UNIVERSITY OF CALIFORNIA
RIVERSIDE

# Identifying Interesting Behaviors from Moving Object Trajectories

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Md Reaz Uddin

March 2014

Dissertation Committee:

Dr. Chinya V Ravishankar, Co-Chairperson
Dr. Vassilis J Tsotras, Co-Chairperson
Dr. Eamonn Keogh
Dr. Vagelis Hristidis

The Dissertation of Md Reaz Uddin is approved:

---

---

---

Committee Co-Chairperson

---

Committee Co-Chairperson

University of California, Riverside

## Acknowledgments

I would like to take this opportunity to express my gratefulness to my advisers Dr. Chinya Ravishankar and Dr. Vassilis Tsotras, without whose help, I would not have been here. They taught me how to do research and helped me with their experience. I thank my committee members Dr. Eamonn Keogh and Dr. Vagelis Hristidis for their help. I would like to thank Dr. Neal Young for helping me to solve critical problems. I truly appreciate the help of Michael Rice and Marcos Vieira during this journey. I would like to mention Amy Ricks as one of the most dependable staffs of the CSE Dept.

To my parents, wife Shathi and sister Meher.

# ABSTRACT OF THE DISSERTATION

Identifying Interesting Behaviors from Moving Object Trajectories

by

Md Reaz Uddin

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2014
Dr. Chinya V Ravishankar, Co-Chairperson
Dr. Vassilis J Tsotras, Co-Chairperson

Understanding moving object behaviors, also known as trajectory semantics, is an important problem that affects many decision making applications. Previous works typically identify such behaviors by using known landmarks, also termed as Regions of Interest (ROIs) (parks, museums, malls, etc.) that are geographically collocated with the trajectory. The main objective of this thesis is to identify trajectory semantics, by looking only at the trajectory data, i.e., without assuming pre-knowledge of ROIs.

We first present a new trajectory behavior, by defining the notion of dwell regions. A region $R$ is a dwell region for a moving object $O$ if, given a threshold distance $d$ and duration $t$, every point of $R$ remains within distance $d$ of $O$ for at least time $t$. Clearly, points within $R$ are likely to be of interest to $O$. We present methods for determining dwell regions for both streaming and archived data. Next, we introduce a novel query that can be used to track conclaves (i.e., secret meetings) of a group of moving object. In this environment we assume only partial observations of the individual object movements, within the context of a local transportation network. This is a realistic assumption due

to sparsely-distributed surveillance cameras or lack of observations in general. Given such limited observations we seek to infer the set of all possible conclaves. The third chapter of the thesis addresses ROI identification. An ROI is typically defined as a region where a large number of moving objects remain for at least a given time interval. Previous methods require sequential scanning of the entire dataset to find ROIs when the semantics (number of objects, time duration) change. Here, we propose a novel method based on object density, to efficiently identify ROIs with arbitrary semantics; this method scans the dataset only once.

We also revisit indexing of trajectories using Hilbert curves. Instead of using minimum bounding rectangles we present methods to use Hilbert curves to index trajectory polylines. Our method outperforms the state of the art methods for spatial range queries by two to fifteen times. Even though such transformation does not preserve the Euclidean distance, we show, that our approach can also be used to efficiently answer kNN queries.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

A *trajectory* is a time ordered sequence of location data of a moving object (with a unique ID) over a certain duration. Usually the location data comes from GPS-enabled devices in the from of (latitude, longitude) coordinate. It is assumed that an object follows a straight line between two consecutivly reported locations.

The widespread use of GPS-enabled devices has enabled many applications that generate and maintain data in the form of trajectories (e.g., [1, 2, 3, 4]). Novel applications [5, 6, 7, 8] allow users to manage, store, and share trajectories in the form of GPS logs, and find travel routes, interesting places, or other people interested in similar activities. Given such large amount of data it is not straightforward to understand the semantics of these movements.

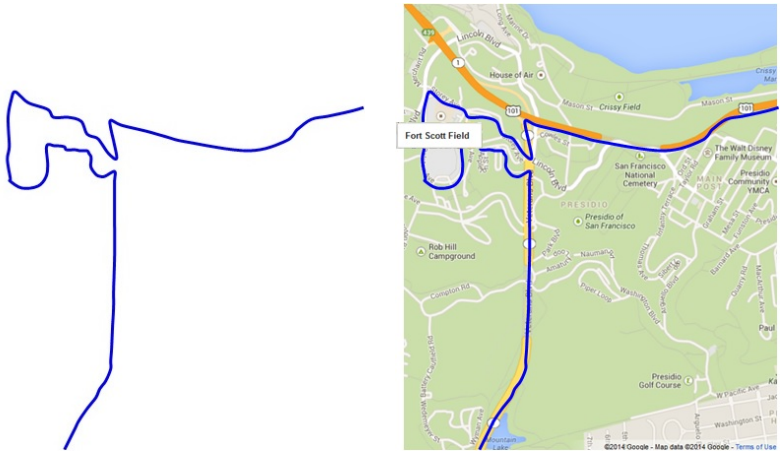
The research effort on moving object trajectory has been mostly on indexing and storing the data[9, 10, 11] or applying data mining algorithms (trajectory data mining) to identify statistically significant patterns[12, 13, 14] from the data. The goal of indexing

and storing trajectories is to be able to efficiently answer range queries (e.g., what are the trajectories that passed through the query specified region), nearest neighbor queries (e.g., what are the nearest trajectories from a query location or region), etc. It includes representing the trajectories e.g., with minimum bounding rectangles, polynomial approximation[11], etc. and then designing a index structure, like MVR-Tree[9], TPR-Tree[10], PA-Tree[11], etc. to efficiently access the data and answer the queries. Answering range and nearest neighbor queries when the exact location of an object is unknown, have also been studied recently[15, 16]. Another interesting development is to evaluate 'complex pattern queries'[17, 18] where multiple query predicates are evaluated at the same time instead of evaluating each of them separately.

Trajectory data mining includes finding frequent movement patterns[12], clustering similar trajectories[14], anomaly detection[19], etc. Most of these works focus on developing distance metrics to measure similarity or dissimilarity between trajectories or parts of trajectories. [20] proposes longest common subsequence as a distance measure [21] uses Chebyshev approximation to find similarity between trajectories. Sympolic approximation is used in [22, 23] for similarity measure. These efforts are based on the spatio-temporal attributes of the trajectory which does not always reveal the semantics of the movement. Until recently, understanding trajectory semantics has been a less explored area.

### 1.0.1 Trajectory Semantics

A common way of understanding trajectory semantics is to annotate trajectories with external information e.g., from geographic or application domain knowledge. For example, figure 1.1(a) shows a trajectory of a moving object which does not reveal any




(a) (b)

Figure 1.1: A Trajectory (a) without any external information (b) with information from external source e.g. map layers.

insight about the activity of the object. However, if we access information from external sources e.g., map layers (figure 1.1(b)) then we understand what intersting places the object has gone by. Without this external knowledge understanding the semantics would not be possible. However, such knowledge may not always be available. In this thesis we consider semantics that does not require any extarnal knowledge or cannot be discovered by using any such knowledge.

We first present a new trajectory semantics *dwell region* and *dwell behavior*. Dwell region has applications in areas such as monitoring and surveillance, as well as to trajectory simplification. We propose an online algorithm to solve this problem, which can handle dynamic addition and deletion of data in logarithmic time. We assume an incoming stream of object positions, and maintain the upper and lower bounds for the radius of the smallest circle enclosing these positions, as points are added and deleted. These bounds allow us to greatly reduce the number of trajectory points we need to consider in the query, as

well as to defer query evaluation. Our method can approximate the radius of the smallest circle enclosing a given subtrajectory within an arbitrarily small user-defined factor. Our experiments show that the proposed method can scale up to hundreds of thousands of trajectories.

Then we present methods for detecting the potential occurrence of secret or private meetings, also known as conclaves which involves the assemblage of various persons of interest (e.g., terrorists, criminals). This is a primary concern amongst many surveillance, security, and criminal-investigation units. However, for many of these persons of interest, we have only partial observations of their individual movements. The movement is assumed to be constrained within a transportation network. Given such limited observations of these individuals moving independently within the network, we seek to infer the set of all possible corridors of travel (for each individual) as well as the set of all possible conclaves, including their potential participants, locations, and durations. We present a formal model for various strategies along with several algorithms for employing these search strategies. Experimental results on real-world road networks show that we can efficiently infer such information typically in only a matter of seconds for very large groups of interest.

Finally, we show how to find regions of interest (ROIs) from trajectory databases instead of retrieving them from any existing domain knowledge. ROIs are regions where a large number of moving objects remain for at least a given time interval. Previous techniques use somewhat restrictive definitions for ROIs, and are parameter-dependent. They require sequential scanning of the entire dataset to find ROIs when the ROI parameters change. Our approach is parameter independent, so that the user can quickly identify ROIs under

different parametric definitions without rescanning the whole database. We also generalize ROIs to be regions of arbitrary shape of some predefined density. We have tested our methods with large real and synthetic datasets to test the scalability and verify the output of our methods. Our methods give meaningful output and scale very well.

### 1.0.2 Trajectory Indexing

We also study indexing trajectories with space filling curves. Traditionally, spatial objects are first abstracted by minimum bounding rectangles (MBRs) which are then inserted in R-trees (or their variations). However, using MBRs to index trajectories introduces large dead space and thus a high number of false positives for queries. Space filling curves (SFC) have been used extensively for indexing points in two and higher dimensional spaces. Their advantage arises from their locality preservation and dimensionality reduction. Extending SFCs to index segments instead of points is not trivial as it may lead to false negatives and/or higher query times. In this thesis we show how use SFCs to index trajectory polylines. We provide algorithms that can be used in any DBMS that provides R-tree support. Furthermore when spatial coordinates are transformed to 1D numbers using any SFC, the Euclidean distance is not preserved, which negatively affects distance based queries (e.g. $k$NN queries). We show, how to implement $k$NN algorithms without converting 1D numbers back to spatial coordinates, thus making $k$NN queries efficient in this environment.

# Chapter 2

# Online Identification of Dwell Regions for Moving Objects

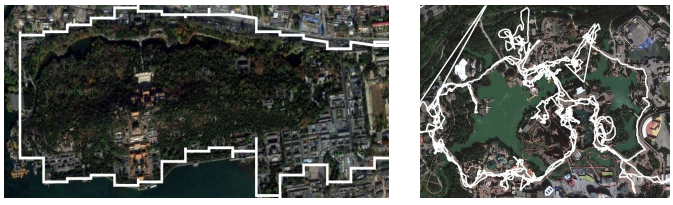
The pervasive deployment of GPS devices has made real-time position data from millions of moving objects readily available. Many applications, especially those involving monitoring and control, require on-line analysis of such data. We need real-time responses to spatiotemporal queries, since positions change rapidly, and queries quickly lose their value. It is essential to keep pace with high incoming data rates.

We consider the problem of online identification of *dwell regions*. A region $\mathcal{R}$ is a dwell region for a moving object $O$ if, given a radius $r_q$ and duration $t$, if $O$ remains within distance $r_q$ from every point in $\mathcal{R}$ for at least time $t$. We also consider the case where $t$ is not specified. Incoming position updates for $O$ are grouped into a subtrajectory $\mathcal{S}$, ensuring that $O$ remains within $r_q$ distance from all points in some region. The problem reduces to computing the smallest enclosing circle $\text{SEC}_{\mathcal{S}}$ of the points in $\mathcal{S}$. Computing $\mathcal{R}$ is hard for

streaming data. We propose approximate methods to compute dwell regions $\mathcal{R}$.

This problem has many real life applications. In surveillance and security applications, the object $O$ may represent a threat, and the region $\mathcal{R}$ may contain potential targets for $O$. For example, $O$ might be collecting information about the region $\mathcal{R}$ or maintaining communication with objects in $\mathcal{R}$ which is only possible within a certain distance from $\mathcal{R}$. Fast detection of $\mathcal{R}$ might be of critical importance. Identifying dwell regions is also important in animal behavior tracking, and may reveal animal territories. Wolf packs are known to stalk prey before attacking it. Identifying such behaviors reveals many interesting facts and is very important to ecosystem researchers [24]. The behaviors we consider in this chapter include both going around a region, as in Figure 2.1(a), or random movement in a certain enclosed region, as in Figure 2.1(b).

Our work also has applications in real time trajectory simplification based on spatio-temporal criteria. One such criterion [25] is the "disk criterion", which collapses into one segment all contiguous trajectory segments that can be enclosed by a fixed size disk. Our data structures and algorithms can be used to maintain a subtrajectory as long as it satisfies the disk criterion. For streaming scenario it is desirable to do this simplification in real time, without storing the data in a physical medium.



(a) (b)

Figure 2.1: Behaviors considered in this chapter.

We assume that every moving object sends regular position updates to a central system. For every moving object we will have a *streaming window* (or just *window*) of recent position records. We are to identify dwell regions as records are being added to and deleted from this window in real time.

Given a window $\mathcal{S}$, our approach approximates the smallest enclosing circle $\text{SEC}_\mathcal{S}$ of the points $\mathcal{S}$ as a polygon with a user-specified number, $k$, of sides. We maintain data structures which are used to compute upper and lower bounds on the radius of $\text{SEC}_\mathcal{S}$. We show that the actual radius is within a factor of $(1 + O(\frac{1}{k^2}))$ of the lower bound. The data structures can be updated for addition/deletion in $\text{O}(k \log n)$ time, $n$ is the window size $|\mathcal{S}|$. We can compute the upper and lower bounds in time $\text{O}(k)$.

Most of the time, we can decide whether $\text{SEC}_\mathcal{S}$ has radius $r_q$ or less from just the upper and lower bounds. When these bounds are insufficient to evaluate the query, we propose a method which allows us to consider only a few points in the window to compute $\text{SEC}_\mathcal{S}$. Computing $\text{SEC}_\mathcal{S}$ only gives the center of the circle, not the complete region $\mathcal{R}$. We hence propose a method for quickly approximating the region $\mathcal{R}$. Our contributions are:

- We maintain an approximation of $\text{SEC}_\mathcal{S}$ in logarithmic update time.
- We propose upper and lower bounds of the radius of $\text{SEC}_\mathcal{S}$, greatly reducing computation time.
- We show that the radius of $\text{SEC}_\mathcal{S}$ is within a factor of $(1 + O(\frac{1}{k^2}))$ of the lower bound, for a user defined $k$.
- We devise a method for selecting a few points using our data structures to compute $\text{SEC}_\mathcal{S}$ exactly.

- We discuss how to compute a fairly good approximation of the dwell region $\mathcal{R}$.
- We extend the online method to evaluate dwell region query on archived data.

The rest of the chapter is organized as follows: Section 2.1 provides the definitions and background while Section 2.2 describes some related works. Our proposed methods and data structures are described in Section 2.3 while Section 2.4 describes the query evaluation algorithms. Section 2.5 presents the preprocessing and query evaluation methods for archived data. Section 2.6 presents the experimental results and Section 2.7 concludes the chapter.

## 2.1 Background

Every moving object has a unique object ID and sends its position updates at certain regular intervals. A position update record contains object ID, spatial coordinate, and a timestamp. A ***trajectory***, with a particular $id$, is a finite sequence of $(x_i, y_i, t_i)$ triples. The $x_i, y_i \in \mathbb{R}^2$ are spatial coordinates, and the $t_i \in \mathbb{R}^+$, are timestamps, with $t_i < t_{i+1}$ for $i = 0, 1, \ldots, n-1$. A ***trajectory segment*** is a straight line between two consecutive tuples $(x_i, y_i, t_i), (x_{i+1}, y_{i+1}, t_{i+1})$ of the same trajectory, where $i \in \mathbb{N}_0$. A ***subtrajectory*** of length $m$ of a trajectory $T = (x_0, y_0, t_0), \ldots, (x_n, y_n, t_n)$, is a subsequence $T' = (x_i, y_i, t_i)$, $\ldots$, $(x_{m+i}, y_{m+i}, t_{m+i})$, of $m$ contiguous trajectory segments, where $i \geq 0, m < n$. A single trajectory segment is a subtrajectory of length one.

For each moving object we maintain a streaming window. A ***streaming window*** of size $n$ is the time-ordered sequence of the latest $n$ positions of the moving object. The length of the window depends on the duration $t$ specified by the query and the frequency

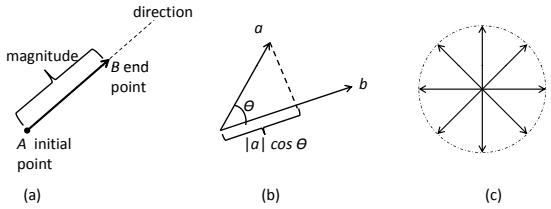



Figure 2.2: (a) A Euclidean vector (b) The scalar projection of $\vec{a}$ onto $\vec{b}$ (c) Eight uniformly spaced vectors around a circle.

of position updates from a moving object. A streaming window is updated by adding the most recent position when a new update record arrives and deleting the least recent point when necessary. For example, in applications like trajectory simplification, records can be added (without any deletion) as long as they satisfy the query condition.

We consider two types of queries. A *region query* $(r_q, t)$ requests the dwell region $\mathcal{R}$, each point of which is within a distance $r_q$ from the trajectory for time at least $t$. A *decision query* $r_q$ asks whether the positions of a given object in the streaming window fall within distance $r_q$ from any point in $\mathbb{R}^2$. This query returns a Boolean value and the center of the smallest enclosing circle. Decision queries are important for applications like trajectory simplification [25].

Consider circles of radius $r_q$ centered at each point in a window $\mathcal{S}$. The intersection of these circles is precisely the dwell region $\mathcal{R}$, since all points of in streaming window are within $r_q$ from any point in $\mathcal{R}$. We hence consider two approaches. The first maintains the overlap region of a set of circles centered at the object positions in $\mathcal{S}$. The other computes $\text{SEC}_{\mathcal{S}}$. Finding $\text{SEC}_{\mathcal{S}}$ suffices to answer decision queries, but not region queries.

A naive approach to maintaining the overlap between circles is to re-compute their intersections whenever a point is inserted into or deleted from the window. However, to the

best of our knowledge, there exists no efficient on-line algorithm to maintain intersection of circles. Additions can be made fast, but a deletion is always $O(n)$ making the update time $O(n)$. Our proposed method is based on approximating the SEC with a polygon of $k$ sides, and calculating the upper and lower bounds of the radius to answer decision queries. We allow the user to make the lower bound arbitrarily close to the actual radius of the SEC by tuning the value of $k$.

Computing $\text{SEC}_{\mathcal{S}}$ gives only the center of $\text{SEC}_{\mathcal{S}}$, not the entire dwell region $\mathcal{R}$. To answer region queries, we use efficient pruning to avoid unnecessary computation. First, no dwell region $\mathcal{R}$ can exist if the lower bound for the radius of $\text{SEC}_{\mathcal{S}}$ exceeds $r_q$. In this case, we do not compute the intersecting region. When the upper bound is less than $r_q$, we compute an overestimate $\mathcal{R}^+$ and an underestimate $\mathcal{R}^-$ for the actual region $\mathcal{R}$. We also identify *critical points* that are more likely to affect the shape of the region. We can efficiently maintain approximations by considering only critical points. Details are described in Section 2.4. We start with some basic definitions.

An $n$ dimensional *vector* is an $n$-tuple, $(v_1, v_2, \ldots, v_n)$, where $v_i$ is its component along the $i^{th}$ axis. We use an overhead arrow to distinguish a vector, as in $\vec{a}$. The *magnitude* $|v|$ of a vector $\vec{v}$ is denoted as $|v| = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}$. The *dot product* of two vectors $\vec{a}$ and $\vec{b}$ is defined as $\vec{a} \cdot \vec{b} = |a||b| \cos\theta = \sum_{i=1}^{n} a_i b_i$. If $\vec{b}$, is a unit vector then the dot product $|a| \cos\theta$ is the length of $\vec{a}$ in the direction of $\vec{b}$, also called the scalar projection of $a$ onto $b$.

If $k$ vectors are uniformly spaced around a circle, the angle between any two adjacent vectors is $\frac{2\pi}{k}$. Figure 2.2 shows (a) a Euclidean vector, (b) scalar projection of $\vec{a}$ onto $\vec{b}$ and (c) eight uniformly spaced vectors around a circle. In a Euclidean space each side

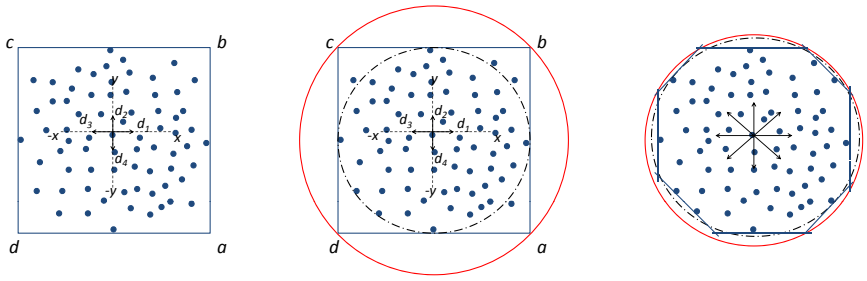
of a straight line is called a *half space*. Given the straight line $ax_0 + by_0 = c$, $(x_0, y_0) \in \mathbb{R}^2$, one half space is $H = \{(x, y) : ax + by \leq c\}$.

## 2.2 Related Work

Trajectories have received much attention recently. The work in [26] considers pattern queries on trajectories, while [27] considers similarity queries for trajecotries. There has also been recent work on finding semantic information from trajectory data [28][29][30]. These works focus on identifying regions where objects have remained for a while, but none of these consider identifying dwell regions. These works also do not consider streaming environments. Their approaches are not readily adaptible to our context.

There has also been work on identifying group behaviors, such as flock patterns [31], convoy detection [32], identifying density of moving objects [33], etc. However we consider individual trajectories instead of group behavior.

The first deterministic linear time algorithm for the smallest enclosing circle appeared in [34]. Several improvements were presented in [35][36][37]. These methods are




(a) (b) (c)

Figure 2.3: (a) MBR of a set of points (b) inner and outer circles (c) better estimation with higher number of directions.

based on linear programming techniques and involve expensive computations, such as solving systems of polynomials. None of these methods was designed for streaming environments, and require repeating the computation for every addition/deletion. Welzl proposed a simple-to-implement randomized algorithm [38] with expected linear run-time. It recursively finds three points on the boundary of the circle. The algorithm can handle addition in constant time but deletion has expected linear time. However, the worst case runtime of Welzl's algorithm is quadratic. We will adapt Welzl's algorithm for computing $\text{SEC}_{\mathcal{S}}$ for a small set of points $\mathcal{S}$.

Algorithm 1 describes the pseudo code of Welzl's algorithm. Given a set of points $\mathcal{S} = \{p_1, p_2, \ldots, p_n\}$ there is a set, $B$, of at most three points that determines $\text{SEC}_{\mathcal{S}}$, e.g., $\text{SEC}_{\mathcal{B}} = \text{SEC}_{\mathcal{S}}$. $B$ is called the basis of $\mathcal{S}$. Computation of basis of a set of at most 4 points can be done in constant time. The algorithm is started with the call MinDisk($\mathcal{S}$, $B$) where $B = \mathit{basis}(p_1, p_2, p_3)$ and the initial circle $D = \text{SEC}_{\mathcal{B}}$. The remaining points in $\mathcal{S}$ are tested one by one whether they are inside $D$. If the next point $p_i$ is inside $D$ then $D$ is the smallest enclosing circle of the set of points seen so far, $P$. Otherwise, a recursive call (line 8) is made to compute $\text{SEC}_{\mathcal{P}}$. This time the basis is $\mathit{basis}(B \cup \{p_i\})$. When this recursive call returns we have the basis and the smallest enclosing circle of points seen so far.

Several heuristics were proposed in [39] for computing circle intersections. However, these heuristics are not useful in environments where points are being added/deleted dynamically. This approach maintains an R-tree [40] for all the *static sites* and computes intersection of circles only when a moving object is out of the *safe zone*. The computation requires traversing the R-tree and making a heap of R-tree entries. With streaming data,

**Algorithm 1** MinDisk($\mathcal{S}$, $B$)

1: $\mathcal{S}$ : a set of points, $\{p_1, p_2, \ldots, p_n\}$.

2: $B$ : may contain at most 3 points, $T \subset \mathcal{S}$.

3: $P = B$ ($P$: set of points seen so far.)

4: $T = B$ ($T$: current basis of $P$.)

5: $D = \text{SEC}_{\mathcal{B}}$

6: **for** each $p_i \in S - B$ **do**

7: $P = P \cup \{p_i\}$

8: **if** $p_i$ is not inside $D$ **then**

9: $T = \text{MinDisk}(P, \mathit{basis}(T \cup \{p_i\}))$

10: $D = \text{SEC}_{\mathcal{T}}$

11: **end if**

12: **end for**

13: return $T$;

the R-tree must be updated and traversed, building the heap for every addition/deletion. Moreover, we will show that our proposed data structures render three of the heuristics in [39] unnecessary.

## 2.3 Data Structures & Algorithms

We present our algorithms and data structures for approximating the smallest enclosing circle $\text{SEC}_{\mathcal{S}}$. We also show how to bound the radius of $\text{SEC}_{\mathcal{S}}$ above and below using circles constructed from the minimum bounding polygon for $\mathcal{S}$. Some symbols used

in this chapter are listed in table 2.1.

### 2.3.1 Minimum Bounding Polygons

Figure 2.3(a) shows a set of points and their minimum bounding rectangle (MBR). MBRs indicate the maximum extents of a set of objects or points $\mathcal{S}$. In the 2-D case, we can construct an MBR for $\mathcal{S}$ as follows. We take four vectors $\vec{d_1}, \vec{d_2}, \vec{d_3}, \vec{d_4}$, spaced 90° apart, and four lines $e_i \perp \vec{d_i}, 1 \leq i \leq 4$. Now we sweep each $e_i$ in the direction of $\vec{d_i}$, in from infinity towards $\mathcal{S}$. We stop when each $e_i$ touches a point $p_i \in \mathcal{S}$. The lines $e_i$ form the edges of the MBR. We denote the set of vertices of the MBR as $\mathcal{V}$.

We can generalize this idea to get tighter upper and lower bounds by using $k$ uniformly spaced vectors $\vec{d_1}, \vec{d_2}, \ldots, \vec{d_k},\ k > 4$. Figure 2.3(c) shows eight uniformly spaced vectors and the corresponding bounding convex octagon. As before, the lines $e_i$ will be swept inwards from infinity until they touch points $p_i \in \mathcal{S}$.

**Definition 1** *The set $\mathcal{F} = \{p_i \in \mathcal{S}\}$ which the lines $e_i$ touch is the set of the frontier points of $\mathcal{S}$ in the directions $\vec{d_i}$.*

We denote the $k$-polygon bounding the set $\mathcal{S}$ by $MBP(\mathcal{S})$. Clearly, if $\mathcal{V}$ is the set of vertices of $MBP(\mathcal{S})$, then $\mathcal{V} \not\subset \mathcal{S}$. However, $\mathcal{F} \subset \mathcal{S}$ by definition.

### 2.3.2 Queries Using $\mathrm{SEC}_{\mathcal{S}}$ and $MBP(\mathcal{S})$

Let $r_{\mathrm{SEC}_{\mathcal{S}}}$ be the radius of $\mathrm{SEC}_{\mathcal{S}}$. We can now get upper and lower bounds for $r_{\mathrm{SEC}_{\mathcal{S}}}$ as follows. The smallest circle $\mathrm{SEC}_{\mathcal{V}}$ enclosing the set $\mathcal{V}$ of vertices of $MBP(\mathcal{S})$ is guaranteed to contain all the points of $\mathcal{S}$, and yields an overestimate for $\mathrm{SEC}_{\mathcal{S}}$ (see Figure

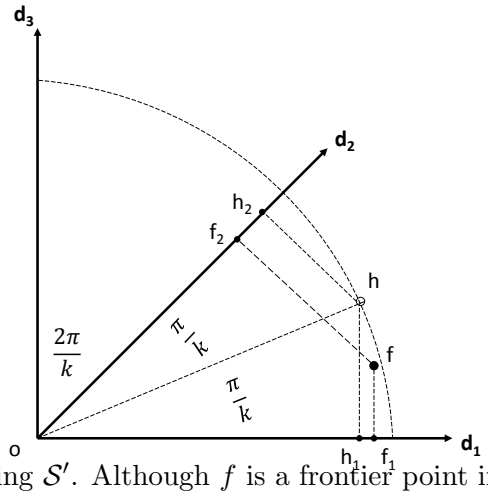


Figure 2.4: Computing $\mathcal{S}'$. Although $f$ is a frontier point in the direction of $\vec{d_1}$, it may not lie on the convex hull. Point $h$ is farther from the center of $\mathrm{SEC}_{\mathcal{S}}$ than $f$. We include all points whose projections exceed $|Of_1| \cos\left(\frac{\pi}{k}\right)$.

2.3(b)). Similarly, $\mathrm{SEC}_{\mathcal{F}}$, the smallest circle enclosing all the frontier points of $\mathcal{S}$ yields an underestimate for $\mathrm{SEC}_{\mathcal{S}}$.

**Definition 2** *Let the bounding k-polygon MBP($\mathcal{S}$) for a set of points $\mathcal{S}$, have vertex set $\mathcal{V}$ and frontier $\mathcal{F}$. We define the under-circle for $\mathcal{S}$ as $\lfloor \mathrm{SEC}_{\mathcal{S}} \rfloor \triangleq \mathrm{SEC}_{\mathcal{F}}$ and the over-circle for $\mathcal{S}$ as $\lceil \mathrm{SEC}_{\mathcal{S}} \rceil \triangleq \mathrm{SEC}_{\mathcal{V}}$. (See Figure 2.3.)*

Let $r_{\lfloor \mathrm{SEC}_{\mathcal{S}} \rfloor}$ and $r_{\lceil \mathrm{SEC}_{\mathcal{S}} \rceil}$ of $\lfloor \mathrm{SEC}_{\mathcal{S}} \rfloor$, and $\lceil \mathrm{SEC}_{\mathcal{S}} \rceil$, respectively. Clearly, $r_{\lfloor \mathrm{SEC}_{\mathcal{S}} \rfloor} \leq r_{\mathrm{SEC}_{\mathcal{S}}} \leq r_{\lceil \mathrm{SEC}_{\mathcal{S}} \rceil}$. We will show that the distance from the center of $\lfloor \mathrm{SEC}_{\mathcal{S}} \rfloor$ to any point in $\mathcal{S}$ will be at most $(1 + O(\frac{1}{k^2}))r_{in}$.

$r_{\lfloor \mathrm{SEC}_{\mathcal{S}} \rfloor}$ and $r_{\lceil \mathrm{SEC}_{\mathcal{S}} \rceil}$ are useful in optimizing decision queries. The query response can be YES or NO whenever $r_q > r_{\lceil \mathrm{SEC}_{\mathcal{S}} \rceil}$ or $r_q < r_{\lfloor \mathrm{SEC}_{\mathcal{S}} \rfloor}$ respectively.

### 2.3.3 Constructing $\mathrm{SEC}_{\mathcal{S}}$

If neither $r_q > r_{\lceil \mathrm{SEC}_{\mathcal{S}} \rceil}$ nor $r_q < r_{\lfloor \mathrm{SEC}_{\mathcal{S}} \rfloor}$ holds, we must construct $\mathrm{SEC}_{\mathcal{S}}$ explicitly. We now show how to construct this with minimal overhead.

**Definition 3** *The convex hull $\mathcal{C}(\mathcal{S})$ of a given set $\mathcal{S}$ is the minimal convex region enclosing $\mathcal{S}$. $\mathcal{H}(\mathcal{S})$ is the set of points defining the boundary of $\mathcal{C}(\mathcal{S})$.*

In our case, we know that the convex hull of $\mathcal{S}$ is a convex polygon whose vertices, $\mathcal{H}(\mathcal{S})$, are all in $\mathcal{S}$.

**Lemma 1** *If $\mathcal{C}(\mathcal{S})$ is the convex hull for a set $\mathcal{S}$, then $\mathrm{SEC}_{H(\mathcal{S})} = \mathrm{SEC}_{\mathcal{S}}$.*

*Proof:* By definition, $\mathrm{SEC}_{H(\mathcal{S})}$ is the minimal circle enclosing $\mathcal{C}(\mathcal{S})$, which is the minimal convex region enclosing $\mathcal{S}$. □

We proceed as follows. Clearly, $\mathcal{C}(\mathcal{S}) \subseteq MBP(\mathcal{S})$, since $MBP(\mathcal{S})$ may include dead space beyond $\mathcal{C}(\mathcal{S})$. We identify a subset $\mathcal{S}' \subseteq \mathcal{S}$ such that $\mathcal{H}(\mathcal{S}) \subseteq \mathcal{S}'$. We will find $\mathrm{SEC}_{\mathcal{S}'}$, which will give us $\mathrm{SEC}_{H(\mathcal{S})}$, and consequently $\mathrm{SEC}_{\mathcal{S}}$. This approach is efficient, since we expect $\mathcal{S}'$ to be much smaller than $\mathcal{S}$.

### 2.3.4 The Algorithm

The quality of the algorithm depends on $k$, the number of uniformly spaced vectors used. The algorithm maintains the frontier point corresponding to each vector. We now show how to identify frontier points, and how to update them dynamically as points are added to and deleted from the window $\mathcal{S}$.

Each point $p \in \mathcal{S}$ defines a vector. To identify a frontier point lying on an edge of $MBP(\mathcal{S})$, we identify a point whose projection onto the corresponding unit vector has

maximum length. We calculate the dot product of each of the $k$ vectors with each point of $\mathcal{S}$, and use the point with maximum projection length for each of the edges. The algorithm works as follows.

Select $k$ unit vectors $\vec{d_1}, \vec{d_2}, \ldots, \vec{d_k}$ uniformly spaced around a circle. For each $\vec{d_i}$, maintain the point $p$ in the current set that maximizes $\vec{d_i} \cdot \vec{p}$, which will be the point $p$ furthest in direction of $\vec{d_i}$. This requires computing $n$ dot products and building a max heap with the values of these dot products. There is one max heap for each vector. The point at the root of a heap is the one that has maximum scalar projection on the corresponding vector. Calculating the dot products is $O(n)$ for each vector and is performed only once, the first time a moving object reaches the required window size. Heap building is also done at the same time and its complexity is $O(n \log n)$. Addition (deletion) of a point from the streaming window requires one addition (deletion) from the heap. This can be done in $\mathrm{O}(\log n)$ time per unit vector i.e. $O(k \log n)$ time for $k$ vectors. The set $\mathcal{F}$ consists of the points at the heap roots. The set $\mathcal{V}$ is computed from the intersection of adjacent edges of $MBP(\mathcal{S})$.

Algorithm 2 describes the initial processing. The dot product between each vector and each point is calculated in line 7. One heap is built with the values of dot products for each vector. The heapify operation at line 9 builds the heap, which takes $O(n \log n)$. The dot product calculation for each vector in lines 6-8 is $O(n)$. As a result, the complexity of this pre-processing for $k$ vectors is $O(kn + kn \log n) = O(kn \log n)$. Note that a particular point will be at different locations in different heaps because of different dot product values with different vectors. If we want to access a particular point $p_j$ and/or its dot product

| | |
|---|---|
| $\mathcal{S}$ | A set of 2D points (possibly from the streaming window). |
| $\text{SEC}_{\mathcal{S}}$ | Smallest Enclosing Circle of a set $\mathcal{S}$ of points |
| $MBP(\mathcal{S})$ | Minimum $k$-polygon bounding $\text{SEC}_{\mathcal{S}}$ |
| $\mathcal{F}$ | Frontier points $\mathcal{S}$ on the edges of $MBP(\mathcal{S})$. |
| $r$ | Radius of $\text{SEC}_{\mathcal{S}}$. |
| $r_{in}$ | Radius of $\lfloor \text{SEC}_{\mathcal{S}} \rfloor$, the inner circle. |
| $r_{out}$ | Radius of $\lceil \text{SEC}_{\mathcal{S}} \rceil$, the outer circle. |
| $H$ | Half space. |
| $r(S)$ | Radius of the $\text{SEC}_{\mathcal{S}}$. |
| $h_i$ | Heap corresponding to unit vector $d_i$ |

Table 2.1: Symbols used in this chapter.

$\vec{d_i} \cdot \vec{p_j}$ of with the vector $\vec{d_i}$ we need to know where is this value in the heap $h_i$. For example we need to access a point when if it is to be deleted. To achieve this we also build a *Lookup Table* ($LT$) while building these heaps. The lookup table contains the location of a point in every heap. For example $LT[\vec{d_i}][\vec{p_j}]$ contains a pointer to $\vec{d_i} \cdot \vec{p_j}$ in $h_i$.

After building the heaps we must update them when inserting and deleting points from the $\mathcal{S}$, Algorithm 3 describes the update step. At each update step, a point $p$ is added to the window $\mathcal{S}$ as the most recent point, and the least recent point, $p_1$, is deleted from $\mathcal{S}$. This requires deleting the record for to $p_1$ from each heap ($LT$ is used). Next, the dot product between $p$ and each vector $\vec{d_i}$ is calculated and the result inserted in the corresponding heap. Insertion and deletion in a heap being $O(\log n)$, update is a $O(k \log n)$ operation.

## 2.4 Query Evaluation

We have described the data structures used to maintain the upper and lower bounds and to approximate the radius of $\text{SEC}_{\mathcal{S}}$. We now show how to evaluate queries. We

**Algorithm 2** buildHeaps($S$, $H$, $d$)

1: $S$ : streaming window, $\{p_1, p_2, \ldots, p_n\}$.

2: $H$ : $k$ heaps, $\{h_1, h_2, \ldots, h_k\}$. One for each direction being tracked.

3: $d$ : $k$ directions, $\{d_1, d_2, \ldots, d_k\}$.

4:

5: **for** each $d_i$ in $d$ **do**

6: **for** each point $p_j$ in $S$ **do**

7: $h_i[j] = d_i.p_j$

8: **end for**

9: heapify($h_i$)

10: **end for**

first consider decision queries, which ask whether or not the current window satisfies the query condition.

As we have seen, we maintain $k$ heaps $h_i$ corresponding to the $k$ vectors $\vec{d_i}$. The root of heap $h_i$ is a frontier point, since it maximizes the extent of $MBP(\mathcal{S})$ in $\vec{d_i}$'s direction. Let the bounding $k$-polygon $MBP(\mathcal{S})$ for $\mathcal{S}$ have vertices $\mathcal{V}$ and frontier points $\mathcal{F}$.

To get $r_{\lfloor \mathrm{SEC}_\mathcal{S} \rfloor}$ and $r_{\lceil \mathrm{SEC}_\mathcal{S} \rceil}$, we use Welzl's algorithm to compute $\mathrm{SEC}_\mathcal{V}$ and $\mathrm{SEC}_\mathcal{F}$ (see Definition 2). As already noted, $r_{\lfloor \mathrm{SEC}_\mathcal{S} \rfloor}$ and $r_{\lceil \mathrm{SEC}_\mathcal{S} \rceil}$ can be used to answer decision queries without actually computing $\mathrm{SEC}_\mathcal{S}$. However, when $r_{\lfloor \mathrm{SEC}_\mathcal{S} \rfloor}$ and $r_{\lceil \mathrm{SEC}_\mathcal{S} \rceil}$ are insufficient for this purpose (when $r_q < r_{\lceil \mathrm{SEC}_\mathcal{S} \rceil}$ and $r_q > r_{\lfloor \mathrm{SEC}_\mathcal{S} \rfloor}$) we must compute $\mathrm{SEC}_\mathcal{S}$.

**Algorithm 3** update($S$, $H$, $d$, $p$)

1: $S$ : streaming window, $\{p_1, p_2, \ldots, p_n\}$.

2: $H$ : $k$ heaps, $\{h_1, h_2, \ldots, h_k\}$. One for each direction being tracked.

3: $d$ : $k$ directions, $\{d_1, d_2, \ldots, d_k\}$.

4: $p$ : next point to be added in $S$.

5:

6: **for** each $d_i$ in $d$ **do**

7: delete LT$[d_i][p_1]$ from $h_i$

8: $val = d_i.p$

9: insert $val$ into $h_i$

10: **end for**

11: delete $p_1$ from $S$

12: add $p$ to the end of $S$

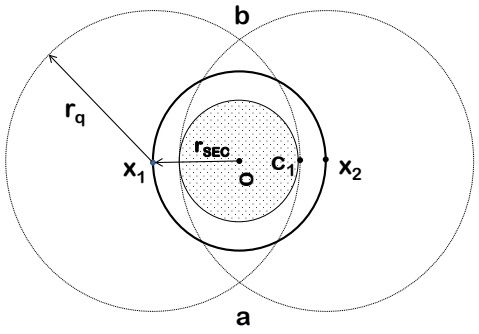



Figure 2.5: The shaded region is $\mathcal{R}^-$.

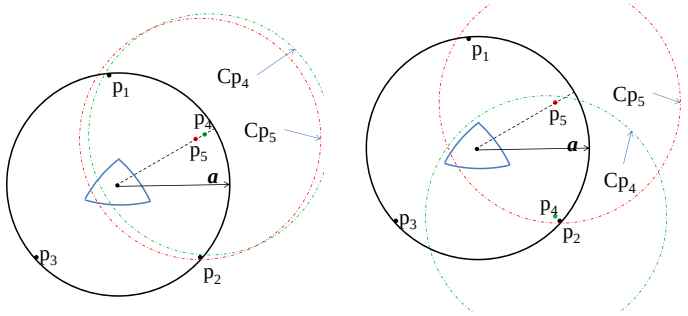



Figure 2.6: Showing the importance of points near the boundary to compute the dwell region $\mathcal{R}$ when the are at (a) the same angular position (b) a different angular position.

### 2.4.1 Efficient Computation of $\mathrm{SEC}_{\mathcal{S}}$

We now show how to compute $\mathrm{SEC}_{\mathcal{S}}$ using only a small subset $S' \subset S$ of points. Our central idea is to identify a set of points $\mathcal{S}'$ that includes all points on the convex hull $\mathcal{C}(\mathcal{S})$. In Figure 2.4, let $O$ be the center of $\mathrm{SEC}_{\mathcal{S}}$, and consider the angular sector between vectors $\vec{d_1}$ and $\vec{d_2}$. Let $f$ be the point in this sector with the maximal projection on to $\vec{d_1}$, so $f$ is the frontier point in the direction of $\vec{d_1}$.

However, $f$ need not be the furthest point in this sector from $O$. As Figure 2.4 shows, there could be a non-frontier point $h$ in this sector such that $|Oh| > |Of|$, but if

we consider their projections $h_1, f_1$ on $\vec{d_1}$, we have $|Oh_1| < |Of_1|$. It is quite likely that $h$ lies on the convex hull $\mathcal{C}(\mathcal{S})$, but we would miss it if we only looked at projections on the vectors $\vec{d_i}$.

Our challenge is to include all such points $h$ in $\mathcal{S}'$. We first note that this situation arises because the angular distance of $f$ from $\vec{d_1}$ is less than that of $h$. (If we consider their projections $h_2, f_2$ on to $\vec{d_2}$, we find $|Oh_2| > |Of_2|$.) We first observe that the projection of $f$ on $\vec{d_1}$ will be largest when $f$ lies on $\vec{d_1}$.

Let the line $Oh$ bisect the sector between $\vec{d_1}$ and $\vec{d_2}$. (We make this choice since it also minimizes the projection of $h$ on both $\vec{d_1}$ and $\vec{d_2}$, and maximizes the likelihood that point $h$ will not be a frontier point.) Let $f$ lie on $\vec{d_1}$ and rotate the line $Of$ so that it coincides with the bisector $Oh$. The projection of $f$ on $\vec{d_1}$ will now be $\lambda_f = |Of| \cos\left(\frac{\pi}{k}\right)$. By selecting all points whose projections on $\vec{d_1}$ are of length at least $\lambda_f$, we are sure to get all points in the half-sector that are at least as far from the center as $f$ is, and remain candidates for $\mathcal{C}(\mathcal{S})$. To get $\mathcal{S}'$, we proceed as follows:

1. Place all frontier points $f_i \in \mathcal{F}$ into $\mathcal{S}'$.
2. Let $f_i$ be the frontier point in the direction of vector $\vec{d_i}$, and let its projection on $\vec{d_i}$ be $\lambda_{f_i}$ Place into $\mathcal{S}'$ all points in the half-sector between $\vec{d_i}$ and $\vec{d_{i+1}}$ whose projections on $\vec{d_i}$ are larger than $\lambda_{f_i} \cos\left(\frac{\pi}{k}\right)$.

We can now state the following result.

**Theorem 2** $\mathrm{SEC}_{\mathcal{S}} = \mathrm{SEC}_{\mathcal{S}'}$.

*Proof:* Since the convex hull $\mathcal{C}(\mathcal{S})$ is the maximal convex region enclosing $\mathcal{S}$, all frontier points $f_i \in \mathcal{F}$ are in $\mathcal{C}(\mathcal{S})$. Step 1) above places $\mathcal{F}$ into $\mathcal{S}'$. However, not all points

in $\mathcal{C}(\mathcal{S})$ are in $\mathcal{F}$. Convexity of $\mathcal{C}(\mathcal{S})$ ensures that such points must be farther from the center of $\text{SEC}_{\mathcal{S}}$ than the frontier points. Step 2) above places all such points into $\mathcal{S}'$. □

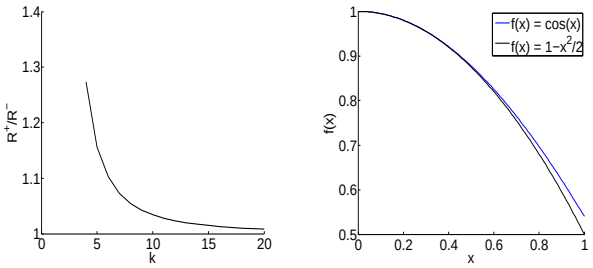
By Theorem 2, we need consider only points in $\mathcal{S}'$, which is much smaller than $\mathcal{S}$. Queries now run much faster.

### 2.4.2 Dwell Region Queries

As we have seen, the dwell region $\mathcal{R}$ is the intersection of all the circles of radius $r_q$ centered at each point of $\mathcal{S}$. When $r_{\text{SEC}_{\mathcal{S}}} = r_q$, the circles centered at the points on the perimeter of $\text{SEC}_{\mathcal{S}}$ will share only its center. In that case, $\mathcal{R}$ will consist of only one point, namely, the center of $\text{SEC}_{\mathcal{S}}$.

Next, consider the case when $r_q \geq r_{\text{SEC}_{\mathcal{S}}}$. Let $C_x$ be a circle of radius $r_q$ centered at a point $x$ on the circumference of $\text{SEC}_{\mathcal{S}}$. Consider the region $\mathcal{R}^- = \bigcap_x C_x$. (See shaded region in Figure 2.5.) If we move the center of $C_x$ along the circumference of $\text{SEC}_{\mathcal{S}}$, the intersection of the resulting circles will be a disk $\mathcal{R}^-$ of radius $\delta = r_q - r_{\text{SEC}_{\mathcal{S}}}$ centered at the center of $\text{SEC}_{\mathcal{S}}$.

We can also calculate a region, $\mathcal{R}^+$, that contains $\mathcal{R}$. The intuition behind our




(a) (b)

Figure 2.7: Trigonometric functions. (a) The ratio $\frac{\mathcal{R}^+}{\mathcal{R}^-} = \frac{k \tan\theta}{\pi}$ as $k$ changes (b)Approximation of $\cos x$ with $1 - \frac{x^2}{2!}$.

method is as follows. Suppose we have a partially computed region $\mathcal{R}_m$, which is the intersection of some $m$ circles. To get $\mathcal{R}$, we must compute the intersection of remaining $|S| - m$ circles with $\mathcal{R}_m$. Now, if any of these circles fully contains $\mathcal{R}_m$, then it will not affect $\mathcal{R}_m$ at all. Our goal is to use those points first that are more likely to intersect $\mathcal{R}_m$, since the remaining circles are less likely to have an effect on $\mathcal{R}_m$.

Intuitively, points closer to the boundary of $\text{SEC}_\mathcal{S}$ are more likely to affect the region $\mathcal{R}_m$, as we will illustrate through an example. Our reasoning is similar to that used to prove Lemma 3 of [39].

Figure 2.6(a) shows $\text{SEC}_\mathcal{S}$ for som set of points $\mathcal{S}$. Assume that points $p_1, p_2, p_3$ are on the boundary and $c$ is the center of the $\text{SEC}_\mathcal{S}$. The partial region $\mathcal{R}_m$ appears near the center as the intersection of three circles of radius $r_q$ centered at $p_1, p_2, p_3$, repectively. Consider two other points $p_4, p_5$ inside $\text{SEC}_\mathcal{S}$ lying on the same radial vector $\vec{a}$. Let $p_5$ be closer to the center of $\text{SEC}_\mathcal{S}$ than $p_4$, and let $C_{p_4}$ and $C_{p_5}$ be circles of radius $r_q$ centered at $p_4, p_5$ respectively. The minimum distances from the center $c$ to any point on $C_{p_4}$ and $C_{p_5}$ are $\delta_1 = r_q - |cp_4|$ and $\delta_2 = r_q - |cp_5|$ respectively. Since $\delta_1 < \delta_2$, $C_{p_5}$ is more likely to fully include $\mathcal{R}_m$ than $C_{p_4}$.

This illustration shows the importance of points closer to the boundary that are at a same angular position in the circle. However, if $p_4$ and $p_5$ had different angular position then $p_5$ might have trimmed $\mathcal{R}'$ more than $p_4$, Figure 2.6(b). Nevertheless, if the points are uniformly distributed, considering them in order from the boundary towards the center will still give us a better chance to get points which are more likely to affect the shape of $\mathcal{R}$. Our experimental evaluation shows that when the answer to the decision query is "yes", points

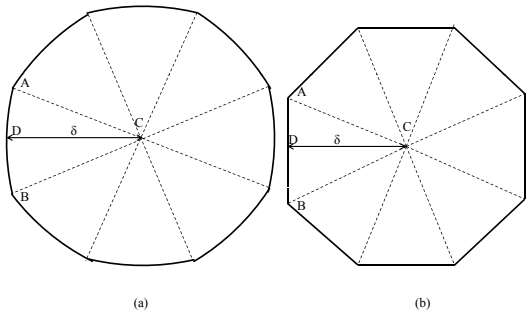



Figure 2.8: Replacing the bounding arcs of the dwell region with straight lines. (a) actual region (b) bounding arcs replaced with straight lines.

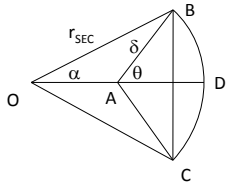



Figure 2.9: Approximating $\mathcal{R}^+$.

are fairly uniformly distributed around the circle and thus the above heuristic applies in practice.

| Time of collection | Number of trajectories | Number of spatial points | Sampling frequency | Description |
|---|---|---|---|---|
| Apr 2007 to Aug 2009 | 165 | 24778552 | 2-5 sec | **GeoLife Data:** |
| April to August of 1993-1996 | 253 | >287,000 | 1hr | **DeerElk Data:** |

Table 2.2: Description of real data set.

To calculate a region $\mathcal{R}^+ \supseteq \mathcal{R}$, it suffices to consider points in a subset of $\mathcal{S}$. Our goal is to make this subset as small as possible, and make $\mathcal{R}^+$ as close to $\mathcal{R}$ as possible. Towards this goal, we select points closer to the boundary of $\text{SEC}_{\mathcal{S}}$ first. The heap data structures we maintain can be used to select points that are closer to the boundary of the

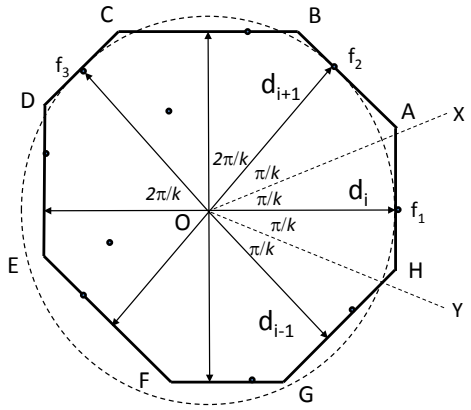
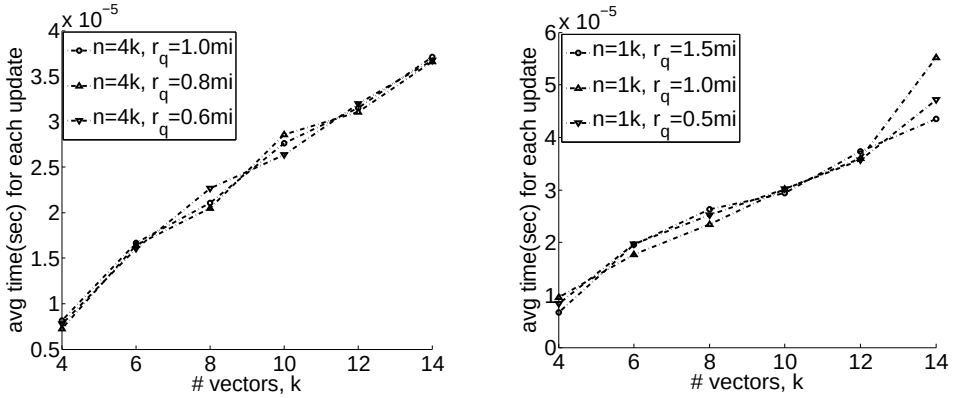



Figure 2.10: Proving Theorem 3. $MBP(\mathcal{S})$ = ABCDEFGH.




(a)GeoLife Data (b)ElkDeer Data

Figure 2.11: Average time required for each update of the data structures and evaluating the query for a moving object.

circle. If we consider only the $k$ points that make up the frontier points (the heap roots), we will get an intersecting region that contains $\mathcal{R}$. If a tighter approximation is required (at the cost of more CPU time) points in the set $\mathcal{S}'$ (figure 2.4) can be considered.

[39] describes five heuristics to discard circles which are not going to affect the intersecting region. Heuristics 2, 3, 5, are used to discard circles that do not have a common intersecting region. For our case since we want to find the intersection only when every circle shares the intersecting region we do not need to consider heuristics 2, 3, 5. Heuristics 1, 4, are used to discard circles that fully contain the intersecting region computed so far and so are not going to affect the region. By considering points in the above mentioned order and applying heuristics 1 and 4 from [39] we can further avoid computing unnecessary circle intersections.

The correctness of the algorithm for the decision query follows from 1) the fact that we actually compute the SEC when upper/lower bounds cannot decide the query answer and 2) theorem 2.

### 2.4.3 Goodness of the Approximation

To measure the goodness of the region approximations, we will attempt to derive the ratio $\mathcal{R}^+/\mathcal{R}^-$. The area of the circular region $\mathcal{R}^-$ is $\pi\delta^2$, where $\delta = (r_q - r_{\mathrm{SEC}_\mathcal{S}})$. We calculate the area of $\mathcal{R}^+$ in the following ideal scenario. We assume $\mathcal{R}^+$ is calculated using $k$ frontier points at the heap roots, so that there are $k$ circular arcs defining the boundary of $\mathcal{R}^+$. We further assume that the arc lengths are equal. Figure 2.8(a) shows $\mathcal{R}^+$ with eight bounding arcs. Each of the sectors in this figure is equivalent to the sector ABDC in Figure 2.9. We obtain the area of this sector as follows.

We begin by noting that $|BC| = 2\delta\sin(\theta)$, and further that

$$\alpha = \arcsin\left(\frac{|BC|}{2r_{\mathrm{SEC}_\mathcal{S}}}\right) = \arcsin\left(\frac{\delta\sin(\theta)}{r_{\mathrm{SEC}_\mathcal{S}}}\right). \tag{2.1}$$

Standard formulas yield the area of the circular segment

$$\mathrm{BCDB} = \frac{r^2_{\mathrm{SEC}_\mathcal{S}}}{2}\left(2\alpha - \sin(2\alpha)\right).$$

Now, applying elementary trigonometry and simplifying,

$$\begin{aligned}\mathrm{ABDC} &= \mathrm{ABCA} + \mathrm{BCDB} \\ &= \frac{1}{4}\delta^2\sin(2\theta) + \frac{r^2_{\mathrm{SEC}_\mathcal{S}}}{2}\left(2\alpha - \sin(2\alpha)\right).\end{aligned}$$

The combined area of all sectors in Figure 2.8 is $k$ times the above area. Hence,

$$\begin{aligned}\frac{\mathcal{R}^+}{\mathcal{R}^-} &= \frac{k\left(\frac{1}{4}\delta^2\sin(2\theta) + \frac{r^2_{\mathrm{SEC}_\mathcal{S}}}{2}\left(2\alpha - \sin(2\alpha)\right)\right)}{\pi\delta^2} \\ &= \frac{k}{\pi}\left(\frac{1}{4}\sin(2\theta) + \left(\frac{r_{\mathrm{SEC}_\mathcal{S}}}{\delta}\right)^2\left(\alpha - \frac{1}{2}\sin(2\alpha)\right)\right)\end{aligned}$$

We can now use the value of $\alpha$ in Equation 2.1.

While the above analysis gives the exact equation of the ratio, we do the following estimation to better understand how it changes with $k$. Consider the area $CADB$ where $C$ is the center of the $\mathrm{SEC}_\mathcal{S}$. Recall that the centers of the bounding arcs are on the boundary of $\mathrm{SEC}_\mathcal{S}$. The closest point from $C$ on arc $AB$ is $D$, with $|CD| = \delta$. The bounding arcs are expected to be very small and hence these arcs can be replaced with straight lines at distance $\delta$ from $C$. The result is a polygon with $k$ boundary lines. Figure 2.8(b) shows the polygon with eight lines which replaces the area of figure 2.8(a). The computation of

$\mathcal{R}^+/\mathcal{R}^-$ is given below-

$$\angle ACD = \frac{1}{2}\angle ACB = \pi/k = \theta$$

$$AD = \delta \tan\theta$$

$$\triangle\, CAD = \frac{1}{2} AD \times CD = \frac{1}{2}\delta^2 \tan\theta$$

$$\Rightarrow \triangle\, CAB = \delta^2 \tan\theta$$

$$\therefore \frac{\mathcal{R}^+}{\mathcal{R}^-} = \frac{k\delta^2 \tan\theta}{\pi\delta^2}$$

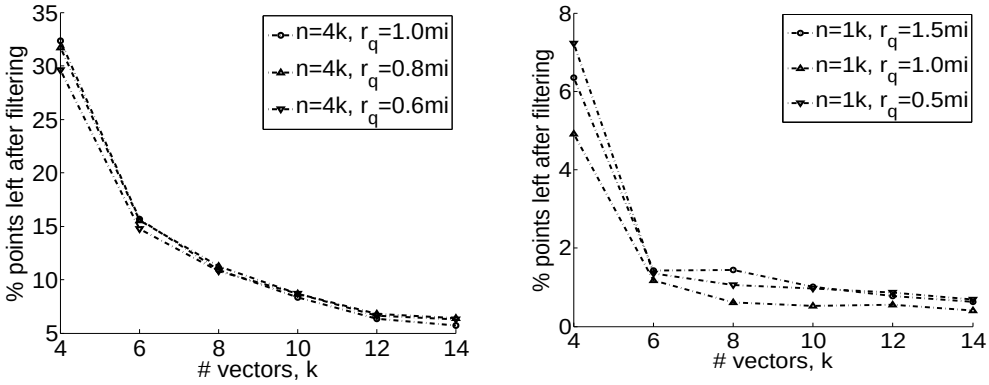
$$= \frac{k \tan\theta}{\pi}$$

We want this ratio to be as close to 1.0 as possible. Figure 2.7(a) shows how this ratio decreases towards 1.0 as $k$ increases. As we increase $k$ linearly, $\tan\theta$ decreases at a faster rate than the rate of increase of $k$. This results in decreasing the ratio with the increase of $k$.

Finally, we prove that $r_{\mathrm{SEC}_\mathcal{S}} \approx (1 + O(1/k^2)) r_{\mathrm{SEC}_\mathcal{F}}$. This means that we can get an arbitrarily good approximation of the radius by maintaining just a constant number $k$ of direction vectors, in $\mathrm{O}(\log n)$ time per update.

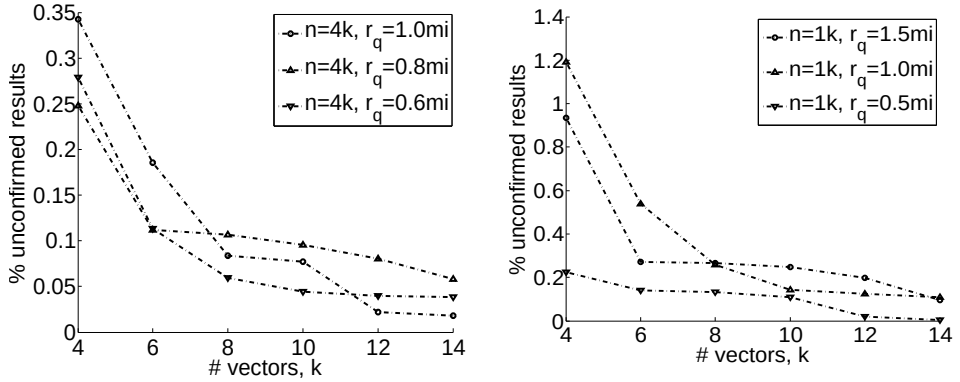
**Theorem 3** *Let $\mathcal{S}$ be the current set of points in the streaming window, and $\mathcal{F}$ be the set of frontier points. Then, $r_{\mathrm{SEC}_\mathcal{S}} \leq r_{\mathrm{SEC}_\mathcal{F}}(1 + O(1/k^2))$.*

*Proof:* Figure 2.10 shows a set of points $\mathcal{S}$, their minimum bounding $k$-polygon $MBP(\mathcal{S})$ = ABCDEFGH, the set $\mathcal{F}$ of frontier points in the directions of vectors $\vec{d_1}, \vec{d_2}, \ldots$, as well as the frontier circle $\mathrm{SEC}_\mathcal{F}$ defined by the frontier points $f_1, f_2$, and $f_3$. Let $O$ be the center of $\mathrm{SEC}_\mathcal{F}$. Clearly, $|Of_1| = r_{\mathrm{SEC}_\mathcal{F}}$. Let $OX$ bisect the angle between $\vec{d_i}$ and $\vec{d_{i+1}}$, and $OY$ bisect the angle between $\vec{d_i}$ and $\vec{d_{i-1}}$.




(a)GeoLife Data (b)ElkDeer Data

Figure 2.12: Fraction of points selected to calculate actual SEC.




(a)GeoLife Data (b)ElkDeer Data

Figure 2.13: Fraction of results where actual SEC has to be computed (e.g., heuristics do not answer the query).

The sector between the bisectors $OX$ and $OY$ is fully contained in the isosceles triangle defined by the lines $OX$, $OY$, and $AH$ (extended as needed). From elementary trigonometry, all points in this isosceles triangle, and hence all points in the sector of $MBP(\mathcal{S})$ defined by $OX$ and $OY$ lie within distance $|Of_1|/\cos\left(\frac{\pi}{k}\right) = r_{\mathrm{SEC}_\mathcal{F}}/\cos\left(\frac{\pi}{k}\right)$ of the point $O$.

Since all points in $\mathcal{S}$ lie within the polygon $MBP(\mathcal{S})$, all points $p_i \in \mathcal{S}$ in this sector are within distance $r_{\mathrm{SEC}_\mathcal{F}}/\cos\left(\frac{\pi}{k}\right)$ of the point $O$. Clearly, this means that we can include all of $\mathcal{S}$ in a circle of this radius.

Hence, $r_{\mathrm{SEC}_\mathcal{S}} \leq r_{\mathrm{SEC}_\mathcal{F}}/\cos\left(\frac{\pi}{k}\right)$. We can now use a Taylor series expansion to obtain $\cos\left(\frac{\pi}{k}\right) = 1 - \frac{\pi^2}{2k^2} + O\left(\frac{\pi^4}{k^4}\right)$, so that $\cos\left(\frac{\pi}{k}\right) = 1 - O\left(\frac{1}{k^2}\right)$ as $\left(\frac{\pi}{k}\right) \to 0$, that is, as $k$ increases. We now have $r_{\mathrm{SEC}_\mathcal{S}} \leq r_{\mathrm{SEC}_\mathcal{F}}\left(1 + O\left(\frac{1}{k^2}\right)\right)$. □

## 2.5 Dwell Region for Archived Data

So far we have described the online algorithm to find dwell region for streaming data. If we want to evaluate the same query on archived data we can still scan the data and use the online algorithm. However, the archived data allows us to save any preprocessed information about the data which might make the query evaluation faster. In this section we present such a preprocessing method for archived data.

### 2.5.1 $R$-lists

We assume a minimum and maximum possible value of query radius $R_{min}$ and $R_{max}$ respectively. We consider $n-1$ equally apart values between $R_{min}$ and $R_{max}$, i.e.

**Algorithm 4** ExtendSubT($S$, $H$, $d$, $r_i$)

1: $S$ : current subtrajectory, $\{p_1, p_2, \ldots, p_n\}$.

2: $H$ : $k$ heaps, $\{h_1, h_2, \ldots, h_k\}$. One for each direction being tracked.

3: $d$ : $k$ directions, $\{d_1, d_2, \ldots, d_k\}$.

4: $r_i$ : current list radius.

5:

6: **for** each $d_i$ in $d$ **do**

7: $val = d_i.p_n$

8: insert $val$ into $h_i$

9: **end for**

10: $X = \text{MBP}(H)$

11: $\lceil \text{SEC}_{\mathcal{S}} \rceil = \text{MinDisk}(X)$

12: **if** $r_{\lceil \text{SEC}_{\mathcal{S}} \rceil} > r_i$ **then**

13: $\text{SEC}_{\mathcal{S}} = \text{effMinDisk}(S, H)$

14: **if** $r_{\text{SEC}_{\mathcal{S}}} > r_i$ **then**

15: Insert an entry in $r_i$-list.

16: Delete $p_1$ from $S$

17: **end if**

18: **end if**

$r_0 = R_{min}, r_1, \ldots, r_{n-1}, r_n = R_{max}$. These values are called list radii. For each list radius $r_i$ we evaluate all possible maximal duration subtrajectories which can be enclosed in a circle of radius $r_i$ ($r_i$-circle). For each radius $r_i$ we maintain a list, $r_i$-list, of these subtrajectories sorted by duration. The collection of all $r_i$-lists is called $R$-lists.

To build $R$-lists we have to scan a trajectory $T$ once for every $r_i$ and use the online algorithm, i.e. each trajectory is scanned $n + 1$ times. We start with the first point $(x_0, y_0, t_0)$ of $T$ and add one by one from remaining points of $T$ in temporal order. Let $S$ be the current subtrajectory of $T$ being considered. After adding each point to $S$ we compute $r_{\lfloor \mathrm{SEC}_S \rfloor}$ and $r_{\lceil \mathrm{SEC}_S \rceil}$ using the online algorithm. We keep adding points to $S$ as long as we have $r_{\lceil \mathrm{SEC}_S \rceil} \leq r_i$. When $r_{\lceil \mathrm{SEC}_S \rceil}$ exceeds $r_i$ we compute $r_{\mathrm{SEC}_S}$ and check if $r_{\mathrm{SEC}_S} \leq r_i$. If yes, we continue adding points. When we find $r_{\mathrm{SEC}_S} > r_i$ for the first time, we make an entry in the $r_i$-list. The entry contains trajectory ID, starting point of $S$ in $T$ and the duration up to the point preceding the last one in $S$. After making the entry we move the start point of $S$ by one and keep adding points to $S$ using the same procedure described above. Algorithm 4 shows the method for building $R$-lists. After scanning for $r_i$, we scan $T$ for $r_{i+1}$.

A $r_i$-list records every maximal duration subtrajectory starting at any point of each trajectory. Consider an entry $e^{r_i}$ in $r_i$-list with radius $r_{e^{r_i}} \leq r_i$ and duration $d_{e^{r_i}}$ starting at the $k^{th}$ point $(x_k, y_k, t_k)$ of $T$. When we will scan $T$ for $r_{i+1}$-list we will have an entry $e^{r_{i+1}}$ starting at $(x_k, y_k, t_k)$ with radius $r_{e^{r_{i+1}}} \leq r_{i+1}$ and $d_{e^{r_{i+1}}}$. Lets $S_{e^{r_i}}$ and $S_{e^{r_{i+1}}}$ be the subtrajectories represented by $e^{r_i}$ and $e^{r_{i+1}}$. Its easy to see that $r_{e^{r_i}} \leq r_{e^{r_{i+1}}}$, $d_{e^{r_i}} \leq d_{e^{r_{i+1}}}$ and $S_{e^{r_i}} \subseteq S_{e^{r_{i+1}}}$. This subset property is important in query evaluation. We

say $e^{r_{i+1}}$ is the *next radius entry* of $e^{r_i}$.

### 2.5.2 Query Evaluation using $R$-Lists

In this section we describe how $R$-lists can be used to efficiently evaluate dwell region queries on archived data. If we plot radius and duration $(r, d)$ pairs from $R$-lists we will get points along a vertical line for each $r_i$-list, figure 2.14. If the query radius $r_q$ is equal to any list radius $r_i$ then query evaluation is straightforward, just look up the $r_i$-list. Entries of $r_{i+1}$-list has radius more than $r_q$. And because of the subset property described above, we do not have to look into any previous lists either. In the following text we describe how to evaluate queries when $r_q$ is not equal to any list radius.

Consider a query point $q = (r_q, d_q)$ where $r_i < r_q < r_{i+1}$. $q$ divides the space into four quadrant, figure 2.14. None of the entries in $SE$-quadrant satisfies the query condition (larger radius but lower duration than those of the query). In $NW$-quadrant all the entries satisfies the query condition. Entries in $SW$-quadrant satisfy the radius condition but have the duration less than $d_q$. So we can extend the corresponding subtrajectories (by adding points from the trajectory) up to duration $d_q$ and check whether any of them meets the radius condition. Similarly, entries in $NE$-quadrant satisfy duration condition but radius is more than $r_q$. We can shorten the corresponding subtrajectories so that they can be enclosed in a $r_q$-circle and check the duration condition.

However, because of the subset property all the entries of $NW$-quadrant do not need to be in the query result. Also we do not need to extend or shorten and check all the entries of $SW$ or $NE$-quadrant respectively. We need to look into $r_i$-list and $r_{i+1}$-list only. Let $r_i^{\uparrow}$ ($r_i^{\downarrow}$)-list be the part of $r_i$-list where any entry $e^{r_i^{\uparrow}}$ ($e^{r_i^{\downarrow}}$) has duration

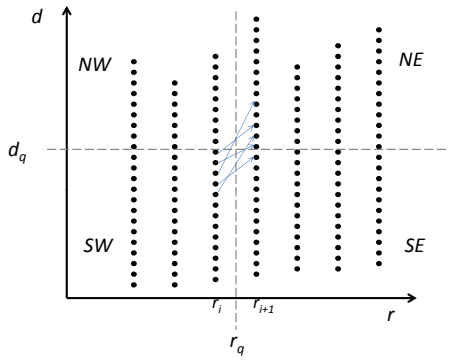



Figure 2.14: *R-Lists.*

$d_{e^{r_i^{\uparrow}}} < d_q (d_{e^{r_i^{\downarrow}}} \geq d_q)$. Similarly, $r_{i+1}$-list is divided into $r_{i+1}^{\uparrow}$ and $r_{i+1}^{\downarrow}$-list by $d_q$. All the entries in $r_i^{\uparrow}$-list goes to the result. This is easy to do, because $R$-lists are sorted by duration. None of the entries in $r_{i+1}^{\downarrow}$-list satisfies the query condition.

Now, we have to check entries in $r_i^{\downarrow}$ and $r_{i+1}^{\uparrow}$-list. Entries in $r_i^{\downarrow}$-list needs to be extended up to at least duration $d_q$ and check if $r_{\mathrm{SEC}_{\mathcal{S}}} < r_q$. Similarly, entries in $r_{i+1}^{\uparrow}$-list needs to be shortened down to radius $r_q$ and check if $d_{\mathcal{S}} > d_q$. This requires to read the corresponding trajectory data from the disk and use the online algorithm. So this step might be an expensive operation.

However, if an entry $e^{r_i^{\downarrow}}$ has it's next radius entry in $e^{r_{i+1}^{\downarrow}}$ then extending $e^{r_i^{\downarrow}}$ will not add to the result. Similarly, if $e^{r_{i+1}^{\uparrow}}$ is a next radius entry of $e^{r_i^{\uparrow}}$ then shortening $e^{r_{i+1}^{\uparrow}}$ is redundant. If an entry $e^{r_i^{\downarrow}}$ has it's next radius entry in $e^{r_{i+1}^{\uparrow}}$ only then extending $e^{r_i^{\downarrow}}$ might add to the result. So it suffices to extend only these entries. To identify which entries are required to extend, at each entry we store the duration of its next radius entry. With this information we can easily identify which entries to extend. Since, we are extending entries from $r_i^{\downarrow}$-list, entries from $r_{i+1}^{\uparrow}$-list are not required to shorten.

## 2.6 Experiments

All the experiments were run in an Intel Xeon 3.0GHz processor running Linux 2.6.18 with 8GB of main memory. In our experiments we use two real datasets. The GeoLife dataset [41] contains public activity data (i.e. shopping, dining, sightseeing, hiking, cycling etc.) in Beijing, China. The DeerElk data contains the trajectories of deer, elk and cattle in the Starkey Experimental Forest and Range in Oregon, USA [42]. Table 4.1 provides the description of the real datasets. Different set of parameter (window size, query radius, and number of vectors) values were considered for different datasets as indicated in the plots.

First, we examine the average time required to update the data structure as the number of vectors changes. The experimental results appear in figure 2.11. As expected, with the increase of the number vectors the time required for each update increases. However the rate of increase is very slow, e.g., using up to 10 vectors the update process is efficient enough to handle orders of hundreds of thousands of moving objects. Since the approximated radius is a factor of $1 + O(\frac{1}{k^2})$, using $k = 10$ can approximate the radius within order of 1% of the actual radius.

To depict the pruning power of our data structures, next, we compute the fraction of points used by our method to evaluate the query when the upper and lower bounds are not enough to answer the query. As seen in figure 2.12, while the number of vectors increases, the fraction of points considered falls sharply. For example with 10 vectors we need to consider only 10% of the window size.

Figure 2.13 shows the percent of unconfirmed results as the number of vectors changes. The fraction of unconfirmed results decreases as the number of vectors increases.

| | | | | |
|---|---|---|---|---|
| 4k, 1.0mi | 0.525011 | 0.514656 | 0.495247 | 0.475605 |
| 4k, 0.8mi | 0.523428 | 0.516502 | 0.486613 | 0.470082 |
| 4k, 0.6mi | 0.522052 | 0.498885 | 0.479042 | 0.461957 |

Table 2.3: Results of Chi-square test.

As a matter of fact, the query is actually evaluated less than 2% of the times, while in the rest it is answered through the heuristics.

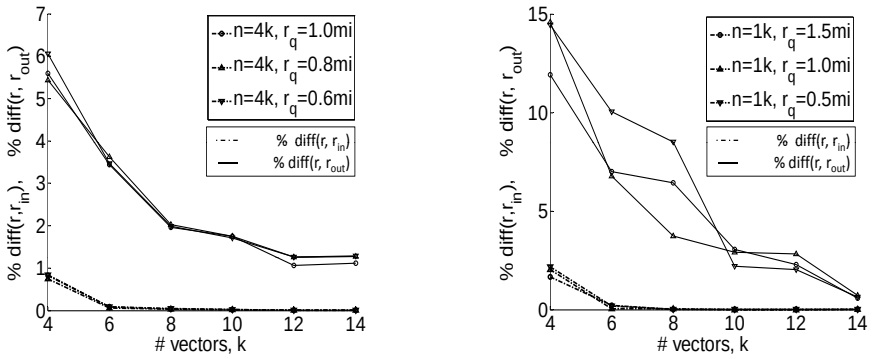
Figure 2.15 depicts the percentage of difference between the actual radius of the SEC and the upper and lower bounds. While the lower bound is always very close to the actual radius the upper bound might be higher.

Finally, we examined the distribution of points around the circle when there is a circle of desired radius. We do Chi-square test of the angular position of the points near the perimeter of the circle. These are the points in the set $\mathcal{S}'$. The null hypothesis for the Chi-square test is "Points are uniformly distributed around the circle". Table 2.3 shows the p-values of Chi-square test. A p-value of 0.05 or less would mean that there is a significant difference between the observed distribution and the theoretical (uniform) distribution. Its equivalent to say that with 95% confidence interval a significant difference is observed. In that case we could reject the null hypothesis. As the experimental results show, p-values are around $0.5 - 0.6$, i.e., we cannot reject the null hypothesis with 95% confidence interval.

## 2.7 Conclusions

This chapter considers a novel problem for moving objects: real time identification of moving objects that are staying within a certain distance from a (unspecified) dwell

region, for at least a certain duration. The main contribution of our work is to propose online method to evaluate the decision query in logarithmic time. Our proposed methods lend to approximating the radius of the smallest enclosing circle of a given subtrajectory within user defined arbitrary factor and efficient approximation of the dwell region. Our experimental evaluation shows that our algorithm is capable of evaluating the query condition on hundreds of thousands of moving objects per second on average.




(a)GeoLife Data (b)ElkDeer Data

Figure 2.15: Comparison between $r_q(= r), r_{\lfloor \mathrm{SEC}_{\mathcal{S}} \rfloor}(= r_{in})$ and $r_{\lceil \mathrm{SEC}_{\mathcal{S}} \rceil}(= r_{out})$.

# Chapter 3

# Corridors and Conclaves: Inferring Group Meetings From Partial Observations

Consider the scenario in which we wish to track the spatio-temporal trajectories of an adversarial group of associated persons of interest (e.g., terrorists, fugitives, etc.). Unfortunately, tracking such groups using available intelligence-gathering means does not always yield full trajectory data due to either adversarial countermeasures or lack of continuous surveillance. However, it may still be possible to obtain partial information about object movement and behavior by integrating data from multiple sensory means.

Although GPS devices are now common, full trajectory data for moving objects is not always available because of privacy issues (or even because GPS may not be available, e.g., within a mall, etc.). Yet, spotty location data for moving objects may still be

available from many sources, such as location sharing in social networks, check-in applications, surveillance cameras, cell-phone usage, sighting reports, etc. Devising techniques for inferring useful intelligence from such spotty data and other available information (e.g., the underlying transportation network, via which the persons of interest must travel) is an important challenge.

As a simple example, in Fig. 3.1, we have observations of 2 objects (persons) of interest, $O_1$ and $O_2$, but only at their respective entry points, $s_1$ and $s_2$, and exit points, $t_1$ and $t_2$. However, we have no information about the possible object trajectories (shown as dashed lines) within the unobserved area (i.e., the "blind" region), nor do we have any information about where these two objects could have possibly met while unobserved (e.g., at node $v$). Regardless, we would like to be able to draw inferences about object behavior inside these unobservable regions, specifically about possible meetings and/or communications between the moving objects. For this, we may assume that we know the underlying network topology within the region (i.e., the grid lines in the Fig. 3.1; in general, the transportation network), but not the actual trajectories that the objects followed.

Given the known associations amongst the group of interest, chances are high that group members may eventually meet somewhere within the unobserved area (in secrecy) to exchange information or for some other tactical purposes, such as planning. Being able identify the place(s) where all or some fraction of them could have visited at the same time could give a useful lead to their eventual apprehension or further surveillance (as they may be likely to meet there again in the future). It is also useful to be able to prioritize these places based on either the number of people that could have met at such places or the

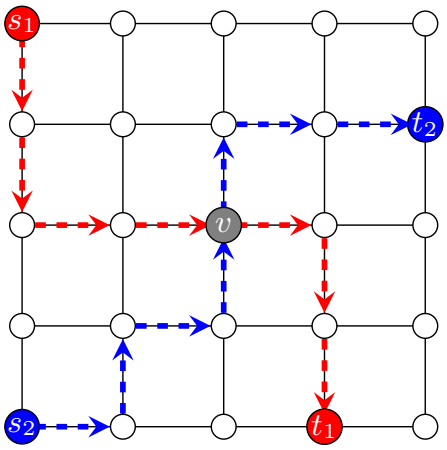



Figure 3.1: Partial observations of moving objects and their possible trajectories and meeting location.

potential duration of the possible meetings.

## 3.1 Related Work

Typically a moving object location is a tuple of the form $(x, y, t)$. Here $(x, y)$ is the spatial coordinate where the moving object was at time $t$. Such tuples can come from GPS sensor updates, social network check-ins, surveillance systems, etc. In between consecutive tuples $(x_1, y_1, t_1)$ and $(x_2, y_2, t_2)$ where $t_2 > t_1$, the object is assumed to follow a straight line. However, this is not always true. For example, in a road network an object can take only certain routes (which may not be a straight line) while in a free space the object can possibly randomly deviate from the straight line. The number possible routes or amount of deviation is limited by the cost of the path taken between $t_1$ and $t_2$, and a maximum possible speed of the moving object.

In the literature, two sources of uncertainty in moving object location have been considered. First one is because of the precision of the GPS sensor (type-I uncertainty). The second one is because of the location update frequency (type-II uncertainty). To handle type-I uncertainty, a disk of a certain radius around each GPS reported location is considered, i.e., the object can possibly be anywhere inside that disk. Type-II uncertainty occurs when two consecutive updates are so far apart that the intermediate position is not known for sure. For moving object in road network, type-II uncertainty exists if there are multiple possible routes between two consecutive location updates. In free space, type-II uncertainty always exists as the location updates occur at discrete timestamps. An elliptical region, with the two consecutive locations as the foci of the ellipse, that contains the possible location of the object is considered to handle this uncertainty. A better model is to have a probability distribution function of the location of the object inside the disk or the elliptical area.

Type-I uncertainty for range query is considered in [43, 44, 45, 46], where [44] provides the detailed model for including uncertainty in the data indexing and query processing. Ni et al. [43] Consider type-I uncertainty about the location of static spatial objects. They consider spatial join of uncertain polygons (representing the boundary of static spatial objects) and propose an R-Tree based method PrTree for the join. [44] provides algorithms for querying both past and future location of the moving object. [47] tries to solve type-I uncertainty (in free space) differently than these previous works by using a stochastic model. According to this model, the location probabilities of an object at a particular time is dependent on its previous location.

Range queries with type-II uncertainty is considered in [16, 15]. [16] considers range query on moving objects in free space while [15] considers snapshot and continuous range queries in network space. [48] considers continuous nearest neighbors query in free space with type-II uncertainty. These works are implicitly using the notion of a corridor between two consecutive location updates. Among these works, [15] is the closest to ours as it considers type-II uncertainty in road network. However, our problem requires to compute intersection of the corridor nodes in the road network of a number of moving objects, which was not attempted in any previous work.

The map matching problem [49] is to match actual GPS data to a digital map or road network. Map matching problem can occure in both type-I and type-II uncertainty. [50, 51, 52] propose map matching algorithms to find the potential routes taken by a moving object under type-II uncertainty in a road network. [50] uses historic trajectories to find potential routes while [51, 52] use road network topology and spatio-temporal attributes of the trajectory (for which map matching is being done). None of the map matching algorithms can be used to identify the corridor between two contiguous GPS samples.

## 3.2 Formal Tracking Model

The underlying transportation network is represented as a weighted, directed graph $G = (V, E, w)$, where $V$ is the set of nodes representing, e.g., road intersections, $E \subseteq V \times V$ is the set of edges representing, e.g., roads crossing between nodes in $V$, and $w \rightarrow \mathbb{R}_{>0}$ is a positive weight function mapping the edges to their respective travel times. Let $d(u, v)$ denote the duration to travel the quickest path from nodes $u$ to $v$.

Let $O = \{o_1, o_2, \ldots o_r\}$ be the set of moving objects being tracked (e.g., individual persons of interest). We use $o_i@\langle u, \tau\rangle$ to denote that the object $o_i$ was at node $u \in V$ at time $\tau$. For every object $o_i \in O$, we know their start and end nodes and the timestamps for when the object was at those nodes. Specifically, for each $o_i \in O$, we know the values $s_i, t_i \in V$ and $\alpha_i, \omega_i, \delta_i \in \mathbb{R}_{\geq 0}$ such that $o_i@\langle s_i, \alpha_i\rangle$ and $o_i@\langle t_i, \omega_i\rangle$, indicating that object $o_i$ takes a duration of $\delta_i = \omega_i - \alpha_i \geq d(s_i, t_i)$ time units to travel from $s_i$ to $t_i$.

For a node $u \in V$, $ea_i(u) = \alpha_i + d(s_i, u)$ is the *earliest-arrival time* at which $o_i$ can arrive at $u$ and $ld_i(u) = \omega_i - d(u, t_i)$ is the *latest-departure time* at which $o_i$ can depart $u$ to reach $t_i$ at $\omega_i$. Let $\rho_i(u) = [ea_i(u), ld_i(u)]$ be the possible *presence interval* of $o_i$ at node $u$. An interval $[a, b]$ is valid iff $a \leq b$. Otherwise, we say that $[a, b] = \emptyset$ represents an "empty" interval. We say $u$ is *reachable* by $o_i$ iff $\rho_i(u) \neq \emptyset$. Let $C_i = \{u \in V \mid \rho_i(u) \neq \emptyset\}$ be the set of *corridor nodes* for $o_i$ (i.e., nodes reachable by $o_i$ on its way from $s_i$ to $t_i$).

Two objects $o_i$ and $o_j$ $(i \neq j)$ can possibly meet at $u$ if their presence intervals overlap at $u$: $\rho_i(u) \cap \rho_j(u) \neq \emptyset$. The longest possible conclave interval for a given set of objects $O' \subseteq O$, at node $u$, is $I_{O'}(u) = \forall_{o_i \in O'} \bigcap \rho_i(u)$. Let the set of *conclave nodes* for a subset of objects $O' \subseteq O$ be defined as $C_{O'} = \{u \in V \mid I_{O'}(u) \neq \emptyset\}$. Note that meeting duration includes the start and end timestamps (e.g., the meeting duration of two objects is considered one time unit when the arrival time of one object coincides with the departure time of another object).

## 3.3 Query Types

Given the tracking model presented above, the most basic types of queries to support within this model are reporting either the set of corridor nodes for a given object $o_i$ (to see where $o_i$ could have traveled) or the set of conclave nodes for a given a subset of objects $O' \subseteq O$ (to see where these objects could have met). However, we may further extend this model to support various other queries of interest by incorporating additional constraints and query objectives.

The first of these extended queries is the $(\gamma, \tau)$-Query, which returns $\{u \in V \mid \exists\, O' \subseteq O, |O'| \geq \gamma, |I_{O'}(u)| \geq \tau\}$. That is, we report the set of conclave nodes at which at least $\gamma$ objects could have met for at least a duration of $\tau$ time units. This allows for finer control over the sizes of groups or durations of meetings which we are interested in tracking.

Other extended queries that we also consider in this model include the Top-$k$-$\gamma$ Query, in which we report the $k$ conclave nodes with the *largest-possible meeting sizes* ($\gamma$) of at least some fixed duration ($\tau$), and the Top-$k$-$\tau$ Query, in which we report the $k$ conclave nodes with the *longest-possible meeting durations* ($\tau$) of at least some fixed meeting size ($\gamma$).

## 3.4 Methods

Let $\Delta = \{\delta_1, \ldots, \delta_r\}$ be the set of object travel times and so $C_O$ be the set of conclave nodes for all objects in $O$ with the travel times given by $\Delta$. We first describe an algorithm for the $(\gamma, \tau)$-query with $\gamma = |O| = r$ and $\tau = 1$. We shall call this the $(r, 1)$-query. After describing the concepts of the basic $(r, 1)$-query algorithm, we then extend

this approach to show how to evaluate the query for any arbitrary value of $(\gamma, \tau)$ and top-$k$ queries.

### 3.4.1 The Naïve Solution

The naïve solution we present to evaluate an $(r, 1)$-query has two main phases: a *search phase* and an *evaluation phase*. Beginning with the search phase, a bidirectional Dijkstra[53] search is carried out by searching forward from the source, $s_i$, and backward from the destination, $t_i$, of each object $o_i \in O$, with $\delta_i \in \Delta$ as the search cutoff bound. Initially, $\forall\, u \in V$, we set $I_O(u) = [I_a, I_b] = [-\infty, \infty]$ and, $\forall\, o_i \in O$, we set $ea_i(u) = \infty$ and $ld_i(u) = -\infty$. Each forward and backward search then sets $ea_i(u) = d(s_i, u)$ and $ld_i(u) = d(t_i, u)$, only when $d(s_i, u) \leq \delta_i$ and $d(u, t_i) \leq \delta_i$, respectively. After the bidirectional search for a given object $o_i$, for each $u \in V$ reachable by $o_i$, we update $I_O(u) = [\max\{I_a, ea_i(u)\}, \min\{I_b, ld_i(u)\}]$. After each bidirectional search, we may also easily establish the set of corridor nodes $C_i$ for the corresponding moving object $o_i$, according to the definition given in Sec. 3.2. Note that a node can be in the corridor of multiple objects and thus be reachable by multiple objects. During the search phase we also keep count of the total number of objects that can reach a node $u$. We call this the *reachability number* $\eta(u)$ of node $u$. We initialize $\eta(u) = 0$ for all $u \in V$ before beginning the search phase. As we do the bidirectional search for each object, $\eta(u)$ is incremented when node $u$ becomes reachable by that object. After running bidirectional searches for all objects in $O$ (i.e., the search phase), we start the evaluation phase. In this phase, we evaluate all relevant nodes to determine if they are possible conclave nodes for all of the objects of interest. Since an $(r, 1)$-query requires nodes at which *all* $r$ objects could have met, we therefore evaluate a

node iff $\eta(u) = r$ (note that this only means that all $r$ objects could *reach* node $u$ within their respective corridors, but it does not guarantee they could have all been present at the same time). Therefore, if a node has $\eta(u) = r$, we then check it's conclave interval $I_O(u)$. If $I_O(u) \neq \emptyset$, then $u$ is a conclave node (i.e., all objects could meet at $u$).

Note that the above method needs to run exactly $2r$ Dijkstra searches for $r$ objects. This implies that a node can be explored by up to $2r$ distinct searches in the worst case. Later in Sec. 3.4.4, we propose an alternate method based on a pre-processing technique for speeding up shortest-path computations, which allows us to instead explore every node at most twice, making for a much faster and more scalable approach overall.

### 3.4.2 Arbitrary $(\gamma, \tau)$-Query

To maintain the minimum meeting duration ($\tau$) condition, we may simply adjust the travel times to $\delta'_i = \delta_i - \tau$ before starting the search phase. Let $\Delta = \{\delta'_1, \ldots, \delta'_r\}$ be the set of adjusted of travel times from this point on unless specified otherwise. This implies that, after the search phase, $ea_i(u) \leq ld_i(u)$ iff $|\rho_i(u)| \geq \tau$ and, for a subset of objects $O'$, if $I_{O'}(u) = [a, b]$ and $a \leq b$, then the objects in $O'$ can meet at $u$ for at least a duration of $\tau$ units.

Let $O^C_u$ be a maximal set of objects that can meet at node $u$. The minimum meeting size ($\gamma$) condition requires that $|O^C_u| \geq \gamma$ for $u$ to be a conclave node. Unlike the $(r, 1)-$query, we cannot update conclave duration for a node because during the bidirectional searches(search phase) we do not know which subset of $O$ is going to satisfy the $(\gamma, \tau)$ condition. So we cannot decide whether a node is a conclave node by checking the reachability number and conclave duration for $u$. Even if $\eta(u) \geq r$, all the objects may not

reach $u$ at the same time and so $|O_u^C|$ may be less than $\gamma$. To evaluate an $(\gamma, \tau)$-query with arbitrary values of $\gamma$ and $\tau$, potential nodes are evaluated (during the evaluation phase) at the end of the search phase.

Nodes with reachability number $\eta(u) \geq \gamma$ are the potential nodes and evaluated in the evaluation phase. Let $O_u^R$ be the set of objects for which $u$ is reachable. A brute force method is to consider all subsets $O' \subseteq O_u^R$ with $|O'| \geq \gamma$ and compute $I_{O'}(u)$. However, this is expensive when many subsets of size $\gamma$ or more is possible(e.g. when $\gamma$ is not close to $|O_u^R|$). To avoid this expensive operation we devise the following *line sweep* algorithm.

We start by selecting nodes $\{u|\eta(u) \geq \gamma\}$. Figure 3.2 illustrates the line-sweep algorithm with 10 objects with $t_a$, $t_b$, $t'_a$ etc. being the $ea$, $ld$ timestamps for a node $u$. To determine whether at least $\gamma$ objects from $O_u^R$ can meet at node $u$, we sort the $ea$ and $ld$ timestamps of all objects in $O_u^R$ in increasing order. Then we "sweep the line" from the earliest to the latest timestamp. As we sweep the line, we keep a counter $c$ of intervals that have started but not ended yet i.e. *live entries*. The value of $c$ gives us the number of overlapping intervals at the current position of the line. The maximum value of the counter, $c_{max}$, gives the maximum number of objects that can meet at node $u$ e.g., 6 in figure 3.2. Note that there might be multiple subsets of objects that satisfy the $\gamma$ condition. If we want to determine all such subsets we have to continue the line sweep up to the last timestamp. However, if we only want to know whether there is at least one subset that satisfies the $\gamma$ condition, we can stop the line sweeping as soon as $c \geq \gamma$. For example, if $\gamma = 3$ then we ca stop at timestamp $t_c$. Another quantity for early stopping is $c$ plus number of future objects $f$. Future objects are the ones whose interval hasn't started yet. If $c + f < \gamma$, then

at least $\gamma$ objects will not be able to meet at node $u$ and we can stop the line sweeping. For example if $\gamma = 8$ then at timestamp $t'_c$ we have $c = 3$ and $f = 4$ and so we can stop the line sweeping. This condition can always be used (i.e., whether we are looking for all subsets that satisfy the query condition or just a yes/no answer).

### 3.4.3 Top-$k$ Queries

We consider two types of top-$k$ queries: *i*) Top-$k$-$\gamma$ with a minimum $\tau$ requirement and *ii*) Top-$k$-$\tau$ with a minimum $\gamma$ requirement.

To evaluate top-$k$-$\gamma$(top-$k$-$\tau$) query a priority queue, $Q^\gamma(Q^\tau)$, for the top-$k$ candidate nodes is maintained in the evaluation phase of the above algorithm for $(\gamma, \tau)$-query. Nodes with a certain reachability number(depending on top-$k$-$\gamma$ or $\tau$ query) are evaluated and the top-$k$ queue is updated if this node can improve the current list of top-$k$ candidates.

Let $\chi_{min}^{Q^\gamma}$ be the minimum conclave size in the priority queue $Q^\gamma$. For top-$k$-$\gamma$ query, a node $u$ is evaluated if $Q^\gamma$ has at least $k$ elements ($Q^\gamma \geq k$) and $\eta(u) \geq \chi_{min}^{Q^\gamma}$. A node is always evaluated if $Q^\gamma < k$. We use the same line sweep algorithm described above to evaluate a node, but we cannot use the same early stopping conditions described above. Because, if the maximum number of objects that can meet at node $u$ is higher than the minimum meeting size in $Q^\gamma$ then we need to update the $Q^\gamma$ by deleting the node with minimum meeting size and inserting the node $u$ with its maximum conclave size. To compute the maximum conclave size at node $u$, we need to complete the line sweep algorithm for $u$. However, as before, if we can figure out early that the node cannot possibly improve the current top-$k$ list then we can stop early. One early stopping criterion we use is $c + f \leq c_{max}$, which means continuing the line sweep algorithm will not improve

the maximum conclave size we have already seen. Note that, if we stop early with this condition we might have $c_{max} \geq \chi_{min}^{Q^\gamma}$ and we will have to update $Q^\gamma$. Consider a top-$k$-$\gamma$ query and $\chi_{min}^{Q^\gamma} = 5$. According to the example in figure 3.2 at time $t_g$ we have $c + f = 6$ which is not going to improve the current best for this node. So we stop at this point but we still update $Q^\gamma$ and $\chi_{min}^{Q^\gamma}$ becomes 6. Another condition is $c + f \leq \chi_{min}^{Q^\gamma}$ which implies that a bigger conclave size than $\chi_{min}^{Q^\gamma}$ cannot be achieved from the current position of the sweeping line. In this case we do not have to update $Q^\gamma$. This time consider the above query with $\chi_{min}^{Q^\gamma} = 9$. At time $t'_f$ we have $c + f = 9$. So we stop at this point and do not update $Q^\gamma$.

To evaluate top-$k$-$\tau$ query with a minimum conclave size $\gamma$ we do not adjust the travel times to account for a minimum coclave duration. We evaluate a node $u$ if $\eta(u) \geq \gamma$. To evaluate a node we modify the above line sweep algorithm. Since our goal is to maximize the meeting duration for a minimum meeting size $\gamma$, considering meeting size more than $\gamma$ may reduce the meeting duration. So, as we sweep the line the start of a meeting duration is recorded as soon as an interval ($\gamma^{th}$ interval) comes alive and the number of live objects increases to $\gamma$. Then we continue the line sweep and do not adjust the meeting duration for meeting sizes larger than $\gamma$. We record the end of the meeting when an interval dies and the number of live objects falls below $\gamma$. The meeting duration is computed from the start time of the $\gamma^{th}$ interval. To do this we need to keep track of the $\gamma^{th}$ interval as we continue the line sweep and intervals start and end. We maintain a list of live intervals sorted by their start time and a pointer to the $\gamma^{th}$ interval in this list. When an interval, with earlier start time than that of $\gamma^{th}$ interval, dies we advance the pointer $\gamma^{th}$ to the $\gamma^{th}$ interval by one

(when there are more than $\gamma$ intervals in the list of live intervals). If an interval that started after $\gamma^{th}$ interval dies we simply delete it from the list and do not modify the pointer.

Consider the example in figure 3.2 again. Consider a top-$k$-$tau$ query with $\gamma = 3$. At time $t_c$ we have 3 alive intervals $\{a, b, c\}$. So $c$ becomes the $\gamma^{th}$ interval and $t_c$ is the start time of a conclave of at least $\gamma$ objects. Then interval $d$, $e$, $f$ starts and interval $f$ ends but the $\gamma^{th}$ entry remains the same (and so does the start time of the conclave). Next, interval $a$ dies at $t'_a$ and the pointer to the $\gamma^{th}$ interval is moved forward to $d$ since $a$ is within the first $\gamma$ intervals. When the end of an interval requires to update the $\gamma^{th}$ interval pointer, we compute the convlave duration and update the maximum conclave duration found so far if necessary. At time $t'_c$ the set of live intervals is $\{b, d, e\}$. The next end of the interval $b$ makes the set of live intervals of size $2 < \gamma$ and the $\gamma^{th}$ interval pointer becomes invalid. At this point the conclave duration of at least $\gamma$ objects is $t'_b - t_e$, which is also the largest duration so far. As the line sweeping proceeds on, conclaves of size at least $\gamma$ start at time $t_g$ and $t_j$ and they end at $t'_e$ and $t'_g$ with duration $t'_e - t_h$ and $t'_g - t_j$. Out of all these conclaves we take the longest one and update the $Q^\tau$ if necessary. Note that, along the line-sweep as we not only find the longest conclave duration, we also know the object IDs of those who were present in that conclave. This line-sweep algorithm can also be used to find all possible conclaves of size at least $\gamma$ or more.

### 3.4.4 CH Based Solution

In this section we show that, the search phase can be significantly improved using our proposed method based on contraction hierarchy. We, again, describe the search phase for $(r, 1)$-query first. The evaluation phase is same for both Dijkstra based and CH based

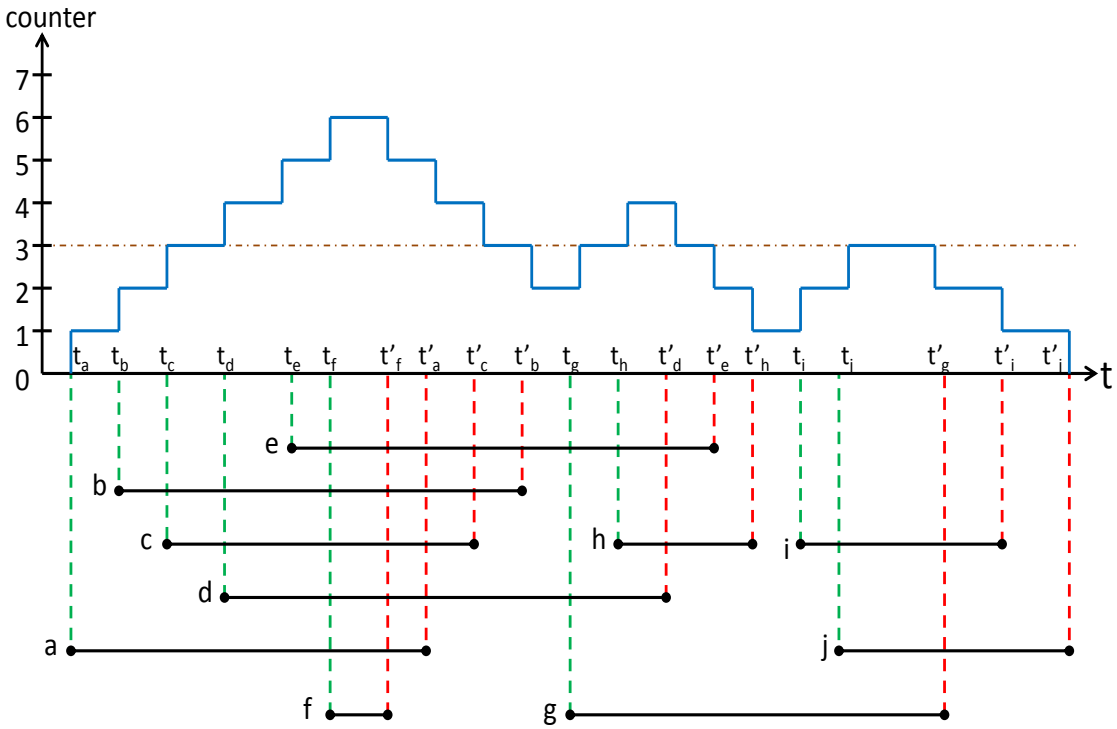



Figure 3.2: The Line-Sweep Algorithm. There are ten intervals $\{a, b, c, d, e, f, g, h, i, j\}$ with there presence interval $[t_a, t'_a]$, $[t_b, t'_b]$ etc. at node $u$.

solutions and the extension to other queries is done by using the line-sweep algorithm in the same way. However, we will show that, for the CH based methods, we do not have to finish the search phase to start the evaluation phase. The evaluation phase can be carried on along with the search phase. A node is evaluated when it is accessed during the search phase. We start with a brief description of CH before presenting our methods in detail.

CH pre-processing has two phases: *i*) ordering the nodes of the graph, and *ii*) contracting nodes one by one in that order. Any arbitrary ordering would preserve the correctness of the algorithm but some orderings improve the performance more than others. After setting up a particular order of the nodes in $G$, $\phi : V \rightarrow \{1, \ldots, |V|\}$, they are contracted in this order. Contracting a node means adding shortcuts to bypass the node when there is a unique shortest path through that node. For example, if there is a pair of incoming and out going edges $(u, v)$ and $(v, x)$ respectively, and $(u, v, x)$ is a unique shortest path then a shortcut edge $(u, x)$ with weight $w(u, v) + w(v, x)$ is introduced. In this way, the contracted node is not deleted from the graph but new edges are introduced to bypass this node during shortest path computation. After contracting all nodes, the graph contains higher number of edges, i.e., $G = (V, E \cup E', w)$ where $E'$ is the set of shortcut edges.

To achieve speedup in shortest path computation from $s$ to $t$ where $s, t \in V$ in the contracted graph, the regular bidirectional Dijkstra algorithm needs to be modified. The forward search from $s$ is run in the graph $G_{\uparrow} = (V, E_{\uparrow})$, where $E_{\uparrow} = \{(u, v) \in E \cup E' | \phi(u) < \phi(v)\}$ and the backward search from $t$ is run at the same time in $G_{\downarrow} = (V, E_{\downarrow})$, where $E_{\downarrow} = \{(u, v) \in E \cup E' | \phi(u) > \phi(v)\}$. A candidate shortest path cost is updated when the two searches meet at some node in the graph. The search in a particular direction may be

stopped when the minimum cost in the priority queue for that direction is higher than the minimum candidate path cost seen so far.

Note, however, that, unlike CH, our first goal is to, compute the corridor nodes for each $(s_i, t_i)$ pair, not just the shortest path. And then, compute the set of conclave nodes which is a subset of the intersection of the corridor nodes of all objects. A two phase algorithm for finding corridor nodes for a given $(s, t)$ pair is described in [54]. The first phase (upward search) runs a forward search from $s$ in $G_\uparrow$ and a backward search from $t$ in $G_\downarrow$. In the second phase (downward search), nodes touched in the first phase are accessed in decreasing order and downward edges are expanded to find the corridor nodes. Both the upward and downward searches are bounded by a threshold (e.g., travel time in our case). To compute corridor nodes for multiple $(s_i, t_i)$ pairs, a straightforward way would be to use the above algorithm once for each $(s_i, t_i)$ pair.

However, because of the structural properties of CH, we can compute the corridor nodes for all $(s_i, t_i)$ pairs simultaneously by running a modified version of the above two phase algorithm (bounded by thresholds in $\Delta$) just once. In the upward search phase we run a forward search from $s_i$ in $G_\uparrow$ and backward search from $t_i$ in $G_\downarrow$ for all $o_i \in O$. Let $d_{s_i}(v)$ $(d_{t_i}(v))$ be the duration of the quickest path found in the forward and backward search to(from) $v$ from(to) $s_i(t_i)$. We start with an increasing rank order queue, *incQ*, containing $\forall_{o_i \in O} s_i, t_i \in V$. We set $\forall_{o_i \in O}\ d_{s_i}(s_i) = 0$, $d_{t_i}(t_i) = 0$, and $d_{s_i}(v) = d_{t_i}(v) = \infty$ where $v \neq s_i, t_i$ at the beginning of the search. When a node $v$ is accessed from *incQ* $i$) it is inserted into a decreasing rank order queue, *decQ* if $\exists_{o_i \in O}\ d_{s_i}(v) \leq \delta_i$ or $d_{t_i}(v) \leq \delta_i$ and $ii$) the upward edges of $v$ are expanded (in the same way as regular Dijkstra search). After the

upward search, we set $V_\Delta = \emptyset$ and access nodes in $decQ$ in decreasing rank order. When a node $v$ is accessed from $decQ$ $i$) if $\forall_{o_i \in O}\ d_{s_i}(v) + d_{t_i}(v) \leq \delta_i$ then we set $V_\Delta = V_\Delta \cup \{v\}$ and $ii$) the edges of $v$ are expanded downward in the following way. For all $(v, x) \in E \cup E'$, $\phi(v) > \phi(x)$ set $\forall_{o_i \in O}\ d_{s_i}(x) = \min\{d_{s_i}(x), d_{s_i}(v) + w(v, x)\}$ and if $x \notin decQ$ and $\exists_{o_i \in O}$, $d_{s_i}(x) + d_{t_i}(x) \leq \delta_i$ set $decQ = decQ \cup \{x\}$. For all $(u, v) \in E \cup E'$, $\phi(v) > \phi(u)$ set $\forall_{o_i \in O}$ $d_{t_i}(u) = \min\{d_{t_i}(u), d_{t_i}(v) + w(u, v)\}$ and if $u \notin decQ$ and $\exists_{o_i \in O}$, $d_{s_i}(u) + d_{t_i}(u) \leq \delta_i$ set $decQ = decQ \cup \{u\}$. The downward search ends when $decQ$ becomes empty.

**Theorem 4** *Upon completion of the above algorithm, we have $u \in V_\Delta$ if and only if $u \in C_O$.*

**Proof:** $u \in V_\Delta \Rightarrow u \in C_O$**:** Note that, the values of $d_{s_i}$, $d_{t_i}$ are the upper bounds (found during the search) of the true shortest path cost from $s_i$ and to $t_i$ respectively for all $o_i$. A node $u$ is included in $V_\Delta$ when $\forall_{o_i}\ d(s_i, u) + d(t_i, u) \leq d_{s_i}(u) + d_{t_i}(u) \leq \delta_i$. So by definition of $C_O$, $u \in V_\Delta \Rightarrow u \in C_O$.

$u \in C_O \Rightarrow u \in V_\Delta$**:** Consider all nodes in $K = C_1 \cap C_2 \cap \ldots \cap C_r = \{u_1, u_2, \ldots, u_x\}$ in decreasing rank order, i.e., $\phi(u_i) > \phi(u_{i+1})$. Consider the highest ranked node $u_1$ in $K$. Let $u_1$ be a corridor node for a subset of objects $o_i \in O' \subseteq O$. We argue that $u_1$ must be the highest ranked node in the path from $s_i$ to $u_1$ and $u_1$ to $t_i$ for all objects in $O'$. Otherwise, there would be a higher ranked node in $K$ according to Lemma 1 in [54].

According to [55] the highest ranked node on a shortest path $s_i - v$ or $v - t_i$ has $d_{s_i}(v) = d(s_i, v)$ and $d_{t_i}(v) = d(v, t_i)$ after the bidirectional upward search in $G_\uparrow$ and $G_\downarrow$ respectively. This implies that we have $d_{s_i}(u_1) = d(s_i, u_1)$ and $d_{t_i}(u_1) = d(u_1, t_i)$ after the first phase and $u_1 \in decQ = \{v \in V | \exists_{o_i \in O}\ d_{s_i}(v) \leq \delta_i, d_{t_i}(v) \leq \delta_i\}$. So $u_1$ will be removed first from $decQ$ and added to $V_\Delta$ if $u_1$ is in the corridor of all moving objects i.e. if $O' = O$

$(\forall_{o_i \in O}\ d_{s_i}(u_1) + d_{t_i}(u_1) = d(s_i, u_1) + d(t_i, u_1) \leq \delta_i)$.

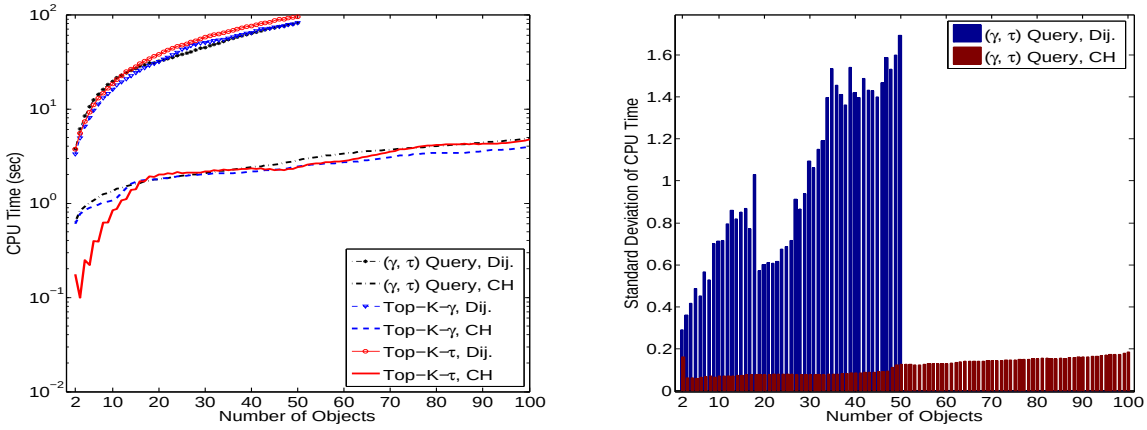
Now, consider $u_k \in K$, $k > 1$. Since we are accessing nodes of $K$ in decreasing rank order, before $u_k$ is accessed all nodes $u_j \in K$, $1 \leq j < k$ are accessed and have $d_{s_i}(u_j) = d(s_i, u_j)$ and $d_{t_i}(u_j) = d(u_j, t_i)$ for all $o_i \in O$. If $u_j$ is the highest ranked node in $s_i - u_j$ and/or $t_i - u_j$ path then the above logic applies for $u_j$ straightforwardly. Otherwise, according to the definition of weakly bitonic path [54], there is a last edge $(v_j, u_k)$, $\phi(v_j) > \phi(u_k)$ along the weakly bitonic path from $s_i$ to $u_k$. In that case, by Lemma 1 in [54], $v_j$ is accessed from *decQ* before $u_k$ and $d_{s_i}(v_j) = d(s_i, v_j)$ is correct by then. Also, when $v_j$ is accessed edge $(v_j, u_k)$ is relaxed and $d_{s_i}(u_k) = d(s_i, u_k)$ is established. Similar arguments hold for all $t_i$. Thus, when $u_k$ is accessed from *decQ* the $s_i - u_k$ and $u_k - t_i$ path costs are correct for all $o_i$ and added to $V_\Delta$ if $u_k$ is in the corridor of all objects. □

## 3.5 Experimental Evaluation

All the experiments were run in an Intel Xeon 3.0GHz quad core processor running Linux 2.6.18 with 8GB of main memory. We present the CPU time for all the experiments. We used the road netowrk of California and Nevada to perform the experiments. There are 1890815 nodes and 4630444 edges and edge weights represent travel times in the road network. We generate queries by randomly selecting a source-destination $(s, t)$ pair for each moving object. The travel time between a pair $(s_i, t_i)$ has to be at least $d_i = d(s_i, t_i)$. However, since the object may not follow the quickest path and/or spend some time at some node we increase $d_i$ to $(1 + \epsilon)d_i$. We experiment the impact of the values of the parameters mentioned in table 3.1.

| Parameter | Value | Default |
|---|---|---|
| Detour length $\epsilon$ | $0.1 - 1$ | 0.5 |
| Num. objects $r$ | 2 to 100 | 20 |
| Num. top nodes in top-$k$ query $k$ | 10 to 100 | 100 |
| Min. conclave size $\gamma$ | $r/10$ to $r$ | $r/2$ |
| Min. conclave duration $\tau$ | $\epsilon d/10$ to $\epsilon d$ | $\epsilon d/10$ |

Table 3.1: Parameters and their default values.




(a) (b)

Figure 3.3: (a) CPU Time vs Number of Objects (b) Standard Deviation of CPU Time vs Number of Objects.

We first experiment the effect of the number of moving objects $r$ being tracked on the query evaluation time for all three types of queries, figure 3.3. We vary the number of objects from 2 to 100 for the CH based methods. For Dijkstra based methods we stop the experiment after 50 objects since it does not scale. For each value of $r$, we compute the average query evaluation time by running 100 random queries (e.g. for 10 objects, we consider 100 different sets of moving objects of size 10). For these experiments we set $\epsilon = 0.5$, $\gamma = r/2$, $\tau = \epsilon d/10$. That means, at least half of the moving objects have to spend at least half of additional travel time($\epsilon d$) in a conclave. The query times are presented in log scale, figure 3.3(a). The query time for the CH based methods rises very slowly compared to the Dijkstra based method. The query times for top-$k$ queries are very similar to $(\gamma, \tau)$-query. This implies that the cost of the line-sweep algorithm and maintaining the queues are very little expensive. Note that, the standard deviation of query time for $(\gamma, \tau)$-query for Dijkstra based method is much higher than that for CH based method, figure 3.3(b). So the query time for Dijkstra based method can vary widely. For the next set of experiments we evaluate the impact of the values of the remaining parameters on $(\gamma, \tau)$-query only.

Next, we experiment the effect of $\epsilon$, which determines the additional travel time to the quickest time. The value of $\epsilon$ determines the size of the search space for our algorithms. We vary $\epsilon$ from 0.1 to 1.0 in steps of 0.1. We set the number of objects to 20, $\gamma = r/2$, $\tau = \epsilon d/10$. The average CPU time is computed by running 100 random queries for each value of $\epsilon$. The effect of $\epsilon$ on the query time for Dijkstra based method is much more than that for CH based methods, figure 3.4. The query time for CH based method rises very slowly while the query time for Dijkstra based method rises very sharply with $\epsilon$.

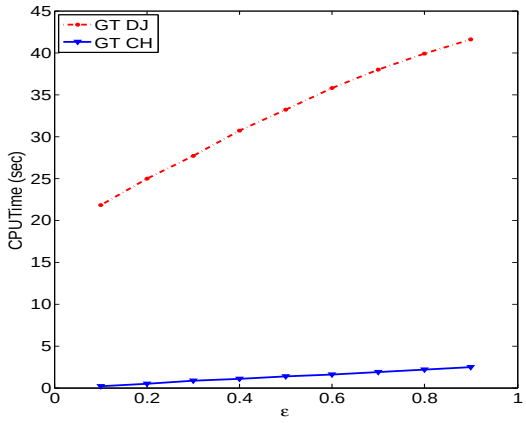



Figure 3.4: $\epsilon$ vs CPU Time.

## 3.6 Conclusion

We introduce a novel problem of identifying potential secret meetings of of moving objects when their exact location is uncertain. We present a number of variations where the query can specify meeting duration or size or seek top-$k$ nodes based on these attributes. We present a formal model for the problem and prove the correctness of our algorithm. We show how to utilize contraction hierarchy to gain orders of magnitude speed up over naïve Dijkstra based methods.

# Chapter 4

# Finding Regions of Interest from Trajectory Data

Regions of interest is a basic concept for trajectory semantics. Previous research efforts have been geared towards understanding and extracting people's activities from trajectory databases. Examples include querying for a certain sequence of activities during traveling [56], mining similarity between travelers based on their activity sequence [57], inferring popular locations from GPS traces [58], etc. Typically, these queries assume that information about the locations of specific activities is provided as a set of Regions of Interest.

There is recent work on discovering Regions of Interest (ROIs) from trajectory databases [57, 58, 59]. However, these methods are aimed at using ROIs to identify user travel similarity, top-$k$ interesting locations, etc. They identify "stay points" (equivalent to ROIs), where a stay point is an $(x, y)$ average of the points of a subtrajectory in which

the object moves less than a prespecified distance threshold $\delta$ and takes longer than a prespecified time threshold $\tau$. If either $\delta$ or $\tau$ changes, the entire trajectory database must be re-scanned. In contrast, our work removes this important limitation.

It is more intuitive to define ROIs in terms of speed. If an object takes at least time $\tau$ to travel at most distance $\delta$, it maintains an average speed no more than $\frac{\delta}{\tau}$ for at least time $\tau$. In our framework, we actually use a speed *range* to define ROIs, as this leads to a more generic definition. Further, we introduce the notion of *trajectory density* to define ROIs. A region is *dense* if the number of objects per unit area is no less than a pre-specified threshold. In summary, our ROI definition uses (1) a range of speed that an object maintains while in an ROI (2) a minimum duration of staying in an ROI area and (3) the density of objects in that area.

We build an index on object speeds to avoid scanning the whole database. Given a range or a particular speed, we first retrieve trajectory segments with that speed using this index. We then verify the minimum stay duration condition. Objects that fulfill the speed and duration condition are *candidate objects*. Finally, we identify dense regions of candidate objects. For this we extend the pointwise density method [33]. Figure 4.1 shows our proposed framework to discover ROIs.

Our contributions are summarized as follows. We provide:

- a generalized ROI definition for trajectories,
- a framework and several approaches to find ROIs efficiently, and
- an extensive experimental evaluation of the proposed methods using one synthetic and three real datasets.

The remainder of the chapter is organized as follows: Section 4.1 presents the basic definitions and formal description of the regions of interest while Section 4.2 provides an overview of related works. Section 4.3 describes the framework for storing trajectories in order to efficiently find ROIs; the proposed methods to find regions of interest are described in Section 4.4. Section 4.5 presents the experimental evaluation and Section 4.6 concludes the chapter.

## 4.1 Background

We begin with some definitions.

**Definition 5** *A **trajectory** is a finite sequence of $(x_i, y_i, t_i)$ triples. The $x_i, y_i \in \mathbb{R}^2$ are spatial coordinates, and the $t_i \in \mathbb{R}^+$, are timestamps, with $t_i < t_{i+1}$ for $i = 0, 1, \ldots, n-1$.*

Each $(x_i, y_i)$ pair represents the position recorded of a moving object at time $t_i$ (typically from a GPS enabled device). Each trajectory has a unique trajectory ID (TID).

**Definition 6** *A **trajectory segment** is a straight line between two consecutive tuples $(x_i, y_i, t_i), (x_{i+1}, y_{i+1}, t_{i+1})$ of the same trajectory, where $i \in \mathbb{N}_0$.*

**Definition 7** *A **subtrajectory** of length $m$ of a trajectory $T = (x_0, y_0, t_0), \ldots, (x_n, y_n, t_n)$, is a subsequence $T' = (x_i, y_i, t_i), \ldots, (x_{m+i-1}, y_{m+i-1}, t_{m+i-1})$, of $m$ contiguous trajectory segments, where $i \geq 0, m \leq n$.*

A single trajectory segment is a subtrajectory of length one. In the rest of the chapter we use *trajectory segment*, *line segment* or *line* interchangeably.

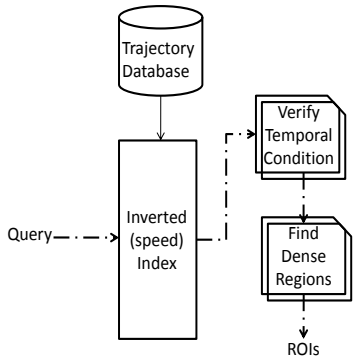



Figure 4.1: Framework for Discovering ROIs

### 4.1.1 Defining Regions of Interest

Conceptually, an ROI is intended to be a region where moving objects pause or wait in order to complete activities that are difficult or impossible to carry out while in motion. Examples of ROIs are restaurants, museums, parks, places of work, and so on. Generally, individual trajectories display idiosyncrasies, so ROIs are best defined in terms of collective behaviors of a collection of trajectories. That is, a collection of trajectories is needed to identify a location as an ROI.

One simplistic approach to define ROIs is to consider places where many trajectories intersect. However, not all intersections may be ROIs. For example, it may not be appropriate to declare a busy road intersection as an ROI. The duration of an object's stay in a location is important in filtering out spurious ROIs, so we will require a *minimum stay duration* for objects at ROIs. Nevertheless, if an object spends a long time in a large spatial region, a city, say, then that large region should not be considered as an ROI either. Hence, we must also consider the geographic extent of the object's movement, that is, the

maximum area within which an object remains (or the maximum distance traveled by an object) during the minimum stay duration.

Since the number of objects visiting a potential ROI is also important, we consider the density of candidate objects in such a region. The problem of finding ROIs can then be viewed as that of finding dense regions of candidate objects.

### 4.1.2 Identifying Point-Wise Dense Regions

We identify dense regions adapting the *point-wise dense region* approach of [33]. In this approach, a region $R \subset \mathbb{R}^2$ is dense if every point in $R$ has a neighborhood which contains a sufficient number of objects. This density approach removes various anomalies (e.g., answer loss, lack of local density guarantee, etc.) that other density computation methods, [60, 61], have. We thus adapt the following definitions from [33]:

**Definition 8** *The $l$-**square neighborhood** of a point $p \in \mathbb{R}^2$ is a square with edge length $l$ centered at $p$, including top and right edges, but excluding bottom and left edges. Figure 4.2(a) shows the $l$-square neighborhood of a point $p$. We assume $l \geq l_{\min}$, where $l_{\min}$ is predefined.*

Since we must find the density around trajectory segments, we extend this definition by defining $l$-neighborhoods around line segments and rectangles.

**Definition 9** *The $l$-**rectangle neighborhood** $r$ of a rectangle $c$ with lower-left and upper-right corners at $(x_l, y_b)$ and $(x_r, y_t)$ is the rectangle (Figure 4.2(b)) with left-bottom corner $(x_l - \frac{l}{2}, y_b - \frac{l}{2})$ and right-top corner $(x_r + \frac{l}{2}, y_t + \frac{l}{2})$. If $c$ is a square, $r$ is the $l$-square neighborhood of $c$.*

**Definition 10** *The $l$-**rectangle neighborhood of a line segment** is the rectangle with edge lengths $l$ and $l+|L|$. The edges parallel with the line $L$ are $l/2$ apart $L$ and have length $l+|L|$ while the other edges are $l/2$ apart from the end points of $L$ having length $l$.*

Figure 4.3(a) illustrates this idea. Figure 4.3(b) shows a variant of this definition that is useful in simplifying the evaluation of certain integrals while computing dense regions. Here, two rectangle boundaries are parallel to the coordinate axes.

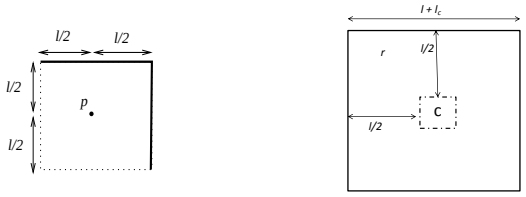
**Definition 11** *A point $p$ in a region is **dense** if at least $N$ different trajectories pass through the $l$-square neighborhood of $p$. The thresholds $l$ and $N$ are specified by the ROI query.*

Given thresholds $l, N, \tau$, we consider $R$ to be an ROI if at least $N$ objects remain for time $\tau$ in the $l$-square neighborhood of every point $p \in R$.

We note that the behavior of an object within a region of interest can be described in terms of its speed and duration of stay. If the object remains within the $l$-square neighborhood of a point for time at least $\tau$, the net speed of the object (in terms of its net displacement) can not exceed $\sqrt{2}l/\tau$ during the time interval $\tau$. Therefore, we define a region of interest in terms of speed.

**Definition 12** *A region $R$ is a **region of interest** if every point $p \in R$ has an $l$-square neighborhood containing segments from at least $N$ distinct trajectories with object speeds in the range $[s_1, s_2]$, and each such object remains in $R$ for at least time $\tau$ before leaving $R$. The parameters $l, N, \tau, s_1, s_2$ are user-defined.*

Our definition also supports timestamps, i.e. weekends or weekdays, lunch or dinner time, etc. This allows the user to distinguish between ROIs with different semantics.




(a) $l$-square neighborhood of a point p.

(b) $l$-square neighborhood of the rectangle $c$.

Figure 4.2:

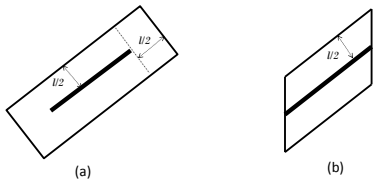



Figure 4.3: $l$-neighbor regions.

ROIs found with long stay duration on the weekends have different semantics than those found on weekdays with short stay duration.

## 4.2 Related Work

Retrieving semantic information from trajectory databases has attracted much research attention. In [56, 62] ROI information is given in a relational database and a join operation between trajectory and ROI relations is performed to evaluate activity sequence queries. [62] and [56] assume that the querying application will specify a finite set of pairs ($\Delta$, $\tau$) of interesting geographic regions $\Delta$ and durations $\tau$. If a trajectory spends at least $\tau$ duration in a specified region $\Delta$, then the portion of the trajectory inside region $\Delta$ is considered as a *stop* area in that trajectory. These stops are similar to ROIs. Nevertheless,

these approaches do not discover new ROIs as they consider only the application specified regions.

Recently, various works on discovering ROIs have appeared, [57, 58, 59, 63, 12], and are discussed below.

In [57, 58] the notion of "stay point" is presented using a maximum distance threshold $D_{threh}$ and a minimum duration threshold $T_{threh}$. In particular subtrajectories are identified that take at least $T_{threh}$ duration to travel no more than $D_{threh}$ distance. A fixed pair of values (e.g., $D_{threh} = 200$m and $T_{threh} = 30$min) is considered for finding these subtrajectories. The $(x,\ y)$ points of these subtrajectories are then averaged to identify stay points (one stay point for each subtrajectory). Note that, these stay points might not be on a trajectory (Figure 4.4). Density based clustering methods are then applied to group spatially collocated stay points. Each cluster is called a "stay region". These stay regions are then used to find similar travel sequences [57], top $n$ interesting locations [58, 59]. However, reducing a subtrajectory into a particular point (possibly not on the trajectory) leads to possible loss of information. For example, if density based methods are applied on stay points to identify stay regions they might generate false negatives. Stay points (solid circles) shown in Figure 4.4 are obtained by taking the average of points in the low speed part of each trajectory. These stay points are too far from each other to form a cluster although there is a dense region (the grey region) of slow moving objects.

[59] takes the first of the two contiguous trajectory points that are logged more than $T_{threh}$ time apart (the empty circles in Figure 4.4). These stay points are always on the trajectory but still subtrajectories are reduced to a single point and thus this approach

also suffers the above problem. In addition, [59] does not consider the situation when an object is moving slowly or stopped but the GPS is frequently recording its positions.

In our approach we find dense regions considering the whole low speed subtrajectories instead of particular points and thus overcome the above problems. We achieve this by identifying as dense points, those points which have a certain number of trajectories in their predefined neighborhood.

In [63] ROIs are discovered for each individual trajectory instead of considering all trajectories and identifying commonly interesting places. The DBSCAN method [64] is modified so that parts of a particular trajectory within a small region and with sufficient stay duration in that region will be considered as clusters (ROIs).

The above methods do not index the data and so they need to rescan the whole database for every different set of values of parameters. If these approaches wanted to index the data to retrieve subtrajectories with arbitrary combination of maximum distance traveled and minimum stay duration then they would have needed an index with all possible combinations of values of $D_{threh}$ and $T_{threh}$, which is not practical. We instead define stay points in terms of the speed of the object and index the trajectory database for speed values. With the speed index, we do not need to access the whole database for every different query: only the low speed (as specified by the query) trajectory segments are accessed.

[12] presents an approach to mine common sequence of locations, ROIs, visited with similar travel times between them. An example of such a sequence is 'Railway Station $\longrightarrow^{15min}$ Book Store $\longrightarrow^{30min}$ University', which says Railway Station to Book Store to University is a common travel sequence with travel time between them 15min and 30min

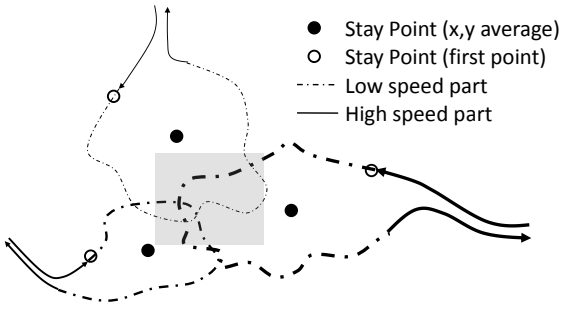



Figure 4.4: Problem of density based clustering methods for trajectories.

respectively. In this work ROI implies dense regions which are visited by a certain fraction (i.e. 10%) of all trajectories. Dense regions are identified by discretization of the space into grids, which can also introduce false negatives [33] i.e., when a dense region spans over multiple cells but none of those cells are individually dense. Moreover, considering all segments of all trajectories will identify places which are not ROIs, i.e. road intersections. Other than density, this method is different from ours because it does not identify ROIs with query specified parameter values.

To summarize, our approach has the following characteristics: 1) it does not assume any a priori knowledge about ROIs, 2) it can identify ROIs for arbitrary values of parameters without rescanning the whole database 3) it identifies commonly interesting places by also considering the number of objects that visited the place.

## 4.3 Indexing Trajectory Segments by Speed

Typically, objects in an ROI will maintain very low (or zero) speed. Hence, if we can quickly retrieve and analyze low speed trajectory segments, we can reduce query costs

**Algorithm 5** BuildIndex($\mathcal{T}$:Dataset, $\mathcal{R}$:Ranges)

1: **for** each trajectory $T \in \mathcal{T}$ **do**
2: $\rho_{prev} = -1$
3: $start = 1$
4:
5: **for** each segment $(p_i, p_{i+1} \in T)$ **do**
6: $\sigma$ =Speed$(p_i, p_{i+1})$
7: $\rho$ =Range($\sigma$, $\mathcal{R}$)
8: **if** $\rho \neq \rho_{prev}$ **then**
9: $length = i - start$
10: $ptr$ =MakePtr()
11: $e$ =indexEntry($T.ID$, $start$, $length$, $ptr$)
12: insertIntoBucket($\rho$, $e$)
13:
14: $start = i$
15: **end if**
16: $\rho_{prev} = \rho$
17: **end for**
18: **end for**

significantly.

Let $s_{\max}$ and $s_{\min}$ be the maximum and minimum speeds specifiable in an ROI query. We partition the speed values into *index ranges* $\mathcal{R} = [s_{\min}, s_1), [s_1, s_2), \ldots, [s_{n-1}, s_{\max})$. These ranges can be of arbitrary length. We maintain one bucket for each index range, with bucket $B_i$ holding trajectory segments with speed range $[s_i, s_{i+1})$.

We consider the segments for trajectory sequentially, and compute speeds assuming linear motion between two successive timestamps. If a series of consecutive segments fall within the same speed range, we combine them into one subtrajectory, and insert it into the index as one entry. Thus each entry in an index bucket points to a subtrajectory all of whose segments fall into within the speed range of the bucket. Figure 4.5 illustrates how subtrajectories are assigned to different buckets. The dotted and dashed lines show the subtrajectories which are contiguous parts of the same trajectory. The pseudo code for building the speed index appears in algorithm 5. Lines 6 and 7 calculate the speed of a segment and decide which of the index ranges contains it. If the speed range is same as that of the previous segment then we proceed to the next segment and so on. Otherwise, a new entry, $e$, is created which points to the last subtrajectory with same speed range; $e$ is then inserted into the appropriate bucket (lines 8 to 15).

We assume trajectories are sorted according to TID, so that subtrajectories in the buckets are also sorted according to TID. Having TID sorted entries in the buckets allows to perform a merge join to reconstruct trajectories from these buckets. When new trajectories are added to the database the index can easily be updated using the above algorithm.

Finding trajectory segments having speed within range $[s_1, s_2)$ is straightforward.

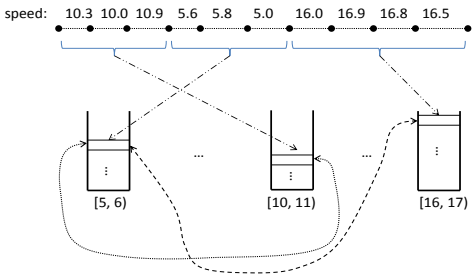



Figure 4.5: Index Structure.

Every bucket whose speed range overlaps with the range $[s_1, s_2)$ is accessed. If the speed range of a bucket $B_i$ is completely contained within the query speed range then all subtrajectories of $B_i$ are considered. If there is partial overlap, the subtrajectories of $B_i$ are checked for containment of speed within the query range.

## 4.4 Finding Regions of Interest

We find ROIs in three steps. First, we retrieve the appropriate buckets from the index. In the second step, we collect subtrajectories spanning multiple buckets by performing a merge-join, and check the stay durations. In the third step, we find regions with line segment density $N/l^2$, where each of $N$ segments has to be from different trajectories.

It is straightforward to retrieve the segments falling into a given speed range $[s_1, s_2)$ using the speed index. No further discussion is needed.

### 4.4.1 Step 2: Verifying the Duration Condition

In this step, we consider only the buckets obtained from the previous step. To verify the duration condition for each trajectory we must join subtrajectories with same

TID from different buckets. Let the query speed range include buckets $B_i$ and $B_j$, and let $S_i \in B_i$ and $S_j \in B_j$ be subtrajectories. Let the start and end timestamps for $S_i$ and $S_j$ be $[t_{i1}, t_{i2}]$ and $[t_{j1}, t_{j2}]$ respectively. If $S_i$ and $S_j$ have the same TID and $t_{i2} = t_{j1}$ or $t_{i1} = t_{j2}$, then $S_i$ and $S_j$ should be merged into a single subtrajectory. The object's stay duration is the interval between the first and the last timestamps of the merged subtrajectory. We discard all subtrajectories with stay duration less than $\tau$ after merge, since they do not fulfil the stay duration condition. Since we have a TID-sorted list in each bucket we need one pass over every bucket entry. The segments that belong to a subtrajectory with stay duration $\tau$ or more are input to the next step.

In addition to minimum stay duration, our implementation also supports other temporal conditions, such as time intervals and weekdays/weekends. For example, ROIs during any weekday with $\tau = 15$ to 30 minutes, carry different semantics than those found in the afternoon or evening of any weekend, with a few hours of stay duration.

### 4.4.2 Step 3: Finding Dense Regions

This step involves finding points $p$ whose $l^2$-neighborhood contains at least $N$ distinct trajectories. For our purpose we extend the Pointwise Dense Region (PDR) method [33] which was originally presented for point objects. We extend those techniques here for line segments. The work in [33] describes two variations: (1) an exact, and (2) an approximate method.

**Exact PDR Method**

The spatial region is assumed to be a $L \times L$ square area. This space is partitioned into $m \times m$ grid, with cell width $l_c = \frac{L}{m}$. Here, $m$ must be such that $l_c \leq \frac{l_{min}}{2}$ where $l_{min} \leq l$. For each cell, $c_{i,j}$ where $1 \leq i, j \leq m$, we maintain a histogram. Initially all histogram values are set to zero. The histogram value for each cell is increased by one, for each distinct overlapping trajectory. We index the trajectory segments obtained from the previous step using an R*-tree [65]. The cost for building such an R*-tree is included in the query cost, which is, according to our experimental evaluations, quite small. For each cell we perform an R*-tree search to determine the number of overlapping trajectories. Histogram values for all the cells form an $m \times m$ matrix, which we call a *histogram matrix*.

PDR [33] is designed for moving objects, and evaluates predictive queries about dense regions at a future timestamp. Hence, the numbers and positions of objects could be different for different queries. As a result, histogram values must be computed for every different query. However, we evaluate the query on historical data (GPS traces), and can calculate histogram values once, and use to evaluate queries. That is, to speed up queries, we pre-compute a histogram matrix for every index range and then use these to answer queries with any arbitrary speed range. As new trajectories are added to the database, the histogram values and the speed index are both updated. We assume that the data is always up-to-date.

Calculating the histogram matrix for a query speed range requires adding histogram matrices whose speed ranges overlap with the query range. If the upper and lower limits of the query range exactly match the limits of index ranges then no histogram values

are calculated. Otherwise we have to calculate one or two histogram matrices for a very small size of data. The following example illustrates the idea.

Let the query range be $[s_{q1}, s_{q2})$, and the index ranges be $[s_0, s_1)$, ..., $[s_i, s_{i+1})$, ..., $[s_j, s_{j+1})$,..., $[s_{n-1}, s_n)$. If $s_{q1} = s_i$ and $s_{q2} = s_j$ where $0 \leq i \leq n-1$, $1 \leq j \leq n$, and $i < j$, then we add up the histograms for the ranges $[s_i, s_{i+1}), \ldots, [s_{j-1}, s_j)$. However, if $s_{i-1} < s_{q1} < s_i$, $s_j < s_{q2} < s_{j+1}$, where $1 \leq i \leq n-2$, $2 \leq j \leq n-1$, and $i < j$, then we have to calculate histogram values for the ranges $[s_{q1}, s_i)$ and $[s_j, s_{q2})$ in addition to adding up histogram matrices for the speed ranges $[s_i, s_{i+1})$, ..., $[s_{j-1}, s_j)$.

**The Filtering Step** Let $\eta_l = \lfloor \frac{l}{2l_c} \rfloor$, $\eta_h = \lceil \frac{l}{2l_c} \rceil$. We use the following definitions from [33]:

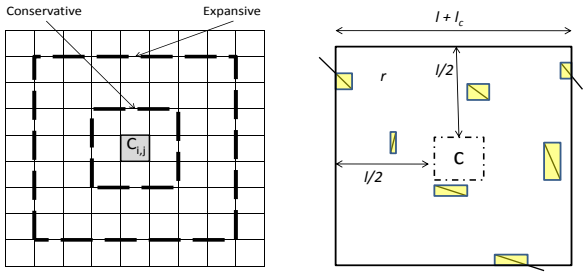
**Definition 13** *The **conservative neighborhood** of a cell $c_{i,j}$ is the union of grid cells $c_{u,v}$ such that $i - \eta_l < u < i + \eta_l$ and $j - \eta_l < v < j + \eta_l$.*

The $l$-square neighborhood of any point inside $c_{i,j}$ fully contains its conservative neighborhood, $C_{i,j}$.

**Definition 14** *The **expansive neighborhood** of a cell $c_{i,j}$ is the union of grid cells $c_{u,v}$ such that $i - \eta_h \leq u \leq i + \eta_h$ and $j - \eta_h \leq v \leq j + \eta_h$.*

The $l$-square neighborhood of any point inside $c_{i,j}$ is fully contained in its expansive neighborhood, $E_{i,j}$. Figure 4.6(a) shows the conservative and expansive neighborhood for a cell $c_{i,j}$, with $\eta_l = 2$ and $\eta_h = 3$.

If the conservative neighborhood of any cell overlaps with $N$ trajectories, then all points inside cell $c_{i,j}$ are guaranteed to be dense, and $c_{i,j}$ is accepted as a dense cell.




(a) Conservative and Expansive neighborhood of a cell $c_{i,j}$.

(b) MBRs in the $l$-square neighborhood of the cell $c$.

Figure 4.6:

On the other hand, if the expansive neighborhood of any cell overlaps less than $N$ line segments, then no point in cell $c_{i,j}$ is dense, and $c_{i,j}$ is rejected. Cells that are neither accepted nor rejected are candidate cells. We use the *FilterQuery* algorithm from [33] to identify accepted, rejected and candidate cells. In our case $\rho l^2 = N$ and there is no query timestamp, $q_t$. Candidate cells are further analyzed to identify dense points inside them.

**Refinement Step** To identify dense points in a candidate cell $c_{i,j}$ first, we will find all segments that overlap with the $l$-square neighborhood, $r$, of the cell $c_{i,j}$.

We do an R*-tree search to retrieve segments overlapping with $r$. We retrieve only the portion of a line that overlaps with $r$. Each segment is represented by its MBR. Figure 4.6(b) shows the region $r$ and the MBRs of the overlapping line segments. We sort the MBRs of line segments according to the $x$ coordinate of their left-bottom corner.

In the refinement step, an *$l$-band* is swept along the $X$ axis for each candidate cell $c_{i,j}$. An $l$-band is a rectangle with width $l$ and height $l + y_t - y_b$. The position of the $l$-band is identified by the position of its vertical median. The plane sweep algorithm along the

$X$-axis starts with the $l$-band's vertical median at the left edge of $c_{i,j}$ and stops when it touches the right edge of $c_{i,j}$. Let $L$ be the sorted list of MBRs that overlap the *l-band* at the beginning. As the $l$-band is swept, when the right edge of the *l-band* touches the left edge of an MBR we insert it into the list $L$. When the left edge of the *l-band* touches the right edge of an MBR we delete it from $L$. The $x$ coordinates of the vertical center line when any edge of $l$-band touches any edge of an MBR are the stopping points. Instead of sweeping through all the $x$ coordinates it is sufficient to consider only the stopping points.

Algorithm 6 describes the plane sweep along the $X$-axis. In this algorithm the $l$-band is placed at each stopping point and the left and right edges of the candidate cell . If the $l$-band overlaps at least $N$ objects then SweepY is called to identify dense regions, where $l$-square neighborhood of each point overlaps $N$ lines. Plane sweep along $Y$ axis proceeds in the same way. An $l$-square is swept instead of an $l$-band.

**Approximate PDR Method**

The exact PDR method requires to run the plane sweep algorithm for all candidate cells, which can be a costly operation. The number of plane sweeps required depends on the number of candidate cells. If most of the cells cannot be accepted or rejected during the filtering step then a large number of plane sweeps is needed. This will significantly increase the query execution time. However in practice the querying application or the user can accept some loss of accuracy and identifying dense regions exactly is not necessary.

The work in [33] therefore presents an alternate method using Chebyshev polynomials of the first kind to approximate the density function $D(x, y)$ of the two dimensional space. $l$ is assumed to be fixed. We adapt this method we adapt to our case. Our experi-

**Algorithm 6** RefineQuery($N$, $c_{i,j}$)

1: $S$ = MBRs in $l$-square neighborhood of $c_{i,j}$

2: sort $S$ according to the left x-coordinate, $x'_l$, of MBRs

3: stopX = $x_l$, $x_r$

4: **for** each MBR($x'_l$, $y'_b$, $x'_r$, $y'_t$) in $S$ **do**

5: insert $x'_l - \frac{l}{2}$ if $x'_l - \frac{l}{2} \in [x_l, x_r]$

6: insert $x'_r - \frac{l}{2}$ if $x'_r - \frac{l}{2} \in [x_l, x_r]$

7: insert $x'_l + \frac{l}{2}$ if $x'_l + \frac{l}{2} \in [x_l, x_r]$

8: insert $x'_r + \frac{l}{2}$ if $x'_r + \frac{l}{2} \in [x_l, x_r]$

9: **end for**

10: $L$ = MBRs inside $l$-band at the initial position

11: $n$ = #elements in $L$.

12: **for** $i = 1$ to $n$ **do**

13: $x_i = L[i]$

14: delete MBR whose $x'_r$ is $x_i - \frac{l}{2}$ from $L$

15: insert MBR whose $x'_l$ is $x_i + \frac{l}{2}$ into $L$

16: **if** $|L| \geq N$ **then**

17: SweepY($L$, $N$)

18: **for** each dense segment $[y_j, y_{j+1})$ **do**

19: $[x_i, x_{i+1}) \times [y_j, y_{j+1})$ is a dense region.

20: **end for**

21: **end if**

22: **end for**

mental results mirror that of [33], and show that the approximation method is very fast, so that approximations for different $l$ can be computed on the fly.

**Chebyshev Polynomials** The Chebyshev polynomial $T_k(x)$ of the first kind is a polynomial in $x$ of degree $k$ and defined by the relationship $T_k(\cos\theta) = \cos(k\theta)$. When $x \in [-1, 1]$, then $\theta \in [0, \pi]$. These polynomials obey the recurrence

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x),\ k \geq 1$$

$$T_0(x) = 1,\ T_1(x) = x.$$

The approximation $\hat{f}(x, y)$ of a function $f(x, y)$ with Chebyshev polynomials of degree $k$ is given by:

$$\hat{f}(x, y) = \sum_{i=0, j=0}^{i+j \leq k} a_{i,j} T_i(x) T_j(y)$$

The Chebyshev coefficients $a_{i,j}$ are computed using the following formula:

$$a_{i,j} = \frac{c}{\pi^2} \int_{-1}^{1} \int_{-1}^{1} \frac{f(x, y) T_i(x) T_j(y)}{\sqrt{1 - x^2}\sqrt{1 - y^2}} dx dy$$

where,

$$c = \begin{cases} 4 \text{ when } i \neq 0, j \neq 0 \\ 2 \text{ when } i = 0, j \neq 0 \text{ or } i \neq 0, j = 0 \\ 1 \text{ when } i = 0, j = 0 \end{cases}$$

The $x$ and $y$ coordinates are normalized to between $+1$ and $-1$, with the bottom-left corner at $(-1, -1)$ and top-right corner at $(1, 1)$. In [33], Chebyshev coefficients are updated for the $l$-square neighborhood of each point as objects are added, so that that the density of the $l$-square neighborhood of a point is increased by $1/l^2$ for this point.

We will use a similar approach, considering the line segments obtained from step 2 one by one, and updating the coefficients. We first describe how to update coefficients for a $l$-square neighborhood of a point and then focus on updating coefficients for a line segment.

| Time of collection | Number of trajectories | Number of spatial points | Sampling frequency | Description |
|---|---|---|---|---|
| Apr 2007 to Aug 2009 | 165 | 24778552 | 2-5 sec | **GeoLife Data** |
| 2008-05-17 to 2008-06-10 | 536 | 11219955 | 10 sec | **TaxiCab Data** |
| April to August of 1993-1996 | 253 | >287,000 | 1hr | **DeerElk Data** |

Table 4.1: Description of real data set.

For each point $p$ we need to update each coefficient $a_{i,j}$, for $i = 0, j = 0$ to $i+j \leq k$, so that the density of $l$-square neighborhood of $p$ is increased by $1/l^2$. It has been shown in [33] that for each point if we can calculate the increment $a^{\delta}_{i,j}$ of coefficient $a_{i,j}$, then the updated coefficient $a'_{i,j}$ is:

$$a'_{i,j} = a_{i,j} + a^{\delta}_{i,j}.$$

$a^{\delta}_{i,j}$ is calculated as follows.

$$\begin{aligned} a^{\delta}_{i,j} &= \frac{c}{\pi^2} \int_{x_l}^{x_r} \int_{y_b}^{y_t} \frac{\frac{1}{l^2} T_i(x) T_j(y)}{\sqrt{1-x^2}\sqrt{1-y^2}} dx dy \\ &= \frac{c}{\pi^2 l^2} \int_{x_l}^{x_r} \frac{T_i(x)}{\sqrt{1-x^2}} dx \int_{y_b}^{y_t} \frac{T_j(y)}{\sqrt{1-y^2}} dy \\ &= \frac{c}{\pi^2 l^2} \int_{x_l}^{x_r} \frac{\cos(i \arccos(x))}{\sqrt{1-x^2}} dx \int_{y_b}^{y_t} \frac{\cos(j \arccos(y))}{\sqrt{1-y^2}} dy \end{aligned}$$

[using the trigonometric representation of $T_i(x)$] (4.1)

$y_b, y_t, x_l, x_r$, are bottom, top, left, and right boundaries respectively of the $l$-square.

However, computing the above integrals for the $l$-rectangle region of a line segment

is complicated, since the boundaries of a line segment's $l$-rectangle are not fixed values. This causes the $y$-limits to become functions of $x$. We will simplify the integrals by assuming that two boundaries of the $l$-rectangle neighborhood are parallel to $y$ axis, as in Figure 4.3(b). Now $x$-limits are fixed values and $y$-limits are linear functions $f_1(x, x_l, x_r)$ and $f_2(x, x_l, x_r)$. The integrals now assume the form

$$\begin{aligned} a^{\delta}_{i,j} &= \frac{c}{\pi^2 l^2} \int_{x_r}^{x_l} \frac{\cos(i \arccos(x))}{\sqrt{1-x^2}} dx \\ &\quad \times \int_{y_b=f_1(x,x_l,x_r)}^{y_t=f_2(x,x_l,x_r)} \frac{\cos(j \arccos(y))}{\sqrt{1-y^2}} dy \\ &= \frac{c}{\pi^2 l^2} \int_{x_r}^{x_l} \frac{\cos(i \arccos(x))}{\sqrt{1-x^2}} \\ &\quad \times \frac{\sin(j f_1(x, x_l, x_r)) - \sin(j f_2(x, x_l, x_r))}{j} dx \\ &= \frac{c}{\pi^2 l^2 j} [I_1 - I_2] \end{aligned} \tag{4.2}$$

where

$$I_k = \int \frac{\cos(i \arccos(x)) \sin(j f_k(x, x_l, x_r))}{\sqrt{1-x^2}}, \ k = 1, 2 \tag{4.3}$$

Unfortunately, using linear $f_1()$ and $f_2()$ in (4.2) results in an elliptical integral, which has no closed form. Numerical integration is expensive, and will not allow us to achieve our goal of quickly approximating the density function.

Since updating the Chebyshev coefficients for the $l$-rectangle neighborhood of a line segment will not be efficient, we proceed using a simpler region around line segments. Our final goal is to approximate the dense regions quickly. We take the middle point $p_m$ of each segment, and update the coefficient for the $l$-square neighborhood of $p_m$. This makes the approximation method simple and quick, although it might harm the goodness of the

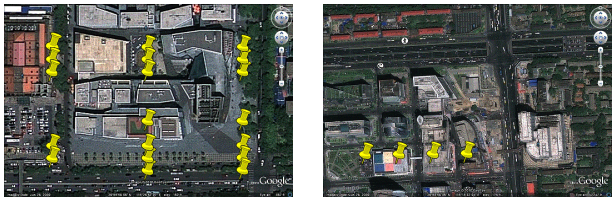
approximation. Usually trajectory segments are much smaller than grid cells, and fully contained within a cell. If a line segment is bigger than grid cell width, $l_c$, then we segment the line into multiple lines of length $l_c$, except the last segment, which might be smaller.

## 4.5 Experimental Evaluation

In our experiments we used three real and one synthetic datasets. All the experiments were run in an Intel Xeon 3.0GHz processor running Linux 2.6.18 with 8GB of main memory. We used the disk manager and R*-tree implementation of the spatial index library [66] with page size 16KB.

Table 4.1 provides the description of the real datasets. The GeoLife dataset [41] contains public activity data (i.e. shopping, dining, sightseeing, hiking, cycling etc.) in Beijing, China. The TaxiCab dataset was collected from GPS equipped taxi cabs in San Francisco, USA [67]. The DeerElk data contains the trajectories of deer, elk and cattle in the Starkey Experimental Forest and Range in Oregon, USA [42]. The synthetic dataset contains two hundred thousand trajectories, each of length 250 recordings, generated for the Chicago metropolitan area road network.

We first consider identifying ROIs in the real datasets. For experiments we used



(a) (b)

Figure 4.7: ROIs identified Beijing with long stay duration in weekends.

the exact PDR method. Table 4.2 shows the temporal conditions used for the experiments on the GeoLife data. The results of our algorithms were then validated using Google Maps. Note that GeoLife data comes from Microsoft Asia employees, visitors, etc. Using a short stay duration (15 to 30 min) we found bus stops, railway and subway stations, the Tsinghua University canteen, etc. We then considered weekends and a longer stay duration (1.5 to 4 hr). This resulted in ROIs in (1) the Sanlitun area which houses many malls, bars and is a very popular place, (2) the Wenhua square which contains churches, theaters, and other entertainment places, and (3) Zhongguancun, referred to as 'China's Silicon Valley', having a lot of IT and electronics markets. Figure 4.7(a) and (b) shows Sanlitun and Zhongguancun area respectively in Beijing. When considered lunch and dinner time we found places that contain many restaurants. Interestingly ROIs found at lunch time contain regions near the Microsoft China head quarters which are absent in dinner time ROIs. Finally we identified ROIs on each individual day from April 2007 to August 2009. These resulted in (1) the Olympic media village, the Olympic sports center stadium during the Olympics 2008, (2) Peking University when the 'Regional Windows Core Workshop 2009 - Microsoft Research' was taking place in the PKU campus, (3) areas near the Great Wall in a weekend, (4) the Beijing botanical gardens, (5) the Celebrity International Grand Hotel, Beijing, etc.

Figure 4.8(a) shows all the ROIs found using the TaxiCab dataset. We further zoom in to ROIs and these are shown in figure 4.8(b) The San Francisco international airport, (c) a car rental, (d)the main downtown, union square (e) hotels: Star Wood, Westin, Mariott, (f) hotel Radisson (g) Ramada Plaza hotel (h) Embarcadero, Regency hotel, (i) San Francisco Caltrain station (j) the yellow cab access road. These were found

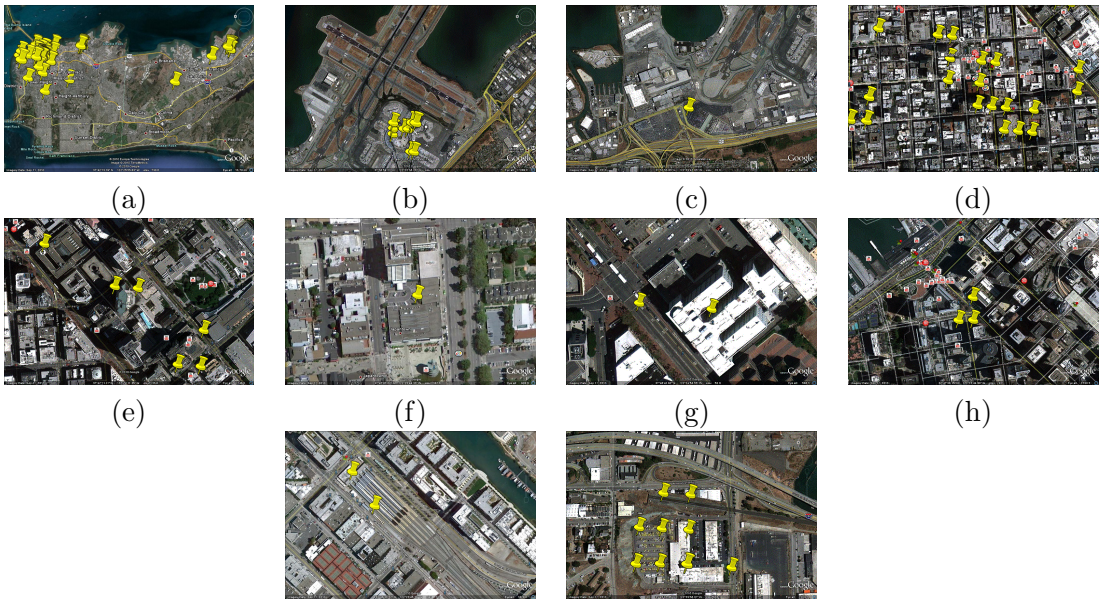



Figure 4.8: ROIs identified for the TaxiCab data.

for short stay duration of 10 minutes. When the stay duration was increased to 12 hours we found only yellow cab access road, while for $2-3$ hours of stay duration we also found the airport.

| | Time Period | Duration |
|---|---|---|
| 1 | Any day | 15-30 min |
| 2 | Weekends | 1.5-4 hr |
| 3 | Lunch time | 0.5-1.5 hr |
| 4 | Dinner time | 0.5-1.5 hr |
| 5 | Any day | 1-4 hr |

Table 4.2: Temporal attributes for the GeoLife data Experiments.

Finally, when using the DeerElk dataset, the ROIs found tend to be near a valley, with the largest ROI being close to a big water body (0.56 miles in length).

Figure 4.9 shows the query evaluation time for different values of parameters on

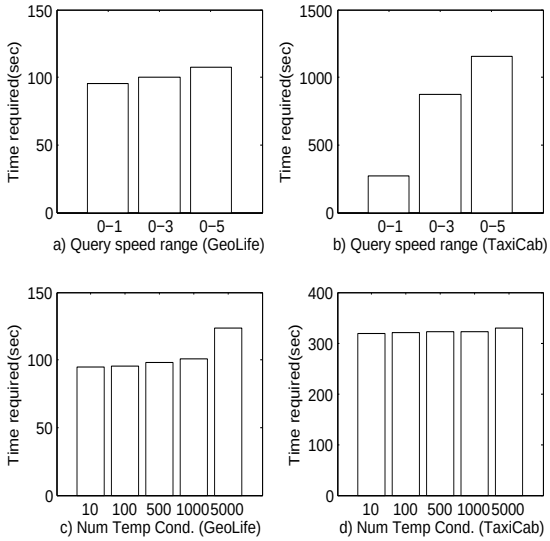



Figure 4.9: Query parameters vs Time.

real data. We do not consider the DeerElk data because there is not much variation of speed in this dataset. As we increase the speed range the query evaluation time increases very slowly for the GeoLife data. However, for the TaxiCab data the evaluation time increases sharply (the reason behind which is explained in the next paragraph). On the other hand, the effect of the number of the temporal conditions on query evaluation time is very small. This is because evaluating the temporal condition requires a sort merge join which is very fast. While varying the number of temporal conditions the amount of selected data from the index was kept the same, which ensures that the subsequent parts of the algorithm after verifying the temporal condition processed the same data.

Figure 4.10 shows the percentage increase in the selected data from the speed index as the query speed range increases from $0 - 1$mph to $0 - 3$mph and from $0 - 3$mph to $0 - 5$mph. Increasing the query speed range results in much higher increase of selected

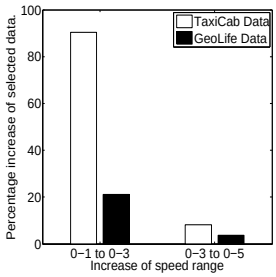
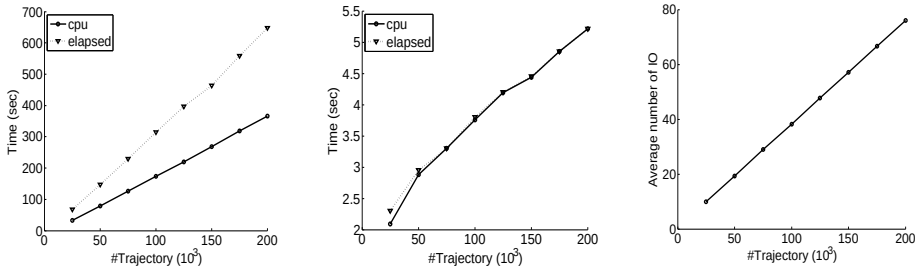



Figure 4.10: Increase in fraction of selected data vs speed range.

data in case of the TaxiCab data than that in the GeoLife data. This explains why the performance of query evaluation in the TaxiCab data is more affected by the query speed range than that in the GeoLife Data.

We also ran experiments on the synthetic dataset to determine the algorithm's behavior over large datasets. Figure 4.11(a) shows the time required to build the speed index. Here we report index construction times for indexing subtrajectories with speeds between 0 and 100, (although we need only low speed segments, i.e. 0 to 5, to answer typical ROI queries). Our index ranges are $[0, 1), [1, 2), \ldots, [99, 100)$. The cost of building the index is linear with the data size. The difference between CPU time and elapsed time is




(a) (b) (c)

Figure 4.11: (a) Index building cost vs Database size. (b) Histogram computation time vs Database size. (c) Histogram computation IO vs Database size.

slowly increasing which is due to the increasing number of disk IO as the data size increases.

Figure 4.11(b) shows the time required for the histogram computation for each index range as a preprocessing step. Computing histogram matrices requires accessing the speed index and building an R*-tree with the selected data. Note that the elapsed time is very close to CPU time. This is because the histogram matrices are typically small and can fit into main memory. Moreover, the data indexed by the R*-tree is also small. Figure 4.11(c) shows the average number of IOs required for computing each histogram matrix.

For comparison purposes we also implemented the CBSMOT approach [63]. CB-SMOT estimates the *eps* parameter using a quantile function where the user has to specify the fraction of trajectory points that is expected to be in an ROI. Since in our approach we assume that the user will specify speed and distance, we can equivalently assume that the *eps*-distance is specified by the user. To compare CBSMOT with our method we ran it for a certain value of *eps* and minimum stay duration $\tau$. Then we ran our method with speed range $[0, \frac{eps}{\tau})$ and minimum stay duration $\tau$. Note that, CBSMOT finds ROIs for each individual trajectory but our method considers all trajectories of the dataset and identifies regions that are commonly interesting. Thus, CBSMOT identifies more ROIs than those identified by our method. To make the results of CBSMOT comparable with ours we extend it (E-CBSMOT) by applying the pointwise density method on its output. Note that, E-CBSMOT, similarly to CBSMOT, cannot evaluate temporal conditions as our approach.

Figure 4.12 shows the regions of interest found in the synthetic data using the exact, approximate and E-CBSMOT methods. ROIs identified by the Chebyshev approximation have more area than those by the exact method. This is due to the fact that

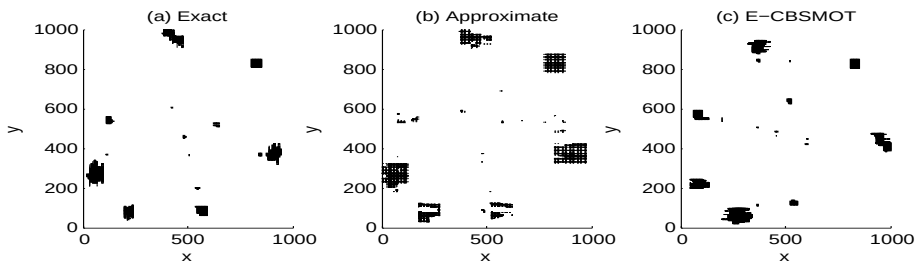



Figure 4.12: ROIs identified by different methods.

the density coefficients are being over estimated because of considering the $l$-square neighborhood around the center of a line segment. The subtle difference between the result of E-CBSMOT and our method is because of the different definitions of ROIs.

Figure 4.13 shows the query evaluation time. As expected, the time for the approximation method is quite less than that of the exact method. Recall that the approximate method computes Chebyshev coefficients and finds dense regions for every different value of $l$. From the experimental results we argue that the whole approximation process is so fast that it is feasible to run the approximation for every different value of $l$. For the exact PDR method we experimented with two cases, when we (1) only add precomputed histogram matrices and (2) need to compute the histogram values after retrieving required subtra-

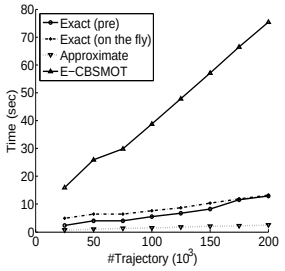



Figure 4.13: Query time vs Database size.

jectories from the speed index. In the later case we need to (i) access the speed index to retrieve the required trajectory segments (ii) build the R-Tree on them and (iii) compute the histogram values. This experiment shows the benefit of precomputing the histogram matrices and thus avoid accessing the speed index. Finally, E-CBSMOT takes much longer than our method since it has to scan the whole dataset.

## 4.6 Conclusions

In this chapter we address the problem of discovering regions of interest from trajectory databases. We give formal definition of ROIs in a more generic way than previous approaches and propose a framework to discover ROIs efficiently. We allow users to specify any arbitrary values for attributes defining ROI. Unlike previous approaches we do not scan the whole database to identify ROIs for a certain set of attribute values defining ROIs, neither assume any spatial information given about these regions.

We also consider a minimum number of objects must stay in an ROI for a minimum duration, which is absent in previous methods. We extend the Pointwise Density method to identify these regions with a minimum density of trajectories. Experimental results show that our proposed methods discover ROIs efficiently and correctly. As a future work we want to address the issue of dealing with data with uncertainty e.g., noisy, low resolution data.

# Chapter 5

# Indexing Moving Object Trajectories With Hilbert Curves

The availability of mobile technologies (ubiquitous cellular networks) combined with highly accurate GPS devices has enabled many applications that generate and maintain data in the form of trajectories. As an example, location-based services have now become common, creating large trajectory repositories. Given the ever increasing size of such repositories, to efficiently access and analyze trajectory data, we need fast indexing methods.

For simplicity, we assume trajectories are created by objects moving in a 2-dimensional space. For our purposes, a trajectory consists of a unique id (corresponding to the moving object that created it) and a poly-line whose endpoints are tuples of the form $(x, y, t)$. Here $(x, y)$ is the spatial coordinate where the moving object was at time $t$. Typically, such tuples come from sensor readings, GPS updates, social network check-ins,

etc. In between consecutive tuples $(x_1, y_1, t_1)$ and $(x_2, y_2, t_2)$ where $t_2 - t_1 < \delta$, we assume that the object followed a straight line. If the time duration between two consecutive tuples from the same object is larger than the threshold $\delta$ we assume that this object starts a new trajectory (or trip). This allows us to create meaningful trajectories per application (using different $\delta$ thresholds). Thus, the same object can have multiple trajectories representing its different trips over time (but non-overlapping in the time domain). By considering an object's movement as a sequence of line segments (instead of simply a collection of endpoints) we can answer queries about the moving object's position in between these endpoints as well.

Space filling curves (Hilbert Curve, Z Curve) have been shown to be advantageous in indexing spatial objects since they can reduce the space dimensionality. In past research, SFCs have been mainly used to index multidimensional points [68, 69, 70] or order arbitrary objects [71, 72]. Orenstein [73, 74] provided a general approach by which any spatial object can be represented as a collection of ranges (where a range starts when the SFC enters the spatial object's area and ends when the SFC exits it). Each such range corresponds to a continuous part of the SFC that overlaps the spatial object. A spatial range query is also translated into a collection of range queries in the SFC space.

When considering a SFC, an important property is how well it performs clustering or preserves proximity (i.e. points that are nearby in the original space are also close to each other in the transformed space). This is important as it reduces the number of disk IO for hierarchical indexing methods. Another important property is the average number of SFC ranges per object. Higher proximity preservation and lower number of ranges result in

better performance. Among all the SFCs, the Hilbert curve has been shown to have better proximity preservation [75, 76, 77], and lower average number of ranges per object [78].

In this chapter, we utilize Hilbert curves to index the trajectory poly-lines. Using a SFC to represent trajectory line segments sets forth a number of *challenges*. First, a line segment does not have an area. Hence, a Hilbert curve either crosses a line segment (i.e. the SFC enters and exits the line segment at the same point, resulting into a degenerate range that contains a single point) or visits very few continuous points on a line segment (since the SFC by construction is not a straight line). As a result, Orenstein's method will transform a line segment mainly to its points (degenerate ranges) and few (small) ranges, resulting in (i) increased data storage, and, (ii) very high query evaluation time.

Since the endpoints of a line segment are important (as they define the limits of the segment), another approach would be to represent the line segment by the Hilbert range of its endpoints. This is intuitive and correct in the Euclidean space but it does not work when using a SFC. Consider the Hilbert numbers $(h_1, h_2)$ assigned to the endpoints $(e_1, e_2)$ of a line segment. The part of the Hilbert curve between $h_1$ and $h_2$ may not go through all the points of the euclidean line $(e_1, e_2)$. As a result, we can have a range query that overlaps $(e_1, e_2)$ but the query's Hilbert transformation may not overlap the $(h_1, h_2)$ range, thus introducing false negatives.

The above approaches can be thought as two extremes. Instead, our proposed approach introduces splits on the original line segments in an effective way that (i) avoids the degenerate ranges of the Orenstein approach and (ii) maintains the segment endpoints while avoiding the false negatives.

When implementing our approach, the next important decision is what space to fill (represent) with the Hilbert curve. One approach is to use the SFC over the full 3-dimensional spatiotemporal space (two spatial coordinates plus time dimension). However, the temporal dimension is semantically different than the spatial ones. One such difference is that time always increases. Typically, the space filled by a SFC has fixed size, a requirement that is easily satisfied for the spatial domain but not the temporal dimension (unless one splits the temporal dimension into fixed intervals and introduce a distinct SFC for each time-space cube; however this approach will introduce unnecessary overhead at query time). Furthermore, the inclusion of time may force closely located positions in the spatial domain to be far away on the Hilbert space. Consider for example a slowly moving object over a long time interval. The trajectory of this object contains locations that are spatially close but if seen in the 3-dimensional space they are not. Hence, a 3-d SFC may assign to such a trajectory ranges that are far away, thus eliminating the desired spatial locality. To this end, we use a SFC over the 2-dimensional spatial domain.

As a result, a trajectory segment (whether original or introduced from a split) with endpoints $(x_1, y_1, t_1), (x_2, y_2, t_2)$ is represented by the MBR $(h_1, t_1), (h_2, t_2)$ where $h_i$ is the Hilbert number allocated to spatial position $(x_i, y_i)$. Such MBRs are then indexed using a $2D$ R-tree. We call this approach the Hilbert Trajectory Index or *HT-Index*.

Most of the previous trajectory indexing approaches are based on R-trees [40, 79, 10] or cell-based partitions [80]. The R-tree based methods (e.g., STR tree, TB tree [79], TPR tree [10] etc.) use minimum bounding rectangles (MBRs) to store trajectory segments. MBRs are known to introduce large dead space when storing 2-dimensional lines which can

generate a lot of false positives for range queries. The problem becomes more significant when adding the time dimension (the dead space increases exponentially). Among the space partitioning approaches, the most efficient has been the TrajStore [80]. It adaptively splits the 2-dimensional space in cells based on the density of trajectories passing within a cell (as time proceeds) and facilitates a quad tree to index them. We compare the HT-Index against both the R-tree and TrajStore approaches; our experiments show that for range queries, the *HT-Index* depicts $2-15$ times speedup over a $3D$ R-tree and $2-3$ times speedup against the TrajStore.

We also considered supporting the temporal dimension by using a temporal access method. We describe a new indexing method called 'Multiversion Interval Tree' (MI-tree). MI-tree combines idea from Interval tree[81] and Multiversion B-tree[82]. Ranges of Hilbert numbers can be indexed with interval tree. To manage the temporal intervals we propose Multiversion Interval Tree which can be implemented with two MVB-Tree. However, this method does not outperform HT-index.

While the above discussion focuses on spatial range queries, another challenge arises from the support of distance-based queries ($k$NN, skyline, etc.) In general, a transformation needs to preserve Euclidean distance so as to avoid false dismissals in the evaluation of distance based queries. Unfortunately, unlike many *orthogonal transformations* [83], the transformation from the 2-d spatial space to SFC ranges does not lower bound the Euclidean distance. Because of this, it has been assumed to be impossible [84] to evaluate distance based queries using SFC transformations, without first converting the SFC ranges back to spatial coordinates. This adds significant overhead to every distance computation during

query evaluation, making the SFC approach impractical. Instead, our proposed method avoids this overhead by calculating a lower bound on the Euclidean distance by inspecting a few bits of the SFC numbers of the segment endpoints. As a result, we can evaluate distance based queries with minimal overhead. Our experimental evaluation showed that for $k$NN queries, the existing methods can be made two times faster using the HT-Index.

Contributions of this chapter are:

- We mitigate the problems of representing line segments using Hilbert curve. We keep the temporal dimension separate from spatial dimensions because of their inherent semantic difference and index spatio-temporal trajectories with a 2D R-Tree. This results in a large performance improvement for range queries and is usable in any available database system with R-Tree implementation.
- We show how to evaluate distance based queries using existing branch and bound algorithms on HT-Index without fully converting Hilbert numbers to Euclidean coordinates for every distance computation.
- We perform extensive experiments and show our proposed method outperforms any state of the art methods.

We also show how to use existing R-Tree cost model [85] to determine grid cell size.

The rest of the chapter is organized as follows: Section 5.2 provides the definitions and background while Section 5.1 describes some related works. R-Tree based methods are described in Section 5.3 while Section 5.4 describes index structures that consider as alter-

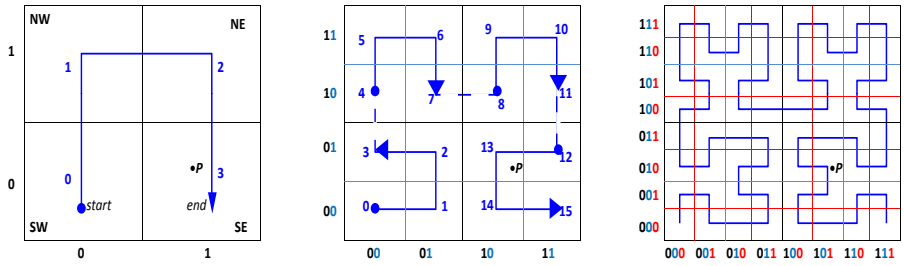



Figure 5.1: Hilbert curves: (a) $1^{st}$ Order (b) $2^{nd}$ Order (c) $3^{rd}$ Order.

native to R-Tree. Section 5.6 presents the experimental results and Section 5.7 concludes the chapter.

## 5.1 Related Work

Orenstein [74] demonstrated a generic method to convert multidimensional objects to a number of disjoint continuous segments (ranges) of the SFC, also called runs [78].

**Spatial Range Queries:** In [68, 70, 69] the Hilbert curve is used to represent locations of moving objects. The B$^x$-Tree [68] indexes current and future moving object positions (i.e., it answers 'predictive' queries about where the moving objects will be in the future). It partitions the time axis into fixed intervals according to the maximum duration between two updates from a moving object. Each interval creates an index partition while the update timestamp determines an object's index partition. The value indexed for each object update is a concatenation of the index partition and the Hilbert representation of the object's position at that time. As time passes the earliest interval (and its index) expires and a new interval is added. The ST$^2$B-Tree [69] also considers predictive queries and uses a

similar approach to the $B^x$-Tree but allows for different HC resolution based on the moving object density. In contrast, the $BB^x$-Tree [70] keeps one $B^+$-Tree for each time interval and thus preserves the past positions of the moving objects; it can thus answer 'historical' queries as well. One key property of these B-tree based methods is that moving object updates need to occur within a maximum interval; that is, a moving object has to report its position within that interval. The methods we propose in this chapter, do not have this limitation.

[71, 72] use Hilbert numbers not to represent but instead to sort spatial objects. The Hilbert R-tree [71] uses a HC to decide object insertion order. The idea is to improve R-tree performance by inserting objects together that are close to each other in the actual space. [72] focuses on indexing trajectories of objects moving on some constrained network (road network, etc.) Edges of the road network are sorted according to the Hilbert number representing their midpoint. Then edges are assigned 1D non-overlapping adjacent intervals of length proportional to the length of the edge. The spatial coordinates of trajectory segments are mapped to 1D intervals by corresponding them to network edges. Network edges and trajectories are indexed separately and evaluating a range query requires searching both indices.

The trajectory indexing methods that do not use SFCs can be categorized by the index they use (an R-tree or cell-based partitioning). Among the R-tree methods, the STR-Tree and TB-Tree [79] focus on clustering segments of the same trajectory close by in the index leaves, while the TPR-Tree [10] is for predictive queries; because of using MBRs to store segments, all these approaches have the dead space disadvantage. TrajStore [80],

facilitates a quad tree to store cells that are created by partitioning the spatial domain based on the trajectory density. The time dimension is not used explicitly in the index (conceptually, all trajectories are projected on their spatial coordinates) but as an interval that is assigned to each data page within a cell. At query time, cells are picked based on the query spatial range; out of the pages that a cell contains, the method considers only those pages with appropriate time interval. Having a good estimate of a cell's density is important for the method's efficiency. However, the cell density is estimated by using all the endpoints of the trajectories crossing the cell, which implies that the location update frequency should be high (so that that there is an endpoint in the cell for a fast object that passes through the cell) and the same for all moving objects (so that fast and slow objects contribute the same in the density estimation). Such requirements may be limiting for many location-based applications.

All the above methods consider time as a separate dimension that is explicitly or implicitly added to the index. Work on indexing temporal data [86] has proposed yet another approach for indexing time, that of partial persistence [87]. In particular, consider an object that was at spatial position $e_1$ at time $t_1$ and moved to $e_2$ at time $t_2$. This move can be approximated as a segment $(e_1, e_2)$ that first appeared at time $t_1$ and 'lived' until time $t_2$. As time proceeds, new segments are created and 'live' until they expire. The problem is then translated into maintaining a data structure that can maintain such segment evolution. This is achieved by taking an 'ephemeral' data structure that can solve the problem for a single time instant and making it partially-persistent [87] so as to also maintain the temporal evolution. This is the approach taken by the multi-version R-tree

(MVR-tree) [9] and the MV3R-tree [88] for storing spatial objects that change over time. The multiversion approach typically provides very fast query times for single time instant (or short time interval) queries. Nevertheless, it introduces additional space through a controlled duplication (so as objects are temporally clustered). Moreover, this approach still uses some dead space, since in the above example the whole segment $(e_1, e_2)$ is approximated from time $t_1$ even though the object was only at $e_1$ at that instant.

**$k$NN Queries:** Among the distance based queries, $k$NN queries are the most popular. Several variations of such queries on spatio-temporal data have been considered in the literature. Approaches are distinguished by *i*) the type of the query object (i.e., static point, trajectory, moving query), *ii*) the type of the data being queried (archival trajectory repository or streaming moving object data), and *iii*) the temporal predicate. The temporal predicate is either a time instant, but more frequently it is a time interval. There are two meanings considered in the interval case: (a) (interval semantics) the query result is a single set with the $k$ closest neighbors over all time instants in the query interval, or, (b) (instant semantics) the query result provides a $kNN$ set for each time instant in the interval.

In [89, 90], both static point and trajectory queries on archived trajectorial data are considered. They consider both interval and instant semantics. [91] considers interval semantics on static point and trajectory queries on archived data while the approach is extended to instant semantics in [92].Instant semantics for trajectory query and static point data is considered in [93]. In contrast, [94] considers instant semantics, trajectory query over archived trajectory data indexed in a 3D-R-tree. Ṅearest neighbor queries on moving point data have been discussed in [95, 96, 97]. These methods are focused on, once the $k$NN

query is evaluated for an initial timestamp, how to update the result as the moving point data or query is being updated.

The above works are build on the classic approaches to perform $kNN$ on static data indexed by an R-tree, namely, the 'branch and bound' [98, 99] and the 'best first' [100] algorithms. In section 5.3.2 we show that both algorithms can be evaluated using a Hilbert curve without converting the SFC ranges back to spatial coordinates, thus providing an efficient way to implement $kNN$ queries in general.

## 5.2 Background

A ***trajectory*** is a polyline with endpoints $T = (x_0, y_0, t_0), \ldots, (x_n, y_n, t_n)$, $t_i < t_{i+1}$ for $i = 0, 1, \ldots, n-1$. A ***trajectory segment*** is a straight line between two consecutive location updates $(x_i, y_i, t_i)$, $(x_{i+1}, y_{i+1}, t_{i+1})$ of the same moving object, where $i \in \mathbb{N}_0$. A ***subtrajectory*** of length $m$ of the trajectory $T$, is a subsequence $T' = (x_i, y_i, t_i), \ldots, (x_{m+i}, y_{m+i}, t_{m+i})$, of $m$ contiguous trajectory segments, where $i \geq 0, m < n$.

A Hilbert curve (HC) [101] is a self avoiding, continuous curve that goes through every point of a discretized multidimensional space exactly once. We thus assume that the spatial domain is represented by a $N \times N$**-grid** where $N = 2^l$, $l = 0, 1, 2, \ldots$. Each cell in this grid corresponds to a point; i.e., there are a total of $N^2$ possible locations (points) in the spatial domain all of which are visited once by the HC. Clearly, the higher the $l$ (also called the *order* of the Hilbert Curve) the higher the space resolution.

To construct a HC of order $l$ we partition the space recursively until we reach the maximum resolution at level $l$. The $1^{st}-$order HC is the smallest Hilbert curve (figure

5.1(a)), also called the **unit shape**. It corresponds to the basic building block, since a higher order HC is constructed by replicating the unit shape with appropriate rotation and direction as we recursively partition the space, and finally, connecting the unit shapes through their start and end points. Figure 5.1(b), (c) shows the $2^{nd}$ order and $3^{rd}$ order HC, respectively.

## 5.3 Methods

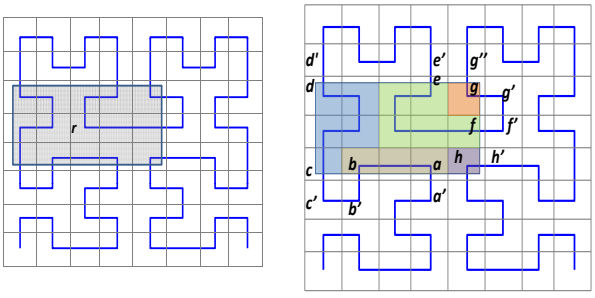
Given a spatial object $S$ (which for the purposes of the remaining discussion can be either a query rectangle or a line segment) and a Hilbert curve $\mathcal{H}$, let $h^S_{min}$($h^S_{max}$) be the minimum (maximum) number on $\mathcal{H}$ overlapping $S$. The run ($h^S_{min}$,$h^S_{max}$) contains all the points on $S$ as well as many points not overlapping with $S$ since this run leaves and enters $S$ many times. Each continuous part of the run ($h^S_{min}$,$h^S_{max}$) that is outside the object $S$ is called a **jump**, while the number of cells a jump contains is its *length*.

An example where the spatial object is a rectangle (query range $R$) appears in Figure 5.2(a); here a $3^{rd}-$order Hilbert curve is used. Based on Orenstein's approach [74], to avoid all points in the run ($h^R_{min}$,$h^R_{max}$) that are not in the rectangle, the run is split into multiple smaller runs, namely: $(a, b)$, $(c, d)$, $(e, f)$, $(g, g)$ and $(h, h)$. This creates the following jumps: $(b', c')$, $(d', e')$, $(f', g')$ and $(g'', h')$, as seen in Figure 5.2(b). The number of cells in a jump is the length of the jump e.g. the length of $(d', e')$ is $e' - d' + 1$.

We observe that most of the jump lengths are very small except a few exorbitantly high. If we *merge* two runs with a small jump between them, it will add a few redundant points (e.g. resulting in a few false positives). On the other hand, merging runs with big

jump between them will result in many false positives and increase the query time. It might be more efficient to run another range query instead. So we split an object only when there is a big jump. To identify big jumps we use a threshold computed based on simple statistics e.g., mean, standard deviation. We choose not to use complex method as this will add extra time to query evaluation. Using this approach the runs for $R$ would be $(a, d)$, $(e, g)$, $(h, h)$. Querying HT-index with these runs adds few false positives but reduces number of search required per range query. We present experimental result on the effects of the threshold value to merge runs in section 5.6. The above approach can be applied for line segments as well. Note that, when a trajectory segment is split, the timestamp at the split point is calculated assuming linear motion between original endpoints of the line.

While the above method reduces number of runs used to represent a spatial object, we still need to avoid false negatives when representing a trajectory segment with its endpoints. Given a line segment $L$, let $h_1^L$ and $h_2^L$ $(h_1^L \leq h_2^L)$ be Hilbert numbers of the end points of $L$. Taking the run $h_1^L$-$h_2^L$ to represent $L$ may result in false negatives. Because the actual run containing all the cells overlapping with $L$ is $(h_{min}^L, h_{max}^L)$ and the numbers $h_{min}^L$ and/or $h_{max}^L$) might not always lie at the end of $L$, i.e. $h_1^L \neq h_{min}^L$ and/or $h_2^L \neq h_{max}^L$.




(a) (b)

Figure 5.2: Runs of a range query.

For example, if we have a range query that maps to Hilbert numbers $12-15$ in figure 5.3(a) and line $ab$ is represented with the Hilbert numbers at the endpoints e.g. $(1, 10)$ we will have a false negative. On the other hand, If we use $(h^L_{min}, h^L_{max})$ to represent $L$ we loose the endpoint information. So, we want to maintain that $h^L_1 = h^L_{min}$ and $h^L_2 = h^L_{max}$.

The above condition can be achieved by recursively splitting a line segment according to the following heuristic. We split a line segment when $h^L_1 \neq h^L_{min} (h^L_2 \neq h^L_{max})$ so that the endpoints of one of the new line segments is $(h^L_1, h^L_{min})$ $((h^L_2, h^L_{max}))$. Then we recursively split the other segment until the above condition is satisfied. The above splitting rules ( to avoid false negatives or big jumps) generates less splits than Orenstein's method leading to less query evaluation time. Finally, we combine time intervals with the runs and index MBRs of $(h, t)$ coordinate pair ($ht-$MBRs) with a R-Tree.

### 5.3.1 Range Query Evaluation

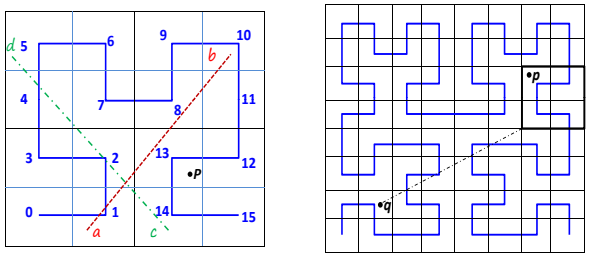
We consider 3D range queries i.e. a spatial range and a temporal interval. The spatial range of the query is, first, mapped to the overlapping cells of the underlying grid (and thus the actual query rectangle is *enlarged* to align with the nearest cell boundaries). The spatial range is, then, split into a number of runs *query runs* according to Orensteins method and then the runs with small jumps are merged (*merged runs*). Finally, the query temporal interval is combined with each of them to obtain the $ht-$MBRs. Effectively, the original query is thus divided into many smaller range queries. These $ht-$MBRs are searched in the HT-index which returns a set of $ht-$MBRs representing spatio-temporal trajectory segments. However, we need to verify these results since they might contain false positives. A straightforward way to detect the false positives would be to take the runs (*result runs*)

from the $ht-$MBRs in the search result , convert them to spatial $(x, y)$ coordinates and then, check if they are within the spatial range specified by the query. However, we use the query runs and result runs to avoid the Hilbert to spatial conversion.

False positives may occur because of 1) query enlargement and 2) merging query runs. To verify the query result we consider two rectangles with edges aligned with boundaries of underlying grid cells. The smallest rectangle $R_{out}$ containing the range query $R_q$ and the largest rectangle $R_{in}$ contained by $R_q$. $R_{mid}$ is the region between $R_{out}$ and $R_{in}$. If a result run $r_h$ does not overlap with any of the query runs then the corresponding trajectory segment is outside $R_{out}$ and resulted because of merging query runs. If $r_h$ overlaps with the $I^{\mathcal{H}}_{R_{in}}$ then it belongs to the query result. We need to convert the Hilbert numbers of $r_h$ and the cells in $R_{mid}$ to spatial coordinates only when $r_h$ is neither inside $R_{in}$ nor outside $R_{out}$. Thus by using $R_{out}$ and $R_{in}$ we can verify most of the results efficiently.

### 5.3.2 $k$NN Query Evaluation

Algorithm 7 shows the pseudo code for incremental $k$NN search [100]. The only problem towards using Inc$k$NN with HT-Index is the distance function $dist(a, b)$ which




(a) (b)

Figure 5.3: (a) False negatives when using Hilbert range. (b) Distance between a query point and a sub-grid.

evaluates Euclidean distance between two spatial objects $a$ and $b$. Evaluating $dist()$ is not possible when the $(x, y)$ coordinates are replaced with Hilbert numbers. Any two points close to each other on the HC is also close to each other in the Euclidean space. However, the reverse is not always true i.e. sometimes there are big difference between Hilbert numbers of two spatially close points. So its not always possible to lower bound Euclidean distance from Hilbert numbers.

The straight forward way to use the Inc$k$NN algorithm is to compute spatial representation from the Hilbert representation of an object or node MBR before computing the Euclidean distance. But it will add significant overhead to $k$NN search. We want to be able to implement $k$NN search with zero of very little overhead.

The Inc$k$NN algorithm efficiently finds out the nearest neighbors by pruning nodes that contains objects that are too far away to belong to the $k$NN result (e.g. if a new object is farther than the current candidate object in the queue then the new object is not considered). So to determine whether a node can be a $k$NN candidate we only need to know a lower bound of the distance to that node(MBR). Consider query point $q$ and a data point $p$ in figure 5.3(b). Suppose the distance of the $k$ farthest object in the queue is $x$. If $dist(p, q) > x$ then $p$ cannot be a candidate for $k$NN. This decision can also be made if we know that the distance between $q$ and a low level grid cell (e.g. a cell of a $2 \times 2$-grid where as the space is actually partitioned into a $8 \times 8$-grid) that contains $p$ is more than $x$.

We can compute a lower bound of Euclidean distance between a query point and a object by inspecting a few bits from the Hilbert representation of the object. Higher the number of bits inspected, tighter is the lower bound.Consider point $p$ in figure 5.1(a).

We will demonstrate how to compute the $3^{rd}-$order Hilbert number $h$ of $P$ from its $(x, y)$ coordinate and the reverse (e.g. $(x, y)$ from $h$). We start with a $2 \times 2$-grid, $1^{st}-$order curve. The coordinate of $P$ in this $2 \times 2-$grid is $(1, 0)$, figure 5.1(a). The Hilbert number for this cell is 11(in binary) which are the two MSBs of $h$. So now we have $h = 11 * * * *$, $x = 1 * *$ and $y = 0 * *$. Now, we further partition all the cells in the current $2 \times 2-$grid and get the $4 \times 4-$grid and corresponding $2^{nd}-$order Hilbert curve, figure 5.1(b). The coordinate of $P$ in the SE $2 \times 2-$grid is $(0, 1)$ and it maps to 01 in the $1^{st}-$order curve which gives us $h = 1101 * *$, $x = 10*$ and $y = 01*$. Finally we partition the cells again and get a $3^{rd}-$curve. The coordinate of $P$ in its current $2 \times 2-$grid is $(1, 0)$ and number 11 along the curve, figure 5.1(c). So we have the final results $h = 110111 (= 55)$, $x = 101$, $y = 010$.

From the above illustration, we can compute a lower level grid cell $b$ containing a given point $p$ by inspecting a few most significant bits of the points Hilbert number $h$. We start by inspecting first two MSBs of

If $dist(q, b)$ is too high to be $k$NN candidate we stop decoding and discard this point. Otherwise we .

The same approach can be used for a pair of Hilbert numbers $(h_1, h_2)$ representing a line segment since they are contained by the same low level grid cell most of the times. Very rarely, we may have to inspect higher number of bits

The real distance is calculated only when an entry $e$ is popped from the queue. After calculating the actual distance $dist(q, e)$, $e$ may not be the top element of the queue. In that case $e$ needs to be inserted in the queue and the next element has to be processed. Another way is not to fully process $e$. Suppose $n$ bits of $e$ have been processed. We can

decode next two bits and check if it still is the top element of the queue. We stop when $e$ is fully decoded or cease to be the top element of the queue. If $e$ is fully decoded then $e$ is the nearest entry to $q$. In this way only entries close to the query point are fully decoded. Entries that are far away are only partially decoded e.g. the farther the entry the less number of bits is decoded. Our experiments suggests this adds very little overhead to $k$NN search.

## 5.4 Alternative Approaches

We consider a number of alternative index structures besides R-Tree to index trajectories represented with Hilbert numbers. First, we consider multi version R-Tree [9] with elongated MBRs of one dimensional Hilbert numbers e.g., an MBR with $(h_1, 0), (h_2, 0)$ as corner points. However, we find that these elongated MBRs are not clustered properly i.e., MBRs too far away from each other are put under the same leaf node and thus a very big MBR with large dead space is created. This in turns, results in many false positives and increase query evaluation time.

Next approach we consider is a combination of a multi-version indexing and a one dimensional indexing. Namely **MI-Tree**: multi-version B-Tree (MVB-Tree)[82] and Interval Tree[81]. Interval tree was developed to index one dimensional intervals. However, our goal is to index runs that also have temporal intervals. Kriegel et al. [102] shows that an Interval Tree can be implemented using two $B^+$-Trees. So, runs (without temporal intervals) can be indexed with Interval Tree which in turn can be implemented with two B-trees. To handle the corresponding temporal intervals associated with these runs we propose using

*multi-version* B-Trees instead of regular B-Trees.

[102] shows that for each range query on interval tree, it requires to do $2h$ range queries on $B^+$-Tree, where the height of the interval tree $h = log(x)$, $x$ = extension of the data space. In our case, $x = 2^{2*ord}$, $ord$ = order of the Hilbert curve being used. So if we use a $10^{th}$ order Hilbert curve we need to do 40 range queries(in the MVB-Tree). Remember, using Hilbert curve already requires multiple index search for each range query. Using MI-Tree will multiply the number of index search required by another significant factor. In our experiments we find that query evaluation in MI-Tree is more than 50 times slower than that in a R-Tree. This is impractical, and so, don't consider this as a competitor.

Finally, we consider a one dimensional indexing method, Snapshot Index [103], to index runs. Snapshot index was proposed to record changes over time e.g., the lifetime of an object starts at time $t_1$ and ends at $t_2$. However, we use snapshot index to record the changes in Hilbert dimension i.e. Hilbert dimension is considered as time dimension. For example a run $(h_1, h_2)$ for an object $o$ is interpreted as $o$ becomes alive at $h_1$ and dies at $h_2$. We split the actual time dimension into $m$ equal intervals and maintain one snapshot index for each interval. Within each interval we ignore time and consider only Hilbert numbers. For example, given an $ht-$MBR $(h_1, t_1)$-$(h_2, t_2)$ for an object $o$, we insert $o$ with the lifetime $(h_1, h_2)$ into each snapshot index with temporal interval overlapping with $(t_1, t_2)$.

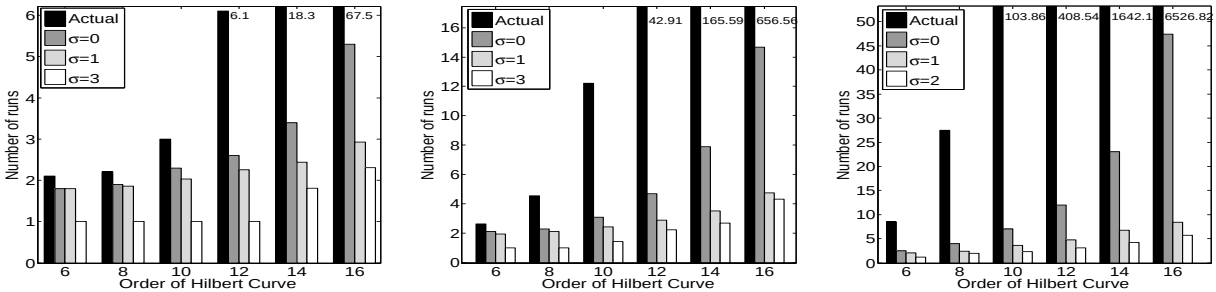
Note that time is always increasing but there is no order among the Hilbert numbers visited by a trajectory. So we need to sort the Hilbert numbers visited by a trajectory. But this makes the update harder. Because updates can never happen in the past but a trajectory can go back at the same location many times. So an update might require to

resort the data and rebuild an index. Moreover, an object $o$ can have same $(h_1, h_2)$ segment multiple times i.e., $o$ needs to be inserted multiple times with the same lifetime. To avoid this we use concatenation of the actual starting time stamp $t_1$ and the object ID of $o$ as the insertion key. So each run is going have a different ID.

## 5.5 Cost Model for HT-Index

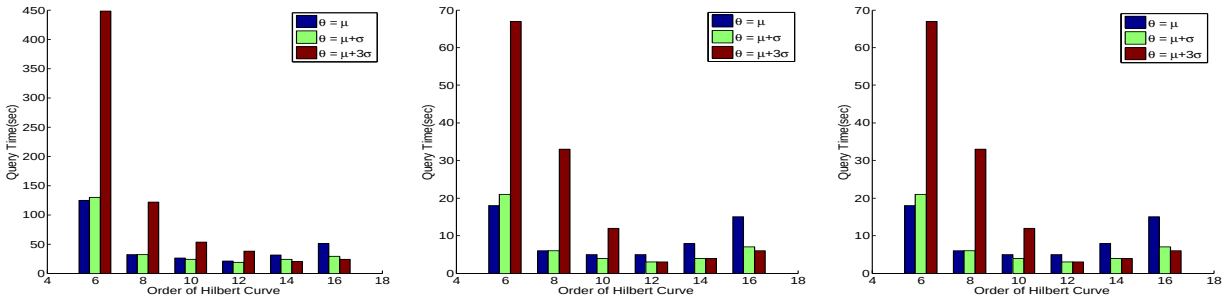
We attempt to relate the order of Hilbert curve to query evaluation performance so that the optimum grid size can be selected for a given data. Cell size determines 1) number of search required for each query 2) data size increment 3) dead space 4) query enlargement.

Smaller cell size implies higher number of runs, consequently higher number of search for each query. Data size increment depends on number split required. The average number of split for each segment depends on the cell size and frequency of location update. Smaller cell size will result in higher number of splits, and lower dead space. For low frequency data (i.e. long line segments) there will be more split. Query enlargement depends



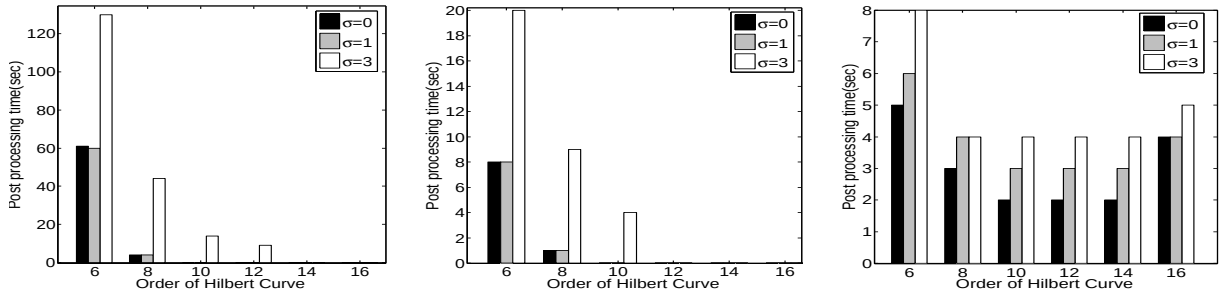

(a)0.1% Queries (b)1% Queries (c)10% Queries

Figure 5.4: Number of runs




(a)0.1% Queries (b)1% Queries (c)10% Queries

Figure 5.5: Index search time.




(a)0.1% Queries (b)1% Queries (c)10% Queries

Figure 5.6: Time for post-processing.

on both cell size and query size. Usually, smaller queries ($e.g.0.1\%$) have higher percent of enlargement than larger queries (e.g. 10%). And, smaller cell size implies lower query enlargement. Query enlargement results in false positives requiring higher post-processing time. Our experiments confirm that post-processing for smaller queries are more affected by cell size.

Post-processing is a very light weight operation compared to disk access(DA) during the search. So we attempt to estimate average disk access for a range query. HT-Index is an R-tree based method. Each Hilbert number represents a cell in the underlying grid. A Hilbert run represents an arbitrary shaped area composed of a number of grid cells. This implies that we can apply the cost model in [85] for R-tree to predict disk access for HT-Index. The formula in [85] uses number of MBRs and MBR density to estimate average number of disk access for range queries. MBR density is defined as average number of MBRs that contain a given point. If the space is a unit area, the MBR density is just the sum of area of MBRs. We can apply the same definition in our arbitrary shaped regions too.

As we discussed before, a trajectory segment might be needed to split and so data size may increase. Computation of the number of segments and density of the regions corresponding to their Hilbert runs for a given order of Hilbert curve, requires one pass over the data. If the data is too big any good sampling algorithm can be used. After computing these values estimating disk access using the formula in [85] is very straight forward. This estimation can help selecting the Hilbert order for a given data set.

## 5.6 Experimental Evaluation

All the experiments were run in an Intel Xeon 3.0GHz processor running Linux 2.6.18 with 8GB of main memory. C++ was used for implementation. In our experiments we use the GeoLife dataset [41]. It contains public activity data (i.e. shopping, dining, sightseeing, hiking, cycling etc.) in Beijing, China.

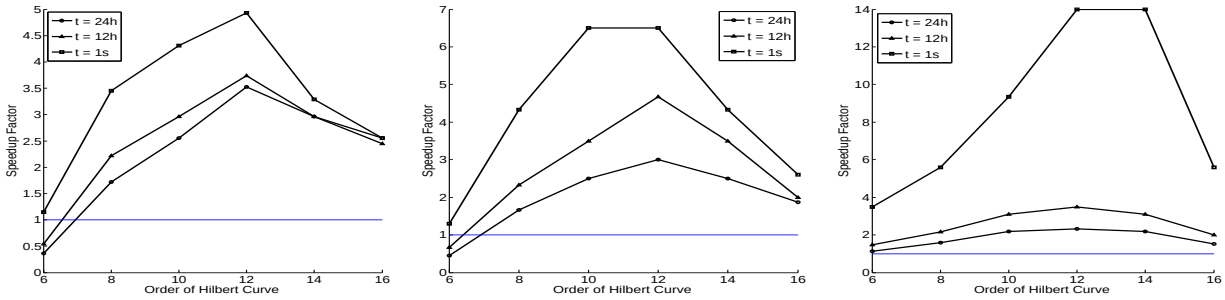
We experimented query performance for queries with spatial extent 0.1%, 1% and 10% of the total area. For temporal extent we use one second (timestamp), 12 hours and 24 hours. We also experiment with different threshold values for merging the runs of a range query. To evaluate query performance for various cell sizes we vary Hilbert order from 6 to 16. We consider is a $4^o$ latitude by $4^o$ longitude area. An order-16 Hilbert curve implies approximately $7m$ by $7m$ grid cells.

We first experiment the effect of the threshold value for merging runs. To identify a threshold that separates small and big jumps we looked into the distribution of these jumps, and their mean ($\mu$) and standard deviation ($\sigma$). We observe that there are very few extremely large jumps. Those large jumps can be eliminated just by using $\mu$ as a threshold. We experiment number of runs with $\mu$, $\mu + \sigma$ and $\mu + 3 * \sigma$ as threshold value, figure 5.4. For lower order and smaller query size, number of ranges is small and so there is not much reduction of number of runs for different threshold values. However, for higher order there is a significant reduction. Also, larger the query size higher the reduction. Experiment shows, we are getting most of the reduction just by using $\mu$ as the threshold. Using $\mu + \sigma$ or $\mu + 3 * \sigma$ results in more reduction, but they will also add more false positives too. Which in turn, will result in higher index search and post processing time. Figure 5.5 and 5.6

shows a big increase in search and post-processing time respectively with low order HC when $\theta = \mu + 3 * \sigma$. From these experiments we select $\theta = \mu + \sigma$ as a threshold to identify big jumps.

Next, we compare our method with 3D R-tree for range query evaluation. We run $100k$, $10k$, and $1k$ instance of queries with 0.1%, 1%, 10% spatial range respectively, figure 5.7. For each spatial extent we use three temporal extents of 1sec, 12hr and 24hr. Higher speed up (14 times) is obtained for timestamp queries, as R-tree does not perform well for timestamp queries. As we increase the order of the HC speedup factor increases and then it starts to decrease. This is because with lower order HC the bigger grid cells results in bigger query enlargement. Also, merging runs with bigger cell size result in more false positives and higher search time. The reason why performance deteriorates at higher order will be explained with the cost model later in this section.

We also compare our method with a quad tree index that has the same cell splitting formula as TrajStore. We note that [80] also suggests adding in TrajStore techniques like: delta compression for trajectories, bundling trajectories on the same path and adaptive query processing; however, all of these optimizations can be applied independently of the underlying indexing (whether it is quad-tree, R-tree or SFC-based). Thus in our comparison with the HT-Index we considered the basic TrajStore implementation with data adaptive cell size. The results appear in figure 5.8. The speedup factor is less than that with 3D R-tree which is consistent with the claim of the authors about TrajStore's speedup over R-tree based method. Our method not only outperforms this quad tree based approach but also has the advantage that




(a)0.1% Queries (b)1% Queries (c)10% Queries

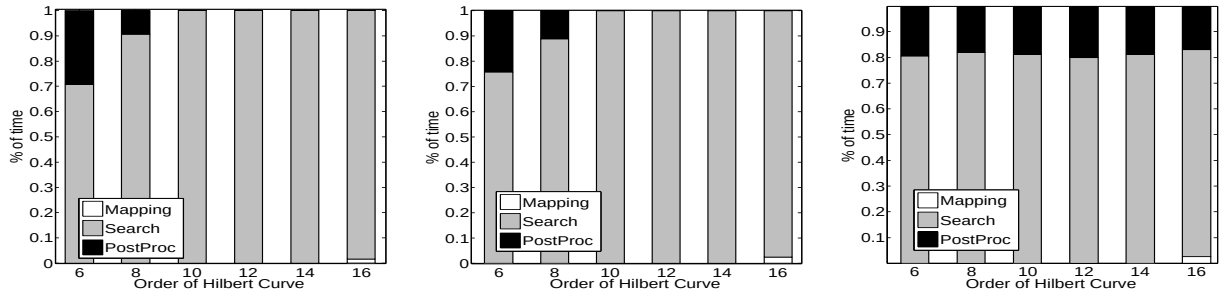
Figure 5.7: Speedup Factor.

Figure 5.9 shows, the fraction of query evaluation time spent for query mapping, search and post processing. Query mapping includes converting the query range to runs and then merging them to reduce the number of search. Mapping time is virtually zero or negligible(for higher order). For higher order, there are higher number of runs and more time required to merge them. The threshold value used to merge runs has no effect on mapping time. Experiments support our expectation that postprocessing (due to query enlargement) is affected more by cell size for smaller queries. A small query is enlarged more for lower order (bigger cells). For bigger queries post-processing time remains same independent of the cell size. These experiments show that post-processing time is very small. Search time (number of search required) dominates the overall performance.

Figure 5.10(a) shows speedup of our method over R-tree in kNN queries. In spite of the little added overhead in distance computation, our method outperforms regular R-tree based method. One possible reason is the better clustering property of Hilbert curve. Better clustering of spatial objects will quickly lead to the nodes containing nearest neighbors. Although our method outperforms R-tree and quad tree based methods the speedup drops as we increase the order. We look for the reasons in the following experiments.


(a)0.1% Queries (b)1% Queries (c)10% Queries

Figure 5.8: Speedup Factor.



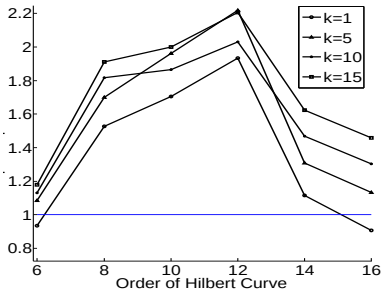

(a)0.1% Queries (b)1% Queries (c)10% Queries

Figure 5.9: Fraction of time used for query mapping, searching and post-processing.

Finally, the reason behind the performance drop of our method above a certain resolution of space can be explained from the estimated disk access($DA$) according to the cost model. As mentioned before DA depends on number of MBR ($N$) and density of MBRs ($\rho$). Figure 5.11(a), (b) and (c) shows number of MBR, MBR density and estimated $DA$ for different orders. With increasing order, $N$ increases but $\rho$ decreases. Increasing $N$ or $\rho$ will increase $DA$ and vice versa. In our experiment, $DA$ increases in a similar fashion as $N$. From the simplified formula in [85] for two dimensional space, its easy to see that DA is more dependent on $N$ than $\rho$. According to the cost model the $DA$ increases sharply after order 12 which is very consistent with the performance drop in our experiments after order 12.

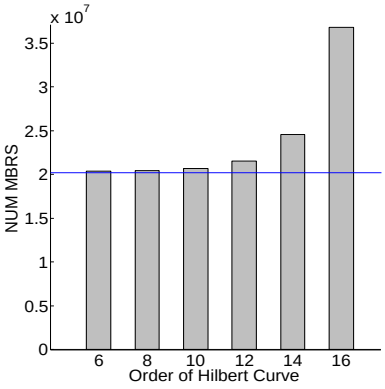
## 5.7 Conclusion

In this chapter we propose technics to improve query performance on trajectory data. We utilize Hilbert curve to represent trajectory polylines. Using Hilbert numbers instead of Euclidean coordinates to represent trajectories is not straightforward. We present techniques to overcome the complexities involved in this approach. We have also shown, for the first time, how to use existing branch and bound algorithms to evaluate $k$NN queries. Our proposed method outperforms traditional $3D$ R-tree and quad tree based method TrajStore. Our proposed method can easily be integrated with any available RDBMS that provides an R-Tree implementation. In this work we mostly focus on spatial dimensions. In future we plan to include temporal dimension as well.

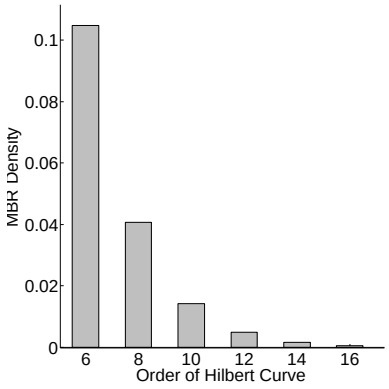



(a)Speedup for kNN queries.

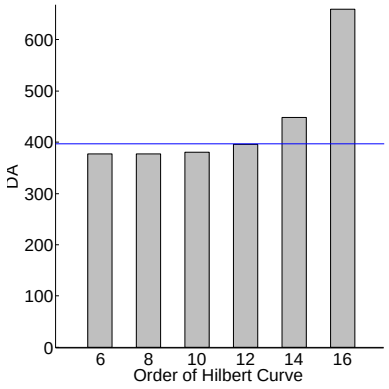
Figure 5.10:




(a)Number of MBR




(b)MBR Density




(c)Estimated DA.

Figure 5.11: Disk access estimation.

**Algorithm 7** Inc$k$NN(R-tree $T$, Query $p$)

```
1: Q : a priority queue.
2: Q.enqueue(T.root, 0)
3: while Q not empty do
4:    e ← Q.dequeue()
5:    if e is a data object or its MBR then
6:       if e is MBR of o and Q not empty and dist(p, o) > Q.min.dist then
7:          Q.enqueue(o, dist(p, o))
8:       else
9:          e (or o) is the next NN.
10:      end if
11:   else if e is a leaf then
12:      for each entry (o, mbr) in e do
13:         Q.enqueue(o, dist(o, mbr))
14:      end for
15:   else
16:      for each entry (n, mbr) in e do
17:         Q.enqueue(n, dist(n, mbr))
18:      end for
19:   end if
20: end while
```

# Chapter 6

# Conclusions

The volume of moving object trajectory data is increasing every day. Data is being produced at a higher rate than we can process and make sense of. More and more intelligent algorithm is required to analyze this data and automatically discover the hidden knowledge.

The contributions of thesis are following.

- We propose online identification of Dwell Region and Dwell Behavior. Dwell Behavior is a new trajectory semantics that we propose for the first time. This problem has applications in surveillance, understanding animal behavior, trajectory simplification, etc. We propose online algorithm to identify Dwell Region and Dwell Behavior. We also show how to estimate Dwell Region when exact computation is not necessary.

- In an effort to infer the movement of a group of objects when their exact location is unknown we propose computing Corridors and Conclaves for these objects. We assume discrete (and possibly distant) locations of these objects are known from diverse sources e.g., surveillance camera, ATM usage, etc. A corridor includes all possible

locations an object could have gone through while traveling between two locations where the object is known to be. The conclaves of a group of object show the potential locations where they could have met. This problem is useful in tracking persons of interest (e.g. fugitives, criminals) who are suspected to have interaction between them and put a collective effort to achieve a common goal.

- We show how to identify regions of interest from trajectory data without accessing any external data. Our method allows querying regions of interest with different semantics with any arbitrary values of the parameters defining the ROIs. We consider trajectory density to estimate the popularity of these ROIs.

- We present methods to represent trajectories with Hilbert curves and index them using several existing index structures. This is not straightforward because of the following reasons. First, it might result in false negatives. In our method we carefully avoid any false negatives. Second, it requires splitting a query in to multiple queries and searching the index multiple times. We show how to minimize the number of searches required per query. Third, evaluating distance based queries was never attempted when Hilbert curve is used to represent spatial objects. This is because the Euclidean distance is not preserved when spatial coordinates are transformed to Hilbert numbers. We present simple techniques to lower bound the Euclidean distance from Hilbert numbers. Finally, we show how to use an existing cost model for R-Tree to determine the order of the Hilbert curve.

Understanding trajectory semantics will have increasing usefulness in the upcoming days. In this thesis we present a few novel semantics out of numerous possibilities.

While existing works do not relate their discovered patterns to any semantics we consider only those patterns that have useful semantics. We believe this thesis will encourage the research community to think of novel and useful semantics.

# Bibliography


[1] www.accutracking.com.

[2] tracNet24, www.isecuretrac.com.

[3] Footpath, www.pathintelligence.com.

[4] GeoChat, instedd.org/geochat.

[5] www.bikely.com.

[6] www.gpsxchange.com.

[7] www.everytrail.com.

[8] www.sports-tracker.com.

[9] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos, "Indexing spatiotemporal archives," in *VLDB Journal*, 2006.

[10] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing positions of continuously moving objects," in *SIGMOD*, vol. 29, no. 2, 2000, pp. 331–342.

[11] J. Ni and C. V. Ravishankar, "Indexing spatio-temporal trajectories with efficient polynomial approximations," in *TKDE*, 2007.

[12] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi, "Trajectory pattern mining," in *ACM KDD*, 2007, pp. 330–339.

[13] F. Giannotti, M. Nanni, and D. Pedreschi, "Efficient mining of temporally annotated sequences," in *SDM*, 2006.

[14] J.-G. Lee, J. Han, and K.-Y. Whang, "Trajectory clustering: A partition-and-group framework," in *ACM SIGMOD*, 2007, pp. 593–604.

[15] G. Trajcevski, "Probabilistic range queries for uncertain trajectories on road networks," in *EDBT*, 2011.

[16] ——, "Uncertain range queries for necklaces," in *MDM*, 2010.

[17] M. R. Vieira, P. Bakalov, and V. Tsotras, "Querying trajectories using flexible patterns," in *EDBT*, 2010, pp. 406–417.

[18] M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. J. Tsotras, "Complex spatio-temporal pattern queries," in *VLDB*, 2005, pp. 877–888.

[19] Y. Bu, L. Chen, A. W.-C. Fu, and D. Liu, "Efficient anomaly monitoring over moving object trajectory streams," in *KDD*, 2009.

[20] M. Vlachos, G. Kollios, and D. Gunopulos, "Discovering similar multidimensional trajectories," in *ICDE*, 2002.

[21] Y. Cai and R. Ng, "Indexing spatio-temporal trajectories with chebyshev polynomials," in *SIGMOD*, 2004.

[22] P. Bakalov, M. Hadjieleftheriou, E. J. Keogh, and V. J. Tsotras, "Efficient trajectory joins using symbolic representations," in *MDM*, 2005.

[23] P. Bakalov, E. J. Keogh, and V. J. Tsotras, "Ts2-tree - an efficient similarity based organization for trajectory data," in *GIS*, 2007.

[24] D. Fortin, H. L. Beyer, M. S. Boyce, D. W. Smith, T. Duchesne, and J. S. Mao, "Wolves influence elk movements: behavior shapes a trophic cascade in yellowstone national park," in *Ecological Society of America*, vol. 86, 2005, pp. 1320–1330.

[25] M. Buchin, A. Driemel, M. van Kreveldz, and V. Sacristn, "An algorithmic framework for segmenting trajectories based on spatio-temporal criteria," in *SIGSPATIAL*, November 2010.

[26] M. R. Vieira, P. Bakalov, and V. Tsotras, "Querying trajectories using flexible patterns," in *EDBT*, 2010, pp. 406–417.

[27] J.-G. Lee, J. Han, and K.-Y. Whang, "Trajectory clustering: A partition-and-group framework," in *ACM SIGMOD*, 2007, pp. 593–604.

[28] X. Cao, G. Cong, and C. S. Jensen, "Mining significant semantic locations from gps trajectory," in *VLDB*, 2010, pp. 1009–1020.

[29] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma, "Mining interesting locations and travel sequences from gps trajectories," in *WWW*, 2009.

[30] M. R. Uddin, C. V. Ravishankar, and V. J. Tsotras, "Finding regions of interest from trajectory data," in *MDM*, 2011.

[31] M. R. Vieira, P. Bakalov, and V. Tsotras, "On-line discovery of flock patterns in spatio-temporal data," in *ACM GIS*, 2009, pp. 286–295.

[32] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, "Discovery of convoys in trajectory databases," in *PVLDB*, vol. 1, no. 1, 2008, pp. 1068–1080.

[33] J. Ni and C. V. Ravishankar, "Pointwise-dense region queries in spatio-temporal databases," in *IEEE ICDE*, 2007, pp. 1066–1075.

[34] N. Megiddo, "Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems," in *FOCS*, Nov 1982, pp. 329 – 338.

[35] K. L. Clarkson, "Las vegas algorithms for linear and integer programming when the dimension is small." in *JACM*, vol. 42, no. 2, March 1995.

[36] M. E. Dyer and A. M. Frieze, "A randomized algorithm for fixed-dimensional linear programming." in *Mathematical Programming*, vol. 44, no. 1-3, May 1989.

[37] R. Seidel, "Linear programming and convex hulls made easy." in *sixth annual Symposium on Computational Geometry*, 1990.

[38] E. Welzl, "Smallest enclosing disks (balls and ellipsoids)," in *New Results and New Trends in Computer Science, Lecture Notes in Computer Science*, vol. 555, 1991, pp. 359–370.

[39] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang, "Multi-guarded safe zone: An effective technique to monitor moving circular range queries," in *ICDE*, March 2010, pp. 189 – 200.

[40] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *ACM SIGMOD*, 1984, pp. 47–57.

[41] http://research.microsoft.com/en-us/projects/geolife/.

[42] http://www.fs.fed.us/pnw/starkey/.

[43] J. Ni, C. V. Ravishankar, and B. Bhanu, "Probabilistic spatial database operations," in *SSTD*, 2003.

[44] G. Trajcevski, "Managing uncertainty in moving objects databases," in *TODS*, 2004.

[45] ——, "The geometry of uncertainty in moving objects databases," in *EDBT*, 2002.

[46] ——, "Probabilistic range queries in moving objects databases with uncertainty," in *MobiDe*, 2003.

[47] T. Emrich, H.-P. Kriegel, N. Mamoulis, M. Renz, and A. Zufle, "Querying uncertain spatio-temporal data," in *ICDE*, 2012.

[48] G. Trajcevski, "Continuous probabilistic nearest-neighbor queries for uncertain trajectories," in *EDBT*, 2009.

[49] J. S. Greenfeld, "Matching gps observations to locations on a digital map," in *Transportation Research Board 81st Annual Meeting*, 2002.

[50] K. Zheng, Y. Zheng, X. Xie, and X. Zhou, "Reducing uncertainty of low-sampling-rate trajectories," in *ICDE*, 2012.

[51] J. Yuan, Y. Zheng, C. Zhang, X. Xie, and G.-Z. Sun, "An interactive-voting based map matching algorithm," in *MDM*, 2010.

[52] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang, "Map-matching for low-sampling-rate gps trajectories," in *ACM SIGSPATIAL GIS*, 2009.

[53] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Numerische Mathematik*, vol. 1, no. 1, 1959, pp. 269–271.

[54] M. N. Rice and V. J. Tsotras, "Parameterized algorithms for generalized traveling salesman problems in road networks," in *SIGSPATIAL*, 2013.

[55] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Workshop on Experimental Algorithms*, 2008, pp. 319–333.

[56] K. Xie, K. Deng, and X. Zhou, "From trajectories to activities: A spatio-temporal join approach," in *LBSN'09: Proc. of the Int'l Workshop on Location Based Social Networks*, 2009, pp. 25–32.

[57] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W.-Y. Ma, "Mining user similarity based on location history," in *ACM GIS*, 2008, pp. 1–10.

[58] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma, "Mining interesting locations and travel sequences from gps trajectories," in *WWW*, 2009, pp. 791–800.

[59] X. Cao, G. Cong, and C. S. Jensen, "Mining significant semantic locations from gps trajectory," in *VLDB*, 2010, pp. 1009–1020.

[60] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras, "On-line discovery of dense areas in spatio-temporal databases," in *SSTD*, 2003.

[61] C. S. Jensen, D. Lin, B. C. Ooi, and R. Zhang, "Effective density queries on continuously moving objects," in *ICDE*, 2006.

[62] L. O. Alvares, V. Bogorny, B. Kuijpers, J. A. F. de Macedo, B. Moelans, and A. Vaisman, "A model for enriching trajectories with semantic geographical information," in *ACM GIS*, 2007, pp. 1–8.

[63] A. T. Palma, V. Bogorny, B. Kuijpers, and L. O. Alvares, "A clustering-based approach for discovering interesting places in trajectories," in *ACM SAC*, 2008, pp. 863–868.

[64] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *ACM SIGKDD*, 1996, pp. 226–231.

[65] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *ACM SIGMOD*, 1990, pp. 322–331.

[66] "Spatial Index Library," http://dblab.cs.ucr.edu/spatialindexlib.html.

[67] crawdad.cs.dartmouth.edu.

[68] C. S. Jensen, D. Lin, and B. C. Ooi, "Query and update efficient $b^+$-tree based indexing of moving objects," in *VLDB*, 2004.

[69] S. Chen, B. C. Ooi, K.-L. Tan, and M. A. Nascimento, "$St^2b$-tree: A self-tunable spatio-temporal $b^+$-tree index for moving objects," in *SIGMOD*, June 2008.

[70] D. Lin, C. S. Jensen, B. C. Ooi, and S. Saltenis, "Efficient indexing of the historical, present, and future positions of moving objects," in *MDM*, 2005.

[71] I. Kamel and C. Faloutsos, "Hilbert r-tree: An improved r-tree using fractals," in *VLDB*, 1994, pp. 500–509.

[72] D. Pfoser and C. S. Jensen, "Indexing of network constrained moving objects," in *GIS*, November 2003.

[73] J. Orenstein, "A class of data structures for associative searching," in *SIGACT*, 1984.

[74] J. A. Orenstein, "Spatial query processing in an object-oriented database system," in *SIGMOD*, 1986.

[75] P. Xu and Tirthapura, "A lower bound on proximity preservation by space filling curves," in *IPDPS*, 2012.

[76] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," in *TKDE*, 2001.

[77] P. Xu and S. Tirthapura, "On optimality of clustering through a space filling curve," in *PODS*, 2012.

[78] H. V. Jagadish, "Analysis of the hilbert curve for representing two-dimensional space." in *Information Processing Letters ELSEVIER*, 1997.

[79] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches to the indexing of moving object trajectories," in *VLDB*, 2000.

[80] P. Cudre-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets," in *ICDE*, 2010.

[81] H. Edelsbrunner, "A new approach to rectangle intersections," in *Institute for Information Processing, Technical University of Graz*, September 1982.

[82] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion b-tree," in *VLDB Journal*, vol. 5, no. 4, December 1996, pp. 264–275.

[83] K.Fukunaga, "Introductionto statistical pattern recognition," in *Academic Press*, vol. 2nd Edition, 1990.

[84] E. Frentzos, N. Pelekis, I. Ntoutsi, and Y. Theodoridis, "Trajectory database systems," in *Mobility, Data Mining and Privacy*, ser. Geographic Knowledge Discovery, F. Giannotti and D. Pedreschi, Eds. Springer, 2008, ch. 6, pp. 151–187.

[85] Y. Theodoridis and T. Sellis, "A model for the prediction of r-tree performance," in *PODS*, 1996.

[86] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," in *ACM Computating Survey*, vol. 31, no. 2, 1999, pp. 158–221.

[87] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," in *Journal of Computer and System Sciences*, vol. 38, no. 1, 1989, pp. 85–124.

[88] Y. Tao and D. Papadias, "The mv3r-tree : A spatio-temporal access method for timestamp and interval queries," in *VLDB*, 2001.

[89] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, "Nearest neighbor search on moving object trajectories," in *SSTD*, 2005.

[90] E. Frentzos, K. Gratsias, N. Pelekis, and Theodoridis, "Algorithms for nearest neighbor search on moving object trajectoriess," in *GeoInformatica*, 2007.

[91] Y.-J. Gao, C. Li, G.-C. Chen, L. Chen, X.-T. Jiang, and C. Chen, "Efficient k-nearest-neighbor search algorithms for historical moving object trajectories," in *Journal of Computer Science and Technology*, vol. 22, no. 2, 2007.

[92] Y. Gao, C. Li, G. Chen, Q. Li, and C. Chen, "Efficient algorithms for historical continuous k nn query processing over moving object trajectories," in *APWeb/WAIM*, 2007.

[93] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *VLDB*, 2002, pp. 287–298.

[94] R. H. Guting, T. Behr, and J. Xu, "Efficient k-nearest neighbor search on moving object trajectories," in *The VLDB Journal*, vol. 19, no. 5, 2010.

[95] X. Xiong, M. F. Mokbel, and W. G. Aref, "Sea-cnn: scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases," in *ICDE*, 2005.

[96] X. Yu, K. Q. Pu, and N. Koudas, "Monitoring k-nearest neighbor queries over moving objects," in *ICDE*, 2005.

[97] G. S. Iwerks, H. Samet, and K. Smith, "Continuous k-nearest neighbor queries for continuously moving points with updates," in *VLDB*, vol. 29, 2003, pp. 512–523.

[98] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *SIGMOD*, 1995.

[99] K. L. Cheung and A. W.-C. Fu, "Enhanced nearest neighbour search on the r-tree," in *SIGMOD*, 1998.

[100] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," in *TODS*, vol. 24, no. 2, 1999.

[101] D. Hilbert, "ber die stetige abbildung einer linie auf ein flchenstck." in *Mathematische Annalen*, vol. 38, 1891, pp. 459–460.

[102] H.-P. Kriegel, M. Ptke, and T. Seidl, "Managing intervals efficiently in object-relational databases," in *VLDB*, 2000.

[103] V. J. Tsotras and N. Kangelaris, "The snapshot index: An i/o-optimal access method for timeslice queries," in *Information Systems*, vol. 20, no. 3, 1995, pp. 237–260.

[104] L. Gomez, B. Kuijpers, and A. Vaisman, "Querying and mining trajectory databases using places of interest," *AIS*, vol. 3, pp. 1–26, 2009.

[105] S.-Y. Hwang, Y.-H. Liu, J.-K. Chiu, and E.-P. Lim, "Mining mobile group patterns: A trajectory-based approach," in *PAKDD*, 2005, pp. 713–718.

[106] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma, "Mining interesting locations and travel sequences from gps trajectories," in *WWW*, 2009.

[107] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," in *ACM SIGMOD Rec.*, vol. 22, no. 2, 1993, pp. 237–246.

[108] M. R. Vieira, P. Bakalov, and V. Tsotras, "On-line discovery of flock patterns in spatio-temporal data," in *ACM GIS*, 2009, pp. 286–295.

[109] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, "Discovery of convoys in trajectory databases," *PVLDB*, vol. 1, no. 1, pp. 1068–1080, 2008.

[110] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle, "Reporting flock patterns," *Comput. Geom.*, vol. 41, no. 3, pp. 111–125, 2008.

[111] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches in query processing for moving object trajectories," in *VLDB*, 2000, pp. 395–406.

[112] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos, "Indexing spatiotemporal archives," in *VLDB*, vol. 15, no. 2, 2006, pp. 143–164.

[113] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *VLDB*, 2002, pp. 287–298.

[114] Y. Cai and R. T. Ng, "Indexing spatio-temporal trajectories with chebyshev polynomials," in *ACM SIGMOD*, 2004, pp. 599–610.

[115] S.-L. Lee, S.-J. Chun, D.-H. Kim, J.-H. Lee, and C.-W. Chung, "Similarity search for multidimensional data sequences," in *IEEE ICDE*, 2000, pp. 599–608.

[116] M. Vlachos, D. Gunopulos, and G. Kollios, "Discovering similar multidimensional trajectories," in *IEEE ICDE*, 2002, pp. 673–684.

[117] A. Anagnostopoulos, M. Vlachos, M. Hadjieleftheriou, E. J. Keogh, and P. S. Yu, "Global distance-based segmentation of trajectories," in *KDD*, 2006, pp. 34–43.

[118] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches in query processing for moving object trajectories," in *VLDB*, 2000, pp. 395–406.

[119] S. Arumugam and C. Jermaine, "Closest-point-of-approach join for moving object histories," in *IEEE ICDE*, 2006, p. 86.

[120] P. Bakalov, M. Hadjieleftheriou, and V. J. Tsotras, "Time relaxed spatiotemporal trajectory joins," in *ACM GIS*, 2005, pp. 182–191.

[121] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data indexed using the hilbert space-filling curve," in *SIGMOD*, 2001.

[122] A. R. Butz, "Alternative algorithm for hilbert's space-filling curve," in *IEEE Transactions on Computers*, 1971.

[123] G. Peano, "Sur une coubre qui remplit toute une aire plane," in *Mathematische Annalen*, vol. 36, no. 1, 1890, pp. 157–160.

[124] Y. Ohno and K. Ohyama, "A catalog of symmetric self-similar space-filling curves," in *Institute of Information Science, Keio*, 1991.

[125] H. Sagan, "On the geometrization of the peano curve and the arithmetization of the hilbert curve." in *International Journal of Mathematical Education*, 1992.

[126] J. Zhu, A. Hoorfar, and N. Engheta, "Bandwidth, cross-polarization, and feed-point characteristics of matched hilbert antennas," in *Antennas and Wireless Propagation Letters, IEEE*, vol. 2, no. 1, 2003, pp. 2–5.

[127] M. F. Mokbel, W. G. Aref, and I. Kamel, "Analysis of multi-dimensional space-filling curves," in *GeoInformatica*, vol. 3, no. 7, 2003, pp. 179–209.

[128] Y. Tao, D. Papadias, and J. Sun, "The tpr*-tree: An optimized spatio-temporal access method for predictive queries," in *VLDB*, 2003.

[129] M. Griebel and G. Zumbusch, "Parallel multigrid in an adaptive pde solver based on hashing and space-filling curves," in *Parallel Computing Systems and Applications*, vol. 25, no. 7, July 1999, pp. 827–843.

[130] http://www.donrelyea.com/hilbert_algorithmic_art_menu.htm.

[131] Y.-J. Gao, B. Zheng, G. Chen, Q. Li, C. Chen, and G. Chen, "Efficient mutual nearest neighbor query processing for moving object trajectories," in *Information Sciences - Elsevier*, vol. 180, no. 11, 2010.

[132] Y.-J. Gao, B. Zheng, G. Chen, and Q. Li, "Algorithms for constrained k-nearest neighbor queries over moving object trajectories," in *GeoInformatica*, vol. 14, 2010.