On-Line Discovery of Hot Motion Paths*

Dimitris Sacharidis
Natl. Technical University
Athens 15780, Greece
dsachar@dblab.ntua.gr

Kostas Patroumpas
Natl. Technical University
Athens 15780, Greece
kpatro@dblab.ntua.gr

Manolis Terrovitis
Natl. Technical University
Athens 15780, Greece
mter@dblab.ntua.gr

Verena Kantere
Natl. Technical University
Athens 15780, Greece
verena@dblab.ntua.gr

Michalis Potamias
Boston University
MA 02215, U.S.A.
mp@cs.bu.edu

Kyriakos Mouratidis
Singapore Mgmt. Univ.
188065, Singapore
kyriakos@smu.edu.sg

Timos Sellis
Natl. Technical University
Athens 15780, Greece
timos@dblab.ntua.gr

ABSTRACT

We consider an environment of numerous moving objects, equipped with location-sensing devices and capable of communicating with a central coordinator. In this setting, we investigate the problem of maintaining hot motion paths, i.e., routes frequently followed by multiple objects over the recent past. Motion paths approximate portions of objects' movement within a tolerance margin that depends on the uncertainty inherent in positional measurements. Discovery of hot motion paths is important to applications requiring classification/profiling based on monitored movement patterns, such as targeted advertising, resource allocation, etc. To achieve this goal, we delegate part of the path extraction process to objects, by assigning to them adaptive lightweight filters that dynamically suppress unnecessary location updates and, thus, help reducing the communication overhead. We demonstrate the benefits of our methods and their efficiency through extensive experiments on synthetic data sets.

1. INTRODUCTION

Location-aware devices are ubiquitous nowadays, ranging from GPS-enabled cell phones and PDAs, to location sensing micro nodes and active RFID tags. This fact enables monitoring of moving objects and real-time analysis of their motion patterns. Positioning devices share a number of common characteristics: (i) typically, they are widely deployed in large areas; (ii) they communicate with central coordinators using an underlying network; (iii) they collect numerous, yet inherently inaccurate, location readings per minute; and (iv) they have limited resources, such as scarce battery life and little to moderate memory and processing power. Having in mind such a dynamic and distributed environment, we present a framework for on-line analysis of objects' movement that extracts motion patterns and identifies frequently traveled paths.

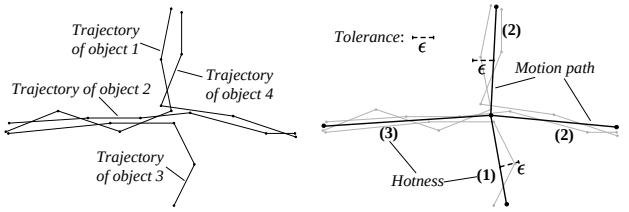
We present two distinct applications that motivate the need for on-line analysis of motion patterns. Consider the case of a mobile phone carrier that wishes to serve targeted advertisements to subscribers. This service would be based on people's profiles and *continuous* market basket information about other clients that follow similar paths. For instance, in case of a major sporting event, many subscribers are expected to move towards the hosting venue. En route, a large number of them may stop by at certain facilities (e.g., rest areas, kiosks, malls) to purchase food, drinks, etc. This buying pattern (i.e., many people shopping for similar types of products at specific locations), can be utilized to promote a particular store that has an advertising deal with the mobile phone carrier. For example, customers passing by the advertised store during the event may be informed about its exact location and its current promotions or discounts.

Motion path extraction could also assist in an emergency situation. Suppose that a fire breaks up in a rural area and spreads rapidly. Nearby villages may be at risk and their residents could panic due to conflicting news and/or guidance. As many people evacuate hastily their homes, they tend to follow similar paths. Authorities need to monitor their trails (tracked by their mobile phones) and immediately identify popular escape routes in order to direct assistance, e.g., ambulances, fire engines, etc.

Both examples discussed above require the on-line discovery and maintenance of paths followed by multiple moving objects. To the best of our knowledge, this is the first work on this compelling problem where we distinguish three challenging issues. First, numerous clients are expected to be on the move, so many object trajectories should be maintained. To enable effective decision making (e.g., advertising, alert scenarios), they need to be grouped and summarized, so that attention is drawn only to the most salient trails. We opt for a solution that consolidates multiple, neighboring trajectories into *motion paths* at the coordinator side. For each of them, we maintain its *hotness*, i.e., the number of objects that have recently traveled through it. Thus, end users are able to visualize/analyze only the hottest paths, and get a quick idea of the current situation. Figure 1 exemplifies this process, illustrating (a) the original object trajectories, and

*This work has been funded by the project PENED 2003, which is cofinanced 75% of public expenditure through EC - European Social Fund, 25% of public expenditure through Ministry of Development - General Secretariat of Research and Technology and through private sector, under measure 8.3 of OPERATIONAL PROGRAMME "COMPETITIVENESS" in the 3rd Community Support Programme.

EDBT'08, March 25–30, 2008, Nantes, France.



(a) Input Trajectories (b) Motion Paths

Figure 1: Motion path extraction

(b) the extracted motion paths and their hotness. As depicted, each motion path corresponds to a set of trajectory segments that evolve similarly, approximating them within a user-specified tolerance ϵ. In our framework, detected motion paths and their hotness are maintained in light-weight index structures enabling fast access. To restrict detection of salient paths to up-to-date readings, we impose a sliding time window of size W, which excludes from consideration any locations received more than W time units ago.

The second challenge in our design regards scalability in terms of communication overhead and computation cost at the coordinator. The naïve approach whereby all objects continuously relay their locations to the coordinator is practically infeasible because it incurs excessive bandwidth consumption, and may also lead to coordinator overloading due to the computational cost for motion path extraction. To alleviate these problems, we propose a distributed approach for processing and filtering location updates. Each object executes locally an algorithm (RayTrace) that compresses on-the-fly its trajectory abiding by a tolerance ϵ. Thus, the object itself reduces the number of locations that will be reported to the coordinator. Method RayTrace sets a permissible spatiotemporal extent, and transmits the recent trail only when the current object location falls outside this filter. Aided by the coordinator, RayTrace then sets a new filter that reflects better its current motion pattern. This approach exploits the computational capabilities at the client side, while substantially reducing the communication overhead (due to fewer location updates) and the processing cost at the coordinator (for summarizing trajectories into motion paths).

Dealing with the inherent inaccuracy of location measurements is the third major consideration. Position readings are imprecise; moreover, they carry different degrees of uncertainty, depending on the handset capabilities and the network infrastructure. A GPS-enabled PDA provides more accurate location tracking than a cell phone, which relies solely on cellular triangulation for estimating its position. Furthermore, phones with just a few surrounding base stations offer less accurate measurements. Our proposed framework takes into account this uncertainty, as well as its varying degrees, and reports motion paths with discrepancy guarantees.

Outline The remainder of this paper is organized as follows. Section 2 surveys previous work related to the motion path discovery problem. Section 3 formulates the problem and outlines the proposed solution. Section 4 describes the RayTrace algorithm for filtering positional updates. Next, Section 5 presents the path extraction process and the associated index structures. Section 6 includes our experimental study. Finally, Section 7 concludes the paper.

2. RELATED WORK

The problem of discovering frequently followed, i.e., hot, routes has also been examined in a recent work [17], but only for the case that objects are confined to move in a known network. A hot route in this context is a sequence of edges, not necessarily adjacent, that share a high amount of traffic. The approach presented in this paper differs in several aspects. First, it assumes unrestricted movement on the xy plane; second, in measuring hotness it considers the time interval that objects crossed each designated path; and, third, it accounts for imprecision in positional measurements.

Our work is relevant to the domain of spatiotemporal data reduction, particularly to the topic of trajectory compression. Most existing algorithms [5, 20] attempt to compress singleton trajectories in isolation, by adapting the off-line Douglas-Peucker algorithm [8]. This line simplification technique drops the least important vertices to achieve a reduced representation. It is widely used in spatial databases, but it requires multiple passes over the data, which yields it inapplicable to on-line streaming applications. Sampling techniques have been proposed in [23] for compressing isolated trajectories. Furthermore, segmentation of multiple trajectories by fitting into axis-parallel rectangles is considered in [1, 11], where dynamic programming, greedy and heuristic techniques are employed so as to minimize the empty space in rectangles or to preserve pair-wise distances among trajectories.

An adaptation of Douglas-Peucker algorithm more suitable to dynamic environments was presented in [20]. Instead of considering the entire trace of an object for applying line generalization, an opening window principle is employed to reduce the amount of timestamped locations considered at each step. The technique starts processing positions in temporal order and progressively produces successive line segments. More specifically, after fixing a starting point, the algorithm examines candidate line segments by setting their floating endpoint as much farther as possible, provided that all intermediate locations are within a given tolerance from the constructed segment. In case this rule is violated, two alternative policies were proposed for fixing the endpoint of the new segment. The conservative approach (DP-nopw) chooses the location that caused the violation, i.e., the one with the greatest distance from the examined segment. The eager approach (DP-bopw) takes the location with the greatest possible timestamp, which is the one just before the floating endpoint. Checking violations is very costly, since all locations between the starting point and the current floating endpoint must be examined each time. Overall, this method is constrained to choose a subset of the reported locations as endpoints and thus, it offers a rather strict trajectory synopsis. In Section 6 we describe an adaptation of this technique to hot motion path computation and use it as a competitor.

Detecting clusters of moving objects, moving clusters and frequent motion patterns has also attracted research interest. Clustering similar objects based on their movement characteristics, e.g., current position and velocity, is discussed in [18, 14]. More related to our problem is the work in [15] for identifying dense clusters of objects which move similarly over a long period of time. According to their definition clusters need not contain the same set of objects all along their lifetime. The difference from the problem we tackle here is twofold. First, although moving clusters evolve across a path that is interesting (i.e., hot), we only

need to identify and maintain the motion paths per se, and not the actual clusters or their constituent objects. The second, and most important, reason is that a motion path may be hot even when no moving cluster crosses it. To justify this, note that a moving cluster requires objects to be close enough to each other at *any time instant* during a sliding window of W time units. In contrast, a motion path can become important as long as a sufficient number of objects have crossed it in the last W time units, no matter if they travel synchronously or not across that path. The work in [19] computes spatial regions containing frequent periodic (e.g., daily, weekly, etc) motion patterns. Besides limitations including a priori knowledge of periodicity, this method treats trajectories merely as sequences of locations (i.e., it eliminates timestamps), hence, being inapplicable to our timestamp-sensitive problem.

Another topic related to our work is trajectory clustering. From a data mining perspective, a methodology was introduced in [9] and it was based on a probabilistic mixture of regression models, which the moving objects are assumed to follow. Also, detecting similarity among trajectories or timeseries has also attracted considerable research interest. Common problems have to do with outliers, local shifts in time, as well as movements of varying total length. Several distance measures have been proposed in order to identify similar trajectories or subsequences, e.g., Time Warping Distance [28], Longest Common Subsequence [26], and Edit Distance on Real Sequences [6]. However, all these techniques are not particularly tailored to handle on-line trajectories, since they require comparisons over large portions of objects' movement, while their objective is to group together trajectories in their entirety. Hence, they fail to identify trajectories that locally follow common routes, because the overall computed distance is greatly affected by distant segments. The most recent approach was presented in [16] and proposes a technique for clustering smaller linear partitions instead of entire trajectories stored in a database. The main idea of the algorithm is that trajectories are first split into several parts at characteristic points and then similar line segments are grouped together into a cluster. For identifying common motion patterns, the minimum description length (MDL) measure is adapted from the domain of pattern recognition, but it seems quite sensitive to appropriate selection of parameters. Moreover, time is ignored and trajectory segments are considered to be spatial polylines.

In this work, we assume a distributed stream environment, where multiple, geographically dispersed objects transmit data to a single coordinator. The latter assumes the role of consolidating, processing and presenting results to users. Such a setting is common in many sensor network applications and data streaming systems. For example, in [21, 7, 2] the coordinator aims to minimize communication cost by appropriately setting and updating filters on data sources that enable on-line calculation of counts, quantiles, and top-k entities, respectively. Filters of a spatial nature have been employed in the literature on continuous monitoring of range [10, 4, 24], nearest neighbor [12, 27] and medoid [22] queries.

Our handling of inaccuracy is aligned with the increasing interest in managing imprecise and uncertain data, such as in [3]. Closely related to our model of uncertainty, albeit not for spatio-temporal data, is the work in [25] that proposes an index for storing and querying imprecise spatial locations modeled by some probability density function.

3. PRELIMINARIES

In this section we formulate the addressed problem and present the assumed system architecture, in Sections 3.1 and 3.2, respectively.

3.1 Problem Formulation

We consider objects moving in the xy plane and hence all spatial locations are points $\mathbf{p}_i = (x_i, y_i)$. A point $\mathbf{p}_i$ accompanied with a timestamp t_i is called a *timepoint* and denoted as $\langle \mathbf{p}_i, t_i \rangle$. The *trajectory* of an object consists of a set of timepoints $\mathbf{T} = \{\langle \mathbf{p}_i, t_i \rangle\}$. The location of an object at time t_i is denoted as $\mathbf{T}(t_i) = \mathbf{p}_i$. Following common practice, between any two consecutive timestamps the object is assumed to move with constant velocity. As a result, the object's location at time t_k, where t_i, t_{i+1} are consecutive timestamps and $t_i < t_k < t_{i+1}$, is considered to lie in the (directed) segment $\mathbf{p}_i\mathbf{p}_{i+1}$ and can be calculated using linear interpolation. In the following, we assume that time is discrete and that all timestamps are multiples of some time granule.

We say that a point $\mathbf{p}_a$ is *close* to an object with trajectory $\mathbf{T}$ if there exists a time t_k such that $\mathbf{p}_k = \mathbf{T}(t_k)$ is within distance ϵ to $\mathbf{p}_a$, where ϵ is a user-specified tolerance parameter. In other words, the object has passed near $\mathbf{p}_a$ at some time t_k. Even though our methods apply to any L_p metric (including the Euclidean), for ease of illustration in the following we assume the max-distance, i.e., the distance between $\mathbf{p}_a$ and $\mathbf{p}_k$ is defined as $\max\{|x_a - x_k|, |y_a - y_k|\}$. Given tolerance ϵ, a directed line segment $\mathbf{p}_a\mathbf{p}_b$ is called a *motion path* if there exists a time interval $[t_a, t_b]$ such that point $\mathbf{p}(\lambda) = \mathbf{p}_a + \lambda(\mathbf{p}_b - \mathbf{p}_a)$ is close (within distance ϵ) to some object's location $\mathbf{T}$ at time $t(\lambda) = t_a + \lambda(t_b - t_a)$ for all $\lambda \in [0, 1]$[1]. We say that the object *crosses* the motion path and, inversely, that the motion path *fits* the object's movement. Intuitively, an object traveling during time interval $[t_a, t_b]$ along motion path $\mathbf{p}_a\mathbf{p}_b$ would always be within distance ϵ to another object.

Figure 2 draws with bold line the trajectory of an object moving along the x axis versus time t. The shaded envelope represents all points that are within distance ϵ to the trajectory (at some timestamp). Figure 2 also shows 4 motion paths, $\mathbf{p}_a\mathbf{p}_b$, $\mathbf{p}_c\mathbf{p}_d$, $\mathbf{p}_e\mathbf{p}_f$, $\mathbf{p}_g\mathbf{p}_h$. The object crosses these motion paths during the time intervals, $[t_a, t_b]$, $[t_c, t_d]$, $[t_e, t_f]$, $[t_g, t_h]$, respectively. A motion path paired with its associated time interval draws a line segment on the xt plane that is completely inside the shaded envelope.

Note that for a single object and for any time interval one could find an infinite number of motion paths. Fix some object i, and let $S_i = \{\langle \mathbf{p}_a\mathbf{p}_b, t_at_b \rangle\}$ denote a set of pairs consisting of a motion path $\mathbf{p}_a\mathbf{p}_b$ that fits the object's movement together with the time interval $[t_a, t_b]$ during which the object crosses it. We say that S_i is a *covering* motion path set for object i if at any time t_k the object either crosses a single motion path, or crosses two motion paths $\mathbf{p}_a\mathbf{p}_b$, $\mathbf{p}_c\mathbf{p}_d$, but $t_k = t_b = t_c$ and $\mathbf{p}_b \equiv \mathbf{p}_c$, i.e., one's start point is the other's end point. A covering motion path set implies that one could construct a hypothetical object whose trajectory is always close to object i's trajectory. For this reason a covering set can be considered as a simplification of the object's movement. A motion path is considered *valid* if it belongs to

[1]Since time is discrete, the λ values are selected so that $t(\lambda)$ is a valid timestamp.

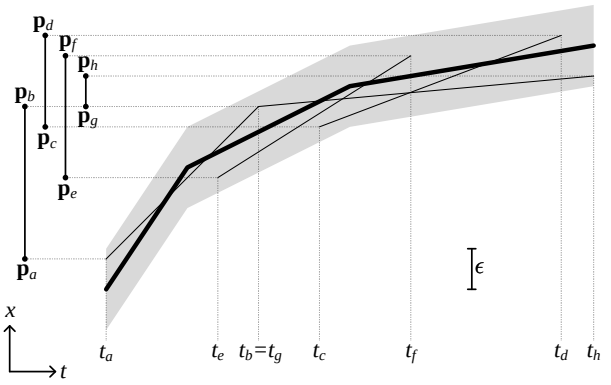


Figure 2: Motion paths example

a covering motion path set for some object. In the remainder of this paper, we only deal with valid motion paths and, thus, omit the valid denotation. Returning to the example in Figure 2, $S = \{\langle \mathbf{p}_a\mathbf{p}_b, t_at_b\rangle, \langle \mathbf{p}_g\mathbf{p}_h, t_gt_h\rangle\}$ is a covering motion path set for the object considered. Indeed, $\mathbf{p}_b \equiv \mathbf{p}_g$ and $t_b = t_g$.

In the previous, we have assumed that the object's location is accurately known. In a more realistic setting, though, the location sensing device reports coordinates with a degree of spatial uncertainty. The position of an object constitutes a random vector $\mathbf{P}_i = (X_i, Y_i)$, where X_i, Y_i are independent random variables. Let us note that there is no uncertainty regarding the timestamp. We repeat the previous definitions considering spatial uncertainty. Given tolerance ϵ and δ, we say that a point $\mathbf{p}_a$ is *close* to an object with trajectory $\mathbf{T}$ if there exists a time t_k such that $\mathbf{P}_k = \mathbf{T}(t_k)$ is within distance ϵ to $\mathbf{p}_a$ with probability greater than $1-\delta$. Assuming the max-distance metric, we require:

$$Pr\left(\max\{|X_k - x_a|, |Y_k - y_a|\} \leq \epsilon\right) \geq 1 - \delta.$$

The motion path in the presence of spatial uncertainty is defined accordingly, considering the aforementioned definition of proximity.

Recall that a motion path could fit multiple objects (or even the same object) during different time intervals. We define the *hotness* of a motion path to be the number of times objects have crossed it during the past W time units. The problem of hot motion path discovery can be formulated as follows:

> **Problem 1 [Hot Motion Paths]** For a set of moving objects, given tolerance ϵ (or ϵ, δ) and a time window of length W, find covering motion path sets and report the top-k hottest motion paths. □

Intuitively, Problem 1 states that we wish to discover motion paths that are crossed frequently by many objects. Depending on the chosen covering motion path sets, the characteristics of the top-k hottest motion path can vary greatly. Since this problem is motivated by the need to identify generalized frequent flows of movement, the best top-k result should ideally contain motion paths that are as large as possible (abiding by the tolerance parameters) and as hot as possible. Hot and large motion paths clearly convey more information (e.g., objects have crossed them and stayed close to each other for a long time), compared to short, but equally hot paths. To assess the quality of the top-k hottest motion paths, we devise a simple metric, termed *score*, that promotes longer paths. The score of a motion path is defined as its hotness multiplied by its length, and the score of the top-k set is the average score of its motion paths.

Given this notion of quality, the discovery process set forth in Problem 1 requires us to carefully construct long motion paths so that they fit as many objects as possible. Considering the freedom in choosing covering motion path sets for each object, this clearly becomes a daunting task. To emphasize on the latter, consider the case of a single moving object. Problem 1 degenerates to summarizing the object's trajectory with the fewest, and hence longest, segments. The solution [13] to this degenerate case requires two passes over the timepoints and requires linear space and time. As we discuss in the next section, such algorithms are prohibitive in our setting since they require storing all timepoints seen so far. Before proceeding to this section and the system model description, we summarize in Table 1 definitions and notation used throughout the paper.

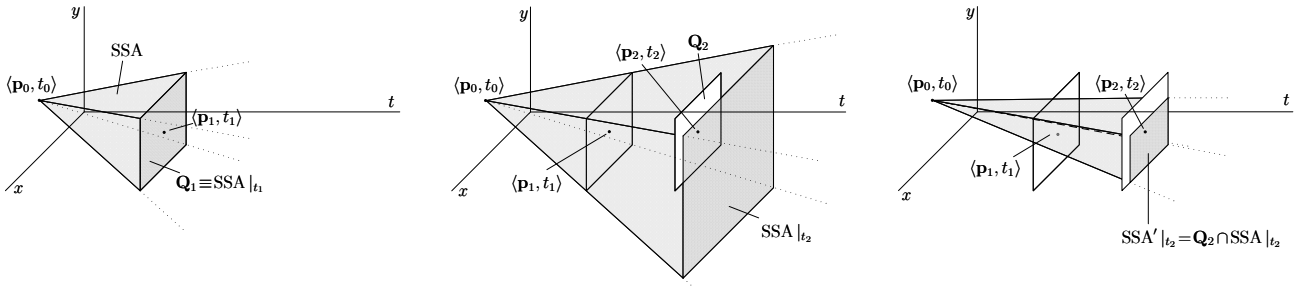
Symbol	Description
$\mathbf{p}_i = (x_i, y_i)$	point in xy space
$\langle \mathbf{p}_i, t_i\rangle$	timepoint in xyt space
$\mathbf{T}_i = \{\langle \mathbf{p}_j, t_j\rangle\}$	trajectory of object i
ϵ, δ	tolerance parameters
$\langle \mathbf{s}^i, t_s^i\rangle$	start of a motion path for object i
$\langle \mathbf{e}^i, t_e^i\rangle$	end of a motion path for object i
Tolerance Square	square of side 2ϵ around point $\mathbf{p}_j$
Spatial Safe Area (SSA)	pyramid $(\mathbf{l}^i(t), \mathbf{u}^i(t))$ in xyt space
Final Safe Area (FSA)	rectangle $(\mathbf{l}^i, \mathbf{u}^i)$ at time t_e
$State_i$	the state transmitted to coordinator
W	time window
Λ	processing epoch
h_j	hotness of motion path $\mathbf{p}_j\mathbf{p}_{j+1}$
$\mathcal{AP}_i$	available motion paths for object i
$\mathcal{CP}_i$	candidate motion paths for object i
$\mathcal{AV}_i$	available vertices for object i
$\mathcal{CV}_i$	candidate vertices for object i

Table 1: Primary symbols and functions

3.2 System Model

We consider an environment where the moving objects are geographically distributed and can communicate with a central coordinator. Each object is capable of sensing its own location with some uncertainty (modeled by tolerance ϵ, δ) and is capable of performing simple processing tasks requiring little memory. In this setting, the coordinator must maintain hot motion paths by collecting information from the objects.

There are two main issues we must take into account in this setting. First, objects have scarce battery life. Sending messages over the communication channel is typically orders of magnitude more power consuming compared to CPU processing. Following common practice, we must strive to minimize communication to and from the coordinator. Furthermore, objects listen for incoming messages only at predefined time instances termed *epochs*, i.e., every Λ time units. The second issue is the streaming nature of location measurements. An object should not store the unbounded stream of measurements, let alone transmit it to the coordinator; rather, it should only store information necessary



(a) $t = t_1$ (b) $t = t_2$, before (c) $t = t_2$, after

Figure 3: Updating the SSA

to discover motion paths. Consequently, all processing must be performed in a single pass over the stream.

We propose a two-tier approach. The first tier involves a one-pass greedy algorithm, termed RayTrace, running on each object independently. The second is a discovery strategy, termed SinglePath, that runs on the coordinator and utilizes a lightweight index structure, termed MotionPath, for storing the hot motion paths. The RayTrace algorithm acts as a filter maintaining a permissible spatiotemporal extent, termed *Spatial Safe Area* (SSA), around the object's trajectory. When a location measurement falls outside the SSA, the current state of the object is sent to the coordinator; a response will arrive in the next epoch. The coordinator executes the discovery strategy in the following manner. It processes messages from all reporting objects and extracts motion paths for each of them using information found in MotionPath. Finally, in the upcoming epoch, it sends a message to each reporting object informing them about the motion path they just crossed. We present in detail the RayTrace algorithm in Section 4, while we discuss the index structures and discovery strategy in Section 5.

4. FILTERING POSITION UPDATES

The RayTrace algorithm constructs a permissible spatiotemporal extent (the aforementioned SSA) around an object's trajectory, given some tolerance. RayTrace is a one-pass greedy algorithm that requires only constant per-measurement processing time and constant space. We first examine the case of tolerance ϵ; the adaptation to uncertainty, modeled by tolerance (ϵ, δ) is presented in Section 4.1.

The SSA is a spatiotemporal extent defined by the area between an initial timepoint $\langle \mathbf{s}, t_s \rangle$ and a rectangle, termed *Final Safe Area* (FSA), at time t_e. The main property of SSA is that a motion path $\mathbf{se}$ exists such that $\mathbf{e}$ lies inside FSA and the object crosses it during $[t_s, t_e]$. The objective of the RayTrace algorithm is to identify the latest timestamp t_e, and hence the largest SSA, such that a motion path can be found for the $[t_s, t_e]$ interval. Once RayTrace determines that the SSA cannot grow larger without violating the tolerance parameters, it notifies the coordinator about its state. The coordinator executes a discovery strategy and responds with a timepoint $\langle \mathbf{e}, t_e \rangle$, which serves as the initial timepoint for the new SSA to be constructed by RayTrace. The requirement that the endpoint is the next initial timepoint guarantees that we construct a covering motion path set.

The SSA is uniquely identified by an initial timepoint $\langle \mathbf{s}, t_s \rangle$ and an FSA at time t_e. Alternatively, we can denote the SSA as a time parameterized rectangle $(\mathbf{l}(t), \mathbf{u}(t))$ for $t_s \leq t \leq t_e$, so that $\mathbf{l}(t_s) \equiv \mathbf{u}(t_s) \equiv \mathbf{s}$ and $(\mathbf{l}(t_e), \mathbf{u}(t_e))$ defines the FSA. We use the notation SSA$|t_i$ to imply the projection of the SSA at time t_i; thus, FSA = SSA$|t_e$.

Algorithm 1 illustrates RayTrace in detail. In the following we describe the most important step in RayTrace, updating the SSA. Given tolerance ϵ, each timepoint $\langle \mathbf{p}_i, t_i \rangle$ is associated with a square $\mathbf{Q}$ of side 2ϵ around $\mathbf{p}_i$, termed *tolerance square*. When examining a timepoint $\langle \mathbf{p}_i, t_i \rangle$, (Lines 24–40 in Algorithm 1), RayTrace must update the SSA so that its projection at t_i is not greater that the tolerance square. It first computes the projection SSA$|t_i$ (Lines 26–27):

$$\mathbf{l}(t_i) = \mathbf{l}(t_s) + \frac{t_i - t_s}{t_e - t_s}(\mathbf{l}(t_e) - \mathbf{l}(t_s))$$

$$\mathbf{u}(t_i) = \mathbf{l}(t_s) + \frac{t_i - t_s}{t_e - t_s}(\mathbf{u}(t_e) - \mathbf{l}(t_s)).$$

RayTrace also constructs the tolerance square $\mathbf{Q}$ (Lines 29–30). Then RayTrace examines if an intersection between SSA$|t_i$ and $\mathbf{Q}$ exists. If it does, then the SSA is updated by setting SSA$|t_i$ to be the intersection (Lines 33–34) and proceeds to process the next timepoint. If an intersection does not exist, the SSA cannot extend any further in time. RayTrace sends its state to the coordinator (Line 38) and goes into waiting mode (Line 36), expecting the server response. The state message $\langle \mathbf{l}(t_s), t_s, \mathbf{l}(t_e), \mathbf{u}(t_e), t_e \rangle$ includes the initial timestamp t_s, the start point $\mathbf{s} \equiv \mathbf{l}(t_s)$, the final timestamp t_e and the FSA $(\mathbf{l}(t_e), \mathbf{u}(t_e))$. As long as an object is in waiting mode, it stores incoming timepoints in a buffer (Lines 37 and 11). When the next epoch arrives, RayTrace receives the final timepoint that becomes the initial timepoint for a new SSA (Lines 13–16) and proceeds with processing new timepoints.

Example 1 Figure 3 illustrates the process of updating the SSA, which is depicted in all figures as the shaded spatiotemporal extent. The initial timepoint is $\langle \mathbf{p}_0, t_0 \rangle$; assume that a new timepoint $\langle \mathbf{p}_1, t_1 \rangle$ arrives, which defines the tolerance square $\mathbf{Q}_1$. Since this is the first timepoint after the initial one, the SSA$|t_1$ becomes equal to $\mathbf{Q}_1$, as demonstrated in Figure 3(a). Next, $\langle \mathbf{p}_2, t_2 \rangle$ arrives defining the tolerance square $\mathbf{Q}_2$, illustrated in Figure 3(b). The projection of the SSA at the $t = t_2$ plane $(\text{SSA}|_{t_2})$ is then intersected with $\mathbf{Q}_2$. Finally, the result of the intersection forms the projection $\text{SSA}'|_{t_2}$ shown in Figure 3(c). □

The RayTrace algorithm requires only constant space to

store the SSA information; a total of three points and two timestamps — i.e., the state of the object. The main task of the algorithm is to maintain and update the SSA. For each newly arriving timepoint, this process (projecting and intersecting) requires only constant time. Also, assuming that a response from the coordinator comes in a timely manner, i.e., at the next epoch, the buffer does not grow indefinitely. Therefore, RayTrace requires $O(1)$ space and $O(1)$ time per processed timepoint.

Algorithm 1 RayTrace algorithm

```
1: Procedure RayTrace
2: Input: Timepoint Stream {⟨p_i, t_i⟩}
3: Input: Initial Timepoint ⟨p_0, t_0⟩
4: Input: Tolerance ε
5: t_s ← t_0; t_e ← t_0 ; // Initialization of SSA
6: l(t_s) ← p_0 ;
7: waiting ← false ;
8: buf ← {} ;
9: while 1 do
10:    Retrieve timepoint ⟨p_k, t_k⟩ ;
11:    buf.pushBack(⟨p_k, t_k⟩) ;
12:    if waiting and time is next epoch then
13:       Retrieve timepoint from coordinator ⟨p_coord, t_coord⟩ ;
14:       t_s ← t_coord; t_e ← t_coord ; // Reset SSA
15:       l(t_s) ← p_coord ;
16:       waiting ← false
17:    end if
18:    while !waiting and buf != {} do
19:       ⟨p_i, t_i⟩ ← buf.popFront() ;
20:       if t_e = t_s then // This is the first timepoint after t_s
21:          t_e ← t_i ;
22:          l(t_e) ← p_i − (ε, ε) ;
23:          u(t_e) ← p_i + (ε, ε) ;
24:       else
25:          // Calculate FSA = SSA|t_i at time t_i
26:          l(t_i) ← l(t_s) + ((t_i − t_s)/(t_e − t_s))(l(t_e) − l(t_s)) ;
27:          u(t_i) ← l(t_s) + ((t_i − t_s)/(t_e − t_s))(u(t_e) − l(t_s)) ;
28:          // Calculate tolerance area (l_i, u_i) around p_i
29:          l_i ← p_i − (ε, ε) ;
30:          u_i ← p_i + (ε, ε) ;
31:          if intersects((l(t_i), u(t_i)), (l_i, u_i)) then
32:             t_e ← t_i ; // Update SSA
33:             l(t_e) ← max{l(t_i), l_i} ;
34:             u(t_e) ← min{u(t_i), u_i} ;
35:          else // Send message to coordinator
36:             waiting ← true // Go into waiting mode
37:             buf.pushBack(⟨p_i, t_i⟩)
38:             Send state ⟨l(t_s), t_s, l(t_e), u(t_e), t_e⟩
39:          end if
40:       end if
41:    end while
42: end while
43: End Procedure
```

4.1 Handling Uncertainty

We first consider the case of a single spatial dimension. A timepoint $\langle X_i, t_i\rangle$ in this case implies that the location X_i of the object at t_i is a random variable. Given tolerance ϵ, δ and assumming that X_i follows a normal distribution with known parameters, we show how to adapt the RayTrace algorithm. The objective is to define a tolerance interval for this timepoint.

The location sensing device reports the mean value x_i and the standard deviation σ_i of a measurement. We assume that the actual location follows a normal distribution, i.e., $X_i \sim N(x_i, \sigma_i^2)$. Let x_i' denote a location that is close to X_i. According to the definition of proximity in Section 3.1 we require:

$$Pr\left(|X_i - x_i'| \leq \epsilon\right) \geq 1 - \delta,$$

or equivalently:

$$Pr\left(X_i \in [x_i' - \epsilon, x_i' + \epsilon]\right) \geq 1 - \delta. \quad (1)$$

Thus, the probability that X_i is in the $[x_i' - \epsilon, x_i' + \epsilon]$ interval must be above $1 - \delta$. Figure 4 illustrates the probability density function (pdf) of X_i. Equation 1 states that the shaded part of the pdf has area more than $1 - \delta$. This area is calculated as:

$$\Phi\left(\frac{x_i' + \epsilon - x_i}{\sigma_i}\right) - \Phi\left(\frac{x_i' - \epsilon - x_i}{\sigma_i}\right),$$

using the standard cumulative distribution function $\Phi(z) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right)$. The error function values $\text{erf}(z)$ are typically precomputed and a table lookup is sufficient for estimating the area.

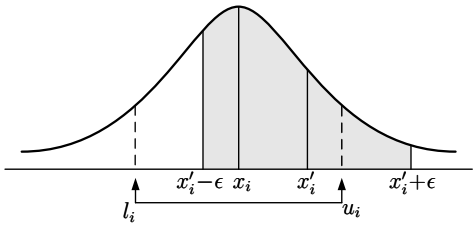


Figure 4: Calculating tolerance square for $\langle X_i, t_i\rangle$

As shown in Figure 4, x_i' should not be far from the mean value x_i; otherwise, the shaded area cannot be larger than δ. Let l_i (u_i) be the lowest (highest) value that x_i' can take without violating Equation 1, i.e., l_i, u_i are the solutions to the equation:

$$\Phi\left(\frac{x_i' + \epsilon - x_i}{\sigma_i}\right) - \Phi\left(\frac{x_i' - \epsilon - x_i}{\sigma_i}\right) = 1 - \delta \quad (2)$$

Equation 2 can be solved numerically in two ways: (i) perform a binary search on Φ's lookup table for those x_i' values satisfying the equation (exploiting Φ's monotonicity); (ii) precompute a lookup table which provides l_i, u_i given ϵ and δ and simply perform a single lookup per instance. The latter option is the most efficient method requiring constant time per timepoint. Note that since lookup tables are given for the $N(0, 1)$ distribution, a simple transformation is required for arbitrary mean and standard deviation.

Observe that $[l_i, u_i]$ serves as the tolerance interval for the timepoint $\langle X_i, t_i\rangle$ with mean value x_i and standard deviation σ_i. Consequently, the RayTrace algorithm can be straightforwardly adapted to construct an SSA given uncertainty in the input data.

An important point is that for given ϵ, δ a timepoint might have standard deviation σ_i such that Equation 2 has no solutions. To avoid this pitfall, a proactive approach would be to set more relaxed tolerance bounds, assuming knowledge of the typical imprecision in the location sensing devices. A retroactive approach would be to assign some predefined minimal tolerance area to these timepoints.

In the case of xy plane, the location $\mathbf{P}_i = (X_i, Y_i)$ of an object at time t_i is a random vector following a joint 2d

normal distribution: $\mathbf{P}_i \sim N(\mathbf{p}_i, \mathbf{\Sigma}_i)$. A location (x'_i, y'_i) is close to (X_i, Y_i) if:

$$Pr\left(\max\{|X_i - x'_i|, |Y_i - y'_i|\} \leq \epsilon\right) \geq 1 - \delta,$$

or equivalently:

$$Pr\left((|X_i - x'_i| \leq \epsilon) \wedge (|Y_i - y'_i| \leq \epsilon)\right) \geq 1 - \delta.$$

Assuming independence among x, y measurements (hence, $\mathbf{\Sigma}_i = \text{diag}\left((\sigma_i^x)^2, (\sigma_i^y)^2\right)$) the last equation becomes:

$$Pr\left(|X_i - x'_i| \leq \epsilon\right) \cdot Pr\left(|Y_i - y'_i| \leq \epsilon\right) \geq 1 - \delta. \quad (3)$$

To simplify Equation 3, we require the failure probability to be less than $\frac{\delta}{2}$ for each dimension, i.e.:

$$Pr\big(X_i \in [x'_i - \epsilon, x'_i + \epsilon]\big) \geq 1 - \frac{\delta}{2},\ Pr\big(Y_i \in [y'_i - \epsilon, y'_i + \epsilon]\big) \geq 1 - \frac{\delta}{2},$$

since $(1 - \frac{\delta}{2})^2$ is marginally larger than $1 - \delta$ for small δ values. Therefore, using such a simplification, it is easy to revert to the single dimensional case and apply the methodology previously described.

5. DISCOVERING HOT MOTION PATHS

The coordinator performs three basic tasks: (i) stores detected motion paths, (ii) maintains their hotness, and (iii) executes a discovery strategy, processing the state of reporting objects. In the following, we discuss each task in detail.

5.1 Storing Motion Paths

We use a lightweight grid-based index to store motion paths. The entire space is partitioned into a predetermined number of cells and the endpoints of each motion path are indexed, rather than the linear shape of the path itself. Every cell contains a list of entries about endpoints that fall inside its area. Apart from storing coordinates of these endpoints, each index entry also stores the respective motion path id and the coordinates of the other endpoint. The list is sorted by motion path id and organized in a hash table. This allows for fast insertions and deletions, requiring constant (on average) time.

5.2 Hotness Maintenance

Recall that the hotness h_i of a motion path $\mathbf{p}_i\mathbf{p}_{i+1}$ is expressed as the number of objects that have crossed it within a sliding window extending to the past W time units from current time. To maintain this count for each motion path, we use a hash table and an event queue. The hash table uses as key the motion path id i, and stores for each i the corresponding number of objects h_i. The event queue updates the hash table when the exit timestamp t_e^i of an object expires from window W.

Assume that we detect that an object has crossed motion path $\mathbf{p}_i\mathbf{p}_{i+1}$ with id i at $[t_s, t_e]$. First, we increase (by one) the counter for $\mathbf{p}_i\mathbf{p}_{i+1}$ in the hash table. The counter will have to be decreased at time $t_e + W$, since the corresponding interval will completely fall outside the window W. To efficiently capture these interval expirations, upon updating h_i, we en-heap tuple $\langle t_e + W, i\rangle$ into the event queue.

The queue is sorted on expiry time, and its head corresponds to the next expiring interval considering all motion paths and intervals in the system. When the current time reaches the expiry time at the head of the heap, then: (i) the top entry is de-heaped, (ii) the hotness of the corresponding motion path is decreased in the hash table, and (iii) if the hotness becomes 0, the motion path is deleted from the grid and the hash table.

Each counter lookup or update in the hash table takes expected constant time. Every en-heap or de-heap operation in the event queue costs time logarithmic to the number of its entries. Thus, the overall computational overhead is low. Regarding space requirements, both structures are relatively concise and can be maintained in main memory.

5.3 The SinglePath Strategy

The SinglePath strategy processes all state messages $\{\langle \mathbf{s}^i, t_s^i, \mathbf{l}^i, \mathbf{u}^i, t_e^i\rangle\}$ received from the reporting objects. Note that we use the superscript index i to refer to object with id i. The objective of SinglePath is to find the hottest motion path that starts from $\mathbf{s}^i$ and finishes somewhere inside the FSA $(\mathbf{l}^i, \mathbf{u}^i)$. The rationale behind this policy is to minimize the number of paths introduced by any single object, utilizing motion paths already discovered and crossed by other objects. In order to exploit existing motion paths, one should first probe the grid index and examine all paths intersecting this FSA, since these paths could be most relevant to the current motion of object i. However, depending on the distribution of objects and the actual pattern of their movement, it may occur that no motion path matches the current state of that object. We distinguish three cases regarding the information retrieved from MotionPath for each object i:

1. There are *available motion paths* starting from $\mathbf{s}_i$ and ending somewhere inside the FSA $(\mathbf{l}^i, \mathbf{u}^i)$.
2. There is no available motion path, but there exist *available vertices*, i.e., motion path endpoints that fall inside FSA $(\mathbf{l}^i, \mathbf{u}^i)$.
3. No available motion path or vertex is found.

Note that Case 1 simply involves updating the hotness for a motion path that will be chosen among the available ones. However, Cases 2 and 3 entail construction of a new motion path for the object at hand and this path must be stored.

SinglePath attempts to identify motion paths and to compute hotness collectively for all objects, in order to reuse existing motion paths (thus, increasing their hotness) and avoid introduction of multiple new ones. The strategy first handles all objects for which available motion paths were identified (Case 1). Afterwards, it takes care of the remaining objects that received no path at all (Cases 2 and 3). All cases follow a two-phase paradigm: *generation of candidates* (motion paths or vertices) and *selection of hottest candidate*. Algorithm 2 shows in detail all steps involved; in the following, we discuss the most critical operations.

Handling candidate motion paths. Throughout this step, a search for qualifying motion paths is performed for each object (Function *GetCandidatePaths* called in Line 5). Initially, a range query is evaluated against the grid index, specifying a rectangle $(\mathbf{l}^i, \mathbf{u}^i)$ for each object i (Line 42). We obtain motion paths $\mathbf{s}^i\mathbf{p}_j$ that intersect $(\mathbf{l}^i, \mathbf{u}^i)$. Their respective hotness values h_j are obtained after performing a single lookup in the hash table. Let $\mathcal{AP}_i = \{\langle \mathbf{s}^i\mathbf{p}_j, h_j\rangle\}$ denote the set of available motion paths retrieved for object i (Lines 43–46). Note that the hotness of each path in $\mathcal{AP}_i$ associated to the i-th object should increase by one (Line 44), implying the potential influence of object i on the significance of any of these motion paths (i.e., if eventually

Algorithm 2 SinglePath Strategy

1: **Procedure** *InsertMotionPaths*
2: **Input:** Object $States = \{\langle \mathbf{s}^i, t_s^i, \mathbf{l}^i, \mathbf{u}^i, t_e^i \rangle\}$
3: $\mathbf{R_{all}} \leftarrow \emptyset$; *//Memory-resident structure for FSA's*
4: **for** each state $\langle \mathbf{s}^i, t_s^i, \mathbf{l}^i, \mathbf{u}^i, t_e^i \rangle$ **do**
5: $\quad \mathcal{CP}_i \leftarrow GetCandidatePaths(\mathbf{s}^i, \mathbf{R}(\mathbf{l}^i, \mathbf{u}^i))$;
6: $\quad \mathbf{R_{all}} \leftarrow \mathbf{R_{all}} \cup \{\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)\}$;
7: **end for**
//Identify overlaps among final safe areas
8: **for all** $\mathbf{R_k} \in \mathbf{R_{all}}$ **do**
9: $\quad$ Calculate overlapping areas $\mathbf{R}_{\{j\}} = \bigcap_{k \in \{j\}} \mathbf{R}_k$
10: $\quad \mathbf{R}_{\{j\}}.count \leftarrow |\ \{j\}\ |$;
11: $\quad \mathbf{R_{all}} \leftarrow \mathbf{R_{all}} \cup \{\mathbf{R}_{\{j\}}\}$
12: **end for**
//Increase hotness of paths that appear in multiple $\mathcal{CP}$'s
13: **for** each motion path $mp_i \in \mathcal{CP}_i$ **do**
14: $\quad mp_i.hotness \leftarrow mp_i.hotness + |\ \{mp_i \in \mathcal{CP}_j, \forall j \neq i\}\ |$;
15: **end for**
//Selection phase
16: **for** each object i in *States* **do**
17: $\quad$ **if** $\mathcal{CP}_i \neq \emptyset$ **then**
18: $\quad\quad$ *//Case 1: Examine available motion paths*
19: $\quad\quad$ Choose motion path $mp_k \in \mathcal{CP}_i$ with *max hotness*
20: $\quad\quad$ Update hotness of mp_k at MotionPath index
21: $\quad$ **else**
22: $\quad\quad \mathcal{CV}_i \leftarrow GetCandidateVertices(\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i))$;
$\quad\quad$ *//Case 2: Check available end vertices of motion paths*
$\quad\quad$ *//Adjust vertex hotness according to potential overlaps*
23: $\quad\quad$ **for** each $\langle \mathbf{p}_j, h_j \rangle \in \mathcal{CV}_i$ **do**
24: $\quad\quad\quad$ Find smallest overlap $\mathbf{R}_k \in \mathbf{R_{all}}$ s.t. $\mathbf{p}_j \in \mathbf{R}_k$
25: $\quad\quad\quad h_j \leftarrow h_j + \mathbf{R}_k.count$;
26: $\quad\quad$ **end for**
$\quad\quad$ *//Case 3: Generate additional candidate vertices*
27: $\quad\quad h_m \leftarrow 0$;
28: $\quad\quad$ **for** each overlap $\mathbf{R}_k \in \mathbf{R_{all}}$ **do**
29: $\quad\quad\quad$ **if** $\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i) \cap \mathbf{R}_k \neq \emptyset$ **and** $\mathbf{R}_k.count > h_m$ **then**
30: $\quad\quad\quad\quad \mathbf{R}_m \leftarrow \mathbf{R}_k; h_m \leftarrow \mathbf{R}_k.count$;
31: $\quad\quad\quad$ **end if**
32: $\quad\quad$ **end for**
33: $\quad\quad v_m \leftarrow Centroid(\mathbf{R}_m)$;
34: $\quad\quad \mathcal{CV}_i \leftarrow \mathcal{CV}_i \cup \{\langle v_m, h_m \rangle\}$;
35: $\quad\quad$ Choose vertex $\mathbf{p}_k \in \mathcal{CV}_i$ with max hotness h_{max}
36: $\quad\quad$ Insert motion path $\langle \mathbf{s}^i \mathbf{p}_k, h_{max} \rangle$ at MotionPath index
37: $\quad$ **end if**
38: **end for**
39: **End Procedure**

40: **Function** *GetCandidatePaths*(vertex $\mathbf{s}^i$, rectangle $\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)$)
41: *Initialization:* $\mathcal{AP}_i \leftarrow \emptyset$
//Search MotionPath *index*
42: $\mathcal{P}_i \leftarrow$ motion paths $\{\mathbf{s}^i \mathbf{p}_j\}$ s.t. $\mathbf{p}_j \in \mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)$;
43: **for** each motion path $\mathbf{s}^i \mathbf{p}_j \in \mathcal{P}_i$ **do**
44: $\quad h_j \leftarrow hotness(\mathbf{s}^i \mathbf{p}_j) + 1$; *//Look-up in hash table*
45: $\quad \mathcal{AP}_i \leftarrow \mathcal{AP}_i \cup \{\langle \mathbf{s}^i \mathbf{p}_j, h_j \rangle\}$;
46: **end for**
47: **return** $\mathcal{AP}_i$
48: **End Function**

49: **Function** *GetCandidateVertices*(rectangle $\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)$)
50: *Initialization:* $\mathcal{AV}_i \leftarrow \emptyset$;
//Search MotionPath *index*
51: $\mathcal{V}_i \leftarrow$ end vertices of motion paths s.t. $\mathbf{p}_j \in \mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)$;
52: **for** each *distinct* vertex $\mathbf{p}_j \in \mathcal{V}_i$ **do**
53: $\quad h_j \leftarrow 0$;
$\quad$ *//Sum up hotness of all converging paths*
54: $\quad$ **for** each motion path $\mathbf{q}\mathbf{p}_j$ terminating at $\mathbf{p}_j$ **do**
55: $\quad\quad h_j \leftarrow h_j + hotness(\mathbf{q}\mathbf{p}_j)$;
56: $\quad$ **end for**
57: $\quad \mathcal{AV}_i \leftarrow \mathcal{AV}_i \cup \{\langle \mathbf{p}_j, h_j \rangle\}$;
58: **end for**
59: **return** $\mathcal{AV}_i$
60: **End Function**

were chosen as hottest). Note that hotness values are only temporarily adjusted in $\mathcal{AP}_i$, leaving intact the contents of the hash table.

As soon as index probing is finished, a new set $\mathcal{CP}_i$ defines the candidate motion paths obtained for object i; hence, $\mathcal{CP}_i = \mathcal{AP}_i$ (Line 5). We stress that other objects may also accentuate hotness of paths in $\mathcal{CP}_i$, since the sets of available motion paths are not disjoint (Lines 13–15). This is reasonable, considering that potential selection of a specific motion path for an object $j \neq i$ could modify hotness ranking among candidate paths for i. Finally, provided that the set $\mathcal{CP}_i$ of candidate motion paths is non-empty, the selection phase simply involves choosing the hottest path for each object i among those collected in its $\mathcal{CP}_i$ (Lines 17–20).

Handling candidate vertices. Recall that this stage affects only objects for which no motion path has been identified during the previous step. For each object i, it provides a set of candidate end vertices for a new path that will have its starting vertex at $\mathbf{s}^i$. Our goal is to choose the hottest possible vertex as the endpoint of this newly discovered motion path for object i. Intuitively, selection of the hottest vertex increases the chances that object i crosses a hot motion path immediately afterwards. Such vertices can be obtained from existing motion paths, while hotness of a vertex is calculated summing the hotness of each incoming motion path (implying multiple segments converging to them).

Let $\mathcal{AV}_i = \{\langle \mathbf{p}_j, h_j \rangle\}$ denote the set of available vertices $\mathbf{p}_j$ and their hotness h_j for object i. The construction of $\mathcal{AV}_i$ is detailed in function *GetCandidateVertices*. Similarly to Case 1, this set is obtained with a range query against the grid (Line 51). The hotness of each vertex is calculated by summing up the hotness of all converging motion paths (Lines 54–56).

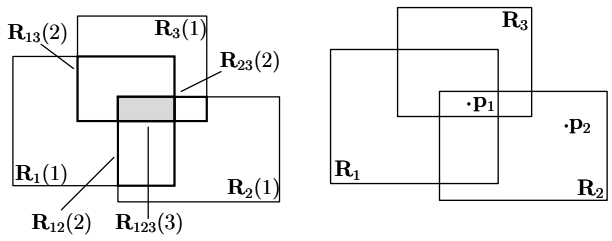
However, it is not sufficient to only consider the vertices of existing motion paths inside the FSA. Specifically:

(i) $\mathcal{AV}_i$ may be empty if no motion path intersects the current FSA, so no vertices will be returned (Case 3). Therefore, a new vertex must be generated, in a way that takes into account motion patterns of other objects as well. This policy increases the chance that new vertices could also serve as endpoints of other motion paths in the future. Thus, we can avoid further segmentation of paths.

(ii) When calculating hotness for a vertex, we must also take into account the possibility that the same vertex may be returned for other objects as well, thus increasing the probability that this vertex might be more suitable for selection.

(iii) Newly generated motion paths for other objects will also provide additional vertices that should not be missed.

We successfully collect additional candidate vertices (besides those in $\mathcal{AV}_i$), by examining intersections of objects' FSA rectangles. We maintain a structure $\mathbf{R_{all}}$ that processes the final safe areas $\mathbf{R}_i$ $(= (\mathbf{l}^i, \mathbf{u}^i))$ of all considered objects (Line 6), and calculates their overlaps $\mathbf{R}_{\{j\}} = \bigcap_{k \in \{j\}} \mathbf{R}_k$ (Lines 8–12). Each rectangle in $\mathbf{R_{all}}$ is associated with a count (its perceived "hotness"), expressing the number of rectangles that it overlaps with, i.e., $c_{\{j\}} = |\{j\}|$ (Line 10). The intuition is that if we are forced to choose an arbitrary vertex, then its hotness should be as high as the count of the smallest stored rectangle in which it resides. This observation is better illustrated with the following example.



(a) Intersections (b) Choosing a Vertex

Figure 5: Considering overlapping rectangles for additional candidate vertices.

Example 2 Consider three objects and their respective FSAs, $\mathbf{R}_1$, $\mathbf{R}_2$ and $\mathbf{R}_3$, that intersect with each other constructing intersections $\mathbf{R}_{12}$, $\mathbf{R}_{23}$, $\mathbf{R}_{13}$ and $\mathbf{R}_{123}$. Figure 5(a) illustrates all original FSA's and their overlaps, along with their counts. Now, assume that there are no available vertices for objects 1 and 3 and that there is a single available vertex $\mathbf{p}_2$ for object 2 with hotness 1. If we choose $\mathbf{p}_2$ as the endpoint for that object's motion path, its hotness will become 2 (one for the existing motion path plus one for the newly discovered path). However, had we chosen a vertex inside $\mathbf{R}_{123}$, say $\mathbf{p}_1$ in Figure 5(b), and used that as the endpoint for the motion path of all objects, its hotness would be 3. Obviously, we should consider introducing additional vertices from the overlapping areas with the highest counts. □

Once $\mathcal{AV}_i$ has been found, we construct $\mathcal{CV}_i = \{\langle \mathbf{p}_j, h_j \rangle\}$, the set of candidate vertices for each object i as follows. The candidate set is initialized to the set of available vertices: $\mathcal{CV}_i = \mathcal{AV}_i$ (Line 22). Fix an object i and consider one of its available vertices $\mathbf{p}_j$ with hotness h_j. Let $\mathbf{R}_k$ be the smallest intersection in which $\mathbf{p}_j$ resides and let its associated count $c_k = \mathbf{R}_k.count$. Had we chosen vertex $\mathbf{p}_j$ as the endpoint of all objects whose FSA overlap with $\mathbf{R}_k$, then its hotness would become $h_j + c_k$. To reflect this potential influence, we increment by c_k the hotness of $\mathbf{p}_j$ in $\mathcal{CV}_i$ (Lines 23–26).

As soon as this update has been performed for all available vertices of all objects, we need to generate additional candidate vertices, as demonstrated in Example 2. In fact, we only need to generate a single additional vertex per object. Let $\mathbf{R}_m$ denote the intersected rectangle with the highest count c_m among those of object i, i.e., among those that fall inside FSA $\mathbf{R}_i$. Then, the newly generated candidate vertex for this object should lie inside $\mathbf{R}_m$ and, thus, must have hotness c_m. We choose one such vertex, e.g., by taking the centroid of $\mathbf{R}_m$, and insert it into $\mathcal{CV}_i$ (Lines 27–34). This scheme guarantees that candidate vertices exist even for objects that received neither a motion path nor a vertex from the index (Case 3). Finally, the selection phase per object i simply involves selecting the hottest candidate vertex among those in $\mathcal{CV}_i$ and inserting the newly created path into the index (Lines 35–36).

6. EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the performance of our framework, focusing on the RayTrace algorithm and the SinglePath strategy. To better gauge its effectiveness, we compare it against a Douglas-Peucker [8] variant, termed DP, that discovers spatial line segments close to the objects' movement. We must note that due to its nature, the output line segments do not constitute proper motion paths because they are disconnected and, in practice, they are hardly interpretable. Hence, DP is not directly comparable to our approach. Rather, as it purposefully benefits already existing line segments and is not bound by the strict covering motion path set requirements, DP is expected to assign higher hotness to segments when compared to our methodology.

The DP Method. As described in Section 2, the windowed variations of Douglas-Peucker algorithm proposed in [20] offer concise trajectory synopses per object. However, unless objects follow exactly the same trajectory, all motion paths extracted will not have hotness greater than one. We choose to relax our requirements. In particular we allow selection of line segments that are close to objects' movements and ignore the time dimension. In this manner, we expect segments to achieve hotness that upper bounds the hotness achieved by motion paths. Whenever a new segment should be created between a starting point and the chosen floating point, we do not store it at once. Instead, we check whether an existing segment (produced earlier by another object) falls completely within the minimum bounding box (MBB) of the candidate segment. Each MBB is expanded by the tolerance value, to cope with uncertainty in objects' locations. In case such a segment exists, we need not store the candidate segment, but we must increase the hotness of the existing path. Otherwise, the new segment is stored with hotness 1. This simple policy can provide an even more dense approximation for each trajectory, with the additional benefit that many segments now belong to multiple object traces. On the other hand, connectivity between successive motion paths for each object is no longer preserved. Note that we measure DP's quality for the sake of comparison with SinglePath and exclude all time measurements. As observed in our evaluation, DP runs significantly faster than SinglePath because it simply performs one range query per discovered segment.

6.1 Experimental Setting

All algorithms were implemented in C++ and compiled with gcc on a 3GHz Intel Core 2 Duo CPU. All processing takes place in main memory.

We generated synthetic datasets for trajectories of moving objects traveling on the main road network of greater Athens that covers an area of 250 km^2. We utilized a simplified graph of the network, assuming that nodes (representing major crossroads) are connected via straight linear links and not curved polylines (as in the real network). This network is illustrated in Figure 6 and consists of 1831 links connecting 1125 nodes in total. Links are ranked with weights, reflecting their significance in vehicle circulation. Thus, links are classified into four categories: motorways, highways, primary roads, and secondary roads.

Each object is initially assigned at a randomly chosen node. Whenever it is allowed to move, this object chooses to follow one of the outgoing links of that node. To make this decision, we calculate a ratio that expresses the relative weight of each such link compared to the total weight of all links connected to the current node. Finally, we randomly choose to follow a link with probability equal to its ratio. We assume that all objects have equal-length displacement s between successive positions, so that the next

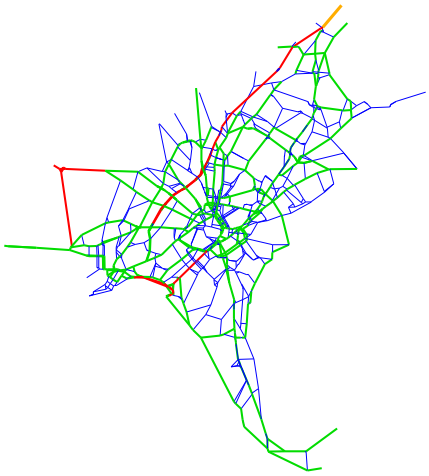


Figure 6: Athens road network links.

location will be along that link or at the opposite end node (at most). In the sequel, as long as an object does not cross a node, it continues its course along that link. Note, though, that movement is also controlled by another parameter that refers to the agility of moving objects. This means that, at each timestamp, only a portion α of the total number N of objects is allowed to move (decided randomly), while the rest remain stopped. Therefore, the inter-arrival time between positional measurements is not fixed for each object, but it fluctuates with time. As in a real traffic scenario, objects tend to follow main roads for large parts of their movement and enter into minor roads less frequently. To capture uncertainty, white noise is then added to object locations. In particular, a value randomly chosen between $-err$ and err is added to both coordinates. Although the data were generated by considering a fixed road network, the algorithms have no knowledge of this fact, and, hence, cannot take advantage of it when discovering motion paths. Intuitively, we expect the algorithms to identify the most frequently traveled parts of the Athens' network.

We run a set of experiments for different parameter values. We consider N = 10,000, 20,000 and 100,000 objects that travel with fixed agility $a = 0.1$. During a timestamp, objects move $s = 10$ meters and take a location measurement with positional error $err = 1$ meter. We model uncertainty with ϵ tolerance[2]; we vary its value from $\epsilon = 1$ to 20 meters. Window size is fixed to $W = 100$ timestamps and at any timestamp we wish to recover the $k = 10$ hottest motion paths. In each experiment we vary a single parameter, while we set the remaining to their default values. The duration of every simulation is 250 timestamps and an epoch corresponds to 10 timestamps. Table 2 summarizes the parameters involved and their ranges; the default values are shown in bold.

To measure the efficiency and quality of the SinglePath strategy we use three metrics. First, we measure the size of the index in terms of motion paths. Second, we calculate the score (see Section 3.1) of the top-k hottest motion paths discovered. Finally, we measure the processing time spent by the coordinator executing the SinglePath strategy.

[2]We do not consider ϵ, δ tolerance since the processing involved is similar, as shown in Section 4.1

Parameter	Values
N	10000, **20000**, 100000 objects
Tolerance (ϵ)	1, 2, **10**, 20 meters
Positional error (err)	1 meter
Agility (α)	0.1
Displacement (s)	10 meters
Window size (W)	100 timestamps
k	10

Table 2: Experimental parameters.

The reported/plotted measurements for the aforementioned performance factors correspond to average values per epoch.

6.2 Experimental results

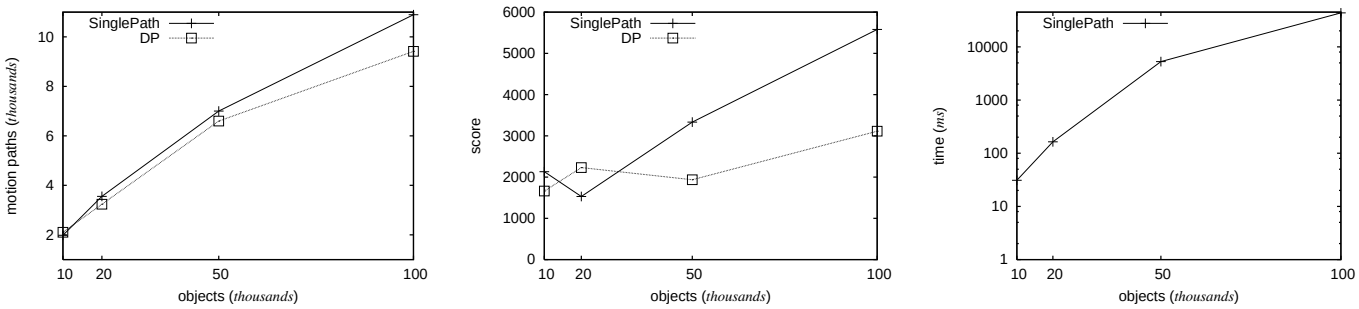
In the first set of experiments we vary the number of objects from $N = 10,000$ to $100,000$ while the tolerance is fixed to $\epsilon = 10$ and show the results in Figure 7. Regarding the number of segments measured by the index size, Figure 7(a) clearly illustrates that DP inserts fewer segments. This is expected as DP enjoys more freedom and is not restricted to finding motion paths. SinglePath, on the other hand, must strictly identify motion paths that fit to some object's movement for a time interval. Note that even for 100,000 objects, SinglePath identifies only 16% more segments compared to DP, i.e., 10,896 versus 9,416.

Figure 7(b) shows the score for the top-10 hottest motion paths returned by the two methods for varying N values. In general, since DP identifies fewer total segments, their average hotness is larger than that of the motion paths found by SinglePath. Interestingly, for $N = 20,000$ SinglePath achieves higher score that DP. This is attributed to the fact that in this setting SinglePath extracts longer motion paths.

Figure 7(c) measures the average per epoch processing time spent by the coordinator running SinglePath. This running time essentially determines what the smallest epoch can be, since all processing must have finished by the next epoch so that objects exit the waiting mode of the RayTrace algorithm as soon as possible. As shown in the figure, for a large number of objects $N > 100,000$, processing time becomes close to 40 secs. To compensate for this behavior, one can choose to increase the tolerance parameter. As discussed in the following, higher ϵ values lead to reduced processing times.

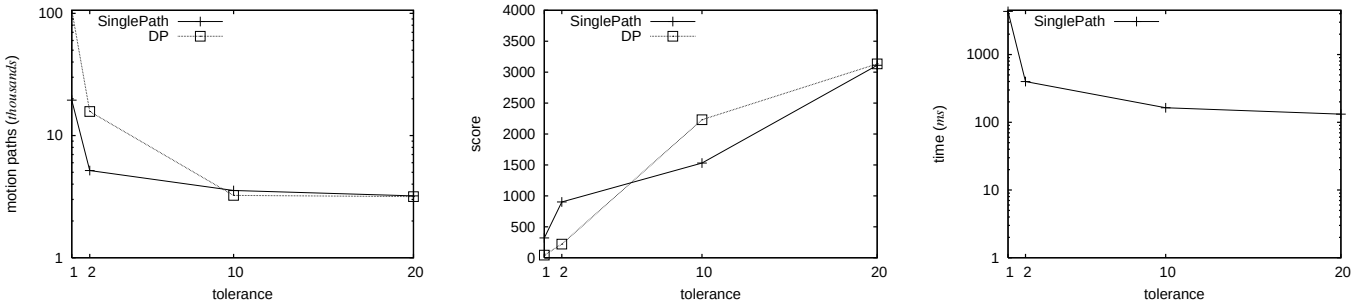
Figure 8 measures the same metrics as before but fixes the number of objects to $N = 20,000$ and varies the tolerance parameter from $\epsilon = 1$ to 20. Figures 8(a) and 8(b) show that RayTrace significantly outperforms the benchmark, i.e., it discovers fewer motion paths that are both hotter and longer. When the tolerance increases, recall that the MBBs of the range queries that DP issues also increase. This results in more freedom when selecting segments. Regarding the scalability of processing time as ϵ increases, Figure 8(c) clearly illustrates the benefits of relaxing tolerance values. The processing time decreases by a factor greater than 3 when the ϵ increases from 2 to 20.

To better illustrate the effectiveness of our methods, Figure 9 draws the entire set of motion paths that have hotness greater than 0 within the time window. Comparing to the entire network shown in Figure 6, the SinglePath strategy manages to accurately extract a set of motion paths that resembles the (unknown to SinglePath) network. Notice that



(a) Index Size vs N (b) Top-k Score vs N (c) Processing Time vs N

Figure 7: Varying the Number of Objects



(a) Index Size vs ϵ (b) Top-k Score vs ϵ (c) Processing Time vs ϵ

Figure 8: Varying the Tolerance Parameter

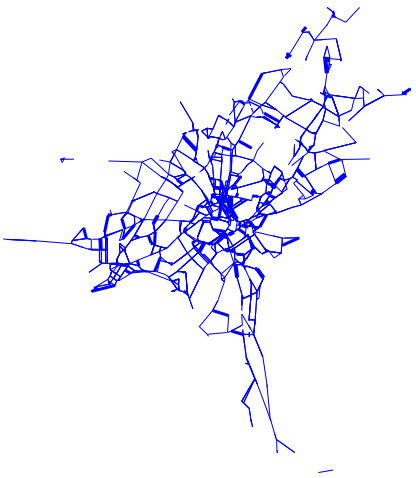


Figure 9: The network as discovered by SinglePath.

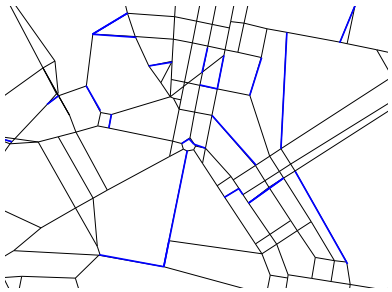


Figure 10: Top 20 hottest motion paths in the center of Athens.

motion paths with larger hotness are drawn with thicker lines. For completeness, Figure 10 focuses on the center of Athens and draws the top 20 hottest motion paths stored in the index.

7. CONCLUSIONS

In this work, we proposed a framework for on-line maintenance of hot motion paths in order to detect frequently traveled trails of numerous moving objects. We consider a distributed setting, with a coordinator that maintains hotness and geometries of these paths in a spatiotemporal index, and many moving clients that issue updates only for important changes in their positions. We focus on motion patterns during the recent past, thus discarding obsolete paths that expire from a sliding time window. We assume freely moving objects, i.e., not restricted by some network, and our techniques take into consideration uncertainty inherent in location readings while providing discrepancy guarantees for the discovered motion paths. Empirical simulations demonstrate the ability of our methodology to provide a dense representation of objects' movement, as well as its efficiency with respect to on-line maintenance of significant motion patterns.

As part of our future work, we intend to explore methods for improving the filtering approach by means of receiving feedback from the server. Currently, each moving object has

knowledge limited to its own state. It is expected that obtaining information about nearby moving objects and hot motion paths could significantly improve the splitting decisions employed by the RayTrace algorithm.

8. REFERENCES

[1] A. Anagnostopoulos, M. Vlachos, M. Hadjieleftheriou, E. J. Keogh, and P. S. Yu. Global distance-based segmentation of trajectories. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 34–43, 2006.

[2] B. Babcock and C. Olston. Distributed top-k monitoring. In *ACM International Conference on Management of Data (SIGMOD)*, pages 28–39, 2003.

[3] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. Uldbs: Databases with uncertainty and lineage. In *International Conference on Very Large Data Bases (VLDB)*, pages 953–964, 2006.

[4] Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *International Conference on Mobile Data Management (MDM)*, pages 27–38, 2004.

[5] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15(3):211–228, 2006.

[6] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *ACM International Conference on Management of Data (SIGMOD)*, pages 491–502, 2005.

[7] G. Cormode, M. N. Garofalakis, S. Muthukrishnan, and R. Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *ACM International Conference on Management of Data (SIGMOD)*, pages 25–36, 2005.

[8] D. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitised line or its caricature. *The Canadian Cartographer Journal*, 10(2):112–122, 1973.

[9] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 63–72, 1999.

[10] B. Gedik and L. Liu. Mobieyes: A distributed location monitoring service using moving location queries. *IEEE Transactions on Mobile Computing*, 5(10):1384–1402, 2006.

[11] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *The VLDB Journal*, 15(2):143–164, 2006.

[12] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *ACM International Conference on Management of Data (SIGMOD)*, pages 479–490, 2005.

[13] H. Imai and M. Iri. An optimal algorithm for approximating a piecewise linear function. *Journal of Information Processing*, 9(3):169–162, 1986.

[14] C. S. Jensen, D. Lin, and B. C. Ooi. Continuous clustering of moving objects. *IEEE Transactions on Knowledge and Data Engineering*, 19(9):1161–1174, 2007.

[15] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 364–381, 2005.

[16] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: A partition-and-group framework. In *ACM International Conference on Management of Data (SIGMOD)*, pages 593–604, 2007.

[17] X. Li, J. Han, J.-G. Lee, and H. Gonzalez. Traffic density-based discovery of hot routes in road networks. In *International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 441–459, 2007.

[18] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 617–622, 2004.

[19] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 236–245, 2004.

[20] N. Meratnia and R. A. de By. Spatiotemporal compression techniques for moving point objects. In *International Conference on Extending Database Technology (EDBT)*, pages 765–782, 2004.

[21] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *ACM International Conference on Management of Data (SIGMOD)*, pages 563–574, 2003.

[22] S. Papadopoulos, D. Sacharidis, and K. Mouratidis. Continuous medoid queries over moving objects. In *International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 38–56, 2007.

[23] M. Potamias, K. Patroumpas, and T. Sellis. Sampling trajectory streams with spatiotemporal criteria. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 275–284. IEEE Computer Society, 2006.

[24] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[25] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *International Conference on Very Large Data Bases (VLDB)*, pages 922–933, 2005.

[26] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *IEEE International Conference on Data Engineering (ICDE)*, pages 673–684, 2002.

[27] W. Wu, W. Guo, and K.-L. Tan. Distributed processing of moving k-nearest-neighbor query on moving objects. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1116–1125, 2007.

[28] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *IEEE International Conference on Data Engineering (ICDE)*, pages 201–208, 1998.