Trajectory Mosaicing

E. Packer
IBM Research - Haifa
University of Haifa Campus
Haifa, Israel 3498825

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

ABSTRACT

The ubiquitous presence of location-based devices and sensors has made it easier to collect data on the trajectories of people, animals, vehicles, and objects. The benefits of collecting this data are clear, whether for traffic analysis, social analytics, business optimization, or safety measures. Unfortunately, much of the data on these trajectories is incomplete or unreliable, and cannot be used Śas isŠ for further analytics. We introduce new methods to combine and process trajectory data. In the core of this work we present a system to analyze ant trajectories annotated manually. Our work reviews the difficulties and challenges that arise in this scenario and presents efficient solutions with which we obtained satisfactory results. These solutions which relate to trajectories and deal with clustering and matching are generic enough to be used in many systems that require trajectory analysis. In particular, systems that process incomplete data or suffer from reasonable noise could benefit from our solutions. We implemented the methods and present experiments with real-world data. TODO It would be good to say something here about the results to provide credibility and give people a reason to read the paper]

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms

Keywords

Trajectories, Matching

1. INTRODUCTION

[Explain the idea of Mosaicing.] Due to the availability of location enabled devices, data with spatial attributes has become more and more widespread, increasing the demand for analytical tools. Numerous technologies are available to provide location data, including GPS, WiFi, cellolar and cameras. Moreover, many applications that make use of this location data have been developed and are available for mobile phones, navigation systems, and more.

(Clearly state the problem here and the importance of your results)

Trajectories are arguably the most interesting pieces of information created by location services. (Move the details to a subsequent section) Let a *trajectory* be defined as a polyline in $\mathbb{R}^2$, represented by a sequence $p_1, \ldots, p_m$ of m 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 the $\mathbb{R}^3$ domain with the coordinates x, y and t. In our settings, we usually parameterize (or normalize) the timestamp along the interval $[0,1]$.

Many applications use the trajectory data as historical resources, including statistics about outdoor activities, common route analysis, and traffic congestion analysis. Other applications use the data on-the-fly for current route statistics, the analysis of current relationships with other trajectories, and more. Trajectories for ships, airplanes, animals, and even people are constantly being collected and used in academic and industrial research.

The availability of trajectory data offer unique opportunities to gain insight. For example, analyzing the complicated relationships among trajectories can help alleviate traffic congestion and thus save time and money, while decreasing pollution. Trajectory data can also be used to examine human mobility and social patterns [17]. Such understanding can, in turn, help advance solutions to large-scale societal problems in fields as varied as telecommunications, ecology, epidemiology, and urban planning. For example, knowing how large populations of people move about can help determine their carbon footprint and subsequently guide policies intended to reduce that footprint.

Over the years, many problems that relate to trajectories have been proposed and solved. Four common problems relate to: measuring the distances between pairs of trajectories, matching sets of trajectories, clustering trajectories, and computing common routes. These problems have been extensively investigated in recent years (see Section 2). In this work we present a system whose major aspect revolves around trajectory analysis and unites the above common problems. We describe the system in detail in Section 3. The objective of this system is to extract trajectories of ants that live in a specific colony for scientific research and stitch Ű- or mosaic – together pieces of information from various trajectories to build a more complete picture that can be used for insightful analysis.

Ant trajectories are generally extracted by volunteers who annotate (possibly inaccurate) input trajectories (for more details refer to [9]). In this setting, the volunteers interact with an online game that plays a video of an ant colony and annotate the trajectory of specified ants via mouse clicks. This task is challenging for computer vision dur to the density and the interactions and collisions of the ants. Unfortunately, the trajectories annotated by the volunteers are not quite useful for further processing. First, the required trajectories span long periods of time, which are much longer than a typical volunteer would be able to comfortably annotate. Hence, we need to stitch together different annotated section. This requires allowing some overlap to enable the stitching. Morever, since the volunteers have some control over the annotation process, their annotations may not be sampled at the same rates or may have gaps. Second, volunteers make mistakes. Beyond innocent mistakes, there are also cases of malicious or purposeful mis-annotation. Thus, it is desirable to obtain several trajectories of the same ant for each period of time. Another difficulty is that since we cannot predetermine the identity of each ant, we cannot fit the annotations to specific ants. This fact complicates our work greatly as we need to decide which annotation belongs to which ant. Since our ultimate goal is to identify the complete trajectories of the ants, our work focuses on solving the following problems.

1. **Clustering trajectories.** Cluster the trajectories so that each cluster corresponds to a specific ant. In this regard, we relate to trajectories for a similar period of time.
2. **Computing common route.** After the trajectories have been clustered, we need to compute a common route that represents the trajectories of the ants during specific periods of time.
3. **Matching trajectories.** To generate the complete trajectories, we must decide which sub-trajectories to stitch together for specific periods of time. We do this by matching the common routes on the overlapping parts.
4. **Measuring the distance between trajectories.** In order to cluster and match the trajectories, we need to use a metric that measures their distances from each other.

While we use prior work that is appropriate for our needs for items 1 and 2, we devised new techniques for items 3 and 4. We note that these techniques are not limited to our ant system, and can be useful for other systems that process trajectories (give examples).

To find matching trajectories, we use the KuhnŰMunkres algorithm algorithm [20] after computing distances between the common routes. However, computing the distance measure is costly, and in particular redundant if one can disqualify the pairing potential of far trajectories Fast. We use two complementary techniques to quickly filter out far enough trajectories: locality sensitive hashing (LSH) and R-tree augmentation. Locality Sensitive Hashing (LSH) filters out far trajectories, but due to its nature it may skip some pairs (see Section **??** for more details).Say why LSH skips in its section. Augmentation of the well-known R-tree algorithm helps us further disqualify far trajectories that were skipped by LSH.

There are many applications that can benefit from a speed up heuristic of this type. (examples) In some of these applications the trajectory of the same moving object is measured using different devices, requiring the data to be merged into a single trajectory to get more reliable and accurate data. Other examples that could benefit from overlaying data from several sources are ones that use GPS data, in which the data is not reliable (e.g., at dense forests or under water) or other data coming from WiFi and cellular networks.

To compute the distance measure between trajectories, we use the popular Fréchet distance (cite). The Fréchet distance is computed for two trajectories, ignoring their associated timestamps. Clearly, there are cases when we might be interested in computing this distance when some of the samples are associated with a timestamp and some are not (see Section 4.1.3). We augmented the Fréchet distance computation to support this kind of data and thus generalize the computation. Because the ant trajectory data has associated timestamps, we show how we can interpolate the trajectories with samples that have no timestamp and achieve potentially better distance quality.

The remainder of this paper is organized as follows. In the next section we discuss related work. In Section 3 we present the system that analyzes ant trajectories. In Section 4 we present the techniques we use for trajectory analysis. In Section 5 we show how we solve the main problem that is described in Section 3. In particular, we describe how we cluster the trajectories, compute their common Routes, and match the latter to create the final results. 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

Two of the most popular analytics works on trajectories relate to measuring the similarity between trajectories and partitioning them into meaningful clusters. The two problems often go hand in hand, since any spatial clustering algorithm depends on the measure of similarity for the entities being analyzed.

Partitioning trajectories into clusters is interesting for various applications such as prediction, classification, identification of common patterns, computation of the mean trajectory, or identification of data characteristics [21]. In many cases it is interesting to cluster sub-trajectories. Lee et al. [14] proposed a technique to segment the trajectories into a set of line segments and then group similar line segments. The distance function the define for the segments depends on the perpendicular, parallel, and angle distances of the segments. Chen et al. [7] used clustering techniques to compress trajectories. The goal was to compute a compact dictionary on sub-trajectories to minimize their representation. Buchin et al. [5] refer to the problem of optimizing clusters of sub-trajectories as a game of three parameters: minimum number of sub-trajectories in the cluster, maximum sub-trajectory pairwise distance, and minimum length of the longest sub-trajectory. Fixing two of the parameters, they define and solve the induced optimization problems in which the other parameter is optimized.

A customary by-product of the clustering process is a common route associated with each cluster. In our work the common route represents a sub-rajectoy of a specific ant. Using the common route thus gives us an efficient temporary data to build the entire trajectories. Lee [14] suggests constructing the common route by sweeping along the major axis of the cluster segments.

Other clustering concepts involve the movement of entities along these trajectories. Gudmunson and van Kreveld [15] relate to the temporal nearness of moving entities. They define a *flock* as a group of entities moving near each other for a certain time, where the size of the flock should be large enough but not necessarily fixed (entities may enter and leave it). They also define a *meeting* as a nearby location where many entities go during some time interval. Kalnis et al. [19] study the general problem of moving clusters. For more interesting work on various angles of this domain, we refer the reader to [22, 16, 13].

Measuring the distance between two trajectories is challenging because it involves synchronizing the movement along the trajectories while minimizing some criteria that is subjective and case dependent. Over the years, various techniques have been proposed and adapated from other disciplines to propose suitable measures.

The Fréchet distance measures the maximum distance between two elements as they synchronize their walk along the trajectories. It is often illustrated as the minimum leash a person needs while walking his dog, where their trajectories are known in advance [3]. (For a more formal discussion see Section 4.1.1.) The Fréchet distance has been used for trajectory clustering, reconstructing road maps from GPS trajectories [4], aligning networks [2], protein matching [18], and more. Recently, the Fréchet distance was studied by Buchin *et al.* [6], who demonstrated how to incorporate time-correspondence and directional constraints. The discrete version can be solved by straightforward dynamic programming of $O(mn)$ time and space where m and n are the link length of the trajectories. Recently, Agarwal et al. [1] proposed the first sub-quadratic algorithm to compute the Fréchet distance. The continuous version, however, is more complicated to solve. A popular solution is to work on the arrangement of the free-space parametric domain. For a given δ, testing whether the Fréchet distance is upper bounded by δ is done by trying to find a monotone path from the source to the destination in the parametric space. The overall time in this case is $O(mnlog(m+n)))$. Faster algorithms are known for curves that satisfy some "reasonable" properties of realistic input models [10].

Dynamic Time Warping (DTW) differs from the Fréchet distance in the sense that it considers the maximum or average distance. This measure has been very popular in a variety of contexts, including: speech recognition, databases, computer vision, and protein structure matching. A major drawback of this technique is that it makes the continuous version very expensive to compute. Efrat et al. [12] proposed an efficient algorithm for a similar measure that is defined over continuous domains. More techniques that have been adapted from other domains are the *longest common sub-sequence* [23], the *edit-distance* [8] and the *sequence alignment* [11]. Their advantages over the Fréchetdistance and DTW is that they do not require all samples to be matched, thus allowing filtering noise and outliers. However, they allow each sample to be matched at most once. Recently Sankararaman et al. [21] removed this restriction with their *local-assignment* model. They show how to support a semi-continuous model in which they interpolate the samples to improve the results. Vlachos et al. [23] list features that a good distance function should satisfy: supporting different sampling rates and different motions, handling outliers, supporting different trajectory lengths, and being efficient enough for computation. Chen et al. [8] claim that the computation similarity is sensitive to noise, shifts, and scaling, which are common results of sensor errors and failures. They introduce a variant of the edit distance measure that stands up in the face of these imperfections.

3. PROBLEM DESCRIPTION

4. TECHNIQUES FOR PROCESSING TRAJECTORIES

4.1 Distance Measures

To measure the distance beween two trajectories, we chose a metric that relates to the maximum distances along them. We distinguish among three cases, each relates to the availability of timestamps on trajectory samples. In the first case, timestamps are not available; this is denoted by No Time Associated Setting (NTAS) and is described in Section 4.1.1. In the second case, timestamps are available; these are denoted by Time Associated Setting (TAS) and described in Section 4.1.2. In the third case, timestamps are partially available; these are denoted by Partial Time Associated Setting (PTAS) and described in Section 4.1.3. Note that for ant trajectories, the time is generally available on the samples and thus corresponds to TAS. However, we show how we can improve the computation precision with PTAS, which is based on NTAS. We also describe other scenarios in which PTAS is useful.

4.1.1 No Time Associated Setting (NTAS)

When the time is not given, the common Fréchet distance is a suitable Measure for the distance of the trajectories. While computing the distance, the algorithm also assigns parametric timestamps on the samples. A common way to illustrate the Fréchet distance is with the man walking a dog analogy, where the trajectories for man and dog are known in advance and both can only move forward along their trajectories.[1] The Fréchet distance is then the length of the shortest leash between the man and the dog that allows 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 γ, we can check whether the Fréchet distance between two trajectories u and v is upper bounded by γ 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; otherwise it is *invalid*. 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 weakly monotone path from $[0,0]$ to $[1,1]$ in H passing through valid grid points; see Fig. 1.

In order to find $D_F(u,v)$, one may perform a binary search over the values of γ, finding the minimum value for which a valid path exists. Unfortunately, the free space diagram is defined with conic arcs and thus must support a range of 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. In this case, the Fréchet distance corresponds to a sequence of steps done

[1]The weak Fréchet distance is defined similarly with the difference that back tracking is allowed.

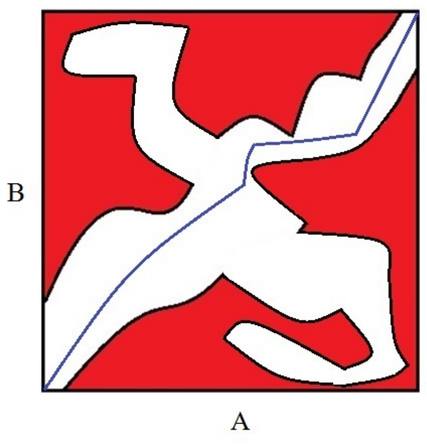


Figure 1: The free space diagram for trajectories A and B and some fixed γ. The valid regions are in white. The blue monotone path shows one way to parameterize the curves so the distance between any two matched points is always less than γ.

on the two trajectories: in each we advance from a station to its subsequent point 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 many cases is relatively close to it.

For a given γ, we query whether the discrete Fréchet distance is smaller than γ 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 and u_i, v_j are the i-th and j-th points of u and v, respectively. Next, we test if there is a weakly monotone path of cells with value 1 from the bottom left of the matrix to its top right, using a traditional dynamic programming pattern (each step in the path is thus horizontal, vertical or diagonal for moving on the first trajectory, second trajectory and both trajectories, respectively). Such a weakly monotone path corresponds to a parameterization valid for γ and thus the existence of such a path is the result of the query.

A simple algorithm for computing the discrete Fréchet is to perform a binary search over γ and find the minimum γ valid for the two trajectories as described above. The algorithm takes $O(m^2 \log D_{max})$ time, where m is the link-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. Note that the above algorithm is not asymptotically optimal; Aggarwal et al. [1] described a more efficient algorithm.

As the discrete version is faster to compute and easier to implement than the continuous one, we use it in the remainder of this work.

4.1.2 Time Associated Setting (TAS)

When timestamps are associated with the trajectory samples, trajectories can be represented as polylines in $\mathbb{R}^3$ (with x, y and t coordinates). To compute the distance, we simply sweep the $t = C$ plane for 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. The maximum value obtained over all pairs of segments is the similarity measure 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.

4.1.3 Partial Time Associated Setting (PTAS)

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. The idea is to solve this variant by modifying the algorithm for NTAS, 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 T_2 cannot reach t_2 before T_1 reaches t_1. This constraint defines a *forbidden rectangle* in the free space diagram; see Figure 2. Given T_1 and T_2, we process them by increasing timestamps. For each timestamp 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 weakly monotone path. It is important to observe that each trajectory samples is associated with at most two rectangles that correspond to constraints with two samples on the other trajectory, appearing before and after it. Thus, the number of constraining rectangles is linear and does not affect the complexity of the free space diagram. These ideas hold for both the continuous and the discrete versions with the necessary changes. Detecting the samples inside the forbidden rectangles in the discrete version can be done in $O(T^3)$ time (recall that T is the trjectory link length), 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. Based on the characteristics of the forbidden rectangles, they form two staircase structures in the free space, touching the bottom-right and the top-left of the free space respectively; see Fig. 2.

PTAS is practical for cases where the timestamp is given partially. This occurs when the data is not reliable or when the sensors that report the location/timestamps need to save energy and thus report only partial data. It can also be useful for improving the precision of the distance as we describe next. Given two trajectories with associated timestamps, instead of computing the distance as proposed in Section 4.1.2, we add sample points in between given sample points. If we knew the real timestamps of those new points, we could clearly compute a possibly more precise Fréchet distance with TAS. Nevertheless, by using PTAS, where only some original points have associated timestamps, we will find optimal parametric time on the new points, potentially improving the precision of the distance. This case is even more interesting when we do not model the movement between samples linearly, but rather with non-linear techniques (e.g., Bazier curves or high order polynomials). Then we could take points along the approximating curves to improve the computation.

4.2 Locality-Sensitive Hashing

(extend it to other settings)

TAS. 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)

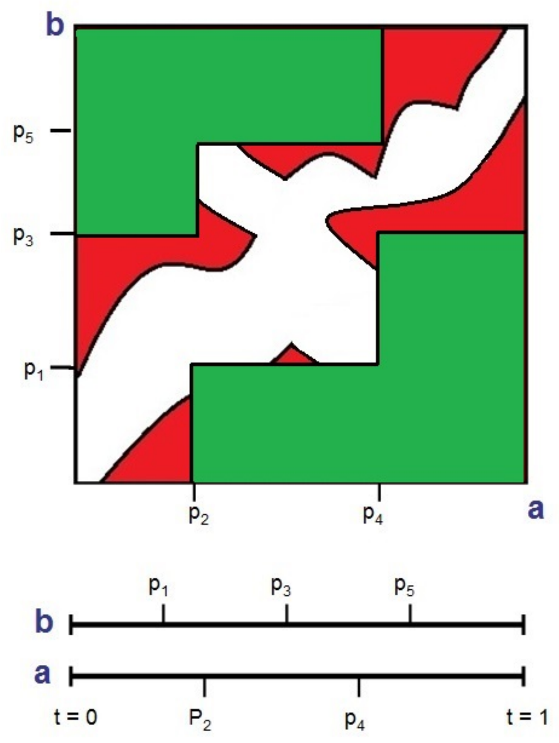


Figure 2: The free space diagram for a pair of trajectories a and b overlayed with forbidden rectangles (in green) that contribute to the invalid parameterization (the red areas). The parameterization of the trajectories are depicted below.

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 $O(n^2R)$ to $O(nR)$, and the complexity of finding a matching is reduced from $O(n^{2.5}\log D_{max})$ to $O(n^{1.5}\log D_{max})$.

NTAS

1. Dimensionality reduction with non-uniform sampling

2. Assume $p_{i,j}$ is the location (vertex) of ant i at time j.

3. Let the number of time slots be denoted by $M/2$. That is, a trajectory could be thought of as a point in $I\!R^M$.

$$p_{i1}.x, p_{i1}.y,\ p_{i2}.x, p_{i2}.y,\ \dots p_{i,\frac{M}{2}}.x, p_{i,\frac{M}{2}}.y$$

   where $p_{i1}.x$ and $p_{i1}.y$ are the x and y coordinates of p_{i1}.

4. Let $\vec{r}$ denote a random vector in R^M. Out goal is to find the projection of each trajectory π on the line $\{t\vec{r} | t \in I\!R\}$. That is, $(t \cdot \vec{r}) \cdot (\pi - t\vec{r}) = 0$ or in other words $t|\vec{r}|^2 = \vec{r} \cdot \pi$, or $t = \frac{\pi \cdot \vec{r}}{|\vec{r}|^2}$.

5. We next show how to obtain the value of t in $O(\log M)$ time, when π is sampled sparsely. Let $\vec{r} = (\gamma_1, \gamma_2 \dots \gamma_M)$.

6. To compute the projection of π' on the line $\{t\vec{r} | t \in R\}$. Assume first π is a single segment (span from all time)

   we need to compute t such that $(t \cdot \vec{r}) \cdot (\pi - t\vec{r}) = 0$ or in other words $t|\vec{r}|^2 = \vec{r} \cdot \pi$, or $t = \frac{\pi \cdot \vec{r}}{|\vec{r}|^2}$.

7. **The case that π is a single segment.** Next assume that π consists of a single segment, and all vertices along π are uniformly spread along it, so $p_{ij} = p_{i1} + j\vec{\mu}$ where $\vec{\mu} = \frac{p_{iM} - p_{i1}}{M}$ and $j = 0, 1, 2 \dots M-1$. Note that $\vec{\mu} \in I\!R^2$. Hence if $\vec{\mu} = (\mu.x, \mu.y)$, then

$$\pi \cdot \vec{r} = \mu.x \cdot \sum_{j=1,3,5\cdots}^{M-1} \gamma_j \ + \ \mu.y \sum_{j=2,4,6\cdots}^{M} \gamma_j$$

   Pre-computing $s_{0..M} = \sum_{j=1,3,5\dots}^{M-1} \gamma_j$ and $\bar{s}_{0..M} = \sum_{j=1,3,5\dots}^{M-1} \gamma_j$, we see that $\pi \cdot \vec{r}$ could be computed in $O(1)$ time.

5. MATCHING ALGORITHM

5.1 Clustering

5.2 Common Route

We want to combine and compute median trajectories from a set of smaller pieces; see Fig. 5.

5.3 Matching using Sequential Bounding Box Balanced Tree

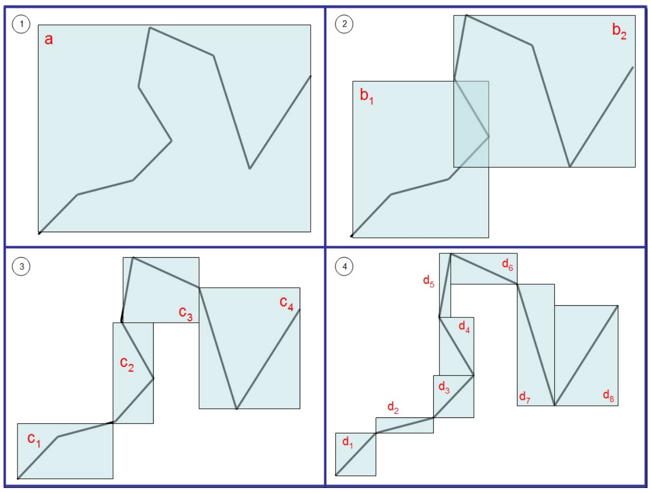
(Need to say something about the two possible matching algorithms we used.) (need an experimental evidence that the approach helps.)

In this section we describe how to match the trajectories from both sets (Rephrase to clarify what those sets are) based on their similarity. Formally, we want to find a matching $M = \{m_1, \dots, m_n\}$, 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 that minimizes the maximum distance between matched trajectories. Our problem, which we denote by TMATCH, is defined as follows.

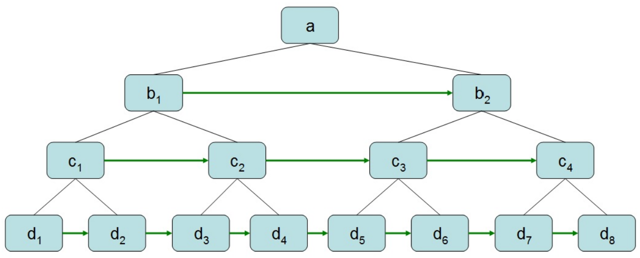
TMATCH: Given two sets of trajectories, $S_1 = \{u^1, \dots, u^n\}$ and $S_2 = \{v^1, \dots, v^n\}$, find a matching $M = \{m_1, \dots, m_n\}$, where $m_k = (u^i, v^j)$ with $u^i \in S_1, v^j \in S_2$ that minimizes

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

Matching two sets of trajectories can be viewed as a process of two steps: computing the pairwise distances of trajectories of different sets and matching the trajectories based on these distances. If the time taken to compute the distances is $(O(R(m)))$ where $R(m)$ is the time needed to compute the distances of trajectories, each of size $\leq m$, then the overall time is $O(n^2R(m))$. Using the algorithm by Agarwal [1], the overall complexity is $O(\frac{n^2m^2 \log\log m}{\log m})$, which might be unacceptable for large n and m. We observe that in real-world cases, two matching trajectories are much closer to each other than to many of the other trajectories. Consequently, it seems desirable to filter out most of the false pairs fast, saving the time for computing their distances. In the previous section we proposed the Locality-Sensitive Hashing to efficiently filter out many far pairs. As there is no guarantee that it will filter out any far pairs, we use another technique based on a geometric data structure that resembles the well known R-tree to filter out far pairs. We describe this data structure as a part of the complete matching algorithm and refer to it as *Sequential Bounding Box Balanced Tree* (SBBBT for short). This technique is applicable for NTAS and PTAS (but not for TAS). Its main usage is 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 of a specific trajectory. (b) The SBBBT of the trajectory. The levels in (b) correspond to the boxes in (a) as the symbols indicate. Green arrows connect vertices of the same level.

DEFINITION 5.1. *A* Sequential Bounding Box Balanced Tree (SBBBT)) *is a binary tree associated with a trajectory and 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. Each level of the tree (that is an order of nodes of the same height) contains corresponding pointers to allow its traversal. See Figure 3 for an illustration of SBBBT.*

Algorithm 1 is the high level algorithm for computing the matching. We first construct the SBBBT with $\log m$ 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. We use SBBBT to construct the graph that we inject into the perfect matching algorithm. It is easy to see that the matching corresponding to the smallest such γ is a bottleneck matching.

```
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

Note that the foreach loop in the algorithm is responsible for determining whether a pair of trajectories is a candidate for a matching and, if so, it is inserted into G. We execute this as follows. 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 define the Fréchet distance between $SBBBT(u,l)$ and $SBBBT(v,l)$ in a similar way as the discrete Fréchet distance between trajectories. The difference is that instead of points (the trajectory samples), we consider rectangles[2]. So instead of jumping from one point to another (on one or two trajectories) we traverse the rectangles accordingly, generating a sequence of rectangle pairs. The Fréchet distance is now the shortest leash connecting any pair of rectangles. 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, the Fréchet distance between u and v also exceeds γ, which is the Fréchet distance of the trajectories. Using this observation, we can 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 and so on. (See Figure 4). Only if the distance is less than γ at all levels,

[2]In principle this technique will work with any other shape as well.

we compute the Fréchet distance of the corresponding 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.

To further save time, we store the following information for pairs of trajectories. (1) If the Fréchet distance is computed at some iteration, we use this value instead of recomputing it during subsequent iterations. (2) If the pair is filtered out for some γ (that is, the Fréchet distance is proven to be 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.

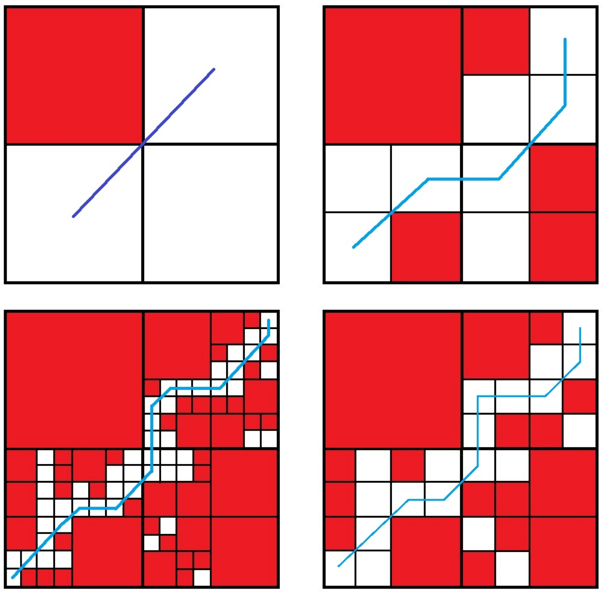


Figure 4: Computing the Fréchet distance with SBBBT. Decreasing levels (from top-left clockwise) of SBBBT are depicted on the parameterization diagram for a pair of trajectories. Red cells represent forbidden parameterization values. In this example, there is a weakly monotone path from bottom-left to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

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 [9]. 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,

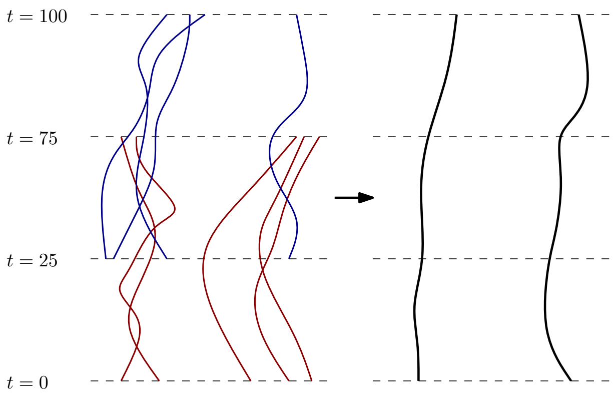


Figure 5: The input (left) is a set of red and blue trajectories. Trajectories of the same color span the same time period. The result (right) is combined black trajectories spanning the entire time period.

the problem is to identify trajectories corresponding to the same ant; see Fig. 6.

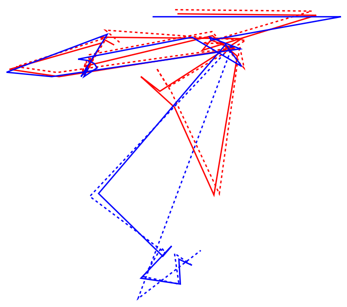
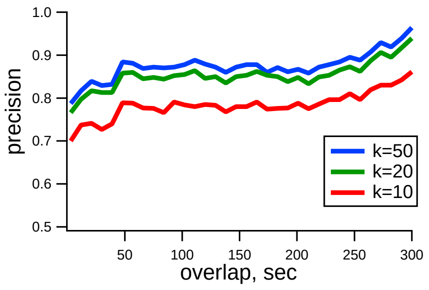


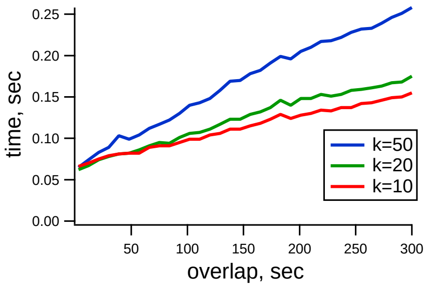
Figure 6: 4 citizen science trajectories corresponding to two different ants. Although the trajectories of the blue ant and the red ants are partially overlap, the matching algorithm is capable to identify 2 pairs of trajectories.

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 4.2. 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. 7(a) and Fig. 7(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



(a)



(b)

Figure 7: 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.

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. 8. 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 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. 9. 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 5.3. We first note that the running time depends on the size of the input; see Fig. 9. 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.

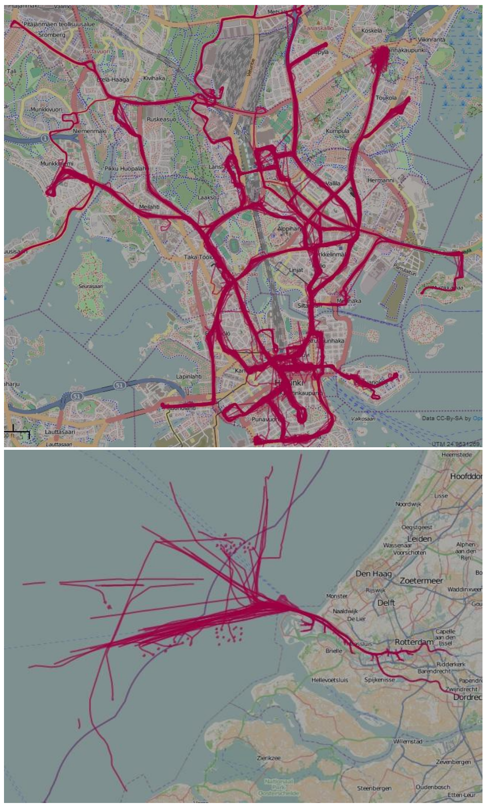
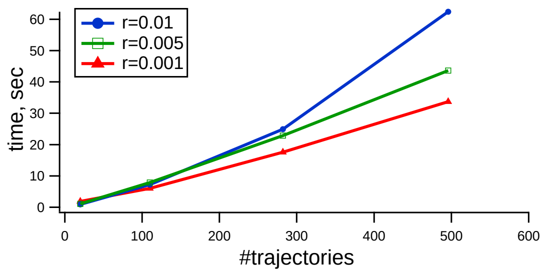


Figure 8: The trajectories we experimented with: Helsinki public transportation (top) and Ships near and at the port of Rotterdam (bottom).

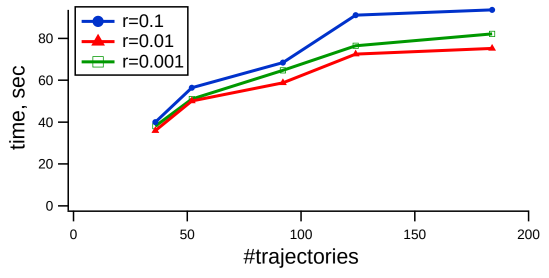
As the most time-consuming step in the matching algorithms is calculation of the Fréchet distance, we also report on how well our heuristic filters Fréchet computation calls. We define a *saving ratio* as the number of the pairs of trajectories for which we computed the Fréchet distance divided by the 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. 10. 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.

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

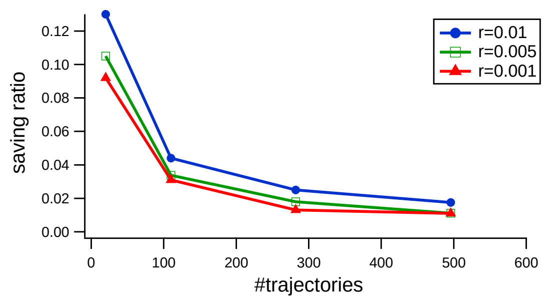


(a)

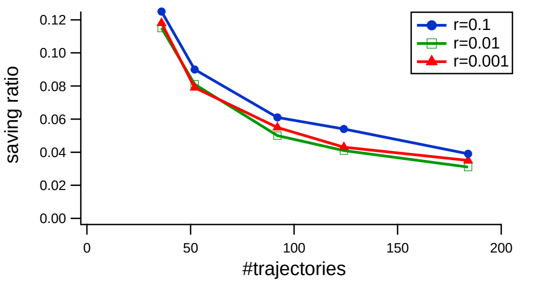


(b)

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



(a)



(b)

Figure 10: The saving ratio of the bottleneck matching algorithm on the (a) Helsinki and (b) Rotterdam datasets for different values of perturbation.

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. 11.

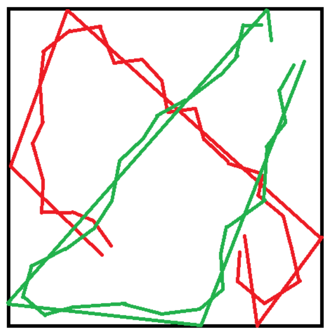


Figure 11: An example of two pairs of generated trajectories inside a square. Each pair is colored with a specific color.

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. 12. 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. 12.

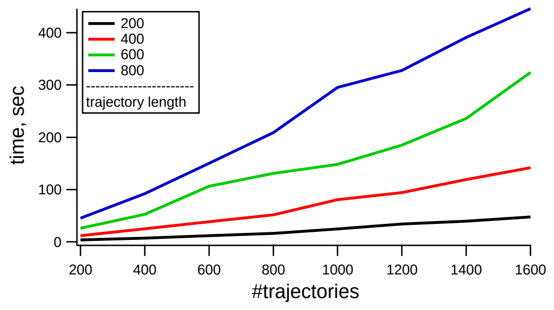


Figure 12: Results of the Artificial Trajectories experiment. The running time is shown as a function of the number of trajectories for different trajectory lengths.

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. 13. 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 $r \leq 2$.

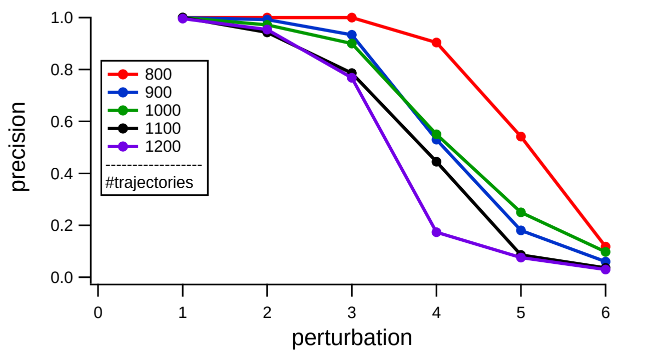


Figure 13: Precision of the results for the Artificial Trajectories dataset.

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. Lï£¡ffler, R. Silveira, C. Wenk, and L. Wiratma. Median trajectories. *Algorithmica*, 66(3):595–614, 2013.

[7] C. Chen, H. Su, Q. Huang, L. Zhang, and L. Guibas. Pathlet learning for compressing and planning trajectories. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'13, pages 392–395, New York, NY, USA, 2013. ACM.

[8] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 491–502, New York, NY, USA, 2005. ACM.

[9] 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.

[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] R. Durbin. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[12] 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.

[13] 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.

[14] J. gil Lee, J. Han, and K.-Y. Whang. Trajectory clustering: A partition-and-group framework. In *In SIGMOD*, pages 593–604, 2007.

[15] J. Gudmundsson and M. van Kreveld. Computing longest duration flocks in trajectory data. In *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems*, GIS '06, pages 35–42, New York, NY, USA, 2006. ACM.

[16] S. Hirano and S. Tsumoto. A clustering method for spatio-temporal data and its application to soccer game records. In *RSFDGrC (1)*, pages 612–621, 2005.

[17] 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.

[18] 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.

[19] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *Proceedings of the 9th International Conference on Advances in Spatial and Temporal Databases*, SSTD'05, pages 364–381, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[21] S. Sankararaman, P. K. Agarwal, T. Mølhave, J. Pan, and A. P. Boedihardjo. Model-driven matching and segmentation of trajectories. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'13, pages 234–243, New York, NY, USA, 2013. ACM.

[22] C. Sung, D. Feldman, and D. Rus. Trajectory clustering for motion prediction. In *IROS*, pages 1547–1552. IEEE, 2012.

[23] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *In ICDE*, pages 673–684, 2002.