Efficient Methods for Registration of Multiple Moving Points in Noisy Environments

E. Packer
IBM Research, Haifa
University of Haifa Campus
Haifa, Israel 31905

S. Pupyrev, A. Efrat, S. Kobourov
Department of Computer Science
University of Arizona
Tucson, AZ, USA

ABSTRACT
Matching sets of trajectories obtained by two different resources is a challenging and well motivated spatio-temporal problem. It arises when the motion of the same set of moving objects is obtained by two sensing devices (e.g. camera or radars) or when data is annotated by different users. The ultimate goal is to pair the trajectories so that each object is associated with two trajectories. Within this context, two main questions arise: (1) how to measure similarities between trajectories, and (2) how to use the similarity measure between trajectories to arrive to a reliable matching. Here we describe computationally efficient methods for several variants of the problem. The proposed methods have been implemented and used in experiments with real-world trajectory data. The results indicate that they are not only theoretically sound, but also work well in practice.

Categories and Subject Descriptors
H.4 [**Information Systems Applications**]: Miscellaneous

General Terms
Algorithms

Keywords
Trajectories, Matching

1. INTRODUCTION
Due to the availability of location enabled devices, data with spatial attributes become more and more widespread, increasing the demand for analysis tools. Numerous technologies provide locations (e.g., GPS, HI-FI, Cellolar, cameras) and many applications that make use of the location data have been developed and are available for mobile phones, navigation systems, etc.

Trajectories are arguably the most interesting pieces of information created by location services. Many applications may need the trajectory data as historical resources (e.g., statistics about outdoor activities, common route analysis and traffic congestion analysis), while others need the data on the fly (current route statistics, analysis of current relationships with other trajectories). Trajectories for ships, airplanes, animals, people have been collected and are used in academic and industrial research.

The availability of trajectory data offer great opportunities. For example, analyzing complicated relationships among trajectories can help alleviate traffic congestion and thus save time and money, while also decreasing pollution. Trajectory data can be used for understanding human mobility and social patterns [16]. Such understanding can, in turn, inform solutions to large-scale societal problems in fields as varied as telecommunications, ecology, epidemiology, and urban planning. As an example, knowing how large populations of people move about would help determine their carbon footprint and in turn help guide policies intended to reduce that footprint.

Trajectory data, together with growing demand to analyze such data, pose many challenges. For example, real-time travel route calculation is a difficult problem. Another challenge is related to location and time inaccuracies associated with input trajectories. Inevitably, location signals are noisy, contain outliers, and have unexpected behavior.

There are many scenarios where the trajectory of the same moving object is measured by different devices. In such cases it is sometimes crucial to merge the data to obtain better localization. For example, while GPS reaches impressive precession in optimal conditions, it might still provide only limited precision or no data within dense forests, indoor environment or underwater. Another limitation of this technology is that sometimes too few satellites are accessible to the receiver, providing poor localization. Other technologies, such as HIFI and Cellolar, have their own limitations that result in unreliable data. Hence, overlaying data from several sources is one natural way to increase the accuracy and reliability of the data.

In some scenarios, objects do not share identities. This could be due to privacy, security, or other reasons. Then the task of merging data from several source becomes more challenging as the the trajectories cannot be matched by merely checking their identities. There are applications that make good use of trajectory matching, such as calibrating two video cameras (direction, focal length, etc.) whose goal is to track the same object. The problem of matching pairs of trajectories naturally arises in the context of extracting accurate trajectories of ants from many (possibly inaccurate) input trajectories annotated by volunteers (citizen scientists) [8]. In this setting, the volunteers play an online game showing a video of an ant colony and indicate the trajectory of specified ants via mouse clicks (a task that is commonly challenging for computer vision techniques). To increase the reliability of such annotation, each ant's trajectory is annotated by two or more scientists. Hence, a robust algorithm for determining which trajectories (annotated by different citizen scientists) refer to the same ant is needed.

In this paper we propose novel methods for computing a good matching between two sets of trajectories. On a high level, the task is divided into two parts: computing a similarity measure between a pair of trajectories and partitioning the trajectories into pairs in a way that optimizes some criteria. We define a *distance* for a pair of trajectories and propose a measure with which we globally minimize the total cost of the matching. Specifically, we focus on the Fréchet distance for computing the distance between trajectories and use a matching algorithm that minimizes the maximum distance between any pair of matched trajectories. As the naive implementation of the algorithm is computationally expensive, we describe two techniques for speeding up the computation. The first technique is based on the idea of *locality-sensitive hashing*, which filters out pairs of trajectories that are "far" from each other. From the remaining pairs, we construct a weighted bipartite graph and find a *bottleneck* matching — a perfect matching that minimizes the maximum weight of a matched edge [12]. To this end, we suggest an algorithm in which computation of the Fréchet distance is optimized.

The rest of this paper is organized as follows. In the next section we discuss related work. In Section 3 we provide background information for this work. In Sections 4 and 5 we describe our algorithms. In Section 6 we provide experimental results and present several applications. We conclude and discuss directions for future work in Section 7.

2. RELATED WORK

The problem of clustering trajectories is closely related to our problem. Several methods for dealing with this problem have been proposed, depending on the distance function [14, 18]. Unlike the traditional setting, in our case all clusters must have exactly two trajectories. Several techniques for computing distances between curves have been studied in the literature. Methods based on the *feature space* take into account the location of knots, loops, points exceeding specific curvatures, and other features when computing distances. These methods are often used in signature verification and other computer vision applications and are difficult to apply in our case. In the setting where accurate timestamps are available, Trajcevski *et al.* [22] use the maximum distance at corresponding times as a measure of similarity between pairs of trajectories, and they describe algorithms for optimal matching under rotations and translations.

In this paper we use a more generic approach that does not require specific prior assumptions about the shape of the curves. Our method is related to Dynamic Time Wrapping (*DTW*), which can be discrete [11] or continuous [14] . The DTW method is often used for clustering data [7, 23]. We do not use the method directly as it is computationally expensive and we are not aware of any fast approximation algorithm.

The Fréchet distance as a measure of similarity for trajectories was introduced to the algorithm community by Alt and Godau [3]. It found its place in trajectory clustering, reconstructing road map from GPS trajectories [4], aligning networks [2], and protein matching [17]. Recently, the Fréchet distance is studied by Buchin *et al.* [6], who show how to incorporate time-correspondence and directional constraints. One of the drawbacks is its relative high computational complexity, with last year's first sub-quadratic algorithm for computing the Fréchet distance between a pair of curves [1]. Faster algorithms are known for curves that satisfy some "reasonable" properties of realistic input models [10].

The problem of finding the most likely "consensus" trajectory, given a set of trajectories is also related to our problem, and it has been considered in many different contexts. Morris and Barnard [19] use a statistical learning approach for finding hiking and biking trails from aerial images and GPS traces. A geometric distance measure to find similar subtrajectories is considered by Buchin *et al.* [5]. Some of the most recent methods include general approaches for tracking cells undergoing collisions by Nguyen *et al.* [20] and specific approaches for tracking insects by Fletcher *et al.* [13].

3. PRELIMINARIES

Define a *trajectory* to be a polyline in $I\!R^2$, represented by a sequence $p_1,\ldots,p_T$ of T points. In addition to the spatial coordinates, temporal information might also be available. Specifically, each point p_i is associated with a normalized timestamp $0 \le t_i \le 1$ and we assume that the trajectory is monotone in time: for each $1 < i < j < T$, it holds $t_1 = 0 < t_i < t_j < t_T = 1$. Hence, a trajectory can be viewed as a monotone (with respect to time) polyline in 3D. We use the term *parameterization* to refer to the location of the timestamp along the interval $[0,1]$. A *distance* between a pair of trajectories is defined as the maximum distance between the corresponding polylines at any timestamp. Let u,v be a pair of trajectories, and $D(u(t),v(t))$ denote the distance between u and v at the timestamp t. Then the distance between u and v is

$$D(u,v) = \max_{t\in[0,1]} D(u(t),v(t)).$$

The input of our problem is two collections of trajectories S_1 and S_2 of size n obtained with different resources. Our goal is to match the trajectories from both sets based on their similarity. Formally, we want to find a matching $M = \{m_1,\ldots,m_n\}$, where $m_k = (u^i,v^j), 1 \le k \le n$ with $u^i \in S_1, v^j \in S_2$ and each trajectory is matched exactly once. Ideally, a good matching identifies pairs of trajectories that belong to the same entity. To this end, we are looking for a matching minimizing the maximum distance between matched trajectories. Our problem, which we denote by TMATCH, is defined as follows.

TMATCH: Given two sets of trajectories, $S_1 = \{u^1,\ldots,u^n\}$ and $S_2 = \{v^1,\ldots,v^n\}$, find a matching $M = \{m_1,\ldots,m_n\}$, where $m_k = (u^i,v^j)$ with $u^i \in S_1, v^j \in S_2$ that minimizes

$$\max_{i,j} D(u^i,v^j).$$

We solve the problem in two steps. First, we compute distances between pairs of trajectories and construct a weighted bipartite graph using the distances. Second, we find a perfect matching on the graph minimizing the maximum weight of matched edges. In the next section, we describe how to perform the first step, that is, how to find a distance between a pair of trajectories; we consider three scenarios depending on the type of input trajectories. In Section 5, we show how to compute the optimal matching efficiently.

4. COMPUTING DISTANCES

The input trajectories may contain timestamps, only partial timestamps, or no timestamps altogether. We consider three scenarios for computing the distance between a pair of curves. In the first one, all points are associated with a timestamp. In the second one, timestamps are missing. The third scenario is a generalization, where some of the timestamps are known and some are not. We consider algorithms for computing distances in all three cases.

4.1 Time Associated Setting (TAS)

This is the simplest setting. Since the timestamps are given, trajectories can be represented as polylines in $\mathbb{R}^3$ (x,y,t coordinates). To compute the distance, we sweep the $t = C$ plane for some constant C from the normalized parameterizations $t = 0$ to $t = 1$, processing segments intersected simultaneously by the sweeping plane. For each such pair, we compute the maximum distance when intersected with the plane. The maximum value obtained over all pairs of segments is the similarity distance of the trajectories. It is easy to verify that the processing time is linear on the trajectory sizes as we traverse both trajectories simultaneously once.

4.2 No Time Associated Setting (NTAS)

In this scenario, the timestamps are unknown and we rely on the Fréchet distance between two curves. A common way to illustrate the Fréchet distance is with the man walking a dog analogy, where the trajectories for man and dog are fixed in advance and both can only move forward along their trajectories. The Fréchet distance is the length of the shortest leash between the man and the dog, which would allow them to get from the beginning to the end of their respective trajectories. The Fréchet distance can be obtained by parameterizing the trajectories optimally. Mathematically, the Fréchet distance is defined as

$$D_F(u,v) = \inf_{\alpha,\beta} \max_{t\in[0,1]} d(u(\alpha(t)), v(\beta(t))),$$

where u and v are the trajectories, d denotes a distance function and α and β range over all monotone parameterizations.

For a given constant γ, checking whether the Fréchet distance between two trajectories u and v is smaller than γ can be done as follows. Define a two dimensional grid $H = [0,1] \times [0,1]$. The value in each point in the grid, $H(t_1,t_2)$, $0 \leq t_1,t_2 \leq 1$ is either *valid* if the distance between $u(t_1)$ and $v(t_2)$ (the points on u and v corresponding to the parameterizations t_1,t_2) is smaller than γ or *invalid*, otherwise. The grid defines a *free space diagram* that indicates the validity of the locations. It follows that $D_F(u,v) \leq \gamma$ if and only if there exists a monotone path from $[0,0]$ to $[1,1]$ in H passing through valid grid points; see Fig. 1.

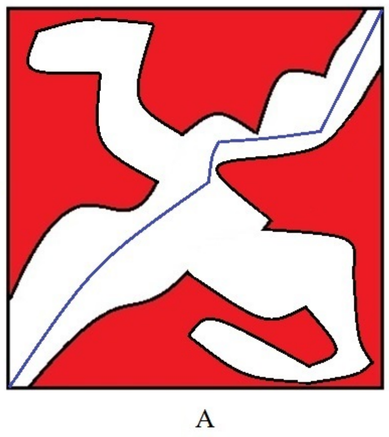


Figure 1: The free space diagram for some trajectories A and B and some fixed constant γ. The valid regions are in white. The blue monotone path shows one way to parameterize the curves so that the distance between any two matched points is always less than γ.

In order to find $D_F(u,v)$, one may perform a binary search on the value of γ, finding the maximum value for which a valid path exists. Unfortunately, the free space diagram is defined with conic arcs and thus requires supporting range queries on arrangements of conic arcs, which is not easy to implement and costly to use.

The discrete Fréchet distance is a variant in which only discrete points (or stations) along the trajectories are considered. Then the Fréchet distance corresponds to a sequence of steps done on the two trajectories: in each we advance from a station to its subsequent on at least one trajectory. The discrete Fréchet distance is the maximum distance between any pair of stations considered in this parameterization. Note that the discrete Fréchet distance gives an upper bound on the continuous variant and in general is relatively close to it. Additionally, the discrete version is faster to compute and easier to implement comparing to the continues one. Therefore, we focus on the discrete Fréchet distance in this work.

The discrete version of Fréchet is defined analogously to the continuous one; instead of considering all points in the range $[0,1]$, we are concerned with discrete points of each trajectory. We check for each pair of points from both trajectories whether the distance between the locations is smaller then γ. Denote by Θ the set of $\mathbb{R}^2$ parameterizations valid for γ. We connect $\alpha \in \Theta$ to $\beta \in \Theta$ if they correspond to a valid motion. Note that the discrete solution does not require special geometric computations making it much more practical.

The algorithm for computing the discrete Fréchet distance is as follows. We use a binary search, and each iteration is performed as follows. Let u and v be two trajectories. We build a two-dimensional matrix M such that $M(i,j) = 1$ if and only if $d(u_i,v_j) < \gamma$, where d is the Euclidean distance, γ is the current distance tested with the binary search, and u_i, v_j are the i-th and j-th points of u and v, respectively. Then we test if there is a monotone path from the lower bottom part of the matrix to its top right, by traversing the matrix as we explain next. The path may consists of horizontal, vertical or diagonal moves for a step on u, a step on v and a simultaneous step on both trajectories. The path is found by iteratively spreading from the lower bottom corner, detecting reachable locations, while maintaining monotonicity. The algorithm takes $O(T^2 \log D_{max})$ time, where T is the length of a trajectory and $D_{max} = \max_{u\in S_1, v\in S_2} D(u,v)$ is the largest distance between any two points in the input.

4.3 Mixed Time Associated Setting (MTAS)

This case is a generalization of TAS and NTAS. Here a point along an input trajectory may or may not be associated with a timestamp. To solve the continuous version, we modify the algorithm for NTAS, by restricting the free space diagram with blocking rectangles. Let T_1 and T_2 be the sequences of timestamps of both trajectories sorted by time. Consider two timestamps $t_1 \in T_1$ and $t_2 \in T_2$ so that $t_1 < t_2$. It follows that the second trajectory cannot pass the location parameterized by t_2 before the first trajectory passes the location parameterized by t_1. This constraint defines a *forbidden rectangle* in the free space diagram; see Fig. 2. Given T_1 and T_2, we process them by increasing values. For each value visited on one of the trajectories v, we take the recent one on the other trajectory and find the corresponding rectangle. Then we subtract those rectangles from the free space diagram to get an instance in which we search for a monotone path. It is important to observe that each trajectory point is associated with at most two rectangles that can exist due to its interaction with locations of the other trajectory that were reported right before or right after it. Thus, the number of constraining rectangles is linear and does not affect the complexity of the free space diagram. The discrete variant is defined analogously. The rectangle definition remains the same, invalidating pairs of points from both trajectories whose distance is smaller than γ. Invalidating pairs can be done in $O(T^3)$ time, since one needs to check $O(T^2)$ pairs against $O(T)$ rectangles. However, by using arrangements of segments in $\mathbb{R}^2$, we can improve the time complexity to $O(T^2 \log T)$, relying on the fact that testing the location of a point within the constraining rectangles takes $O(\log T)$ time. Note that based on the characteristics of the forbidden rectangles, they form two staircase structure in the free space, touching the bottom-right and the top-left of the free space respectively; see Fig. 2.

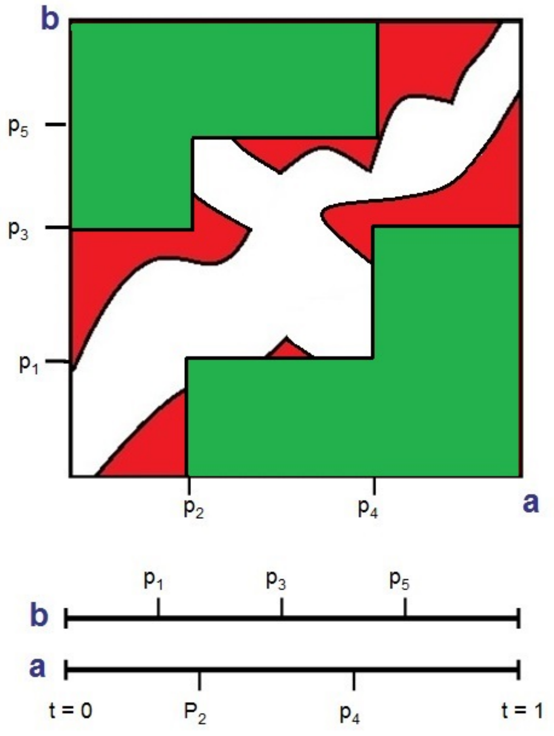


Figure 2: The free space diagram for a pair of trajectories a and b overlayed with forbidden rectangles (in green) that contribute to the invalid parameterization (the red areas). The parameterization of the trajectories are below.

5. MATCHING ALGORITHM

Let $S_1 = \{u^1, \ldots, u^n\}$ and $S_1 = \{v^1, \ldots, v^n\}$ be two sets of trajectories. A straightforward approach (referred to as *brute-force*) for solving the TMATCH problem consists of two steps. First, we build a complete bipartite graph G with $2n$ vertices corresponding to the trajectories. For an edge (u^i, v^j) in G, we compute its weight as the distance between the trajectories u^i and v^j as discussed in the previous section. The next step is to compute a perfect matching in G. It is easy to see that the problem of finding a perfect matching minimizing the maximum weight can be solved in polynomial time using the classical Hopcroft-Karp algorithm [15] or by finding a maximum flow in G [9].

Let us discuss complexity of the approach. Suppose computing a distance for a pair of trajectories requires R steps. As discussed earlier, $R = O(T)$ for TAS, $R = O(T^2 \log D_{max})$ for NTAS, and $R = O(T^2 \log T)$ for MTAS, where T is the length of trajectories and D_{max} is the largest distance between input trajectories. Hence, computing all pairwise distances requires $O(n^2 R)$ time. A standard algorithm for finding a perfect matching minimizing the maximum weight uses a binary search on the maximum distance between trajectories, and hence, requires $O(n^{2.5} \log D_{max})$ time. Therefore, the total time complexity of the algorithm is $O(n^{2.5} \log D_{max} + n^2 R)$. As the length T of trajectories may be large, this is often inefficient in practice. Fortunately, in most cases we expect most of the trajectories to be far enough (location-wise or time-wise) to be candidates for pairing. We next present two approaches to improve the running time of the algorithm, depending on the availability of time information.

5.1 Locality-Sensitive Hashing

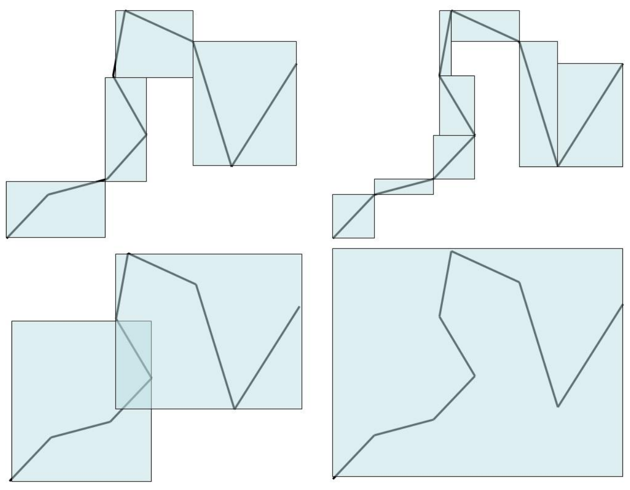
Let us consider the scenario in which timestamps are available. We improve the running time of the brute-force algorithm by limiting the number of trajectory pairs that can be potentially matched. On a high level, the idea is based on *locality-sensitive hashing*. For each trajectory $u \in S_1 \cup S_2$, we compute a hash $h(u)$ so that "similar" trajectories (those that can be potentially matched) get close values. To this end, we consider a trajectory u as a point in R^T, and choose a random line in R^T with origin $p \in R^T$ and direction $q \in R^T$. Given a trajectory $u \in R^T$, let $h(u) \in R$ be so that $p + h(u)q$ is the projection of u onto the line; that is, the nearest point on the line to u. The projection can be found using the expression $u \cdot (u - q - h(u)u) = 0$, where $\cdot$ denotes a dot product. It is easy to see that such a hash is easy to compute in linear time for each trajectory. Note that similar trajectories correspond to close points in R^T, and therefore, get similar hashes. However, it is also sometimes possible that for the points lying far apart, the computed hashes are close.

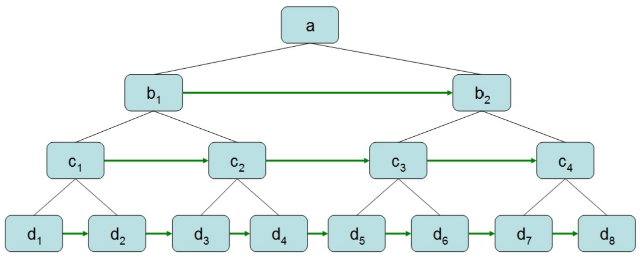
Now, instead of considering all pairwise distances, we fix a constant k and for each trajectory $u \in S_1$, find k trajectories $v \in S_2$ with the closest hashes, that is, having the smallest values $|h(u) - h(v)|$. It is easy to see that this results in computing $kn \ll n^2$ distances and, hence, a graph G with kn edges. Thus, the complexity of computing pairwise distances is reduced from $O(n^2 R)$ to $O(nR)$, and the complexity of finding a matching is reduced from $O(n^{2.5} \log D_{max})$ to $O(n^{1.5} \log D_{max})$.

5.2 Bottleneck Matching

Here we describe how to solve TMATCH when timestamps are not available or are incomplete (NTAS, MTAS). We start by presenting a geometric data structure called *Sequential Bounding Box Balanced Tree* (SBBBT), which is similar to the well-known $R-$

tree [21]. We use it to bound each trajectory in a hierarchical fashion. In the following, we assume that a bounding box is always axis-aligned.


(a)



(b)

Figure 3: (a) Four levels of bounding boxes for a trajectory. (b) The SBBBT of the trajectory. The vertices of the tree are marked according to the boxes in (a) that they correspond to. Green arrows connect vertices of the same level.

DEFINITION 5.1. *A* Sequential Bounding Box Balanced Tree *is a binary tree associated with a trajectory constructed as follows. The root contains the bounding box of the trajectory. The left and right children of the root correspond to the first and the second part of the trajectory. Each child contains the bounding box of the corresponding trajectory part. The rest of the tree is defined recursively in the same manner; see Fig. 3.*

We use SBBBT to speed up the computation of a matching. Our algorithm has the following high-level steps; see Algorithm 1. First we construct the SBBBT with $\log T$ levels for every trajectory. Then we use binary search to find the minimum value of γ for which there exists a perfect matching with all distances between matched trajectories less than γ. It is easy to see that the matching corresponding to the smallest such γ is a bottleneck matching.

The main idea of the algorithm is to avoid time-consuming computation of the Fréchet distance between trajectories. Consider a SBBBT tree constructed for a trajectory u. For any level $l \geq 0$, let $SBBBT(u,l)$ be a set of bounding boxes (rectangles) on level l of the tree. For a pair of trajectories $u \in S_1, v \in S_2$ and for $l \geq 0$, we

```
Input  : Trajectories $S_1$ and $S_2$
Output: The optimal bottleneck matching
/* computing SBBBT */
foreach trajectory $u \in S_1 \cup S_2$ do create $SBBBT(u)$
/* binary search on $\gamma$ */
$\gamma_l \leftarrow 0$, $\gamma_r \leftarrow D_{max}$
while $\gamma_l < \gamma_r$ do
    $\gamma \leftarrow (\gamma_l + \gamma_r)/2$
    /* computing candidate pairs */
    foreach pair $u \in S_1, v \in S_2$ do
        if $D_F(SBBBT(u), SBBBT(v)) \leq \gamma$ then add $(u,v)$ to $G$
    end
    /* matching */
    if a perfect matching with $D_F \leq \gamma$ exists in $G$ then
        $\gamma_l \leftarrow \gamma$
    else
        $\gamma_r \leftarrow \gamma$
    end
end
```

Algorithm 1: Bottleneck Matching

define the Fréchet distance between $SBBBT(u,l)$ and $SBBBT(v,l)$ in a similar way as the discrete Fréchet distance between trajectories. Given two sets of rectangles Z_1 and Z_2 and four points $s_1 \in Z_1, s_2 \in Z_2$ (sources) and $t_1 \in Z_1, t_2 \in Z_2$ (destinations), the Fréchet distance between Z_1 and Z_2 is the length of the shortest leash that allows two entities from different zones to walk from the sources to the destinations, while connected to each other with the leash. Notice that bounding boxes of the tree on some level completely cover the corresponding trajectory. Hence, if the Fréchet distance between $SBBBT(u,l)$ and $SBBBT(v,l)$ for any level $l \geq 0$ exceeds γ, then the Fréchet distance between u and v also exceeds γ. Using the observation, we may check whether $D_F(u,v) \leq \gamma$ by traversing the trees as follows. If $D_F(SBBBT(u,0), SBBBT(v,0)) > \gamma$ then the distance between u and v also exceeds γ; otherwise, proceed with the next level; see Fig. 4. Only if the distance is less than γ at all levels, we compute the actual Fréchet distance for the pair of trajectories. Note that the computation cost increases as we descend in the tree. Hence, disqualifying the trajectories early results in significant savings in the running time.

Let us now describe how we check for the existence of a perfect matching for a giving value γ in Algorithm 1. First, we compute a bipartite graph G with $2n$ vertices in which an edge between two trajectories is present if and only if the trajectories can be potentially matched. To this end, we compute the Fréchet distance between $SBBBT(u,l)$ and $SBBBT(v,l)$ for levels $0 \leq l \leq 3$, and add an edge (u,v) to G if the distance is less than γ. Clearly, this operation can be done efficiently since the number of boxes at the levels is relatively small. We expect to filter out most of the pairs of trajectories at this step, and construct a sparse G. Second, we check whether there exists a perfect matching in G. For this purpose, we use the maximum flow algorithm [9] by adding source and sink vertices to G, and then iteratively finding augmenting paths in the graph. During the search for an augmenting path, we use the corresponding edge (u,v) if the Fréchet distance between u and v is less than γ. Here, we again use the trees $SBBBT(u)$ and $SBBBT(v)$ to speed up the running time. We check up to $\log T$ levels of the trees, where T is the length of trajectories. We stress that the actual computation of the Fréchet distance is performed only if $D_F(SBBBT(u,l), SBBBT(v,l)) \leq \gamma$ for all $0 \leq l \leq \log T$. If the

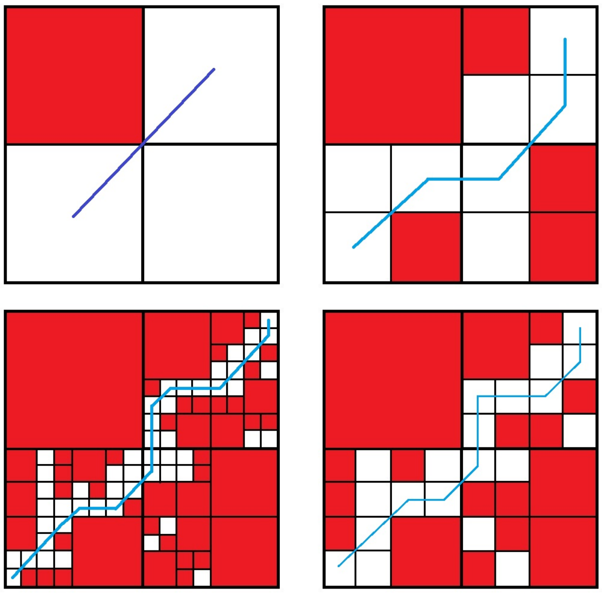


Figure 4: Computing the Fréchet distance with SBBBT. Decreasing levels (from top-left clockwise) of SBBBT are depicted on the parameterization diagram for a pair of trajectories. Red cells represent forbidden parameterization values. In the example, there exists a blue monotone path from bottom-left to top-right in all levels.

maximum flow in G is equal to n, then we conclude that there exists a perfect matching for the current value γ.

Let us mention an important aspect of our implementation of the algorithm. For each pair of trajectories, we store the following information to improve performance. (1) If the Fréchet distance is computed at an iteration, we use the value to prevent consideration of the same pair of trajectories at the subsequent iterations. (2) If the pair is filtered out for some γ (that is, the Fréchet distance is greater than γ), then we do not check it again for larger values of γ. Overall, in the worst case the algorithm for computing a bottleneck matching has the same complexity as the brute-force algorithm. However, we found significant speedup in practice, as discussed in Section 6.

6. EXPERIMENTS

We evaluate our algorithms on four datasets. We start with a real-world collection of ant trajectories generated by citizen scientists. We then consider two datasets constructed from route data of vehicle trajectories. The last one is an artificial dataset that is designed to highlight some features and insights about our methods.

6.1 Citizen Science Trajectories

The problem of matching pairs of trajectories naturally arises in the context of extracting accurate average trajectories of ants from many (possibly inaccurate) input trajectories contributed by citizen scientists [8]. In this setting, a citizen scientist plays an online game showing a video of an ant colony; the goal is to generate a trajectory of a specified ant via mouse clicks. Citizen scientists track ants during short video segments. In order to reconstruct a complete ant trajectory, one needs to stitch together short pieces of overlapping trajectories. Equivalently, given partially overlapping trajectories, the problem is to identify trajectories corresponding to the same ant; see Fig. 5.

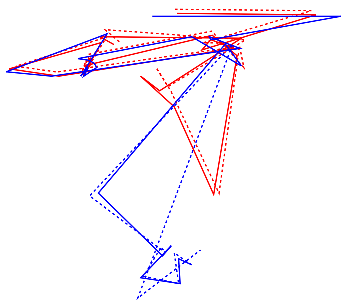


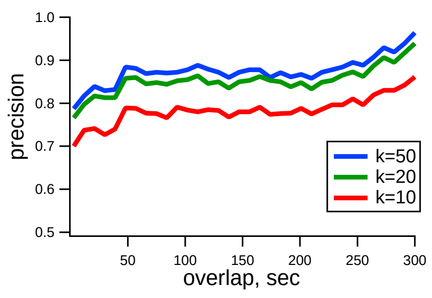
Figure 5: 4 citizen science trajectories corresponding to two different ants. Although the trajectories of the blue ant and the red ants are partially overlap, the matching algorithm is capable to identify 2 pairs of trajectories.

To evaluate our algorithms, we work with a video of a *Temnothorax rugatulus* ant colony containing 10,000 frames, recorded at 30 frames per second. Our dataset consists of 252 citizen-scientist-generated trajectories for 50 ants, with between 2 and 8 trajectories per ant. From the data, we construct 150 inputs for our matching algorithm. Every input consists of 50 pairs of trajectories with different length of overlapping segments varying from 300 seconds (corresponding to 100 timestamps) to 5 seconds (corresponding to 1 common timestamp). Since the trajectories contain associated time information, we compute distances as in TA setting and use the matching algorithm described in Section 5.1. The bipartite graph G is constructed by applying locality-sensitive hashing and using k closest trajectories for $k = 50$, $k = 20$, and $k = 10$. Note that the case with $k = 50$ is equivalent to the brute-force algorithm.

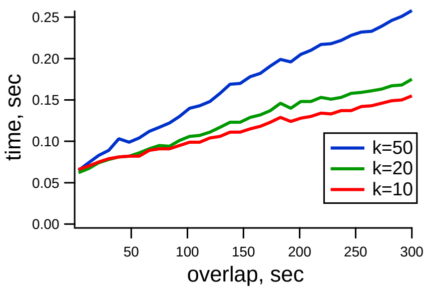
We compare the results by measuring the precision of the result and the running time of the algorithm; see Fig. 6(a) and Fig. 6(b). The *precision* is the percentage of correctly identified pairs of trajectories, which we know from ground-truth data. As expected, precision is higher for the inputs in which pairs of trajectories have longer overlap. The brute-force algorithm correctly matches over 95% pairs of trajectories having 5-minute overlap. As expected, using locality-sensitive hashing significantly improves the running time, whie only slightly reducing the accuracy. Specifically, in the setting where input trajectories have 1-minute overlap and for $k = 20$ (which corresponds to 60% fewer considred possible trajectories) we achive 85% precision (which corresponds to only 5% reduction in accuracy).

6.2 Vehicle Trajectories

We work with two datasets constructed from real-world vehicle trajectories. The first one contains public transportation trajectories (busses and trams) on a specific day in Helsinki. The second one contains ship trajectories at the port of Rotterdam; see Fig. 7. In both cases the trajectories are represented by geographic coordinates (longitude and latitude), and no time information is available. The Helsinki dataset contains 4 collections with 20, 110, 282, and 496 trajectories; the Rotterdam dataset contains 5 collections with 36, 52, 92, 124, and 184 trajectories. The length of trajectories varies from 11 to 2800 points with 1400 on average. Given a trajectory, we create its *paired* trajectory by randomly moving every point within a disk of radius r. We call the radius a *perturbation* and use $r \in \{0.001, 0.005, 0.01, 0.1\}$ in our experiments. Note that the



(a)



(b)

Figure 6: The results on the Citizen Science Trajectories dataset. k is the number of "close" trajectories used in the locality-sensitive hashing approach. (a) Precision ($\frac{\text{correctly identified pairs}}{\text{all pairs}}$) is higher for trajectories with longer overlap. (b) The running time of our algorithm.

value $r = 0.01$ corresponds approximately to 10 kilometers. Using the data, we constructed $\approx 10^6$ inputs with $40-992$ pairs of trajectories per input.

The results of our experiments are given in Fig. 8. Here we compare the bottleneck matching algorithm designed for NTA setting computing the discrete Fréchet distance between trajectories. We observe that for the perturbation value $r \leq 0.01$ the algorithm correctly identifies all pairs of trajectories, that is, achieves 100% precision. On the other hand, for $r \geq 1$ none of the algorithms has a chance to recover correct pairs of trajectories. Hence, in this section we primarily focus on measuring the running time for smaller values of perturbation. We compare the basic brute-force algorithm against our new bottleneck matching algorithm described in Section 5.2. We first note that the running time depends on the size of the input; see Fig. 8. The running time also depends on the noise (perturbation value); larger noise results in longer running time. This can be explained by the fact that larger noise results in more pairs of trajectories that can be potentially matched to each other.

As the most time-consuming step in the matching algorithms is calculation of the Fréchet distance, we also report on how well our heuristic filters Fréchet computation calls. We define a *saving ratio* as the number of the pairs of trajectories for which we computed the Fréchet distance divided by the total number of pairs of trajectories. The lower value corresponds to a better filtering, and the brute-force algorithm has the saving ratio 1. The results are presented in Fig. 9. We note that for a dataset with ≥ 100 trajectories, our heuristic using SBBBT filters out most of the pairs of trajectories. The actual computation of the Fréchet distance is done only for $\leq 2-5\%$ of all pairs. This corresponds to the $20-50x$ speedup compared to the brute-force algorithm.

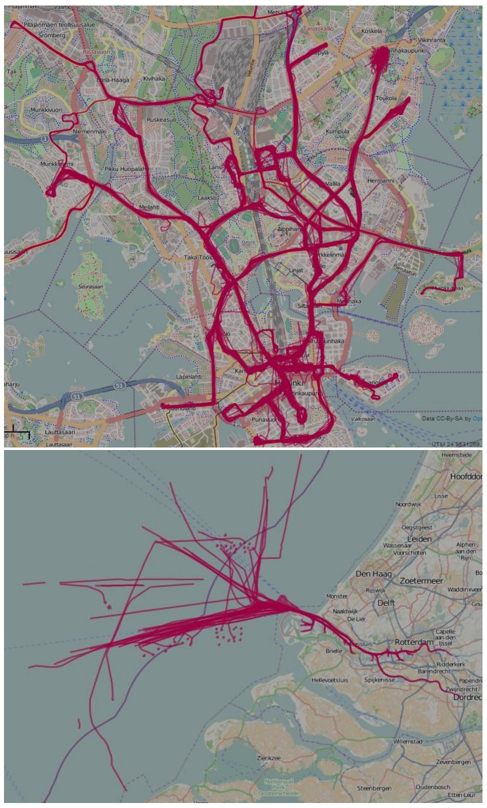
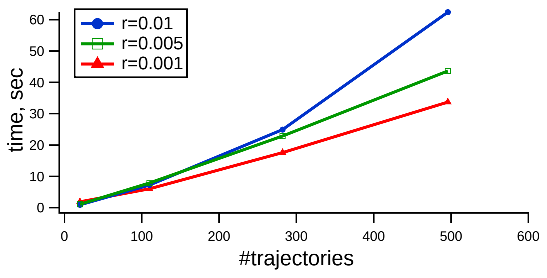
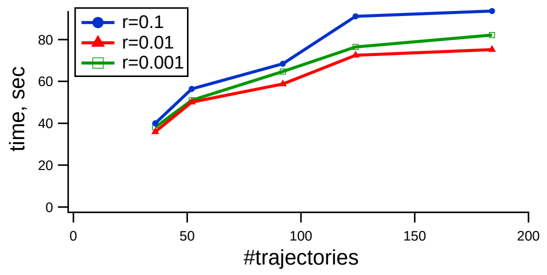


Figure 7: The trajectories we experimented with: Helsinki public transportation (top) and Ships near and at the port of Rotterdam (bottom).

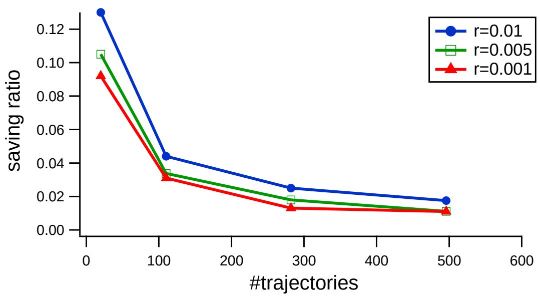


(a)

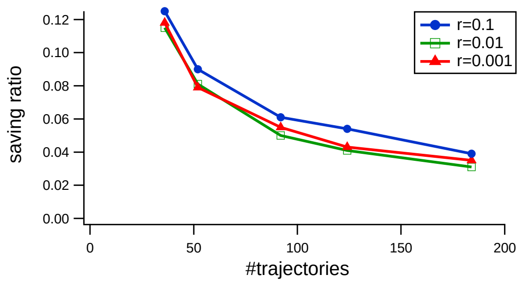


(b)

Figure 8: The running time of the bottleneck matching algorithm on the (a) Helsinki and (b) Rotterdam datasets for different values of perturbation.



(a)



(b)

Figure 9: The saving ratio of the bottleneck matching algorithm on the (a) Helsinki and (b) Rotterdam datasets for different values of perturbation.

6.3 Artificial Trajectories

We generated an artificial dataset to further analyze the improvement of running time and the effect on the quality of the output. The sets S_1 and S_2 of trajectories are constructed as follows. Each trajectory $u \in S_1$ is constructed inside a square of size 10. Its starting position and direction d are chosen randomly. Then u grows along d with steps of unit length until it hits the boundary of the square. When it happens, d is reflected with slight perturbation to prevent trajectory repetitions. We then generate a *paired* trajectory $v \in S_2$ by duplicating u and perturbing each of its points inside a disc centered at the point with radius r; see Fig. 10.

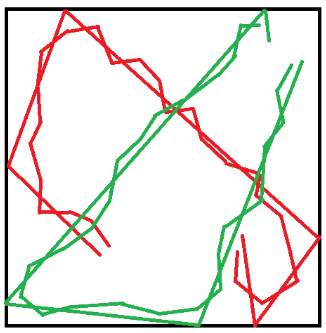


Figure 10: An example of two pairs of generated trajectories inside a square. Each pair is colored with a specific color.

The motivation of experimenting with this dataset is as follows. Any point along a trajectory $u \in S_1$ may have many other trajectories close to it at any given timestamp. However, in the long run, as we move along u, we expect that only its paired trajectory v will be close to u. Consider a square of edge length 4. On average, 25% of the trajectories are close to any point of u. On the other hand, the number of trajectories that are close to it after moving from any point decreases exponentially. Thus, we expect that non-paired trajectories are filtered relatively fast using SBBBT. Hence, the Fréchet distance is computed rarely compared to the brute-force algorithm. With this in mind, we analyze the bottleneck matching algorithm using SBBBT in the NTA setting.

Running Time. First, we evaluate the running time. In the experiments, we varied the number of trajectories and the length of a trajectory; see Fig. 11. We use perturbation radius $r =$ for which our algorithm correctly matches all pairs of trajectories, that is, it has 100% precision. We observed that the number of pairs for which the algorithm computes the Fréchet distance linearly depends on the input size (the number of trajectories). This explains the linear growth of the running time in Fig. 11.

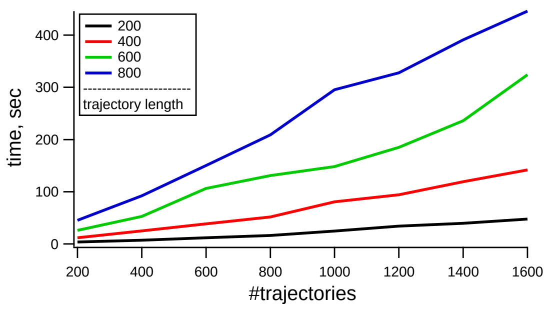


Figure 11: Results of the Artificial Trajectories experiment. The running time is shown as a function of the number of trajectories for different trajectory lengths.

Precision. In this experiment we set the trajectory length $T = 100$, while varying the input size $n \in \{800, 900, 1000, 1100, 1200\}$ and the perturbation radius $r \in \{1, 2, 3, 5, 6\}$. We analyze how the perturbation affects precision, that is, the number of correctly identified pairs divided by the total number of pairs. The results are shown in Fig. 12. Not surprisingly, the precision is lower for inputs with larger noise, and it decreases faster for inputs with more trajectories. Note, however, that the precision remains steadily over 95% for the entire dataset if $r \leq 2$.

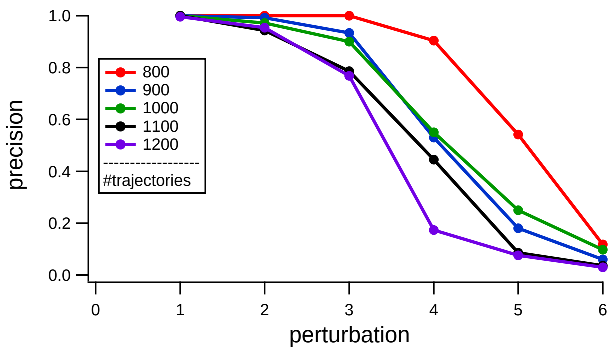


Figure 12: Precision of the results for the Artificial Trajectories dataset.

7. CONCLUSION AND DISCUSSION

We presented a novel approach for matching trajectories generated by two independent resources. We considered three scenarios: (1) timestamps are associated with the points of the trajectories, (2) timestamps are missing, and (3) some of the points include timestamps. In every scenario, we presented an efficient algorithm by solving two subproblems. The first one is to measure similarities of two trajectories, and the second one is to match the trajectories based on the computed similarities. When timestamps are available, we used locality-sensitive hashing to find a solution with high accuracy. When no timestamps are known, we measured the similarity using the Fréchet distance. In order to match the trajectories, we use bottleneck matching (minimizing the maximum distance) and a sequential bounding box balanced tree. To support partial timestamp data, we modified the solution by introducing additional constraints. We demonstrated experimentally that our algorithms yield good results.

An important application of the presented algorithms is tracking ants (or other insects) in long videos. For long videos (e.g., hundreds or even thousands of hours), automated tracking methods are not reliable. Whenever such algorithms loose tracking, the error quickly accumulates and a trajectory cannot be recovered. Trajectory matching can be used to resolve this problem. We ask citizen scientists to solve the "hard ants" or "hard trajectory segments", while tracking "easy ants" and/or "easy trajectory segments" automatically. Then we apply the matching algorithm for trajectories and stitch together many short pieces of overlapping trajectories.

A great deal of challenging problems remain. In this work we described how to match two sets of trajectories. In general, we may have more than two sets. Already for three sets of trajectories the problem changes dramatically. First, it is interesting to compute the Fréchet distance in $\mathbb{R}^3$. Second, for matching the trajectories we would need to solve a tripartite matching problem (a matching in hypergraphs in which vertices contain three elements), which is known to be NP-hard. Also, our use of augmenting paths will not fit so nicely with more than two sets. Another direction for future research is to wisely select the number of levels in the SBBBT. In this paper, we obtained good results with four levels in the tree. It would be interesting to establish a good criteria for the determining the number of levels so as to further improve performance.

8. REFERENCES

[1] P. K. Agarwal, R. B. Avraham, H. Kaplan, and M. Sharir. Computing the discrete Fréchet distance in subquadratic time. In *24th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–167, 2013.

[2] H. Alt, A. Efrat, G. Rote, and C. Wenk. Matching planar maps. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 589–598, 2003.

[3] H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *International J. of Computational Geometry & Applications*, 5(1-2):75–91, 1995.

[4] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *31st International Conference on Very Large Data Bases*, pages 853–864, 2005.

[5] K. Buchin, M. Buchin, J. Gudmundsson, M. Löffler, and J. Luo. Detecting commuting patterns by clustering subtrajectories. *International J. of Computational Geometry & Applications*, 21(3):253–282, 2011.

[6] K. Buchin, M. Buchin, M. Kreveld, M. Löffler, R. Silveira, C. Wenk, and L. Wiratma. Median trajectories. *Algorithmica*, 66(3):595–614, 2013.

[7] G. Chen, Q. Wei, and H. Zhang. Discovering similar time-series patterns with fuzzy clustering and dtw methods. In *Joint 9th IFSA World Congress and 20th NAFIPS International Conference*, volume 4, pages 2160–2164, 2001.

[8] L. D. L. Cruz, S. G. Kobourov, S. Pupyrev, P. Shen, and S. Veeramoni. Angryants: An approach for accurate average trajectories using citizen science. *CoRR*, abs/1212.0935, 2013.

[9] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math Doklady*, 11:1277–1280, 1970.

[10] A. Driemel, S. Har-Peled, and C. Wenk. Approximating the Fréchet distance for realistic curves in near linear time. *Discrete & Computational Geometry*, 48(1):94–127, 2012.

[11] A. Efrat, Q. Fan, and S. Venkatasubramanian. Curve matching, time warping, and light fields: New algorithms for computing similarity between curves. *Journal of Mathematical Imaging and Vision*, 27(3):203–216, 2007.

[12] A. Efrat, A. Itai, and M. J. Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica*, 31(1):1–28, 2001.

[13] M. Fletcher, A. Dornhaus, and M. Shin. Multiple ant tracking with global foreground maximization and variable target proposal distribution. In *WACV'11*, pages 570–576, 2011.

[14] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 63–72. ACM, 1999.

[15] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[16] S. Isaacman, R. A. Becker, R. Cáceres, S. G. Kobourov, J. Rowland, and A. Varshavsky. A tale of two cities. In *HotMobile*, pages 19–24, 2010.

[17] M. Jiang, Y. Xu, and B. Zhu. Protein structure-structure alignment with discrete Fréchet distance. In *5th Asia-Pacific Bioinform. Conference*, pages 131–141, 2007.

[18] X. Li, W. Hu, and W. Hu. A coarse-to-fine strategy for vehicle motion trajectory clustering. In *18th International Conference on Pattern Recognition*, volume 1, pages 591–594. IEEE, 2006.

[19] S. Morris and K. Barnard. Finding trails. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[20] N. Nguyen, S. Keller, E. Norris, T. Huynh, M. Clemens, and M. Shin. Tracking colliding cells in vivo microscopy. *IEEE Trans. on Biomedical Engineering*, 58(8):2391–2400, 2011.

[21] H. Samet. *Foundations of Multidimensional And Metric Data Structures*. Morgan Kaufmann. Elsevier, 2006.

[22] G. Trajcevski, H. Ding, P. Scheuermann, R. Tamassia, and D. Vaccaro. Dynamics-aware similarity of moving objects trajectories. In *ACM-GIS'07*, pages 1–8, 2007.

[23] Z. Zhang, K. Huang, and T. Tan. Comparison of similarity measures for trajectory clustering in outdoor surveillance scenes. In *18th International Conference on Pattern Recognition*, volume 3, pages 1135–1138, 2006.