Technical Report
NWU-EECS-08-01
January 9, 2006

Efficient Similarity Join of Large Sets of Spatio-temporal Trajectories

Hui Ding, Goce Trajcevski, Peter Scheuermann

Abstract

We address the problem of performing efficient similarity join for large sets of moving objects trajectories. Unlike previous approaches which use a dedicated index in a transformed space, our premise is that in many applications of location-based services, the trajectories are already indexed in their native space, in order to facilitate the processing of common spatio-temporal queries, e.g., range, nearest neighbor etc. We introduce a novel distance measure adapted from the classic Frechet distance, which can be naturally extended to support lower/upper bounding using the underlying indices of moving object databases in the native space. This, in turn, enables efficient implementation of various trajectory similarity joins. We report on extensive experiments demonstrating that our methodology provides performance speed-up of trajectory similarity join by more than 50% on average, while maintaining effectiveness comparable to the well-known approaches for identifying trajectory similarity based on time-series analysis.

Northrop Grumman Corp., contract: P.O.8200082518
NSF grant IIS-0325144/003

Keywords: Spatio-temporal trajectory, similarity join

Robust and Fast Similarity Join of Large Sets of Moving Object Trajectories

Hui Ding, Goce Trajcevski and Peter Scheuermann

Dept. of EECS, Northwestern University

2145 Sheridan Road

Evanston, IL 60208, U.S.A.

Abstract

We address the problem of performing efficient similarity join for large sets of moving objects trajectories. Unlike previous approaches which use a dedicated index in a transformed space, our premise is that in many applications of location-based services, the trajectories are already indexed in their native space, in order to facilitate the processing of common spatio-temporal queries, e.g., range, nearest neighbor etc. We introduce a novel distance measure adapted from the classic Fréchet distance, which can be naturally extended to support lower/upper bounding using the underlying indices of moving object databases in the native space. This, in turn, enables efficient implementation of various trajectory similarity joins. We report on extensive experiments demonstrating that our methodology provides performance speed-up of trajectory similarity join by more than 50% *on average, while maintaining effectiveness comparable to the well-known approaches for identifying trajectory similarity based on time-series analysis.*

1 Introduction

The advances in Global Positioning Systems, wireless communication systems and miniaturization of computing devices have brought an emergence of various applications in Location-Based Services (LBS). As a result, there is an increasing need for efficient management of vast amounts of location-in-time information for moving objects. An important operation on spatio-temporal trajectories, which is fundamental to many data mining applications, is the similarity join [17, 6]. Given a user defined *similarity measure*, a similarity join identifies all pairs of objects that are similar based on a join predicate. Efficient similarity joins are especially desirable for spatio-temporal trajectories, because the distance calculation between trajectories is generally very expensive due to the intrinsic characteristics of the data.

Previous research efforts on efficient similarity search in time series data sets mainly follow the GEMINI framework [14, 21, 27, 10]: given a similarity measure on the time series, each trajectory is transformed into a point in a high-dimensional metric space and an index is constructed in the transformed space using the defined measure (or the lower-bounding measure if one is proposed). These *transformed space approaches* have been proved efficient for a large number of different similarity measures in a variety of time series application domains.

However, when it comes to *moving object trajectories* which constitute a special category of time series data, we observe that one can perform the similarity join more efficiently using a different approach. The transformed space approaches incur extra overheads building dedicated index structures and applying trajectory transformations. On the other hand, we can exploit the fact that trajectories are often already indexed in their *native space*, in order to facilitate processing of the common spatio-temporal queries such as range, nearest neighbor, etc. [24, 8, 22]. The main focus of this work is to provide efficient and scalable similarity joins of spatio-temporal trajectories.

Our main contributions can be summarized as follows:

- We introduce a novel distance measure based on the Fréchet distance [2], which is highly effective in identifying spatio-temporal similarity of trajectories.
- We propose lower and upper bounding approximations of the exact distance measure, which are straightforwardly applicable to the spatio-temporal indices and can prune a significant portion of the search space.
- We present an efficient trajectory similarity join in the native space, which combines the distance calculations with incremental accesses to the spatio-temporal indices.
- We conduct extensive experimental evaluations to show the efficiency and effectiveness of our proposed techniques.

The rest of this paper is organized as follows. Section 2 provides the necessary background. Section 3 formally defines our distance metric and the approximation bounds. Section 4 elaborates on our index-based trajectory join framework. Section 6 presents our experimental results. Section 7 reviews related work and concludes the paper.

2 Preliminary

In this section, we introduce the concept of spatio-temporal trajectories, and discuss the existing similarity measures and the indexing of trajectories using R-tree.

2.1 Trajectories and Similarity Measures

We assume that objects move in a two-dimensional space, and that a trajectory Tr is a sequence of points $p_1, p_2, ..., p_i, ..., p_n$, where each point p_i represents the location of the moving object at time t_i, and is of the form (x_i, y_i, t_i), for $t_1 < t_2 < ... < t_i < ... < t_n$. For a given trajectory, its number of points is called the *point length* (p-length) of the trajectory. The time interval between t_1 and t_n is called the *duration* of the trajectory, denoted by ΔTr. The portion of the trajectory between two points p_i and p_j (inclusive) is called a *segment* and is denoted as s_{ij}. A segment between two consecutive points is called a *line segment*.

Several distance measures for trajectories have been proposed in the literature. The L_p*-norms* [14] are the most common similarity measures. For example, given two trajectories Tr_i and Tr_j of the same p-length, one can define the similarity measure based on the Euclidean distances between the corresponding points as: $\mathrm{L}_2(Tr_i, Tr_j) = \sqrt{\sum_{k\in[1,n]} dist(p_k^i, p_k^j)}$, where $dist(p_k^i, p_k^j) = (p_k^i.x - p_k^j.x)^2 + (p_k^i.y - p_k^j)^2$. While L_2 can be calculated in time linear to the length of the trajectories, it is sensitive to noise and is lack of support for local time shifting, i.e., trajectories with similar motion patterns that are out of phase. The *Dynamic Time Warping* (DTW) distance [21] overcomes the above problem by allowing trajectories to be stretched along the temporal dimension, and is recursively defined as: $DTW(Tr_i, Tr_j) = dist(p_1^i, p_1^j)$ $+min(DTW(Rest(Tr_i), Rest(Tr_j)),\ DTW\ (Rest(Tr_i), Tr_j),\ DTW(Tr_i,\ Rest(Tr_j)))$, where $Rest(Tr_i) = p_2^i, ..., p_n^i$. To reduce the impact of the quadratic complexity of DTW on large data sets, a lower-bounding function together with a dedicated indexing structure was used for efficient pruning [21]. Similar to DTW, other distance measures have also been proposed, e.g., the *Edit Distance on Real Sequence* (EDR) [10] and the distance based on *Longest Common Subsequence* (LCSS) [27]. The commonality is that they all follow the transformed space approach, and are not designed to utilize the spatio-temporal indices available in the native space. Recently,Pelekis *et al.* [23] identified several different similarity distance for trajectories, and argued that each of them is more appropriate than the others in different settings.

2.2 R-tree Based Indexing of Trajectories

The R-tree and its variants have been widely used for indexing arbitrary dimensional data [22]. An R-tree is a B+-tree like access method, where each R-tree node contains an array of *(key, pointer)* entries where *key* is a hyper-rectangle that minimally bounds the data objects in the subtree pointed at by *pointer*. In a leaf node, the *pointer* is an object identifier, while in a non-leaf node it is a pointer to a child node on the next lower level.

When indexing spatio-temporal trajectories with the transformed space approach, each trajectory is first transformed into a single point in a high-dimensional (metric) space and a high-dimensional indexing structure is used to index these points. Under this GEMINI framework [14], a high-dimensional R-tree is but one optional index structure.

However, spatio-temporal trajectories can also be indexed in their native space. Several such implementations have been developed in the moving object database literature [24, 8, 22] for processing various spatio-temporal queries. Directly indexing the entire trajectories may introduce large dead space and decrease the discriminating power of the index, hence the general idea is to split a long trajectory into a number of segments, and index the segments [22]. Each leaf node in the R-tree contains a number of 3-dimensional minimum bounding hyper-rectangles (MBR) that enclose the segments generated from splitting, together with unique identifiers that match each segment to its corresponding trajectory. The segments of the trajectories do not have to be of the same length, and a particular leaf node may contain segments from different trajectories. The problem of optimally splitting the trajectories to support efficient data mining has recently been investigated in [3] and is beyond the scope of this paper.

3 Spatio-Temporal Distance of Trajectories

Existing similarity measures are not directly applicable to spatio-temporal indices in the native space. Hence, in this section we introduce our new distance measure based on the classical Fréchet distance [2], a popular illustration of which is given by the *man walking dog* example. The distance between the motion of the man and the dog is the minimal length of leash needed. Spatio-temporal trajectories in real settings consist of series of coordinate points at discrete time stamps, and the location of a moving object between these points is obtained via interpolation when needed. Hence it suffices to define a *discrete* version of the Fréchet distance as follows [12]:

Let $Tr_1 = (p^1_1, \ldots, p^1_n)$ and $Tr_2 = (p^2_1, \ldots, p^2_n)$ be two trajectories. A *coupling* C between Tr_1 and Tr_2 is a sequence $\{(p^1_{a_1}, p^2_{b_1}), (p^1_{a_2}, p^2_{b_2}), \ldots, (p^1_{a_k}, p^2_{b_k})\}$ of distinct pairs such that $a_1 = 1, b_1 = 1, a_k = n, b_k = n$ and for all a_i and b_i we have $a_{i+1} = a_i$ or $a_{i+1} = a_i + 1$, $b_{i+1} = b_i$ or $b_{i+1} = b_i + 1$, i.e., the matching is monotonically non-decreasing. The *length* $\|C\|$ of the coupling C is the maximum *link* of all the pairs in the coupling C, where a link is defined as the Euclidean distance between the two points in the pair. That is, $\|C\| = \max_{i=1,\ldots,k} dist(p^1_{a_i}, p^2_{b_i})$. Finally, the discrete Fréchet distance between two trajectories Tr_1 and Tr_2 is defined as $\delta_{dF}(Tr_1, Tr_2) := \min\{\|C\| : \text{C is a coupling of } Tr_1 \text{ and } Tr_2\}$. An important observation is that exploring all the possible couplings is exhaustive plus by considering all pairs of $(p^1_{a_i}, p^2_{b_i})$ without paying attention to their temporal distances distorts the real spatio-temporal similarity of moving objects. Motivated by this, we use a temporal matching window to constrain the possible point matching and define the *w-constrained discrete Fréchet distance* (*w*DF) as follows:

DEFINITION 3.1. *Given two trajectories Tr_1 and Tr_2, their w-constrained discrete Fréchet distance* $\delta_{wDF}(Tr_1, Tr_2) :=$ $\min\{\|C_w\| : C_w$ *is a w-constrained coupling of Tr_1 and Tr_2, s.t.* $\forall(p^1_{a_i}, p^2_{b_i}) \in C_w \Rightarrow \|p^1_{a_i}.t - p^2_{b_i}.t\| \leq w\}$, *where w is a parameter that determines the limit of the temporal matching window.*

The importance of the temporal dimension is emphasized by the matching window. An idea similar to ours (temporal matching window constraint) has also been used for other similarity measures [28], where a window size of $5\% - 20\%$ of the entire trajectory duration ΔTr is reported sufficient for most application in terms of finding similar trajectories. Further stretching the temporal matching window not only result in longer execution time of the distance function, but may deteriorate the accuracy of the distance measure due to over matching. δ_{wDF} has the following properties:

(1) $\delta_{wDF}(Tr_1, Tr_1) = 0$, (2) $\delta_{wDF}(Tr_1, Tr_2) = \delta_{wDF}(Tr_2, Tr_1)$ and (3) $\delta_{wDF}(Tr_1, Tr_2) \leq \delta_{wDF}(Tr_1, Tr_3) + \delta_{wDF}(Tr_3, Tr_2)$. Hence, we have:

PROPOSITION 3.1. δ_{wDF} *defines a pseudo-metric on the set of spatio-temporal trajectories.*

Due to space limit, the proofs of the claims are omitted from this paper and are presented in [11].

The *w*DF distance can be computed by Algorithm 1 using dynamic programming and has a complexity of $O(\frac{w}{\Delta Tr} n^2)$. However, unlike DTW and EDR, *w*DF is a pseudo-metric and can utilize the triangular inequality for pruning during similarity search [1]. More importantly, *w*DF has led us to the derivation of two approximation distances that provide even greater pruning power, which we discuss next.

3.1 Efficiency and Approximation of *w*DF

For long trajectories, the brute force computation of *w*DF can be costly. We propose two efficient approximations that can bound the exact *w*DF distance and are much faster to compute: one that guarantees a lower-bound and one that guarantees an upper-bound for the exact *w*DF distance, respectively. The proposed approximations make use of a coarser representation of the spatio-temporal trajectories, obtained through splitting a given trajectory into segments and representing them by the sequence of MBRs that enclose the corresponding segments.

Consider two trajectories Tr_1 and Tr_2, each approximated by a sequence of MBRs, e.g., $M_1 = \{MBR^1_1, \ldots, MBR^1_t\}$, $M_2 = \{MBR^2_1, \ldots, MBR^2_s\}$, the *lower-bound coupling* C^L_w between M_1 and M_2 is defined as a monotonically non-decreasing matching between the pairs of MBRs from each sequence. In particular, the link of a pair in the lower-bound coupling C^L_w is defined as the *MinDist* between the two composing MBRs, i.e., the minimum spatial distance between any two points from the respective MBR (c.f. Figure 1 (a)). The length $\|C^L_w\|$ of the lower-bound coupling C^L_w is $\max\{MinDist(MBR^1_{u_i}, MBR^2_{v_i})\}$

Algorithm 1 Computing the δ_{wDF} Distance

```
Input: Trajectory $Tr_1 = (p_1^1, \ldots, p_n^1)$, $Tr_2 = (p_1^2, \ldots, p_n^2)$, matching window $w$
Output: $\delta_{wDF}(Tr_1, Tr_2)$
 1: $dF(1:n, 1:n) \Leftarrow -1.0$ // initialize $n$ by $n$ array $dF$
 2: return $ComputeWDF(n, n)$

 3: ComputeWDF(i, j)
 4: if $dF(i,j) > -1.0$ then
 5:     return $dF(i,j)$
 6: else if $\|p_i^1.t - p_j^1.t\| > w$ then
 7:     $dF(i,j) := \infty$
 8: else if $i == 1$ and $j == 1$ then
 9:     $dF(i,j) := dist(p_1^1, p_1^2)$
10: else if $i > 1$ and $j = 1$ then
11:     $dF(i,j) := max(ComputeWDF(i-1,1), dist(p_i^1, p_1^2))$
12: else if $i = 1$ and $j > 1$ then
13:     $dF(i,j) := max(ComputeWDF(1,j-1), dist(p_1^1, p_j^2))$
14: else if $i > 1$ and $j > 1$ then
15:     $dF(i,j) := max(min(ComputeWDF(i-1,j), ComputeWDF(i-1,j-1), ComputeWDF(i,j-1)), dist(p_i^1, p_j^2))$
16: return $dF(i,j)$
```

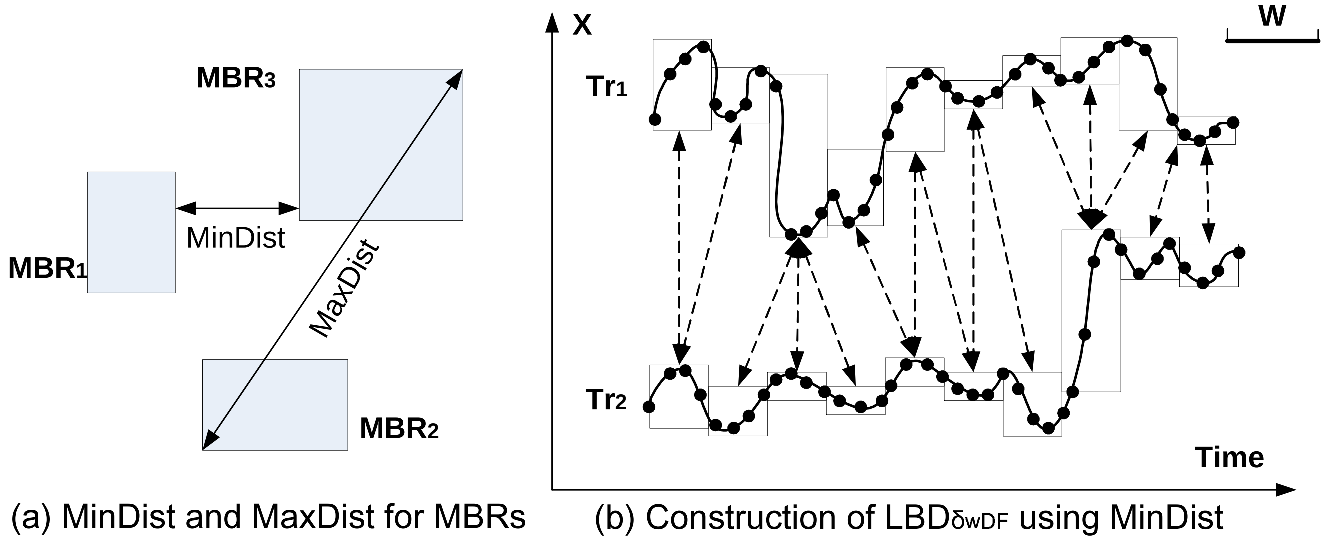

MBR3
MBR1
MinDist
MaxDist
MBR2
(a) MinDist and MaxDist for MBRs
X
W
Tr1
Tr2
Time
(b) Construction of LBDδwDF using MinDist

Figure 1: *Bounding the exact wDF distance with MBRs*

where $u_1 = v_1 = 1$, $u_k = t$ and $v_k = s$. The w-constraining condition is specified over the time intervals of MBRs. Assuming that MBR_i^1 and MBR_j^2 enclose segment $(p_{i_1}^1, ..., p_{i_k}^1)$ and $(p_{j_1}^2, ..., p_{j_m}^2)$ respectively, they will be considered as a possible pair in a w-constrained coupling only if $\exists p_{i_a}^1, p_{j_b}^2$ *s.t.* $\|p_{a_i}^1.t - p_{b_i}^2.t\| \leq w\}$. Formally:

DEFINITION 3.2. *Given two sequences of MBRs M_1 and M_2 for trajectories Tr_1 and Tr_2 respectively, the lower-bound distance of Tr_1 and Tr_2 is: $LBD_{\delta_{wDF}}(M_1, M_2) := \min\{\|C_w^L\|$: C_w^L is a $w-$constrained lower-bound coupling of M_1 and $M_2\}$.*

Similarly, we define an *upper-bound coupling C_w^U* on the two sequences of MBRs M_1 and M_2, where the link of a pair is defined as the *MaxDist* between the two composing MBRs, provided that the temporal w-constraint holds:

DEFINITION 3.3. *Given two sequences of MBRs M_1 and M_2 for trajectories Tr_1 and Tr_2 respectively, the upper-bound distance of Tr_1 and Tr_2 is: $UBD_{\delta_{wDF}}(M_1, M_2) := \min\{\|C_w^U\|$: where C_w^U is a $w-$constrained upper-bound coupling of M_1 and $M_2\}$.*

The construction of $LBD_{\delta_{wDF}}$ between two trajectories from their MBRs is illustrated in Figure 1 (b), and the relationship of these two distance bounds and the exact distance is given by the following:

THEOREM 3.1. *Given two trajectories Tr_1 and Tr_2, and the corresponding sequences of MBRs, M_1 and M_2, that approximate them, for any matching window w the following holds: $LBD_{\delta_{wDF}}(M_1, M_2) \leq \delta_{wDF}(Tr_1, Tr_2) \leq UBD_{\delta_{wDF}}(M_1, M_2)$.*

We note that Theorem 3.1 applies to arbitrary trajectory splitting strategies, and the problem of optimally splitting the trajectories is beyond the scope of this paper [3]. For simplicity, we assume in the rest of this paper that the trajectories are uniformly split into segments of equal length l. From their definitions, $LBD_{\delta_{wDF}}$ and $UBD_{\delta_{wDF}}$ can be computed using the same dynamic programming algorithm for computing wDF, except that the MinDists/MaxDists between MBRs are used instead of Euclidean distance between points. However, the amount of distance computation involved can be greatly reduced because of the coarser representation. This can be illustrated by using the *warping matrix* concept [25] to describe the relevant coupling between two trajectories. The values in the cells of the warping matrix denote the distances between the corresponding matching MBRs/points. Figure 2 (b) shows the warping matrix between the MBRs of the two trajectories Tr_1 and Tr_2, and Figure 2 (c) shows the warping matrix between the actual points of the two trajectories. Intuitively, calculating $LBD_{\delta_{wDF}}(UBD_{\delta_{wDF}})$ or wDF is the process of finding a *path* [25] from the lower-left corner to the upper-right corner that minimizes the maximum value over all cells on the path. The amount of computation for $LBD_{\delta_{wDF}}(UBD_{\delta_{wDF}})$ is significantly less because of the reduced matrix size. Instead of computing the exact δ_{wDF} distance each time, we compute the $LBD_{\delta_{wDF}}$

and $UBD_{\delta_{wDF}}$ with only $O(\frac{w}{\Delta Tr}\frac{n^2}{l^2})$ time, and exclude unqualified candidates, substantially reducing the number of the *w*DF distance calculations. Theorem 3.1 ensures that the MBR-based pruning will not introduce any false negtives.

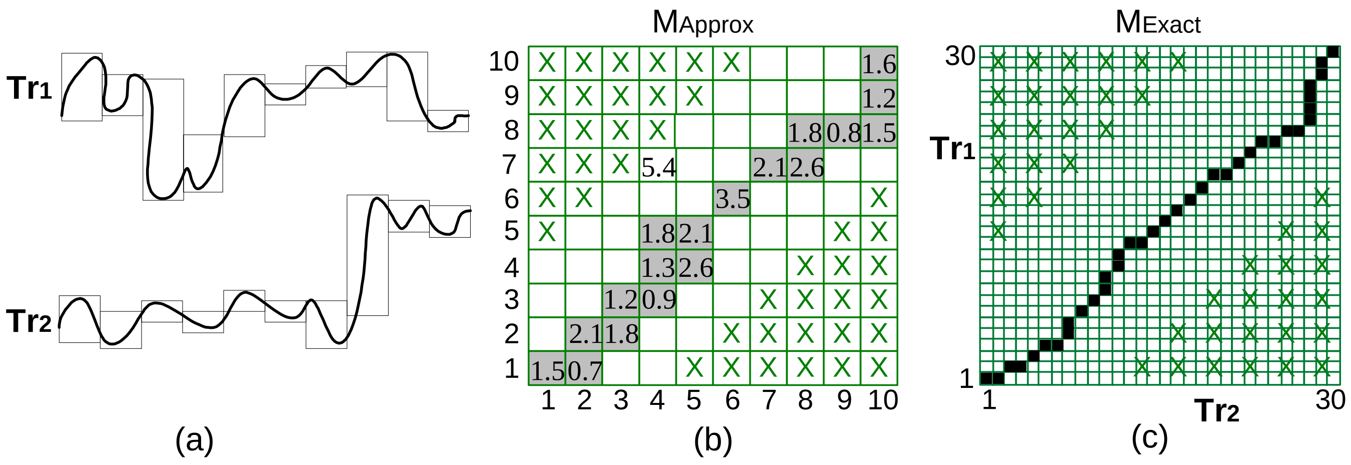


Figure 2: *Warping matrices for calculating $LBD_{\delta_{wDF}}$ /$UBD_{\delta_{wDF}}$ and wDF: X cells are automatically excluded by the temporal matching window, grey cells are potentially useful and black cells are on final path*

Moreover, we can do even better with the approximation distances by further limiting the search space, using an idea similar to *early abandoning* [30]. Consider the warping matrix M_{Approx} in Figure 2 (b) for calculating $LBD_{\delta_{wDF}}$ between Tr_1 and Tr_2. Initially, it only consists of "x" cells and white cells and all the white cells are assigned a value of ∞. We access MinDists between the MBRs in ascending order, and update the values of the corresponding cells, e.g., cell$(1,2)$=0.7, cell$(8,9)$ = 0.8, cell$(3,4)$ = 0.9, ...(grey cells). After each update, we invoke the dynamic programming algorithm to compute the $LBD_{\delta_{wDF}}$. At first the algorithm will quickly return ∞. After updating a cell (i,j), if we obtain the first path connecting the two corners in the matrix, then this is the optimal path (since any path formed later will use a cell updated after cell (i,j) and will have a larger MinDist value than cell (i,j)). Consequently, the $LBD_{\delta_{wDF}}$ distance is equal to the MinDist value of cell (i,j). At this point the distance calculation has been completed and the rest of the cells can be pruned. In the example of Figure 2 (b), the critical cell after which we could find the path is $cell(6,6)$ and as a result, $LBD_{\delta_{wDF}}(Tr_1,Tr_2)$ equals 3.5. Note that cells such as $(7,4)$ are never considered at any time since its value is greater than 3.5. This important observation is formalized as follows:

THEOREM 3.2. *When calculating $LBD_{\delta_{wDF}}$($UBD_{\delta_{wDF}}$) between two trajectories Tr_1 and Tr_2, if the pairwise distances between the MBRs are incrementally accessed in ascending order of their MinDists (resp. MaxDists), the number of MinDists (resp. MaxDists) calculated is minimum.*

Theorem 3.2 requires that the MinDists/MaxDists are sorted in ascending order, which may incur an extra overhead. However, the key observation is that such an ordering can be naturally obtained by maintaining a priority queue while accessing the MBRs in the R-tree [17]. The worst case complexity is still bounded by $O(m^2)$, where m is the number of MBRs in each trajectory. However, we observed in our experiments that in practice significant speed up can be achieved, since not all m^2 cells of the warping matrix need to be evaluated. In addition, although both $LBD_{\delta_{wDF}}$ and $UBD_{\delta_{wDF}}$ can be calculated using Theorem 3.2, in practice we only invoke dynamic programming once to calculate $LBD_{\delta_{wDF}}$. The path for $LBD_{\delta_{wDF}}$ can then be used to calculate an upper-bound on $UBD_{\delta_{wDF}}$, which in practice approximates the actual $UBD_{\delta_{wDF}}$ distance very well.

3.2 Improving *w*DF Against Noise

While *w*DF distance can be speeded up by applying $LBD_{\delta_{wDF}}$ and $UBD_{\delta_{wDF}}$ to existing spatio-temporal indices, one of its weakness is that it is sensitive to noises/outliers in the trajectories. To see this, consider the definition of *w*DF distance (c.f. Definition 3.1) where the similarity of trajectory Tr_1 and Tr_2 is essentially determined by the minimum of the *maximum distance* between the two sets of points from the respective trajectory [18], i.e., the link with the maximum value for a given coupling. Hence, when there is noise at a single point, two otherwise close trajectories may be made arbitrarily distant away. DTW also suffers from the same problem, however, the impact is mitigated since DTW uses the sum of distances from all pairs of points between two trajectories which can in effect smoothen the data from single noise/outlier. EDR and LCSS distance measures are generally recognized as being stronger against noises/outliers, because they compute a similarity score based on a matching threshold, and can leave noises/outliers unmatched to avoid their negative impact on the similarity computation.

In this section we propose a novel technique, called *time-aware adaptive median filtering* (TAMF), to improve the robustness of *w*DF. The purpose of TAMF is to effectively identify outliers in the data and exclude them from the distance computation. Furthermore, TAMF is very lightweight and only poses a slight overhead compared to the $O(n^2)$ complexity of computing *w*DF. Existing adaptive median filtering algorithms usually make certain global assumptions about the data in order to detect noise candidate [16, 9]. However, if we consider a moving object trajectory as a stochastic process over time, its statistical characteristics is usually time-varying. Hence, using global assumptions about the data may lead to inaccurate noise detection and removal. TAMF exploits the statistics about the most recent past of the moving object, in order to improve the precision when picking noise candidates.

The basic idea of TAMF is as follows: when calculating the *w*DF, instead of using only the Euclidean distance between any pair of points in a coupling, we also take into account the Euclidean distances among their neighboring points. Recall that $dist(p_i^1, p_j^2)$ denote the Euclidean distance between the ith and the jth points of trajectory Tr_1 and Tr_2 respectively, let $S_{i,j}^h$ be a window of size $h \times h$ centered at (i, j), i.e., $S_{i,j}^h = \{(k,l) : \|k-i\| \leq h, \|l-j\| \leq h\}$. We compute $dist(p_k^1, p_l^2)$ for all the pairs of points in $S_{i,j}^h$, and sort these distance values and. We use the median of this sorted list to determine the “distance” between the two points p_i^1 and p_j^2. As an example, suppose that we have two one dimensional trajectories: $P = [(t_1,1),(t_2,2),(t_3,100),(t_4,2),(t_5,1)]$ and $Q = [(t_1,1),(t_2,2),(t_3,3),(t_4,2),(t_5,1)]$. When calculating the “distance” between the third points of P and Q with a median filter window size 1, we obtain a list of distance values $(0,0,0,0,1,1,97,97,98)$. Using the median of this list, we determine that the “distance” between the second points of P and Q is 1 instead of 97 and successfully remove the impact of the outlier. We call the wDF distance defined using TAMF the *median wDF* (m-wDF).

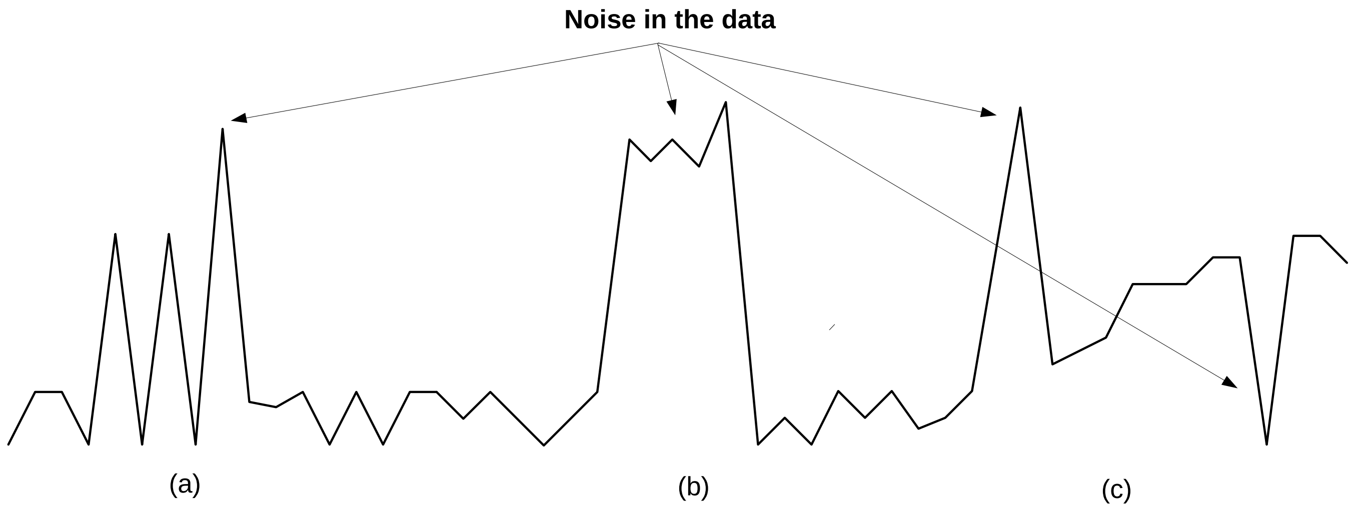


Figure 3: *Noise and outlier patterns that may exist in the moving object trajectories*

While the outlier in the example above can be easily filtered out with a simple median filter, the situation is much more complicated in trajectories from real applications. TAMF considers three aspects of the noise that may exist in the data:

- A series of bursts of impulse noises (c.f. Figure 3.a): in order to remove such noises we need to adaptively adjust the median filter window size: while large filter window size may lead to unnecessary overhead, small window size may result in treating noisy values as correct output of the filter.
- Noisy patches that span several data samples (c.f. Figure 3.b): in order to detect such outliers we need to correctly detect the size of the noise as well as its rising and falling edges, and adjust the median filter window size accordingly.
- Time-varying amplitude of the impulse noise (c.f. Figure 3.c): this has been overlooked by previous works on adaptive

median filtering and may affect the precise detection and removal of the previous two types of noises. As we have explained, trajectories are usually time-varying stochastic processes and their characteristics develops over time, e.g., the amplitude of the impulse noise are not uniformly the same for the entire trajectory. For example, suppose we are tracking vehicles on road network, a data sample indicating a speed of 80mph when the object moves in urban area is a likely candidate for outliers, whereas when the object moves on an interstate highway, a data sample indicating a speed of 30mph may suggest a (negative) impulse noise. This illustrates that in order to correctly identify noise candidates from trajectories, we need to incorporate the dynamic local statistics of the trajectory data.

TAMF achieves time-awareness using a sliding window based approach. When computing median *w*DF, we maintain a sliding window over the recent past outputs produced from the median filter. Suppose the size of the sliding window is N, these values are simply stored using a FIFO queue in the main memory and are dynamically updated with every new output from the median filter. We maintain three statistics about the queue, the maximum value in the queue s_{max}, the minimum value in the queue s_{min} and the average of the difference between any two neighboring values v_{avg}. These statistics are used to detect the presence of noise in the data.

The algorithm first detects whether there is a impulse in the median filter output in the while loop. If there is no impulse in the median filter output, the algorithm then detect whether the center distance value itself is corrupted by noise/outlier, using the minimum and maximum distance values as well as the average speed values maintained by the sliding window. If there is impulse in the median filter output, we increase the size of the median filter until h_{MAX}. The value of h_MAX is determined by the noise occurrence probability [16, 9] and is usually set to 7. If there is impulse output from the median filter after the window size has reached the maximum, the algorithm then examine whether there is a large noise patch with more than one data sample. The maximum size of a noise patch that can be removed is specified by k_{MAX}, and is usually set to 3. Impulses that span more than 3 data samples are treated as intrinsic patterns of the data and are not removed. The complexity of TAMF is dependent on the maximum size of the median filter window. As for the sliding window Our empirical study shows that a sliding window size of ?? is most effective. The maintenance of the sliding window takes $O(1)$ time and $O(N)$ space.

M-*w*DF can still be bounded using the MBRs in the native space, however, it is no longer a pseudo-metric, i.e., it does not satisfy the triangular inequality which can be used for pruning. Hence, it is slower to perform similarity search or similarity join using m-*w*DF, compared to using *w*DF. This actually presents a trade-off between the effectiveness and efficiency of the similarity measure. We provide a more detailed experimental evaluation in Section 6.

Algorithm 2 Computing m-*w*DF Distance

```
Input:  Trajectory $Tr_1 = (p^1_1, \dots, p^1_n)$, $Tr_2 = (p^2_1, \dots, p^2_n)$, warping matrix $df[n][n]$ where $df[i][j] = dist(p^1_i, p^2_j)$
Output: $\delta_{m-wDF}(Tr_1, Tr_2)$
 1: for each pair of points $(p^1_i, p^2_j)$ do
 2:     let $s_{min}$, $s_{max}$ and $v_{avg}$ denote the current sliding window statistics
 3:     initialize filter window $S^h_{i,j}$ size $h$ to 1
 4:     while $h \leq h_{MAX}$ do
 5:         sort the $df$ elements within the window, let $x_{min} \leftarrow$ minimum distance value within $W$, $x_{max} \leftarrow$ maximum distance value within $W$, $x_{med} \leftarrow$
            median value within $W$
 6:         if $x_{med} > x_{min}$ and $x_{med} < x_{max}$ then
 7:             $no\_impulse \leftarrow true$
 8:             break
 9:         else
10:             increase $h$
11:     if $no\_impulse$ is $true$ then
12:         if ($df[i][j] > s_{min}$ and $df[i][j] < s_{max}$) and ($df[i'][j'] - df[i][j] < \lambda \times v_{avg}$ or $df[i][j] - x_{med} < \lambda \times v_{avg}$) then
13:             keep the original $df[i][j]$ value
14:         else
15:             $df[i][j] \leftarrow x_{med}$
16:     else
17:         for $k$ from 1 to $k_{MAX}$ do
18:             if $df[i-k][j-k] - df[i][j] < \lambda \times v_{avg}$ or $df[i][j] - x^{-k}_{med} < \lambda \times v_{avg}$ then
19:                 there is a noise patch of size $k$, increase the filter size
20:             else
21:                 increase k
22: $dF(1:n, 1:n) \Leftarrow -1.0$ // initialize $n$ by $n$ array $dF$
23: return $ComputeWDF(n,n)$
```

4 Index-Based Trajectory Join Under *w*DF

In this section, we present our framework for spatio-temporal similarity join of trajectories under the *w*DF distance measure. Assuming that each trajectory is uniformly split into segments that are indexed by a 3-dimensional R-tree, we describe the nearest neighbor join algorithm, and present several important variants.

4.1 Nearest Neighbor Join

The *nearest neighbor join* retrieves pair of trajectories, where the second trajectory in a result pair is closer to the first one than any other trajectory from the second data set, using *w*DF distance as the similarity measure.

The main inputs to the algorithm are the two trajectory sets $\mathbb{S}_1$ and $\mathbb{S}_2$, indexed by disk-based R-trees R_1 and R_2, respectively. The algorithm accesses both trees in a manner similar to the incremental distance join [17]: descending from their roots simultaneously, and concurrently assigning the segments from the second set to the closest trajectory from the first set. The algorithm maintains a set of pairs, where the first item in each pair is from R_1 and the second one from R_2. Each item can be either a node of the R-tree, or a leaf node entry, i.e., the MBR of a particular segment. The set of pairs is managed by a priority queue, such that the dequeue operation will always remove the pair with the smallest *MinDist*. In addition to the priority queue, the algorithm utilizes two data structures. The first is the *warping matrix directory* (WMD) that maintains an entry for each trajectory from $\mathbb{S}_1$, storing a list of incomplete $LBD_{\delta_{wDF}}$ and $UBD_{\delta_{wDF}}$ warping matrices between that trajectory and a trajectory from $\mathbb{S}_2$. Each entry in WMD also maintains an *upper bound distance*, which is the maximum possible distance allowed to become an answer candidate. In addition, each entry has a flag that indicates whether the nearest neighbor for this particular trajectory has been found. The second structure is the *candidates table* (CT) that stores for each trajectory from $\mathbb{S}_1$ its candidate answers in a sorted list, in ascending order of the $LBD_{\delta_{wDF}}$.

The join process is illustrated in Algorithm 3. After initializing the relevant data structures, the main body of the algorithm is a while loop that continuously processes the next pair dequeued:

- When both elements in the pair are MBRs of trajectory segments (line 4-17), it first checks whether the corresponding entry from WMD is *complete* and if so, simply discards the pair from further consideration. Otherwise, it performs early abandoning by checking whether the MinDist between the two MBRs is less than the upper bound distance (line 7). Then the relevant warping matrices in the corresponding WMD entry are updated and the algorithm examines whether the update generates a complete path in the $LBD_{\delta_{wDF}}$ warping matrix. If so, the $LBD_{\delta_{wDF}}$ and $UBD_{\delta_{wDF}}$ distances are calculated. $LBD_{\delta_{wDF}}$ is used to insert a new entry into the candidates table, and $UBD_{\delta_{wDF}}$ is used to update the upper bound distance of the entry. Finally, if the pair's MinDist is greater than the entry's upper bound distance, this WMD entry is flagged *complete* and the relevant warping matrices are discarded.
- When only the first element in the pair is the MBR of a segment (line 18-22), the algorithm checks whether the corresponding entry in WMD is flagged *complete*, and if so the pair is discarded since it (and any new pair formed by further descending the R-tree) may not produce a better answer than the existing candidate. Otherwise the second node is

Algorithm 3 Index-Based Trajectory Join

```
Input: R-tree $R_1$, $R_2$; Trajectory set $\mathbb{S}_1$, $\mathbb{S}_2$; temporal matching window $w$
    /* filtering stage */
 1: priority queue $Q.ENQUEUE(R_1.root, R_2.root, 0)$
 2: while ! $Q.ISEMPTY$ do
 3:     $(e1, e2, mindist) \Leftarrow Q.DEQUEUE$
 4:     if both $e1$ and $e2$ are segment MBRs then
 5:         $Tr_1 \Leftarrow$ trajectory of $e1$, $Tr_2 \Leftarrow$ trajectory of $e2$
 6:         if $WMD[Tr_1]$ flagged $incomplete$ then
 7:             if MinDist($e1$,$e2$) $\leq E.upper\_bound\_dist$ then
 8:                 insert MinDist, MaxDist of $e1$, $e2$ into $WMD[Tr_1]$
 9:                 if a path exists for the MinDist warping matrix between $Tr_1$ and $Tr_2$ then
10:                     compute $LBD_{\delta_{wDF}}$ and $UBD_{\delta_{wDF}}$ between $Tr_1$, $Tr_2$
11:                     if $UBD_{\delta_{wDF}} < E.upper\_bound\_dist$ then
12:                         $E.upper\_bound\_dist \Leftarrow UBD_{\delta_{wDF}}$
13:                     insert $Tr_2$ and $LBD_{\delta_{wDF}}$ into $CT[Tr_1]$
14:             else
15:                 set flag of $WMD[Tr_1]$ as $complete$
16:         else if $WMD[Tr_1]$ flagged $complete$ then
17:             discard the pair $(e1, e2)$
18:     else if $e1$ is segment MBR then
19:         if $WMD[Tr_1]$ flagged $complete$ then
20:             discard the pair $(e1, e2)$
21:         else
22:             $expandElement(e1, e2, Q)$
23:     else if $e2$ is segment MBR then
24:         $expandElement(e2, e1, Q)$
25:     else if both $e1$ and $e2$ are node then
26:         $expandBalancedElement(e1, e2, Q)$
    /* refinement stage */
27: for every entry $Tr_i$ in CT do
28:     compute $\delta_{wDF}(Tr_i, Tr_j)$ for each candidate $Tr_j$ until the nearest neighbor is found
```

expanded by calling the function *expandElement*. *ExpandElement* expands one of the input nodes by pairing each one of its children with the other input element if they are temporally within w, and inserts the resulting pair into the priority queue Q.

- When only the second element in the pair is the MBR of a segment (line 23-24), the first node is expanded by calling *expandElement*, with *elem*1 and *elem*2 exchanged.
- When a pair of nodes is processed (line 25-26), the algorithm chooses to expand one of the nodes by calling *expandBalancedElement* which tries to keep the balance of the depth when descending the two trees. The node to expand is the one with a shallower depth or with a larger area if both nodes are at the same depth [17].

After the while loop terminates, the refinement step is performed on the CT using the triangular inequality of *w*DF for pruning (line 27-28). For every entry of the candidates table, we examine the candidate trajectories in ascending order of their $LBD_{\delta_{wDF}}$ and calculate the exact *w*DF distance, until either all the candidate trajectories have been examined, or the next $LBD_{\delta_{wDF}}$ is greater than the largest computed *w*DF distance value.

4.2 Variants of the Similarity Join

Algorithm 3 for nearest neighbor join requires minor modifications to calculate other similarity joins among trajectories in our framework.

- **k-nearest neighbor join** (kNN join) [6]: A kNN join finds for each trajectory from $\mathbb{S}_1$ its k nearest neighbors from $\mathbb{S}_2$ in terms of the *w*DF distance. For each trajectory from $\mathbb{S}_1$, after the first k candidates are added to the candidate table, the minimum of their $UBD_{\delta_{wDF}}$ is used as the upper bound. We continue to add new candidates as long as their $LBD_{\delta_{wDF}}$ distances are smaller than the current upper bound, and update the upper bound with the new tighter $UBD_{\delta_{wDF}}$ if necessary. In the refinement stage, we calculate the exact *w*DF distance for every candidate and select the k trajectories with the smallest distance values.
- **Range Join** [6]: A range join finds for each trajectory from $\mathbb{S}_1$ all the trajectories from $\mathbb{S}_2$ that are within a given *w*DF distance of it. For this extension, we simply retrieve for each trajectory in $\mathbb{S}_1$ all the candidates whose $LBD_{\delta_{wDF}}$ is less than the given distance threshold during the filtering stage, and refine the answers using the exact *w*DF distance.

We also note that our framework can straightforwardly support the time interval join [5], where the kNN or range predicate is defined using only some portions (segments) of trajectories within a specified time interval of interest. In this case we retrieve only the index nodes and leaf node entries that intersect with the given time interval from the same index structure.

5 Applying Similarity Join for Efficient Clustering of Trajectories

In this section we demonstrate how our proposed techniques can be used to support efficient clustering of spatio-temporal trajectories. Various clustering algorithms have been proposed in the literature [18, 19, 13]. These algorithms unanimously require frequent distance computation between objects and scaling to large trajectory sets may be costly. However, their efficiency can be significantly improved when equipped with our trajectory join framework. In the following, we first discuss the application of our methodology to the partition-based algorithm *k-medoids* [18], and then we outline the similar benefits in the context of *Chameleon* [19] and *DBSCAN* [13].

5.1 K-medoids Clustering for Trajectories

The k-medoids algorithm is a partition-based method that divides the trajectories into k groups and iteratively exchanges objects between them until a predefined function, which evaluates the quality of the clusters, reaches a local optimum. The most expensive operations in k-medoids are the initial assignment of the trajectories to their nearest medoids, and the iterative reassignment of the trajectories after randomly replacing an existing medoid with a new one. However, these operations actually correspond to a nearest neighbor join between the set of trajectories and the set of medoids [6], hence we can make use of the nearest neighbor join algorithm presented in the previous section to expedite the process. The modification of the standard k-medoids algorithm [18], which utilizes spatio-temporal inexing and the *w*DF distance, is described as follows:

```
1. Randomly pick $k$ trajectories as initial medoids and create an R-tree on the medoid set;
2. Call Algorithm 3 with the R-trees of trajectories & medoids as input;
3. Compute the quality measure $Q_C = \sum_{1<i<k} \sum_{Tr_j \in C_i} \delta_{wDF}(m_i, Tr_j)$;
4. Randomly pick trajectory $m_{replace}$ to replace a randomly selected medoid $m_i$ in the medoids R-tree;
5. Call Algorithm 3 with the new medoids R-tree;
6. Update $Q_C$ after the replacement and commit the change if it decreases;
7. Repeat 4-6 until no further improvement in $Q_C$.
```

The crucial benefit of our approach is received in step 5 of the above algorithm, where many trajectories remain closest to the same medoid from the previous assignment. Hence, we can *incrementally* update the nearest medoid only when necessary, instead of running Algorithm 3 from the scratch every time. The trajectories affected by the medoid replacement were either assigned to the cluster whose medoid is replaced, or assigned to other clusters but are now closer to the new medoid. All other trajectories remain in the same clusters. By identifying such trajectories at an early stage during the replacement process, we can prune a large number of distance calculations. This is achieved by modifying Algorithm 3 to cater for step 5, as described in Algorithm .

Algorithm 4 Incremental Reassignment

```
... ...
/* refinement stage */
$m_i \Leftarrow$ the medoid to be replaced by $m_{replace}$
Hash table $H \Leftarrow$ trajectories that were in the cluster of $m_i$
for every entry $Tr_i$ in CT do
    if $Tr_i \in H$ or candidates contain $m_{replace}$ then
        compute $\delta_{wDF}(Tr_i, Tr_j)$ for each candidate $Tr_j$
```

We utilize an additional hash table H to store the trajectories that were in the replaced cluster. In the refinement stage, we only calculate the distance for an entry in CT if: either the trajectory is in H, or the replaced medoid is one of the candidates. These changes ensure that the distance calculation for the trajectories that remain in their old cluster will not be repeated, and can significantly speed up the reassignment process which is dominated by the expensive distance calculation in the refinement stage.

5.2 Hierarchical and Density-Based Clustering

- **Index-based Chameleon**: The Chameleon algorithm [19] explores graph partitioning and dynamic modelling to achieve high quality hierarchical clustering. The most expensive step, when applying Chameleon to spatio-temporal trajectories, is to build the hyper-graph where each vertex of the graph represents a trajectory, and there exists an edge between two vertices (trajectories) if one trajectory is among the k nearest neighbors of the other. However, the step of building this k-nearest neighbor graph can be transformed to that of performing a k-nearest neighbor self-join, and we can utilize the techniques presented in Section 4.2 to expedite the process.

- **Index-based DBSCAN**: To cluster trajectories using density-based algorithms such as DBSCAN [13], the critical task is to identify dense regions using the input parameters ε and $MinPts$, by performing a range query for each trajectory. However, instead of executing the individual range queries separately, we can apply the range join algorithm proposed in Section 4.2 to concurrently perform the range queries for all the trajectories in the data set, where the join distance equals ε. The result of this join can be stored in a matrix for look-up during the execution of the classic DBSCAN algorithm.

Note that for trajectories whose $UBD_{\delta_{wDF}}$ distance is smaller than ε, we know that these trajectories are in the ε neighborhood for sure and the exact distance computation can be completely avoided.

6 Experimental Results

In this section, we empirically evaluate the efficiency and effectiveness of our proposed techniques.

We have implemented our similarity join framework in Java. All our experiments are executed on a PC with a Pentium IV 3.0 GHz CPU and 1 GB of memory. To evaluate the efficiency of the proposed algorithms, we use the network-based traffic generator [7] and produce moving object trajectories based on the road networks of Oldenburg (OB) and San Francisco (SF). To obtain some quantitative observations about the potential use of our framework for data mining applications, we use the *w*DF distance for classification in data sets provided by the UCR Time Series Collection [20]. We index the trajectories with a 3-dimensional R-tree and uniformly split each trajectory into segments. The resulting segments are then inserted into the R-tree, where each data entry in the leaf node contains one segment. The page size is set to 4KB, and an LRU buffer with 1000 pages is used. Unless stated otherwise, the *w* window size is set to 15% of the entire trajectory duration.

6.1 Efficiency of Similarity Join

Although our results are independent of the trajectory splitting strategy adopted, before evaluating the performance of our similarity join framework, we need to determine the trajectory splitting size for our data sets in order to remove its impact from further experiments. Increasing the number of splits implies tighter bounds but may also increase the costs for calculating them, whereas decreasing the number of splits deteriorates filtering effectiveness. We perform a nearest neighbor join for 200 trajectories generated from the road networks of OB and SF, with 400 and 1000 points respectively. We vary the number of points contained in each segment from 5 to 200 and the results are shown in Figure 4. Based on the results, we fix the number of points in each segment to 20 in the following experiments.

With the uniform split model, we then evaluate the tightness of the two distance bounds. We use the road networks of OB and SF to generate varying number of trajectories, and randomly pick one trajectory to perform a nearest neighbor query, using the two distance bounds for pruning. We record the total number of times the exact *w*DF distance is calculated, and divide this number by the total number of trajectories in the data set. The result ratio is shown in Figure 4. It can be observed that using our approximate distance bounds, we only need to perform less than 2% of the *w*DF distance calculation.

Next we evaluate the efficiency and the scalability of our trajectory join algorithm. Due to the limited space, we focus on the nearest neighbor join only. The next two sets of experiments compare the efficiency of three different approaches: (1) our framework using the *w*DF distance, (2) the metric-based join [29] (essentially a sequential scan over the entire data set but uses the triangular inequality for pruning as much as possible) with *w*DF as the distance metric and (3) similarity join on

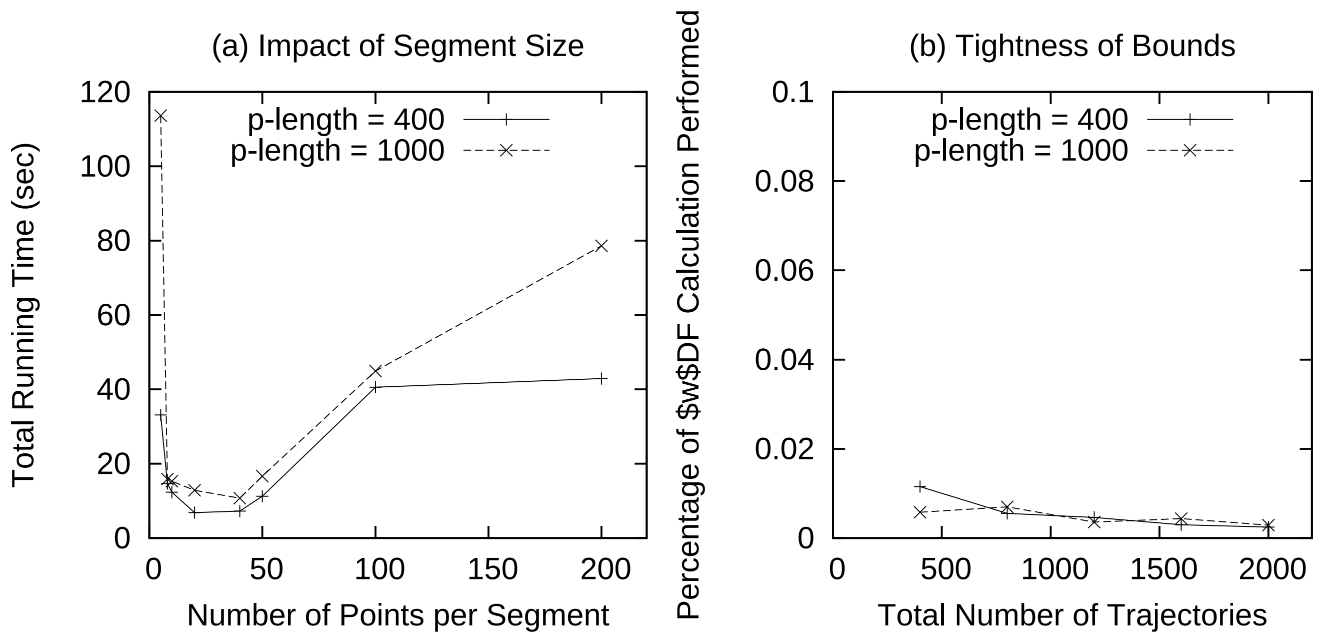


Figure 4: *Impact of Segment Size and Tightness of Bounds*

DTW distance with lower-bound indexing [21], as a representative of the transformed space approach. For the DTW based approach, we implement the join as a batch of similarity search queries where each query is a trajectory from the first data set that is used to search for its nearest neighbor in the second data set. We use the same parameters, e.g., the number of points in each segment/piece-wise approximation, the R-tree parameters and splitting strategy, etc. in both our framework and the DTW implementation. We also take into account the time it takes for approach (1) and (3) to build the index structure.

Our first set of experiments reports the total running time of the nearest neighbor join as a function of the number of trajectories. Figure 5 compares the performance of the three approaches on trajectories generated from road networks of OB and SF, respectively. Each OB trajectory contains 400 points and each SF trajectory contains 1000 points. We observe that our join framework clearly outperforms the metric-based join, yielding a speed-up of up to 10 times. Furthermore, our approach scales well to large trajectory sets since the running time grows linearly with respect to the number of trajectories, whereas the running time for metric-based join grows quadratically. While the DTW based approach also outperforms the metric space based approach by a large factor, it is on average more than 2 times slower than our approach. This discrepancy becomes even larger on the SF data set. The main reason is that when the number of points in each trajectory increases, the dimensionality of the index structure used to index the trajectories in the transformed space, i.e., the index of the DTW distance, also grows. This will reduce the selectivity of the index and admit more false positives that will need to be eliminated with the expensive DTW distance calculation. Increasing the number of points per segment/piece-wise approximation can

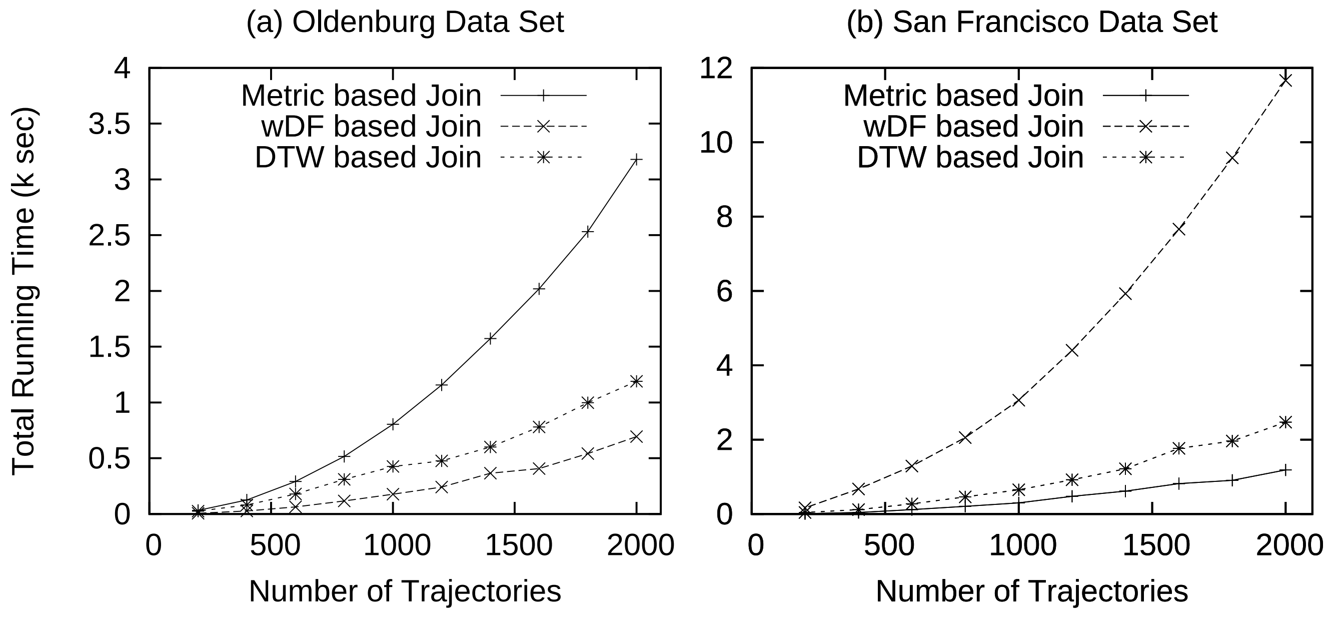


Figure 5: *Scaling with Number of Trajectories*

not solve the problem, as it will yield a wider bounding envelop used by DTW and loosen the lower-bounds [21]. Working in the native space, our approach does not have this problem of dimensionality. When the number of points per trajectory increases, it only increases the total number of segments and the size of the R-tree structure. However, the extra accesses to the indices are paid off by the reduction of false positives because of the lower/upper-bounds.

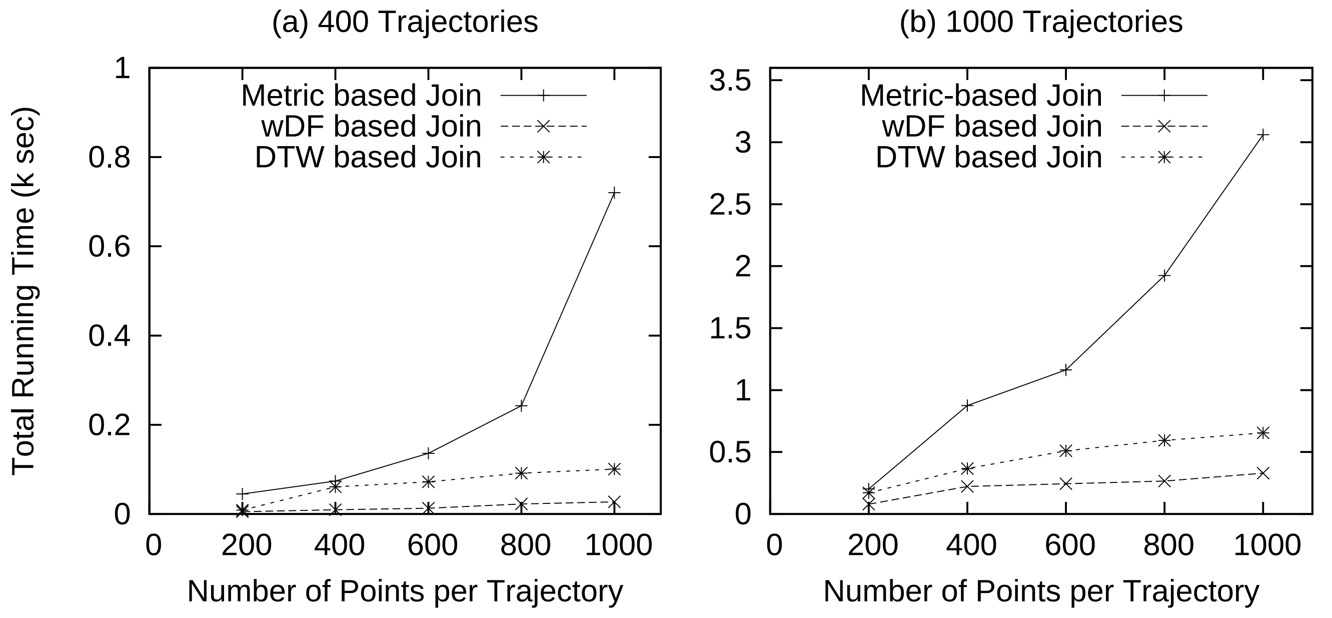


Figure 6: *Scaling with Trajectory Length*

Our next set of experiments investigates the similarity join performance with respect to the number of points per trajectory. We fix the number of trajectories in OB and SF to 400 and 1000 respectively and increase the number of points in each trajectory. From Figure 6, we can observe that our approach scales very well with the number of points per trajectory, and consistently delivers a speedup of more than 2 with respect to the DTW based approach. The speedup increases as the number of points per trajectory grows from 200 to 1000.

6.2 Effectiveness of *w*DF

In order to evaluate the effectiveness of our proposed similarity measure, we use a one-nearest neighbor classification algorithm as suggested by Keogh *et al.* on 20 different data sets [20]. These data sets cover various application domains, e.g., robotics, industry, botany etc. For each group of data, a training set and a testing set are provided together with the correct cluster labels. We compare the classification error ratio of *w*DF against that of L_2-norm and DTW from [20], as shown in Table 1.

The classification error ratio of *w*DF is obtained by finding the optimal warping window size for the purpose of this comparison (and so does DTW), and the percentages in parentheses indicate the ratio of matching window size w to the trajectory duration. We observe that the classification error rates yielded by *w*DF are clearly superior to L_2-norm, and is comparable with DTW (*w*DF wins in 7 data sets, ties in 2 data sets and loses the rest). This is because while *w*DF can handle local time shifting, it is more sensitive to noise than DTW. We note that using a uniform window size of 15% yields only slightly different results [11].

Since *w*DF is sensitive to noise, one can alleviate this problem by apply some filtering technique similar to EDR and LCSS [27, 10] when determining *w*DF. We have considered using a median filter to protect the warping matrix from noise. Our preliminary experiments indicate that the median filter substantially improves the effectiveness of *w*DF for classification purposes, as illustrated in Table 2. However, there are two important issues that we need to address: (1) choosing the optimal filter size, or properly adjusting it (for adaptive algorithms); (2) median filters need not yield metric distance, which may slow down the refinement step of Algorithm 3. We will focus on these issues in the future work.

6.3 Clustering Speed-up

To demonstrate the application and benefits of our framework, we evaluate the performance of the three clustering algorithms presented in Section 5, equipped with our index-based trajectory join. We compare our approach only with the metric-based approach where the triangular inequality is used to prune the distance computation. The trajectories used in

Data set	L_2-norm	DTW	*w*DF (*w*)
Synthetic Ctrl.	0.12	0.017	0.02 (13%)
Gun-Point	0.087	0.087	0.027 (0.7%)
CBF	0.148	0.004	0.027 (14.8%)
Face(all)	0.286	0.192	0.142 (2.3%)
OSU Leaf	0.483	0.384	0.421 (3%)
Swedish Leaf	0.213	0.157	0.182 (4.7%)
50 Words	0.369	0.242	0.301 (4.4%)
Trace	0.24	0.01	0 (4.4%)
Two Patterns	0.09	0.0015	0.0045 (14.5%)
Wafer	0.005	0.005	0.0045 (13.2%)
Face(Four)	0.216	0.114	0.307 (2.3%)
Lightning-2	0.246	0.131	0.229 (2.8%)
Lightning-7	0.425	0.288	0.329 (3.8%)
ECG	0.12	0.12	0.13 (1%)
Adiac	0.389	0.391	0.381 (6.25%)
Yoga	0.17	0.155	0.143 (2.1%)
Fish	0.217	0.16	0.181 (1.7%)
Beef	0.467	0.467	0.467 (2.3%)
Coffee	0.25	0.179	0.0714 (2.8%)
OliveOil	0.133	0.167	0.167 (0.5%)

Table 1: *Effectiveness of wDF Distance*

the experiments are generated from the SF road networks, with 600 points each.

We first compare the effects of incremental reassignments in k-medoids. We fix the number of trajectories to 6400 and increase the number of clusters from 2 to 64. Figure 7 (a) shows that the incremental reassignment can speed up each iteration over the naive reassignment by up to 3.5 times. The speed-up increases with the number of clusters since the number of trajectories that need to be reassigned becomes smaller.

Next, we study the speed-up achieved by implementing the three clustering paradigms using our trajectory similarity join framework. In all the experiments, we fix the number of clusters to be 10. For k-medoids, we force the reassignment process to terminate after 20 iterations in each run. The results are shown in Figure 7 (b): the similarity join can dramatically speed

Data set	DTW	*w*DF	*w*DF with median filter
CBF	0.004	0.027	0.011
OSU Leaf	0.384	0.421	0.393
Swedish Leaf	0.157	0.182	0.166
50 Words	0.242	0.301	0.273
Two Patterns	0.0015	0.0045	0.00075
Lightning-7	0.288	0.329	0.29
Beef	0.467	0.467	0.433

Table 2: *Effectiveness of Median Filter on wDF*

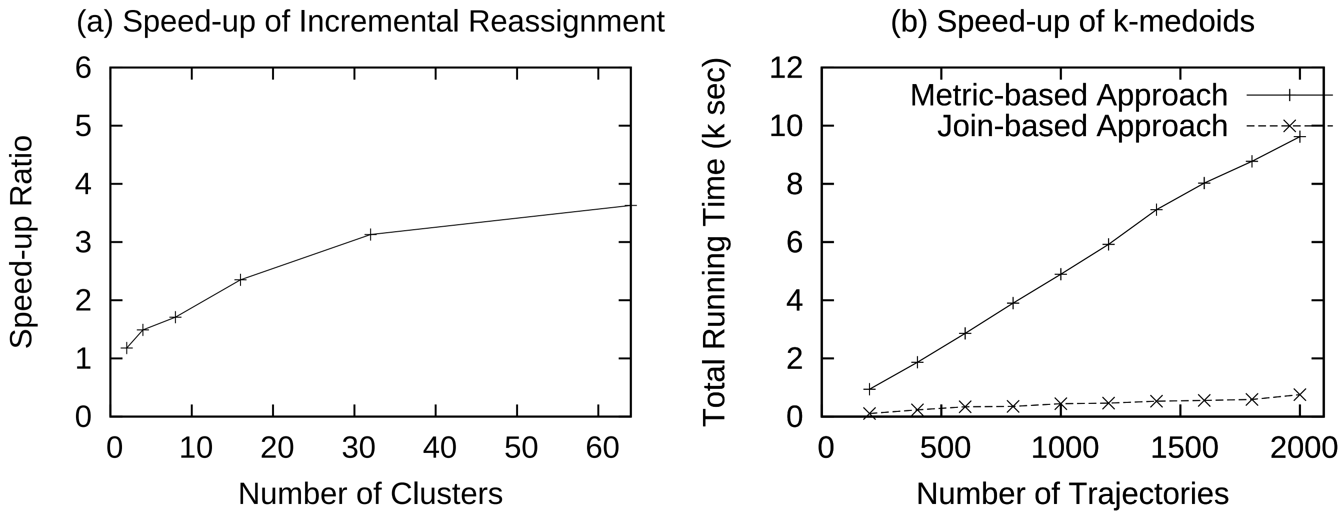
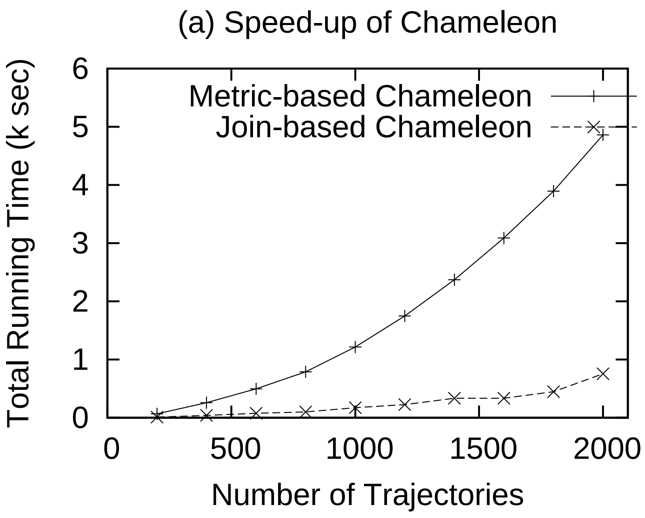


Figure 7: *Speed-up of k-medoids*

up the k-medoids algorithm compared to the metric-based approach by more than an order of magnitude. For Chameleon, we use a 5-NN join to build the hyper-graph for agglomotive clustering. For DBSCAN, we set ε to ten times the maximum speed of the moving objects and *MinPts* to 3. The results for Chameleon and DBSCAN are shown in Figure 8. We observe that due to the expensive cost to compute the *w*DF distance, the kNN join and the range join dominate the running time of Chameleon and DBSCAN, by 99.5% and 99% respectively (not shown in figure). The performance of Chameleon can be improved by more than 5 times, and DBSCAN by more than 8 times. Furthermore, the tendency of improvement keeps growing with the number of trajectories.



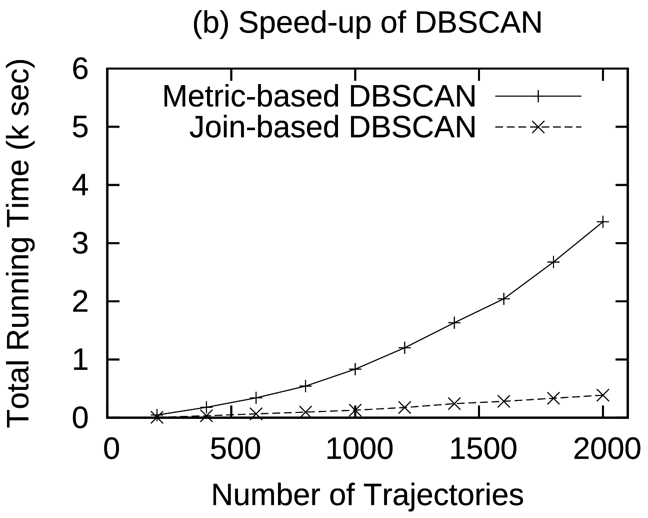


Figure 8: *Speed-up of Chameleon* & *DBSCAN*

7 Related Work & Concluding Remarks

The problem of turbo-charging data mining process by similarity join has been investigated in [6] for low-dimensional data. In this work, we focus on joining spatio-temporal trajectories and the main goal is to utilize the index structure to prune a large number of expensive distance calculation which dominates the join process. A trajectory join using the L_p-norms and a specialized index structure was presented in [5]. However, the approach can not be straightforwardly extended to support different spatio-temporal similarity join.

In [28] the indexing of LCSS and DTW using MBRs of trajectory segments is explored. However, the proposed lower-bound distance are calculated in conjunction with a query sequence, which makes the efficient extension to similarity join questionable. The issue of what is a semantically appropriate distance measure for trajectory similarity is addressed in [23]. [15] considers similarity search for trajectories using spatio-temporal indices and proposes a novel distance measure, however the work does not address the similarity join of trajectories. We note that our idea of lower-bounding *w*DF using mindists of MBRs is similar to that proposed in [26], and the idea of early abandoning proposed in [30] is similar to our Theorem 3.2. The key contribution of our work is the combination of similarity joins and spatio-temporal indices in the native space of moving object trajectories, which yields an increased efficiency.

In this paper, we introduced a new similarity measure *w*DF for location-related time series data, based on Fréchet distance. In order to compute the distance efficiently, we proposed two approximations for effective upper/lower-bounding. We then combined these approximations with spatio-temporal indices in the native space for pruning, and presented a similarity join framework under our distance measure that supports a number of different similarity join variants. Our experimental results have demonstrated the efficiency and scalability of our proposed technique in the context of moving object trajectories, and verified the effectiveness of our distance measure.

One immediate extension of this paper is to improve the robustness of our distance measure against outliers in the data. Another interesting avenue of future work is to extend our approach towards more general types of motion and richer representations of the trajectory models.

References

[1] Lei Chen 0002 and Raymond T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.

[2] Helmut Alt and Michael Godau. Computing the fréchet distance between two polygonal curves. *Int. J. Comput. Geometry Appl.*, 5:75–91, 1995.

[3] Aris Anagnostopoulos, Michail Vlachos, Marios Hadjieleftheriou, Eamonn J. Keogh, and Philip S. Yu. Global distance-based segmentation of trajectories. In *KDD*, pages 34–43, 2006.

[4] Axel Mosig. *Efficient Algorithms for Shape and Pattern Matching*. Ph.D thesis, 2004.

[5] Petko Bakalov, Marios Hadjieleftheriou, Eamonn J. Keogh, and Vassilis J. Tsotras. Efficient trajectory joins using symbolic representations. In *Mobile Data Management*, pages 86–93, 2005.

[6] Christian Böhm and Florian Krebs. The *k*-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 2004.

[7] Thomas Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[8] V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing large trajectory data sets with SETI. In *CIDR*, 2003.

[9] R. H. Chan, Chung-Wa Ho, and M. Nikolova. Salt-and-pepper noise removal by median-type noise detectors and detail-preserving regularization. *IEEE Trans. on Image Processing*, 14(10), 2005.

[10] Lei Chen, M. Tamer Özsu, and Vincent Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD Conference*, pages 491–502, 2005.

[11] Hui Ding, Goce Trajcevski, and Peter Scheuermann. Efficient similarity join of spatio-temporal trajectories. In *Technical Report NWU-EECS-08-01, Northwestern University*, 2007.

[12] Thomas Eiter and Heikki Mannila. Computing discrete fréchet distance. In *Technical Report CD-TR 94/64, Technische Universitat Wien*, 1994.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[14] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD Conference*, pages 419–429, 1994.

[15] Elias Frentzos, Kostas Gratsias, and Yannis Theodoridis. Index-based most similar trajectory search. In *ICDE*, 2007.

[16] H.Hwang and R. A. Haddad. Adaptive median filters: new algorithms and results. *IEEE Trans. on Image Processing*, 4(4), 1995.

[17] Gísli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD Conference*, pages 237–248, 1998.

[18] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, CA, 2005.

[19] George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *IEEE Computer*, 32(8):68–75, 1999.

[20] E. Keogh, X. Xi, L. Wei, and C.A. Ratanamahatana. The UCR Time Series dataset. In *http://www.cs.ucr.edu/ eamonn/time_series_data/*, 2006.

[21] Eamonn J. Keogh. Exact indexing of dynamic time warping. In *VLDB*, pages 406–417, 2002.

[22] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, and Y. Theodoridis, editors. *R-trees: Theory and Applications*. Springer-Verlag, 2006.

[23] Nikos Pelekis, Ioannis Kopanakis, Gerasimos Marketos, Irene Ntoutsi, Gennady L. Andrienko, and Yannis Theodoridis. Similarity search in trajectory databases. In *TIME*, pages 129–140, 2007.

[24] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000.

[25] Yasushi Sakurai, Masatoshi Yoshikawa, and Christos Faloutsos. Ftw: fast similarity search under the time warping distance. In *PODS*, pages 326–337, 2005.

[26] Michail Vlachos, Dimitrios Gunopulos, and Gautam Das. Rotation invariant distance measures for trajectories. In *KDD*, 2004.

[27] Michail Vlachos, Dimitrios Gunopulos, and George Kollios. Discovering similar multidimensional trajectories. In *ICDE*, 2002.

[28] Michail Vlachos, Marios Hadjieleftheriou, Dimitrios Gunopulos, and Eamonn J. Keogh. Indexing multidimensional time-series. *VLDB J.*, 15(1):1–20, 2006.

[29] Jason Tsong-Li Wang and Dennis Shasha. Query processing for distance metrics. In *VLDB*, pages 602–613. Morgan Kaufmann, 1990.

[30] Li Wei, Eamonn J. Keogh, Helga Van Herle, and Agenor Mafra-Neto. Atomic wedgie: Efficient query filtering for streaming times series. In *ICDM*, pages 490–497, 2005.

APPENDIX:

Proposition 3.1 proof (sketch, c.f.[4]): Obviously, we have $\delta_{wDF}(Tr_1, Tr_1) = 0$ and $\delta_{wDF}(Tr_1, Tr_2) = \delta_{wDF}(Tr_2, Tr_1)$, it remains to prove the triangular inequality.

Let $Tr_1 = (p^1_1, \ldots, p^1_n)$, $Tr_2 = (p^2_1, \ldots, p^2_n)$ and $Tr_3 = (p^3_1, \ldots, p^3_n)$, we show that $\delta_{wDF}(Tr_1, Tr_3) \leq \delta_{wDF}(Tr_1, Tr_2) + \delta_{wDF}(Tr_2, Tr_3)$. By definition of δ_{wDF}, it suffices to construct a w-constrained coupling (u, v) such that $dist(p^1_i, p^3_{u(i)}) \leq \delta_{wDF}(Tr_1, Tr_2) + \delta_{wDF}(Tr_2, Tr_3)$ and $dist(p^1_{v(j)}, p^3_j) \leq \delta_{wDF}(Tr_1, Tr_2) + \delta_{wDF}(Tr_2, Tr_3)$ for all $i, j \in [1:n]$.

By definition of δ_{wDF}, there exists a w-constrained coupling (μ, ν) between Tr_1 and Tr_2 such that $dist(p^1_{\mu(s)}, p^2_{\nu(s)}) \leq \delta_{wDF}$ for any s. Similarly, there exists a w-constrained coupling (κ, λ) between Tr_2 and Tr_3 such that $dist(p^2_{\kappa(t)}, p^3_{\lambda(t)}) \leq \delta_{wDF}$ for any t.

Next, we construct $u(i)$ as follows: given $i \in [1:n]$, let α be the first point in (μ, ν) that is matched with p^1_i, and let $u(i)$ be the first point in (κ, λ) that is matched with p^2_α. Similarly, we construct $v(j)$ as follows: given $j \in [1:n]$, let β be the first point in κ, λ that is matched with p^3_j, and let $v(j)$ be the last point in μ, ν that is matched with p^2_{beta}. We have now defined $u(i)$ and $v(j)$ for all $i, j \in [1:n]$. From the construction process, we know $(i, u(i))$ and $(v(j), j)$ form two monotonically non-decreasing couplings between Tr_1 and Tr_3. Furthermore, we know that for any i, p^1_i, $p^3_{u(i)}$ and α forms a triangle and hence we have: $dist(p^1_i, p^3_{u(i)}) \leq dist(p^1_i, \alpha) + dist(\alpha, p^3_{u(i)}) \leq \delta_{wDF}(Tr_1, Tr_2) + \delta_{wDF}(Tr_2, Tr_3)$, from the definition of (μ, ν). Similarly, we have $dist(p^2_{v(j)}, p^3_j) \leq dist(p^2_{v(j)}, \beta) + dist(\beta, p^3_j) \leq \delta_{wDF}(Tr_1, Tr_2) + \delta_{wDF}(Tr_2, Tr_3)$. □

Theorem 3.1 proof (sketch): Let C be the coupling between Tr_1 and Tr_2 which has the minimum length. By definition, there exists a pair of points in C, say (p^1_i, p^2_j), whose distance is larger than any other pair in C, i.e., $\delta_{wDF}(Tr_1, Tr_2) = dist(p^1_i, p^2_j)$. Further assume that for a given M_1 and M_2, p^1_i is contained in MBR^1_u and p^2_j is contained in MBB^2_v. We have $MinDist(MBB^1_u, MBB^2_v) \leq dist(p^1_i, p^2_j)$. By definition of δ_{wDF} and by transitivity, for any other pair of MBRs we can show that their MinDist is bounded by $dist(p^1_i, p^2_j)$. Hence, $dist(p^1_i, p^2_j)$ is greater than the length of any lower-bound coupling between the two sequences of MBRs. The proof of $\delta_{wDF}(Tr_1, Tr_2) \leq UBD_{\delta_{wDF}}(M_1, M_2)$ can be carried out similarly. □