is p.

Based on these assumptions, we prove the propositions.

Proof of Proposition 1: After adding $1/p$ time series to the database, there may be one series that breaks the bound. Then this series is added to the buffer. After adding about B/p time series the buffer is full and the database needs to be refreshed. The refreshing cost is $O(Mnlogn)$, so the average refreshing cost of adding one time series is $O(pMnlogn/B)$. Thus, Proposition 1 has been proved.

Proof of Proposition 2: The NN query time is $O(Mn)$ and $O((1+2+...+B)n/B) = O(Bn)$ in the database and buffer, respectively. As a result, the average query time for a time series is $O((M+B)n)$. Hence Proposition 2 has been proved.

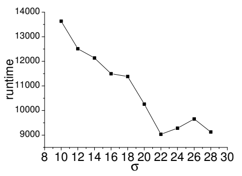
B. SUPPLEMENTARY EXPERIMENTS

B.1 The Impact of σ and ϵ on Efficiency

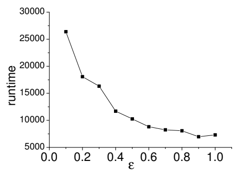
In this experiment, we show the efficiency of the naive STS3 when three parameters change.

To show the impact of σ, we fix ϵ=0.5 and vary σ. The results are shown in Figure 6(a). From the figure, we observe that when σ grows, the runtime of STS3 decreases. This is because a big σ causes more points to locate in one cell and the cell number gets smaller.

We fix σ=20 and vary ε. The experimental results are shown in Figure 6(b). The trend is similar to Figure 6(a). The reason for this phenomenon is the same as that for σ.



(a) Runtime VS. σ



(b) Runtime VS. ε

B.2 The Whole Accuracy Comparison Result

This is the whole accuracy comparison result on UCR Archive datasets. Since our program will be terminated after one hour for each dataset, some datasets have no result and '-' is used to fill the blank. Since the runtime of LCSS in most datasets is over one hour, we do not list it in the table. The results of DTW are obtained directly from [11]. The meaning of each column is as follows:

1. Train-size: the number of time series in the TRAIN dataset.
2. Test-size: the number of time series in the TEST dataset.
3. TS-Len: the length of the time series.
4. #Classes: the number of different classes.
5. ED, DTW, STS3: the error rate of ED, DTW and STS3 when the TRAIN and TEST datasets are used to to train parameters and test the error rate, respectively.
6. DTWfixed, STS3fixed: the error rate of DTW and STS3 when the workload is relatively fixed. That is, the error rate when the TRAIN and TEST datasets are both used to train parameters.

Table 8: Error Rate of different Approaches in UCR Archive

Name	Train-size	Test-size	TS-Len	#Classes	ED	DTW	STS3	DTWfixed	STS3fixed
50words	450	455	270	50	0.369	0.242	0.299	0.235	0.270
Adiac	390	391	176	37	0.389	0.391	0.435	0.358	0.412
Beef	30	30	470	5	0.333	0.333	0.433	0.300	0.200
BeetleFly	20	20	512	2	0.250	0.300	0.200	0.250	0.000
BirdChicken	20	20	512	2	0.450	0.300	0.350	0.250	0.050
Car	60	60	577	4	0.267	0.233	0.300	0.217	0.183
CBF	30	900	128	3	0.148	0.004	0.157	0.002	0.001
ChlorineConcentration	467	3840	166	3	0.350	0.350	0.395	0.350	0.373
CinC_ECG_torso	40	1380	1639	4	0.103	0.070	0.063	-	0.018
Coffee	28	28	286	2	0.000	0.000	0.036	0.000	0.000
Computers	250	250	720	2	0.424	0.380	0.316	0.304	0.196
Cricket_X	390	390	300	12	0.423	0.228	0.308	0.221	0.277
Cricket_Y	390	390	300	12	0.433	0.238	0.326	0.233	0.279
Cricket_Z	390	390	300	12	0.413	0.254	0.300	0.223	0.272
DiatomSizeReduction	16	306	345	4	0.065	0.065	0.105	0.033	0.016
DistalPhalanxOutlineAgeGroup	139	400	80	3	0.218	0.228	0.233	0.200	0.155
DistalPhalanxOutlineCorrect	276	600	80	2	0.248	0.232	0.265	0.225	0.202
DistalPhalanxTW	139	400	80	6	0.273	0.272	0.308	0.273	0.213
Earthquakes	139	322	512	2	0.326	0.258	0.317	0.252	0.180
ECG200	100	100	96	2	0.120	0.120	0.160	0.100	0.070
ECG5000	500	4500	140	5	0.075	0.075	0.075	0.075	0.066
ECGFiveDays	23	861	136	2	0.203	0.203	0.225	0.165	0.096
ElectricDevices	8926	7711	96	7	0.450	0.376	-	-	-
FaceAll	560	1,690	131	14	0.286	0.192	0.267	0.183	0.250
FaceFour	24	88	350	4	0.216	0.114	0.114	0.114	0.057
FacesUCR	200	2050	131	14	0.231	0.088	0.195	0.077	0.150
FISH	175	175	463	7	0.217	0.154	0.246	0.149	0.194
FordA	1320	3601	500	2	0.341	0.341	-	-	-
FordB	810	3636	500	2	0.442	0.414	0.488	-	-
Gun_Point	50	150	150	2	0.087	0.087	0.227	0.027	0.027
Ham	109	105	431	2	0.400	0.400	0.429	0.400	0.276
HandOutlines	370	1000	2709	2	0.199	0.197	0.213	-	-
Haptics	155	308	1092	5	0.630	0.588	0.653	0.571	0.575
Herring	64	64	512	2	0.484	0.469	0.438	0.344	0.266
InlineSkate	100	550	1882	7	0.658	0.613	0.696	0.584	0.587
InsectWingbeatSound	220	1980	256	11	0.438	0.422	0.454	0.422	0.412
ItalyPowerDemand	67	1029	24	2	0.045	0.045	0.126	0.045	0.063
LargeKitchenAppliances	375	375	720	3	0.507	0.205	0.312	0.203	0.304
Lighting2	60	61	637	2	0.246	0.131	0.295	0.082	0.098
Lighting7	70	73	319	7	0.425	0.288	0.301	0.233	0.219
MALLAT	55	2345	1024	8	0.086	0.086	0.135	0.054	0.043
Meat	60	60	448	3	0.067	0.067	0.100	0.067	0.017
MedicalImages	381	760	99	10	0.316	0.253	0.357	0.246	0.316
MiddlePhalanxOutlineAgeGroup	154	400	80	3	0.260	0.253	0.218	0.248	0.200
MiddlePhalanxOutlineCorrect	291	600	80	2	0.247	0.318	0.327	0.247	0.237
MiddlePhalanxTW	154	399	80	6	0.439	0.419	0.456	0.414	0.353
MoteStrain	20	1252	84	2	0.121	0.134	0.167	0.121	0.113
NonInvasiveFatalECG_Thorax1	1800	1965	750	42	0.171	0.185	-	-	-
NonInvasiveFatalECG_Thorax2	1800	1965	750	42	0.120	0.129	-	-	-
OliveOil	30	30	570	4	0.133	0.133	0.200	0.133	0.100
OSULeaf	200	242	427	6	0.479	0.388	0.467	0.380	0.372
PhalangesOutlinesCorrect	1800	858	80	2	0.239	0.239	0.268	0.239	0.260
Phoneme	214	1896	1024	39	0.891	0.773	0.873	-	-
ProximalPhalanxOutlineAgeGroup	400	205	80	3	0.215	0.215	0.190	0.195	0.122
ProximalPhalanxOutlineCorrect	600	291	80	2	0.192	0.210	0.203	0.192	0.186
ProximalPhalanxTW	205	400	80	6	0.292	0.263	0.235	0.260	0.213
RefrigerationDevices	375	375	720	3	0.605	0.560	0.539	0.541	0.435
ScreenType	375	375	720	3	0.640	0.589	0.563	0.560	0.499
ShapeletSim	20	180	500	2	0.461	0.300	0.467	0.300	0.383
ShapesAll	600	600	512	60	0.248	0.198	0.225	0.192	0.198
SmallKitchenAppliances	375	375	720	3	0.659	0.328	0.363	0.307	0.328
SonyAIBORobotSurface	20	601	70	2	0.305	0.305	0.388	0.266	0.200
SonyAIBORobotSurfaceII	27	953	65	2	0.141	0.141	0.297	0.135	0.118
StarLightCurves	1000	8236	1024	3	0.151	0.095	-	-	-
Strawberry	370	613	235	2	0.062	0.062	0.062	0.057	0.046
SwedishLeaf	500	625	128	15	0.211	0.154	0.160	0.152	0.152
Symbols	25	995	398	6	0.100	0.062	0.210	0.050	0.052
synthetic_control	300	300	60	6	0.120	0.017	0.050	0.007	0.027
ToeSegmentation1	40	228	277	2	0.320	0.250	0.180	0.184	0.149
ToeSegmentation2	36	130	343	2	0.192	0.092	0.146	0.092	0.054
Trace	100	100	275	4	0.240	0.010	0.120	0.000	0.000
Two_Patterns	1,000	4,000	128	4	0.090	0.002	0.032	0.000	0.028
TwoLeadECG	23	1139	82	2	0.253	0.132	0.317	0.096	0.134
uWaveGestureLibrary_X	896	3582	315	8	0.261	0.227	0.253	-	-
uWaveGestureLibrary_Y	896	3582	315	8	0.338	0.301	0.323	-	-
uWaveGestureLibrary_Z	896	3582	315	8	0.350	0.322	0.332	-	-
UWaveGestureLibraryAll	896	3582	945	8	0.052	0.034	0.041	-	-
wafer	1,000	6,174	152	2	0.005	0.005	0.009	0.004	0.004
Wine	57	54	234	2	0.389	0.389	0.481	0.389	0.241
WordsSynonyms	267	638	270	25	0.382	0.252	0.346	0.251	0.312
Worms	77	181	900	5	0.635	0.586	0.558	0.530	0.459
WormsTwoClass	77	181	900	2	0.414	0.414	0.392	-	0.298
yoga	300	3000	426	2	0.170	0.155	0.196	0.151	0.162