Continuous Monitoring of Nearest Trajectories

Dimitris Sacharidis
Institute for the Management of Information Systems
R.C. ATHENA
dsachar@imis.athena-innovation.gr

Dimitrios Skoutas
Institute for the Management of Information Systems
R.C. ATHENA
dskoutas@imis.athena-innovation.gr

Georgios Skoumas
Knowledge and Database Systems Laboratory
National Technical University of Athens, Greece
gskoumas@dblab.ntua.gr

ABSTRACT

Analyzing tracking data of various types of moving objects is an interesting research problem with numerous real-world applications. Several works have focused on continuously monitoring the nearest neighbors of a moving object, while others have proposed similarity measures for finding similar trajectories in databases containing historical tracking data. In this work, we introduce the problem of continuously monitoring nearest trajectories. In contrast to other similar approaches, we are interested in monitoring moving objects taking into account at each timestamp not only their current positions but their recent trajectory in a defined time window. We first describe a generic baseline algorithm for this problem, which applies for any aggregate function used to compute trajectory distances between objects, and without any restrictions on the movement of the objects. Using this as a framework, we continue to derive an optimized algorithm for the cases where the distance between two moving objects in a time window is determined by their maximum or minimum distance in all contained timestamps. Furthermore, we propose additional optimizations for the case that an upper bound on the velocities of the objects exists. Finally, we evaluate the efficiency of our proposed algorithms by conducting experiments on three real-world datasets.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

General Terms

Algorithms

Keywords

moving objects, nearest neighbors, continuous monitoring

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SIGSPATIAL'14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3131-9/14/11 ...$15.00
http://dx.doi.org/10.1145/2666310.2666408

1. INTRODUCTION

The increasingly widespread use of GPS enabled devices and other positioning technologies has made possible the tracking and monitoring of various types of moving objects, such as cars, people or animals. This has consequently lead to the study of a broad range of queries in a multitude of settings and applications. Retrieving objects whose motion is "similar" to that of a target query object is one of the most basic and useful analytical queries. In the literature, two important types of such moving object similarity queries have been proposed, the *Nearest Neighbor* (NN) and the *Nearest Trajectory* (NT), also known as trajectory similarity, queries. Both types define a similarity (or equivalently a distance) metric between moving objects, and return the top-k most similar (or equivalently least distant) objects with respect to a specified moving query object. The distinguishing characteristic is the definition of the metric. Generally speaking, NN queries, e.g., [5, 8], are concerned with the distance between *individual locations* of moving objects, i.e., at some particular time instance, whereas NT queries, e.g., [19, 21] take into account the distance between the *trajectories* of moving objects, i.e., for the sequence of locations over a time interval.

Both query types have been extensively studied for *historical* data, which can be stored on disk and indexed by specialized data structures. To the best of our knowledge, however, only NN queries have been considered in a *continuous monitoring* setting, where new object locations continuously arrive, and the result must be accordingly updated. This work introduces and studies *Continuous Nearest Trajectory* (CNT) queries. Given a *trajectory distance*, i.e., a metric aggregating individual location distances within a specified time window, a CNT query continuously returns the set of k objects that have the smallest trajectory distance to a given query object. CNT queries are a natural extension of both continuous NN queries, in the sense that the recent trajectory (and not only the last location) of objects is considered, as well as of historical NT queries, in that the result is computed and maintained in real-time.

We note that existing approaches do not extend for CNT queries. This is obvious for methods designed for historical data, as they take advantage of specialized index structures (which are not suitable for highly dynamic streaming data), and have the entire trajectory completely known upfront. Moreover, algorithms for continuous NN queries cannot be adapted for CNT. The main reason being that these methods assume that either the objects [34] or the query [31, 29, 17] is stationary, and define *validity* or *influence spatial regions*, which guarantee that the result will not change as long as the query object remains inside the region, in the former case, or that objects do not cross the region, in the latter case. Note that the latter methods can handle moving queries but only by treating them as new queries. This means that previous computations are

no longer useful and the result needs to be computed from scratch, a scenario which is only tolerable when the query object changes location infrequently. Therefore, a validity/influence region-based approach is not possible for CNT queries, where both the query object and the other objects move continuously and freely. However, we show that, in a specific setting (concerning the definition of the trajectory distance and assuming maximum velocities), it is possible to determine the minimum expected time when a moving object can influence the result.

There are some other works dealing with different types of continuous queries on moving objects, which however do not extend for CNT queries either. In a setting similar to ours, [2] defines a trajectory distance metric, and continuously computes the spatiotemporal trajectory join, i.e., determines pairs of objects whose trajectory distance does not exceed a given threshold. In other words, the underlying computation is answering a *range query* under a *hard threshold*, which is always easier to process as the search space is restricted. In contrast, the k-th trajectory distance in CNT queries is not known beforehand and can be arbitrarily high, making the methods of [2] inapplicable for CNT queries. Another work [27] proposes an online method to determine groups of objects that move close together, i.e., within a disk of a given radius. In their problem, only the distance between individual locations is taken into account and the threshold is also hard, making their ideas not suitable for CNT queries.

Given these observations, we propose a generic baseline algorithm, termed BSL, for processing CNT queries. This approach makes no assumptions regarding the underlying trajectory distance function or the movement of the objects. Thus, it serves as a framework for adapting and optimizing algorithms to more specific cases.

Building upon this, we derive an optimized algorithm, called XTR, for the cases where the trajectory distance between two moving objects is defined based on the extrema (maximum or minimum) of individual location distances. The maximum-defined CNT query establishes a distance guarantee that spans the time window within which trajectories are examined, and can be used, for example, to determine how far the nearest objects have strayed. The minimum-defined CNT query determines objects that have come close to the query at any time during the recent past, and can be thought of as a continuous NN query with "memory". On the other hand, using both the minimum and the maximum location distances, gives a more informative description of the movement of an object, as it determines the tightest annulus (donut) around the query that contains the object's trajectory.

Moreover, we study the aforementioned case when a global maximum velocity for the objects is known. This is a reasonable assumption given that most moving objects have upper bounds on their attainable velocities. For this particular setting, we introduce the HRZ algorithm, which computes distance bounds in order to determine the earliest possible time, termed *horizon*, when an object may influence the result.

Our main contributions can be summarized as follows:

- We introduce and formally define the problem of continuously monitoring the objects with the k-nearest trajectories to a given query object, where trajectory distances take into consideration the objects' recent locations.
- We present a generic baseline algorithm (BSL) for the problem, which defines the types of events and operations needed for the computation.
- We propose the XTR algorithm optimized for the case when the trajectory distance is determined by the maximum or minimum individual location distance between objects. We also discuss some other related trajectory distance definitions.
- We present the HRZ algorithm that introduces further optimizations assuming that the moving objects have bounded velocities.
- We experimentally evaluate the proposed algorithms using real-world datasets.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 formally defines the problem. Then, Sections 4–6 present our algorithms. Finally, Section 7 presents our experimental evaluation, and Section 8 concludes the paper.

2. RELATED WORK

We discuss various types of queries for moving objects, distinguishing between NN variants in Section 2.1 and NT methods in Section 2.2.

2.1 NN Queries on Moving Objects

Given a (stationary) query object location and a set of (stationary) object locations, the k-Nearest Neighbor (NN) query retrieves the k objects which are closer to the query location. There are many different ways to extend the NN query for moving objects during some time interval, evident by the rich bibliography on the subject.

A first classification is based on where the interval of interest lies with respect to the current time. If it is in the past, the queries are termed *historical*, as they concern stored trajectory data. If the interval is placed in the future, the queries are further classified into *predictive*, when it can be assumed that objects move in a known manner (i.e., with constant velocity, or along a line) and thus their future locations can be extrapolated, or *monitoring*, when no assumptions are made on the moving patterns and thus location updates are issued. Processing historical and predictive NN queries is generally less challenging compared to monitoring queries, because the former essentially have at query time the entire trajectories of the moving objects.

The second classification is based on the semantic of the NN query during an interval. A *snapshot* NN query reports the objects that are closest to the query object at *any* time instance within the interval; e.g., find the object that comes closest to some location within the next 10 minutes. A *continuous* NN query reports the objects that are closest to the query object at *every* time instance within the interval; e.g., report the objects that were at some time closest to the query object during the past 10 minutes. Note that in the data stream literature, the term continuous (or long standing) query [1] refers to the case when the result of a query must be continuously updated as streaming tuples arrive; in the context of NN queries, these requirements essentially correspond to the continuous monitoring NN query.

Regarding predictive queries, [13] presents a dual plane method for predictive snapshot NN queries, in the case that all objects move in 1-D space, or are restricted to move within the same segment (i.e., road). [23] studies continuous predictive variants for various spatial queries, including NN, and describe a method to return the initial result and its validity period (i.e., the time at which the result will change). [24] studies continuous predictive NN queries assuming that only the query is moving along a line, while all other objects are stationary. [10] and [3] also deal with continuous predictive NN queries, but they are able to handle updates on the motion patterns of objects, without computing the result from scratch.

For continuous monitoring NN queries, [22] and [34] handle the case when only the query object is moving. The former retrieves $m > k$ nearest neighbors hoping that the result at a future time is among these m objects, provided that the query does not move much. The latter returns a Voronoi-based validity region such that the result does not change as long as the query remains within the region. [31], [29] and [17] present incremental grid-based methods

for general continuous monitoring NN queries, i.e., when all objects move in a non-predictive manner; the last two works feature shared execution techniques to handle multiple NN queries.

In the case of historical trajectory data, R-tree based trajectory indices (e.g., 3D R-tree [26], TB-tree [20]) are typically used to expedite the NN query processing. [6] handles historical snapshot NN queries, while [5], [8] process historical continuous NN queries.

Another line of work concerns NN queries over uncertain data. For example, [25] processes continuous monitoring NN queries for objects with uncertain locations. [9] handles continuous predictive NN queries with updates for objects with uncertain locations and speeds. [18] deal with historical snapshot and continuous NN queries for objects with uncertain locations.

Finally, there has been some interest on identifying groups of moving objects, such as moving object clusters [12], flocks [7, 27], convoys [11], and swarms [15]. Generally speaking, these groups consist of objects that are close to each other (e.g., within a disk of a given radius) at each time instant. These methods however cannot be used for processing CNT queries.

2.2 Nearest Trajectories

There exist many approaches for defining distance (or similarity) metrics for trajectories. All of them also propose methods to identify the most similar trajectory to a given query trajectory, which can be extended to retrieve the top-k similar ones, but their techniques only operate on historical data. A useful survey on the topic is included in [19].

While the Euclidean distance (or some other L_p norm) is typically used to quantify closeness of two locations, the extension for the case of multiple locations within trajectories is not straightforward. In addition, a trajectory distance must take into account the temporal aspect of the locations. [30] defines the trajectory distance as the L_2 norm of individual Euclidean location distances, after re-sampling the trajectories to account for different reporting intervals. [16] ignores the temporal dimension and defines spatial trajectory distance as the average of the Euclidean distances computed between a location in one trajectory and its closest location in the other (termed the one way distance).

The previous trajectory distances can be computed in linear time with respect to the trajectory length. On the other hand, there exist more complex metrics, inspired from sequence similarity measures, that require quadratic time. [28] uses the Longest Common Subsequence (LCSS) similarity measure, an edit distance variant, that allows the matching of locations that are close in space at different time instants, provided that they are not far in time, and also allows for locations to be unmatched, e.g., accounting thus for location imprecisions or small deviations. In a similar manner, [4] defines the Edit Distance on Real Sequence (EDR) that captures the minimum number of edit operations (insert, delete, replace locations) necessary to transform one trajectory into the other.

An approach for finding historical top-k similar trajectories is presented in [21]. The basic algorithm prioritizes object examination aiming to avoid distance computations for objects not in the result. In addition, approximate techniques are also presented.

Another related problem is trajectory clustering, where the goal is to group trajectories based on a trajectory distance metric. For this problem, however, the basic underlying operation is typically a range query (retrieve trajectories within a given distance threshold) rather than a top-k similarity query. For example, in [14] the goal is to partition historical trajectories into sub-trajectories and then group them to construct dense clusters according to a metric that composes a perpendicular, a parallel, and an angle distance.

To the best of our knowledge, continuous monitoring of top-k similar trajectories has not been addressed in the past. The only work that handles continuous monitoring of a trajectory defined query is [2], which deals with spatiotemporal trajectory joins. The underlying trajectory distance metric is the maximum among all Euclidean location distances, and the goal is to find pairs of trajectories that are within a given trajectory distance threshold. That is the core query is a range rather than a top-k similarity query. Therefore, their approach is not applicable to our problem.

3. PROBLEM DEFINITION

Consider a set O of moving objects, whose locations are continuously monitored and reported at fixed discrete times, called *timestamps*. Location updates have the form $\langle o, t, x, y\rangle$, meaning that object o at timestamp t is at location $o[t] = (x, y)$. We assume that updates always arrive in increasing order of their timestamps, but we do not assume that for each timestamp updates are received for all objects.

We denote as $\mathcal{T}(o)$ the set of timestamps at which updates for o were received. For simplicity and without loss of generality we assume that for any timestamp t' for which no update for o was received, i.e. $t' \notin \mathcal{T}(o)$, the location of o is the same as its last reported location, i.e. $o[t'] = o[t]$, where $t \in \mathcal{T}(o)$ is the latest timestamp before t'. Essentially, this corresponds to assuming that object o has not moved during time $[t', t]$; making other assumptions can also be handled accordingly, e.g., by issuing artificial updates for the objects based on inferred locations.

Definition 1. The *location distance* between two objects o and o' at timestamp t is given by the Euclidean metric, i.e.,

$$d(o, o', t) = \sqrt{(x - x')^2 + (y - y')^2}.$$

where $o[t] = (x, y)$ and $o'[t] = (x', y')$ are the respective (reported or extrapolated) locations of o and o' at t.

The above definition measures the distance between two objects at a single timestamp. However, we are interested in comparing the recent trajectories of the objects, hence their distances over a series of consecutive timestamps within a specified time window. For this purpose, we introduce the following definition.

Definition 2. Given two objects o and o', a time window w, and an aggregate function $\mathcal{G}$, the *trajectory distance* of o and o' is defined by applying $\mathcal{G}$ on the location distances of o and o' at each timestamp within the time window of length w ending at timestamp t:

$$D(o, o', t, w, \mathcal{G}) = \mathcal{G}_{\tau \in [t-w,t]} d(o, o', \tau).$$

Function $\mathcal{G}$ can be any aggregate function, e.g., minimum, maximum, average, among others.

We now formally define the problem of continuously reporting the objects with the k-nearest trajectories to a moving query object.

Problem Statement. The *Continuous Nearest Trajectory* (CNT) query $\langle O, q, \mathcal{T}, k, w, \mathcal{G}\rangle$, where O is a set of moving objects, $q \in O$ a query object, and $\mathcal{T}$ a series of consecutive future timestamps, returns for each timestamp $t \in \mathcal{T}$ the k objects in O that have the smallest trajectory distance to q w.r.t. the time window w and the aggregate function $\mathcal{G}$, i.e., $\forall t \in \mathcal{T}$ it returns a subset $O_k \subseteq O$ of size k, such that $\forall o \in O_k, o' \in O \setminus O_k$:

$$D(q, o, t, w, \mathcal{G}) \leq D(q, o', t, w, \mathcal{G}).$$

4. BASELINE FOR CNT

We first describe a generic *baseline* (BSL) method for answering continuous nearest trajectory queries. BSL operates under *any* ag-

Algorithm 1: BSL

```
1  foreach t ∈ T do
2      O_A ← ∅ // the set of affected objects at t
3      if QUpd then
4          q.loc ← (x_q, y_q) // update q's current location
5          O_A ← O // mark all objects for processing
6      else
7          foreach OUpd and OExp do
8              O_A ← O_A ∪ o // add the referred object in O_A
9      foreach o ∈ O_A do
10         D_o ← BSL_ProcessObject(o)
11         if D_o has changed then
12             if o in the results R then
13                 update o's entry in R
14             else if D_o smaller than the trajectory distance of R's last entry
               then
15                 delete R's last entry
16                 insert an entry for o in R
17     report t, R
```

gregate function $\mathcal{G}$ and follows an event driven process, where the events to be handled are specified below:

- *Query location updates* (*QUpd*). This is an update $\langle q, t, x, y\rangle$ to the location of the query object q, specifying its new location (x, y) for the current timestamp t. This may result in changes in the trajectory distances of the objects, and subsequently changes in the current set of nearest trajectories (NTs). When objects are allowed to move arbitrarily, their new distances to the new query location have to be computed, and the new aggregate distances and NTs have to be evaluated.
- *Object location updates* (*OUpd*). This is an update $\langle o, t, x, y\rangle$ to the location of an object o, specifying its new location (x, y) for the current timestamp t. As a result, the current location distance of the object to the query has to be evaluated, which may affect its trajectory distance within the window w. If this changes, it may in turn affect the inclusion or not of the object in the result set. In addition, the system needs to remember to purge this location distance when it becomes obsolete, i.e., concerns a location outside the window. Therefore, it generates a corresponding expiration event that will be triggered at timestamp $t + w$ as described next.
- *Object distance expiration* (*OExp*). Unlike *QUpd* and *OUpd*, which are events received by the external environment, *OExp* events are generated and triggered by the system as part of handling *OUpd* events. *OExp* events have the form $\langle o, t\rangle$, and mean that a location distance for object o is set to expire at timestamp t (this location distance was computed for a location update received at timestamp $t - w$). Similarly to a location update, such a removal may affect the trajectory distance of the object, and consequently its inclusion in the set of NTs.

In the following, we describe in detail how BSL handles the above events to evaluate a CNT query.

BSL makes use of the following in-memory data structures. For the query, it only stores its latest location $q.loc$. For each object, it stores its latest location $o.loc$, as well as a list $o.hist$ of location distances and their corresponding timestamps, ordered by time. In addition, it uses an event queue Q to store and process *OExp* location distance expiration events, i.e., for removing distances for timestamps outside the time window w. An *OExp* event $\langle o, t\rangle$ means that at time t, BSL needs to purge an expired distance for object o. This is the least recent location distance in the list $o.hist$. Events in Q are inserted and processed in a FIFO manner, i.e. they are ordered by time. Finally, BSL maintains a results list R of size k, where each entry corresponds to an object and its trajectory distance, updated at every timestamp. Any object that does not appear in the list at time t has trajectory distance not less than the largest trajectory distance in R.

Algorithm 2: BSL_ProcessObject

```
1  if OExp event for o was triggered then
2      remove the expired location distance from o.hist
3  if QUpd or OUpd event for o was received then
4      update o.loc, if changed
5      compute new location distance d
6      add d to o.hist
7      create OExp event for o at timestamp t + w
8  update trajectory distance D_o
9  return D_o
```

Algorithm 1 shows the pseudocode for BSL. Since this is a continuous query, BSL executes in a loop for every timestamp $t \in \mathcal{T}$ (line 1), i.e. as long as the query is standing, and at each iteration it reports the current result set R (line 17). The input at each timestamp is the set of *QUpd*, *OUpd*, and *OExp* events that have been received for processing. Based on these events, BSL determines the set of *affected objects* O_A that require processing at this timestamp (line 2). Note that the set O_A contains not only objects that have received location updates, but also objects for which an expiration event was triggered, or, in the case of a query update, all objects.

If the query object has moved to a new location, then $q.loc$ is updated and new object distances need to be computed (lines 3–5). Otherwise, only those objects for which an update or expiration happened are marked for processing (lines 6–8). Subsequently, each affected object $o \in O_A$ is processed (lines 9–17). First, the procedure `BSL_ProcessObject` is invoked (line 10), which updates $o.loc$ and $o.hist$ accordingly, and recomputes the object's location distance and trajectory distance (see Algorithm 2 below). Then, BSL checks if the returned trajectory distance has changed (line 11). If so, then the result set R may need updating. In particular, if o was in the result, then its entry in R must be updated (lines 12–13) with the new trajectory distance. Otherwise, if the new trajectory distance is smaller than any trajectory distance in R, this means that o should be (tentatively) inserted in R, evicting the last entry (lines 14–16).

We next describe the procedure `BSL_ProcessObject`, shown in Algorithm 2, in more detail. If an expiration event has occurred for o, then the expired location distance is removed from $o.hist$ (lines 1–2). If the object's location has changed, $o.loc$ is updated (line 4). The new location distance of o is computed and added to the history (lines 5–6). Moreover a corresponding expiration event is added in Q (line 7). Finally, the new trajectory distance for o is computed and returned (lines 8–9).

5. EXTREMA-DEFINED CNT

In the following, we assume that the aggregate function $\mathcal{G}$ defining the trajectory distance is $\max$ or $\min$ over location distances, or, more generally, any other function taking as input only the extrema ($\max$, $\min$) location distances. In these instances, the trajectory distance is determined by one (or two) location distances within the time window. Note, however, that these location distances may change over time, as new locations arrive and old ones expire. Nonetheless, we show that processing of extrema-defined CNT queries can be streamlined. We first start our discussion considering the case of the $\max$ function; the case of $\min$ can be handled in a similar manner, and is hence omitted. We then discuss the necessary changes to process CNT queries for any extrema-defined aggregate function.

Algorithm 3: XTR_ProcessObject

```
1  if OExp event for o was triggered then
2      remove the expired location distance from o.hist
3  if QUpd or OUpd event for o was received then
4      update o.loc, if changed
5      compute new location distance d
6      add d to o.hist
7      remove from o.hist all location distances less than d
8      t' ← the earliest timestamp in o.hist
9      if Q contains OExp event for o then
10         update OExp's time to t' + w
11     else
12         insert in Q the event ⟨o, t' + w⟩
13 return D_o ← earliest location distance in o.hist
```

When the aggregate function $\mathcal{G}$ is $\max$, the trajectory distance of an object o is determined by the largest location distance within the time window w. In that case, we show that it is possible to discard some location distances which cannot influence the trajectory distance during their lifespan. Based on this observation, we describe the *Extrema* (XTR) algorithm, which is based on the BSL framework but reduces the number of location distances stored per object, and, consequently, the number of events generated and processed. The key observation of XTR is captured by the following lemma.

Lemma 1. Given an object o, where $d < d'$ are two location distances at timestamps $t < t'$ for $t' - t \leq w$, the location distance d does not contribute to the trajectory distance of o for any timestamp after t'.

Proof. Location distance d is valid, i.e., may contribute to the trajectory distance, during its lifespan ending at timestamp $t+w$. During the time interval $[t', t+w]$, location distance d' is also valid and greater, and thus dominates d. As a result, the trajectory distance, i.e., the maximum location distance, must be at least $d' > d$. □

The XTR algorithm uses the same data structures and variables as BSL and performs the same main operations described in Algorithm 1. However, XTR differs from BSL in the way it processes objects. In particular, we discern the following main differences. First, XTR only keeps the non-dominated location distances in $o.hist$, as Lemma 1 suggests. Second, at any time t, the event queue Q contains only a single entry per object o, and its semantics can be viewed differently: it now schedules trajectory distance recomputations rather than location expirations. By purging a priori those earlier location distances that are smaller than d, XTR avoids unnecessary triggering of the corresponding *OExp* events, thus avoiding unnecessary processing of objects whose trajectory distance cannot yet change.

The processing for an object o in XTR is handled by the procedure `XTR_ProcessObject` outlined in Algorithm 3. Its first tasks, removing expired location distances, updating the object's location, and recomputing the object's location distance to the query, are identical to BSL's (lines 1–6). In addition, based on Lemma 1, XTR removes any location distances less than d from $o.hist$ (line 7). Let t' be the earliest timestamp that remains (line 8). XTR inserts in Q an event to expire the location distance at t', if no event for o in Q already exists, otherwise it resets the scheduled time of the existing event (lines 9–12). This event essentially schedules the next trajectory distance recomputation necessary for o (assuming that the trajectory distance is not affected by newer location updates until then). Finally, the trajectory distance is set to the earliest location distance and returned (line 13).

We now discuss the general case where the aggregate function $\mathcal{G}$ is some function over the extrema ($\min$ and $\max$) location distances. One example of such a function is the average of the minimum and maximum location distances recorded for an object within the current time window. Recall that Lemma 1 identifies location distances which are irrelevant for the $\max$ case; an analogous lemma holds for the $\min$ case. Therefore, when both the $\max$ and the $\min$ location distance contribute to the trajectory distance, we can discard location distances which are irrelevant for both extrema cases. Following this observation, we propose the following changes to the `XTR_ProcessObject` algorithm. For each object o, we maintain its minimum and maximum location distances for the current time window, denoted as $o.min$ and $o.max$, respectively, i.e. $o.min = \min\{o.hist\}$ and $o.max = \max\{o.hist\}$. In addition, we keep a time marker t_m which is the earliest timestamp of either $o.min$ or $o.max$. In the event queue Q, we still need to keep only one entry for each object o, set to $\langle o, t_m + w\rangle$, to trigger a reevaluation of its trajectory distance when either $o.min$ or $o.max$ expires. Moreover, the early removal of unnecessary entries in $o.hist$ is now done as follows. When $o.min$ or $o.max$ changes, and t_m is set accordingly, we remove all entries from $o.hist$ with timestamp earlier than t_m. The reason for this is that for any location distance d with timestamp $t < t_m$ it holds that $o.min < d < o.max$ (otherwise, d would be the current min or max) and d cannot become a future $o.min$ or $o.max$ since it expires before them.

6. EXPLOITING BOUNDED VELOCITIES FOR EXTREMA-DEFINED CNT

This section considers extrema-defined trajectory distances and assumes that there exists a global upper bound v_{max} on the velocity of a moving object[1]. Under this realistic assumption, we show that it is possible to derive a more efficient algorithm than XTR for processing CNT queries. The proposed *Horizon* (HRZ) algorithm takes advantage of the velocity bound to further reduce the number of location updates that need to be processed. Similar to Section 5, we assume that the trajectory distance is the maximum location distance within the time window; the case of $\min$ is similar, while the more general case of extrema-defined functions can be handled in a straightforward manner.

The basic idea behind HRZ is the following. For ease of exposition, assume $k = 1$ and consider two objects o and o'. Let $\underline{D}[t]$ and $\overline{D}'[t]$ denote, respectively, a lower and an upper bound on the trajectory distances of o and o' to the query q at time t. Clearly, if $\underline{D}[t] > \overline{D}'[t]$ for any timestamp t within a time interval, then o cannot be in the result during that interval. Hence, in Section 6.1, we derive lower and upper bounds on trajectory distances. Then, in Section 6.2, we discuss the computation of the time horizon, which determines a time interval during which a particular object may not be a result. Finally, in Section 6.3, we put our ideas together and present the HRZ algorithm.

6.1 Bounds on Trajectory Distances

Let t be the current timestamp, and consider an object o for which the most recent location distance is d received at timestamp $t_d \leq t$, and its current trajectory distance is $D \geq d$, valid since timestamp $t_D \leq t_d$. A lower bound for the trajectory distance of o at any future timestamp $t' > t$ can be computed assuming that object o moves at maximum velocity v_{max} towards the query q, while q also moves at maximum velocity v_{max} towards o. As a result, since the last known update at t_d, the location distance of o to q is decreasing at a maximum rate of $2v_{max}$. Notice however that this will affect its trajectory distance only after both D and d have

[1]Note that the extension to differing maximum velocities across objects is straightforward and thus omitted.

expired. The trajectory distance in this setting is clearly a lower bound for the trajectory distance of o for any possible motion of o and q. Thus, we derive the following lemma.

Lemma 2. Given an object o at current timestamp t, with latest location distance d at timestamp $t_d \leq t$ and current trajectory distance $D \geq d$ valid since $t_D \leq t_d$, its trajectory distance for any future timestamp $t' \geq t$ is lower bounded by the function:

$$\underline{D_o}[t'] = \begin{cases} D & \text{if } t \leq t' \leq t_D + w \\ d & \text{if } t_D + w < t' \leq t_d + w \\ \max\{d - 2v_{max} \cdot (t' - t_d), 0\} & \text{if } t' > t_d + w. \end{cases}$$

Proof. First, observe that at time t the history of the object (i.e., during the time interval $[t - w, t]$) certainly contains a location distance with value D at time t_D (determining the current trajectory distance) and another with value d at time t_d. It may also contain other location distances, which however must have values between d and D. Consequently, it is easy to see that the lemma holds for the first two clauses.

Regarding the third clause, we need to show that for any future timestamp $t' > t$, the lower bound holds. Consider the location distances valid during the future time window $[t' - w, t']$; recall that location distance d is no longer valid. Let d_m be the largest valid location distance with timestamp $t_m \in [t' - w, t']$. Therefore, the trajectory distance at time t' is defined as d_m. Due to the bound on the velocity of objects, it holds that any object, o or the query q, from timestamp t_d (of o's known location update in the past) up to timestamp t_m cannot have traveled a distance greater than $v_{max} \cdot (t_m - t_d)$. As a result, the location distance of o cannot have decreased more than $2v_{max} \cdot (t_m - t_d)$ (but not become less than zero), which is the case that o and q travel towards one another (and travel together once they reach each other). Therefore, the location distance at t_m cannot be less than $d_m \geq d - 2v_{max} \cdot (t_m - t_d)$, and is also greater than zero. Since $t_m \leq t'$, the lower bound holds. □

In a similar way, we can also derive an upper bound for the trajectory distance of o in a future timestamp t'. This can be computed assuming that object o moves at maximum velocity v_{max} away from the query q, while also q moves at maximum velocity v_{max} away from o. As a result, since the last known update at t_d, the location distance of o to q is increasing at a rate of $2v_{max}$. Again, any updates will come into effect only as long as there exists no previous value that is still valid and greater. The trajectory distance in this setting is clearly an upper bound for the trajectory distance of o for any possible motion of o and q. Thus, we derive the following lemma.

Lemma 3. Given an object o at timestamp t, with latest location distance d at timestamp $t_d \leq t$ and current trajectory distance $D \geq d$ valid since $t_D \leq t_d$, its trajectory distance for any future timestamp $t' \geq t$ is upper bounded by the function:

$$\overline{D_o}[t'] = \begin{cases} \max\{D, d + 2v_{max} \cdot (t' - t_d)\} & \text{if } t \leq t' \leq t_D + w \\ d + 2v_{max} \cdot (t' - t_d) & \text{if } t' > t_D + w. \end{cases}$$

Proof. Consider the first clause, and a timestamp $t' \in [t, t_D + w]$; the corresponding time window is $[t' - w, t']$ and D is still valid. Let d_m denote the largest valid location distance with timestamp $t_m \in [t' - w, t']$. Trivially, if d_m is D, the upper bound holds. Assume otherwise, i.e., $d_m > D$. Using similar reasoning as in Lemma 2, the location distance of o from timestamp t_d up to timestamp t_m cannot have increased more than $2v_{max} \cdot (t_m - t_d)$. Therefore, $d_m \leq d + 2v_{max} \cdot (t_m - t_d)$, and the upper bound also holds for this case because $t_m \leq t'$. The second clause is proved in a similar way, given that D has now expired. □

6.2 Time Horizon of Objects

We now proceed to derive the minimum time required for an object $o \notin R$ to enter the result set. We refer to this as the *time horizon* of an object, corresponding to the earliest time for which the object's trajectory distance may become equal to (or less than) the trajectory distance of some object in R. Using Lemmas 2 and 3, the time horizon is formally defined as follows.

Definition 3. Given the current result set R at timestamp t, the *time horizon* t_h for an object $o \notin R$ is defined as the earliest possible time that the trajectory distance of o becomes lower than that of any object in R, i.e.:

$$t_h = \min\{t' \geq t \mid \exists o' \in R : \underline{D_o}[t'] \leq \overline{D_{o'}}[t']\}$$

An important remark regarding the previous definition is that it does not suffice to just consider the trajectory distance upper bound of the k-th object in R. As location updates may not occur at all timestamps, it is possible for two objects $o_i, o_j \in R$ with trajectory distances $D_i < D_j$ to have at some future timestamp t' upper trajectory bounds such that $\overline{D_i}[t'] > \overline{D_j}[t']$. This can occur, for example, when the objects' last location distances and timestamps satisfy the conditions $d_i > d_j$ and $t_i < t_j$.

As a result, computing the time horizon for an object requires considering the trajectory distance upper bounds for all objects in R, which is time consuming given that the time horizon needs to be computed at each timestamp for each affected object not in the result set. We thus propose an alternative method for determining the time horizon. The key idea is the following lemma, which derives a single upper bound on the trajectory distance of any object in the result set R.

Lemma 4. Consider a set of objects R at current timestamp t, where, for the i-th object, d^i denotes its latest location distance at timestamp t_d^i and $D^i \geq d^i$ denotes its current trajectory distance valid since timestamp $t_D^i \leq t_d^i$. Define object $o^+ \in R$ to be the one with the largest trajectory distance, and object $o^* \in R$ to be the one that can have the largest possible location distance at current timestamp t, i.e.,

$$o^+ = \underset{o_i \in R}{\operatorname{argmax}} D^i \text{ and } o^* = \underset{o_i \in R}{\operatorname{argmax}} \left(d^i + 2v_{max} \cdot (t - t_d^i) \right).$$

Then, the trajectory distance of any object in R for any future timestamp $t' \geq t$ is upper bounded by the function:

$$\overline{D_R}[t'] = \begin{cases} \max\{D^+, d^* + 2v_{max} \cdot (t' - t^*)\} & \text{if } t \leq t' \leq t + w \\ d^* + 2v_{max} \cdot (t' - t^*) & \text{if } t' > t + w, \end{cases}$$

where D^+ is the trajectory distance of o^+, and d^* is the latest location distance of o^* computed at timestamp t^*.

Proof. It suffices to show that the upper bound on the trajectory distance of each object in R according to Lemma 3 is always (i.e., for any $t' > t$) not greater than the upper bound provided by this lemma. Consider an object $o_i \in R$ and its trajectory distance upper bound:

$$\overline{D^i}[t'] = \begin{cases} \max\{D^i, d^i + 2v_{max} \cdot (t' - t_d^i)\} & \text{if } t \leq t' \leq t_D^i + w \\ d^i + 2v_{max} \cdot (t' - t_d^i) & \text{if } t' > t_D^i + w. \end{cases}$$

First note that $t_D^i < t$, and consider a future timestamp t' during the time interval $[t, t_D^i + w]$. Comparing the first clause of the two bounds, we can see that $D^+ \geq D^i$ from the definition of object o^+. On the other hand, from the definition of o^* we derive that $d^* + 2v_{max} \cdot (t - t^*) \geq d^i + 2v_{max} \cdot (t - t_d^i)$. Adding $2v_{max} \cdot (t' - t)$ to both sides of the inequality, we derive that the lemma holds.

Algorithm 4: HRZ

```
1  foreach t ∈ T do
2      O_A ← ∅ // the set of objects marked for processing at t
3      if QUpd then
4          q.loc ← (x_q, y_q) // update q's current location
5          O_A ← O // mark all objects for processing
6      else
7          foreach OUpd and OExp do
8              O_A ← O_A ∪ o // add the referred object in O_A
9      foreach o ∈ O_A ∩ R do
10         D_o ← HRZ_ProcessObject(o)
11         update o's entry in R
12     identify objects o+ and o* in R // from Lemma 4
13     foreach o ∈ O_A \ R do
14         if t < o.t_h − w then continue
15         D_o ← HRZ_ProcessObject(o)
16         if D_o smaller than the trajectory distance of R's last entry then
17             delete R's last entry
18             insert an entry for o in R
19     report t, R
```

Algorithm 5: HRZ_ProcessObject

```
1  if OExp event for o was triggered then
2      remove the expired location distance from o.hist
3  if QUpd or OUpd event for o was received then
4      update o.loc, if changed
5      compute new location distance d
6      if o ∉ R then
7          compute t_h
8      else t_h ← t
9      if t < t_h − w then
10         clear state of o
11         remove o's entry in Q
12         return D_o ← ∞
13     else
14         add d to o.hist
15         remove from o.hist all location distances less than d and with
           timestamps before t − w
16         if t < t_h then
17             return D_o ← ∞
18         else
19             t' ← the earliest timestamp in o.hist
20             if Q contains OExp event for o then
21                 update OExp's time to t' + w
22             else
23                 insert in Q the event ⟨o, t' + w⟩
24 return D_o ← earliest location distance in o.hist
```

Next, consider a future timestamp t' during the time interval $[t^i_D + w, t + w]$, and compare the second clause of $\overline{D^i}[t']$ to the first clause of $\overline{D_R}[t']$. With similar reasoning as before, we have that $d^* + 2v_{max} \cdot (t' - t^*) \geq d^i + 2v_{max} \cdot (t - t^i_d)$, and since the first clause of $\overline{D_R}[t']$ is always greater than the left-hand side of the inequality, the lemma holds.

In the case of a future timestamp $t' > t + w$, when the second clauses of the bounds apply, it is easy to see, using similar reasoning as before, that the lemma holds. □

Using the bound on the trajectory distance of any object in R, it is possible to efficiently compute a timestamp that never overestimates the time horizon, as the next lemma suggests.

Lemma 5. Given the current result set R at timestamp t, the time horizon t_h for an object $o \notin R$ is not less than the following value:

$$t_h \geq \min\{t' \geq t \mid \underline{D_o}[t'] \leq \overline{D_R}[t']\}.$$

Proof. Denote as A the set from Definition 3, i.e., $A = \{t' \geq t \mid \exists o' \in R : \underline{D_o}[t'] \leq \overline{D_{o'}}[t']\}$, and as B the set from this lemma, i.e., $B = \{t' \geq t \mid \underline{D_o}[t'] \leq \overline{D_R}[t']\}$. We claim that $B \subseteq A$ to prove the lemma. Since it holds that $\overline{D_R}[t'] \geq \overline{D_{o'}}[t']$ for any $o' \in R$ from Lemma 4, the condition of set B is harder to satisfy than that of A, and thus the claim $B \subseteq A$ holds. □

Lemma 5 suggests that we can compute, in constant time, a timestamp not greater than the time horizon as the solution of the equation $\underline{D_o}[t'] = \overline{D_R}[t']$. Henceforth, to simplify the presentation of HRZ, whenever we refer to the time horizon t_h or its computation, we mean the solution of this equation instead of Definition 3.

6.3 The HRZ Algorithm

Having a method to compute the time horizon of an object, we next detail the HRZ algorithm, highlighting its differences with respect to XTR. The data structures and variables that HRZ uses are as in XTR, with the exception that for each object HRZ additionally stores its time horizon t_h indicating the time after which the object may appear in the result set R. The computation of t_h is based on Lemma 5. The HRZ algorithm takes advantage of the time horizon to reduce the number of events processed as follows. At any timestamp before $t_h - w$, HRZ ignores updates for the particular object. During the time interval $[t_h - w, t_h]$, HRZ only stores the locations and location distances, since these are necessary to compute the trajectory distance at time t_h. However, it does not compute the trajectory distance, since it is guaranteed to be greater than those in R, and it does not add any events in Q. After the time horizon t_h, HRZ operates similar to XTR.

Algorithm 4 shows the pseudocode for HRZ. The main difference from BSL and XTR is that it handles the processing of affected objects in two phases. In the first phase (lines 9–11), HRZ considers only objects that are in R, i.e., objects that were reported as results in the previous timestamp. For these objects, the processing (handled by HRZ_ProcessObject) is essentially identical to XTR, as we later explain. Once processing is completed, the object's entry in R is updated if its trajectory distance changed.

Between the first and second phase, HRZ scans all objects in R, and determines objects o^+ and o^* as defined in Lemma 4 (line 12). Then, during the second phase (lines 13–18), HRZ considers the remaining affected objects, i.e., not in R. If the current time is more than w timestamps before the time horizon $o.t_h$ of an object o, HRZ essentially ignores o (line 14). For each other affected object, its processing (handled by HRZ_ProcessObject at line 15) differs significantly from XTR. Once it concludes, HRZ checks whether the object should be included in the result set R provided that its trajectory distance has sufficiently decreased (lines 16–18).

We next describe the HRZ_ProcessObject procedure, shown in Algorithm 5. As in XTR, the procedure removes expired location distances if an event from Q was triggered (lines 1–2). The main operations of the procedure occur when either an object or a query location update were received (lines 3–23). First, the object's location is updated, if it changed, and its location distance is computed (lines 4–5). If the object o under processing did not belong in the result at the previous timestamp (line 6), the procedure computes the time horizon t_h by applying Lemma 5 (line 7); otherwise t_h is set to current time (line 8), meaning that object o may belong in the result. Since the time horizon is now recalculated taking into account the object's current location distance, it is necessary to check again if the object should be ignored (line 9). If the check succeeds, all stored information for object o is cleared, its entry in the event queue is removed and an infinite trajectory distance is returned (lines 10–12).

In the following operations (lines 14–23), it holds that the current time is $t \geq t_h - w$, hence HRZ needs to store locations and location distances. The object's current location distance d is stored

(line 14), and all location distances less than d are removed (line 15) as in XTR. Subsequently, if the current time falls in the interval $[t_h - w, t_h]$ (line 16), finding the actual trajectory distance during this interval is not necessary, as the object is guaranteed to not be in the result set. Therefore, HRZ simply returns an infinite trajectory distance (line 17) and, to increase efficiency, it does not create a corresponding expiration event. A consequence is that at future timestamps after the current time horizon, there may exist expired location distances. Therefore, the procedure may also have to remove such distances (line 15). Otherwise, if the current time is not before the time horizon (line 18), the processing is identical to XTR. That is, the earliest timestamp is identified, and the event queue is properly updated (lines 19–22). The last operation is to compute the trajectory distance from the earliest location distance and return it (line 24). As a final note, observe that if the object was not in the result at the previous timestamp, its processing is identical to XTR, as its time horizon is set to current time (line 8).

7. EXPERIMENTAL EVALUATION

To evaluate the efficiency of the proposed algorithms for the continuous nearest trajectories query, we conduct experiments using three real-world trajectory datasets. In the following, we first present the datasets used for the evaluation and then we report the results of our experiments.

7.1 Datasets

To cover a variety of cases regarding the shapes of trajectories, the type of the objects, and the speed and type of movement, we use three different real-world datasets in our experiments. We refer to these datasets as *Beijing taxis*, *Aegean ships*, and *Athens vehicles*. These datasets vary in their characteristics, ensuring that our methods are robust across diverse settings. For example, in the *Beijing taxis* dataset, the shape of the trajectories exhibits a relatively high regularity due to the grid-like structure of the underlying road network. At the other end, the Athens road network is highly irregular, resulting in diverse trajectories with constantly varying headings. Finally, the *Aegean ships* trajectory dataset comprises relatively long trajectories with medium degree of heading variations.

A typical issue in trajectory datasets is the often high variation of the sampling rate, caused, for example, by weak GPS signal, or when the user manually switches off their personal tracking devices (e.g., to save battery or for privacy). In our datasets, to reduce such gaps, when the time interval between two consecutive reported locations exceeds a specified threshold (set to 30 seconds) but is not greater than a maximum threshold (set to 120 seconds), we use linear interpolation to create intermediate location updates.

We next detail the used datasets.

- *Beijing taxis.* These trajectories are from the T-Drive trajectory dataset, which contains GPS tracking data from taxis moving in the area of Beijing [32, 33]. A total of 1,023,924 trajectories are used. These trajectories belong to a total of 569 taxis recorded in the period 2/2/2008 – 4/2/2008. Each trajectory comprises on average 3,017 points (i.e. location updates).
- *Aegean ships.* This dataset contains GPS tracking data from ships moving in the Aegean sea[2]. A total of 986,275 trajectories are used, obtained from 887 ships in the period 31/12/2008 – 02/01/2009. On average, each trajectory comprises 1,101 points.
- *Athens vehicles.* This dataset contains GPS tracking data from vehicles moving in the area of Athens, recorded in the context of the SimpleFleet project[3]. 667,421 trajectories are used, coming from 2,497 vehicles on 01/10/2012. Each trajectory comprises 157 points on average.

[2]http://www.chorochronos.org/?q=node/8

[3]http://www.simplefleet.eu/

7.2 Results

The goal of the experimental evaluation is to study the efficiency of the proposed algorithms, and in particular to compare the speedup achieved by the XTR and HRZ algorithms with respect to the more generic baseline BSL algorithm. For this purpose, we conduct a series of experiments, using the datasets previously described. The trajectory distance metric used in all experiments is the maximum of all valid location distances. We note that the performance of BSL is identical for all metrics, as the method is distance agnostic. On the other hand, XTR and HRZ have roughly the same performance for any extremum-defined trajectory distance metrics.

The main performance metric is the total execution time, i.e., the time spent processing a CNT query over its entire lifespan. To better investigate the performance gains of XTR and HRZ with respect to BSL, we also report their relative improvement in execution time, and the percentage of events (location updates and expirations) that they process compared to BSL. The investigated parameters affecting the performance of the algorithms is the number k of nearest trajectories requested, the size w of the time window, and the number $|O|$ of objects. In all settings, the reported performance metrics (time and number of events) are the average of 10 executions involving randomly selected query objects. The answer to a CNT query is calculated at each timestamp that an update or an expiration event occurs.

7.2.1 Varying the number of nearest trajectories

In this experiment, we measure the total execution time of each of the three algorithms with respect to the number k of nearest trajectories returned. The total monitoring time $\mathcal{T}$ is set to 60 minutes, and the size w of the time window for keeping each object's history is set to 5 minutes. The results are presented in Figure 1.

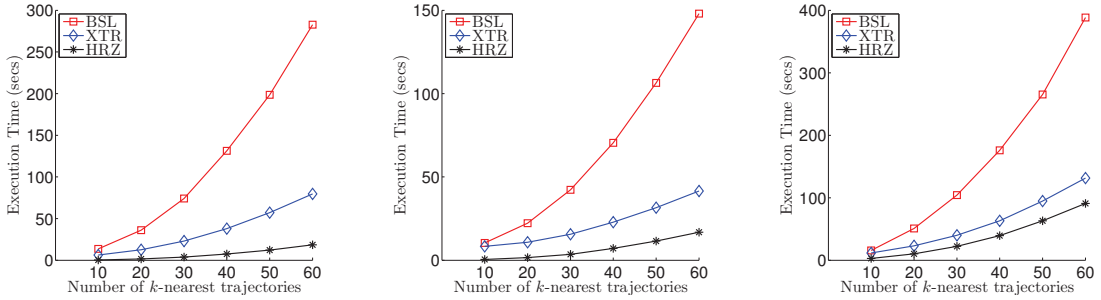
The first important observation is that for all datasets, the execution times of both XTR and HRZ are significantly lower than for BSL, clearly showing in practice the effectiveness of the corresponding optimizations for these cases. Furthermore, HRZ has also a clear benefit over XTR. The differences are more pronounced in the *Beijing taxis* dataset, which shows that, due to the regularity in the movement of objects imposed by the underlying grid-like structure of the Beijing road network, more effective pruning of location updates and distance recomputations can be achieved. In contrast, the differences become relatively smaller in *Athens vehicles*, where the road network is less uniform.

A second observation is that for all algorithms the execution time increases with k. This is expected since k regulates the size of the ordered list R that has to be maintained by the algorithm at each timestamp. However, this increase is lower for XTR and, even more so for HRZ, which is an additional evidence that XTR and HRZ need to process fewer events, and hence perform fewer lookup and sort operations on R.

7.2.2 Varying the size of the time window

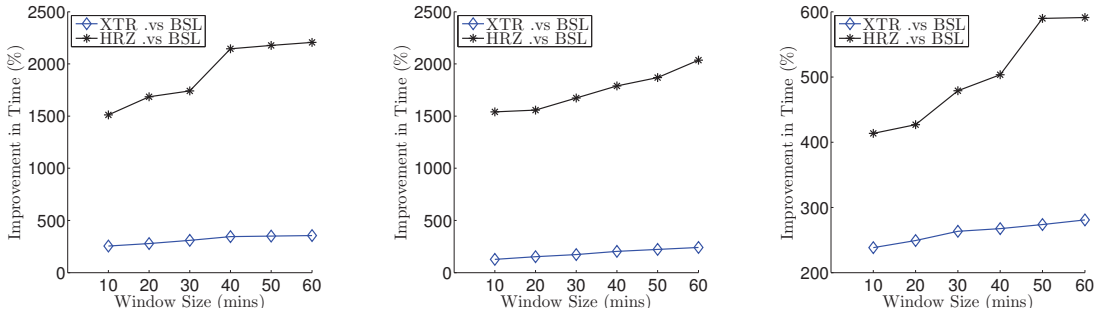
In the next experiment, we compare the execution time of the three algorithms with respect to the window size w during which the past location distances of an object remain valid and contribute to the trajectory distance. As previously, the total monitoring time $\mathcal{T}$ was set to 60 minutes, and k was set to 10. To better illustrate the improvement in execution time achieved by XTR and HRZ with respect to BSL, we plot the speedup of XTR and HRZ compared to BSL. The results are shown in Figure 2.

As illustrated, XTR shows a speedup of almost up to 5 times over



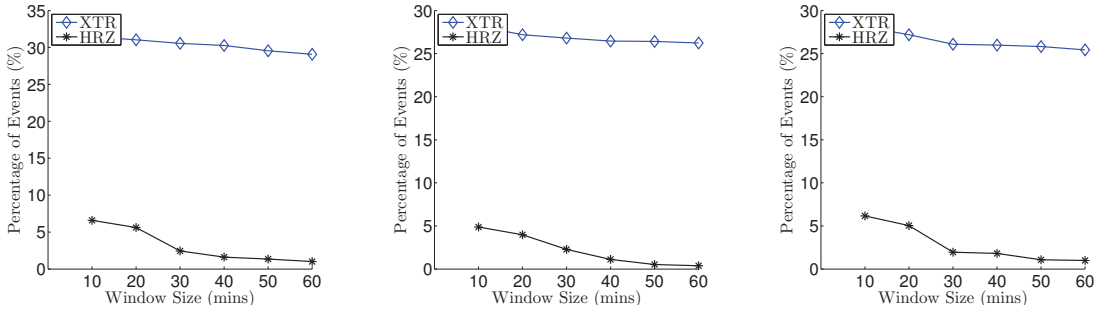
(a) Beijing taxis (b) Aegean ships (c) Athens vehicles

Figure 1: Execution time of BSL, XTR and HRZ w.r.t. the number k of nearest trajectories returned at each timestamp.



(a) Beijing taxis (b) Aegean ships (c) Athens vehicles

Figure 2: Execution time speedup of XTR and HRZ compared to BSL w.r.t. the size w of the time window.



(a) Beijing taxis (b) Aegean ships (c) Athens vehicles

Figure 3: Percentage of events handled by XTR and HRZ compared to BSL w.r.t. the size w of the time window.

BSL, while for HRZ it is even higher, in the range of 15× to 22× for the first two datasets and 4× to 6× for the *Athens vehicles*. Notice that in this setting $k = 10$, so when these results are considered in conjunction with those illustrated in Figure 1, these speedups are expected to be increasingly higher for higher values of k.

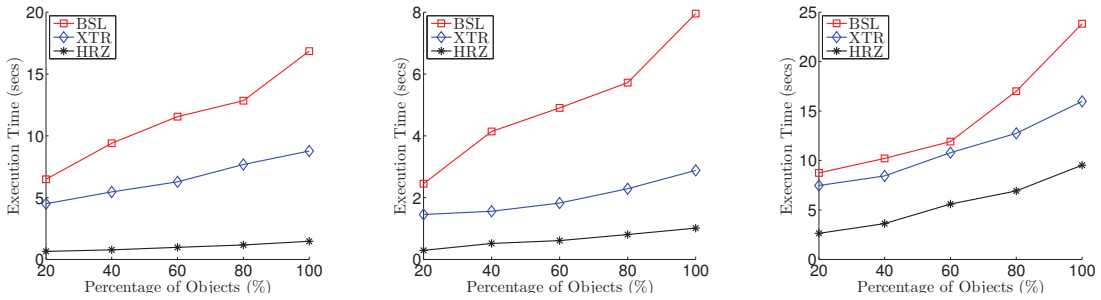
Moreover, the speedup for both algorithms increases as the window size w increases. This behavior is because XTR and HRZ only consider the maximum or minimum value in each object's history, so the gain is higher for larger time windows. The gain for HRZ is even higher as w increases, since HRZ is able to set time horizons for objects later in the future, thus ignoring more location updates and further decreasing the total events to be handled.

To better illustrate the reduction of the number of events that XTR and HRZ process, and how this is affected by the size of the time window, we also report the number of events in the event queue Q that are created and processed by XTR and HRZ with respect to those by BSL. The results are shown in Figure 3. Indeed, the results are in agreement with those in Figure 2, showing that XTR needs to process only about 30% of the events processed by BSL, while HRZ fewer than 5%.

7.2.3 *Varying the number of objects*

In the last set of experiments, we measure the performance of the algorithms with respect to the number of objects. For this purpose, we create subsets of the original datasets, containing a specific portion of randomly selected objects, and ran the algorithms on these subsets. The other parameters are set to $\mathcal{T} = 60$ minutes, $k = 10$, and $w = 5$ minutes. The results are plotted in Figure 4.

As expected, the execution time of all algorithms increases as the size of the dataset increases. However, XTR and, especially, HRZ show better scalability. Especially HRZ for the cases of the *Beijing taxis* and the *Aegean ships*, where the movement of the objects is relatively more regular, proves to be quite robust with respect to the total number of objects, which verifies that it can successfully avoid unnecessary examinations of objects that cannot qualify as candidates for the result set.



(a) Beijing taxis (b) Aegean ships (c) Athens vehicles

Figure 4: Execution time of BSL, XTR and HRZ w.r.t. the size $|O|$ of the dataset.

8. CONCLUSIONS

This paper introduced and studied the problem of continuously reporting moving objects with similar recent trajectories to a given query object. This problem extends the case of continuous nearest neighbor monitoring and of discovering similar trajectories in historical data. We proposed a generic baseline method that operates for any aggregate trajectory distance metric; the extension to other metrics is left as future work. Then we turned our attention to instances where the distance between the trajectories of two objects is determined by the extrema (minimum and maximum) of their individual location distances. For these instances, we described two more efficient algorithms, with the latter taking into account a given bound on the velocities of objects. Our experimental study on real-world datasets showed that our methods exhibit up to 22 times performance gain compared to the baseline.

Acknowledgements This work was supported by the EU FP7 Projects GEOCROWD (FP7-PEOPLE-2010-ITN-264994) and GEOSTREAM (FP7-SME-2012-315631), and by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

9. REFERENCES

[1] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3), 2001.

[2] P. Bakalov and V. J. Tsotras. Continuous spatiotemporal trajectory joins. In *GSN*, 2006.

[3] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *VLDB J.*, 15(3), 2006.

[4] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, 2005.

[5] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Algorithms for nearest neighbor search on moving object trajectories. *GeoInformatica*, 11(2), 2007.

[6] Y. Gao, C. Li, G. Chen, L. Chen, X. Jiang, and C. Chen. Efficient k -nearest-neighbor search algorithms for historical moving object trajectories. *J. Comput. Sci. Technol.*, 22(2), 2007.

[7] J. Gudmundsson and M. J. van Kreveld. Computing longest duration flocks in trajectory data. In *GIS*, 2006.

[8] R. H. Güting, T. Behr, and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *VLDB J.*, 19(5), 2010.

[9] Y.-K. Huang, S.-J. Liao, and C. Lee. Efficient continuous k-nearest neighbor query processing over moving objects with uncertain speed and direction. In *SSDBM*, 2008.

[10] G. S. Iwerks, H. Samet, and K. P. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, 2003.

[11] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *PVLDB*, 1(1), 2008.

[12] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *SSTD*, 2005.

[13] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *STDBM*, 1999.

[14] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, 2007.

[15] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *PVLDB*, 3(1), 2010.

[16] B. Lin and J. Su. Shapes based trajectory queries for moving objects. In *GIS*, 2005.

[17] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.

[18] J. Niedermayer, A. Züfle, T. Emrich, M. Renz, N. Mamoulis, L. Chen, and H.-P. Kriegel. Probabilistic nearest neighbor queries on uncertain moving object trajectories. *PVLDB*, 7(3), 2013.

[19] N. Pelekis, I. Kopanakis, G. Marketos, I. Ntoutsi, G. L. Andrienko, and Y. Theodoridis. Similarity search in trajectory databases. In *TIME*, 2007.

[20] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, 2000.

[21] G. Skoumas, D. Skoutas, and A. Vlachaki. Efficient identification and approximation of k-nearest moving neighbors. In *SIGSPATIAL/GIS*, 2013.

[22] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, 2001.

[23] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD*, 2002.

[24] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, 2002.

[25] G. Trajcevski, R. Tamassia, H. Ding, P. Scheuermann, and I. F. Cruz. Continuous probabilistic nearest-neighbor queries for uncertain trajectories. In *EDBT*, 2009.

[26] M. Vazirgiannis, Y. Theodoridis, and T. K. Sellis. Spatio-temporal composition and indexing for large multimedia applications. *Multimedia Syst.*, 6(4), 1998.

[27] M. R. Vieira, P. Bakalov, and V. J. Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *GIS*, 2009.

[28] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, 2002.

[29] 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.

[30] Y. Yanagisawa, J. Akahani, and T. Satoh. Shape-based similarity query for trajectory of mobile objects. In *MDM*, 2003.

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

[32] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *KDD*, 2011.

[33] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *GIS*, 2010.

[34] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, 2003.

Towards Enabling Outlier Detection in Large, High Dimensional Data Warehouses*

Konstantinos Georgoulas and Yannis Kotidis

Athens University of Economics and Business
76 Patission Street, Athens, Greece
{kgeorgou,kotidis}@aueb.gr

Abstract. In this work we present a novel framework that permits us to detect outliers in a data warehouse. We extend the commonly used definition of distance-based outliers in order to cope with the large data domains that are typical in dimensional modeling of OLAP datasets. Our techniques utilize a two-level indexing scheme. The first level is based on Locality Sensitivity Hashing (LSH) and allows us to replace range searching, which is very inefficient in high dimensional spaces, with approximate nearest neighbor computations in an intuitive manner. The second level utilizes the Piece-wise Aggregate Approximation (PAA) technique, which substantially reduces the space required for storing the data representations. As will be explained, our method permits incremental updates on the data representation used, which is essential for managing voluminous datasets common in data warehousing applications.

1 Introduction

Assuring quality of data is a fundamental task in information management. It becomes even more critical in decision making applications, where erroneous data can mislead to disastrous reactions. The data warehouse is the cornerstone of an organization's information infrastructure related to decision support. The information manipulated within a data warehouse can be used by a company or organization to generate greater understanding of their customers, services and processes. Thus, it is desirable to provision for tools and techniques that will detect and address potential data quality problems in the data warehouse.

It is estimated that as high as 75% of the effort spent on building a data warehouse can be attributed to back-end issues, such as readying the data and transporting it into the data warehouse [1]. This is part of the Extract Transform Load (ETL) processes, that extract information pieces from available sources to a staging area, where data is processed before it is eventually loaded in the data warehouse local tables. Processing at the data staging area includes cleansing, transformation, migration, scrubbing, fusion with other data sources etc.

In this paper, we propose a novel framework for identifying outliers in a data warehouse. Outliers are commonly defined as rare or atypical data objects that do not behave like the rest of the data. Often, erroneous data points appear as outliers when projected

* This work was funded by Project EICOS.

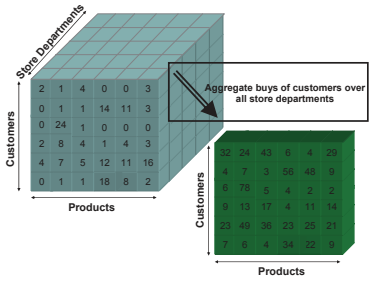


Fig. 1. Customer dimension projected onto the Product dimension

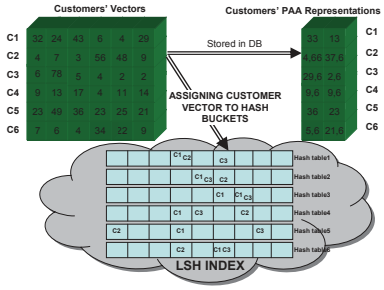


Fig. 2. Framework's Overview

on a properly derived feature space. In our work, we exploit the dimensional modeling used in a data warehouse and let the user examine the data under selected dimensions of interest. This way, our definition of what constitutes an outlier has a natural interpretation for the policy makers that interact with this data. Moreover, our techniques are tailored for the massive and periodic schedule of updates that occur with each ETL process. Clearly, techniques that require substantial pre- or post- processing of data are not suitable for handling massive datasets such as those in a data warehouse.

2 A Framework for Detecting Outliers in a Data Warehouse

Given a data warehouse with multiple dimensions d_1, d_2,..., d_n, each organized by different hierarchy levels h_k the data warehouse administrator my select a pair (d_{aggr}, h_{aggr_level}) so as to define the requested aggregation and, similarly, a pair (d_{proj}, h_{proj_level}) in order to denote the space that these aggregates should be projected upon. For instance the aggregate dimension can be customer at the hierarchy level of customer-type and the projected dimension product at the product-brand level. These pairs indicate our intention to compare different customer types based on cumulative sales of the brand of products they buy in order to search for outliers. Clearly, a data warehouse administrator may define multiple such pairs of dimensions in order to test the data for outliers. An example presented in Figure 1 where the customer (aggregate) dimension is projected onto the product dimension. This projection leads to a high dimensional vector for each customer that summarizes all his buys over the whole list of products.

An $O(D, M)$ distance based outlier is defined [2] as a data item O in a dataset with fewer than M data items within distance D from O. The definition, in our domain, suggests that range queries need to be executed in the data space defined by the projected dimension in order to compute the number of data items that lay inside a range of D from item O. As has been explained, the projected space can have very high dimensionality (i.e. equal to the number of all products in the data warehouse, which is in the order of thousands), which renders most multidimensional indexing techniques ineffectual, due to the well documented curse of dimensionality [3].

In order to address the need to compare data items on a high-dimensionality space when looking for outliers, we adapt a powerful dimensionality reduction technique

called LSH [4]. LSH generates an indexing structure by evaluating multiple hashing functions over each data item (the resulting vector when projecting a customer on the space of products she buys). Using the LSH index, we can estimate the k nearest neighbors of each customer and compute outliers based on the distances of each customer from its k neighbors. We thus propose an adapted approximate evaluation of distance-based outliers that treats a data item O as an outlier if less than M of its k nearest neighbors are within distance M from O. Please notice that this alternative evaluation permits us to utilize the LSH index for a k-NN query (with $k > M$) and restrict the range query on the k results retrieved from the index. Thus, the use of the LSH index permits effective evaluation of outliers, however it introduces an approximation error, because of collisions introduced by the hashing functions. There have been many proposals on how to tune and increase performance of LSH (e.g. [5, 6]), however such techniques are orthogonal to the work we present here.

The use of LSH enables computation of outliers by addressing the curse of dimensionality. Still, an effective outlier detection framework needs to address the extremely high space required for storing the resulting data vectors. The size of these vectors is proportional to the size of a data cube slice on the selected pair of dimensions. Moreover, these vectors need to be updated whenever the data warehouse is updated with new data. We address both these issues (space overhead, update cost) using the PAA representation instead of the original vectors. Utilizing PAA, we store vectors of lower dimensionality than the real ones, thus gaining in space. We can also compute the distances between each data item and its nearest neighbours through their PAA representations much faster than using the real data items without losing too much in accuracy as we will show in our experimental evaluation. PAA [7] represents a data item of length n in $\mathbb{R}^N$ space (where $n > N$) achieving a dimensionality reduction ratio $N{:}n$. Given that each data item X is a vector with coordinates $x_1, .., x_n$, its new representation will be a new vector $\bar{X}$ of length N and coordinates the mean values of the N equisized fragments of vector X. So according to PAA a vector $X = x_1, .., x_n$ is represented by $\bar{X} = \bar{x}_1, .., \bar{x}_N$ where $\bar{x}_i = \frac{N}{n} \sum_{j=\frac{n}{N}(i-1)+1}^{\frac{n}{N}i} x_j$.

Beside the space savings provided by PAA, its adaptation has another important advantage in our application. Because of its definition, PAA representations are linear projections that permit incremental updates whenever new data arrives at the data warehouse. Let PAA_{old} denote the PAA vector of a customer and PAA_{delta} the PAA representation of the customer's buys in the newly acquired updates. Then, in order to compute the new representation PAA_{new} for this customer we can simply add the two vectors, i.e. PAA_{new}=PAA_{old}+PAA_{delta}. This property is vital for data warehouses, where incremental updates are of paramount importance [8].

3 Experiments and Concluding Remarks

In our experimental evaluation, we used a clustered synthetic dataset that represents orders of 10,000 customers over a list of 1200 products. Each cluster contains customers with similar behavior. In particular, the customers within each cluster have a randomly

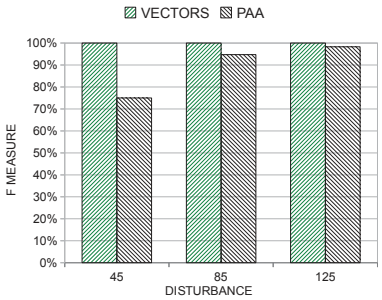


Fig. 3. F-Measure

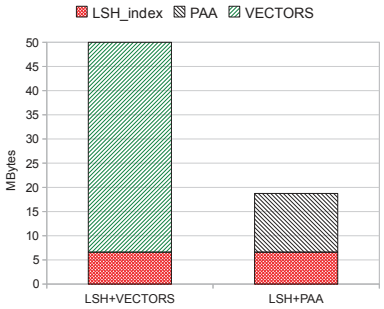


Fig. 4. Space for the LSH Index, the Vectors or PAAs

(pre)selected set of "hot" products that represent 20% of the whole product list. For a customer in the cluster, 80% of her orders are on that 20% subset of products. Different clusters have different set of hot products. The frequencies of customers' orders follow the normal distribution with different mean values per cluster. In order to evaluate the performance of our method we injected in the dataset outliers in the form of spurious orders. We created three infected datasets. In the first one, the spurious orders add low disturbance (measured by the number of spurious orders) to the original data. while in the second medium and in the third large. In Figure 3 we depict the f-measure ($\frac{2\times recall\times precision}{recall+precision}$) in detecting the injecting outliers for the tree datasets. We compare two variants. The first uses the LSH index and the data vectors generated by projecting the customers on the product dimension. The second setup, instead of the original vectors, it only stores their PAA representations of one quarter of the original vector length. In the Figure we observe that the accuracy of the PAA method is quite similar to the other method that stores the actual vectors, while the storage of PAA is much smaller, as it shown in Figure 4. Moreover, the size of the LSH index is very small, while its accuracy in computing distance-based outliers is at least 98%.

References

1. Kimball, R.: The Data Warehouse Toolkit. John Wiley & Sons (1996)
2. Subramaniam, S., Palpanas, T., Papadopoulos, D., Kalogeraki, V., Gunopulos, D.: Online Outlier Detection in Sensor Data Using Non-Parametric Models. In: VLDB. (2006)
3. Korn, F., Pagel, B.U., Faloutsos, C.: On the 'Dimensionality Curse' and the 'Self-Similarity Blessing'. IEEE Trans. Knowl. Data Eng. **13**(1) (2001) 96–111
4. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: FOCS, IEEE Computer Society (2006) 459–468
5. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In: VLDB. (2007) 950–961
6. Georgoulas, K., Kotidis, Y.: Distributed Similarity Estimation using Derived Dimensions. VLDB J. **21**(1) (2012) 25–50
7. Keogh, E.J., Chakrabarti, K., Pazzani, M.J., Mehrotra, S.: Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. Knowl. Inf. Syst. **3**(3) (2001) 263–286
8. Roussopoulos, N., Kotidis, Y., Roussopoulos, M.: Cubetree: Organization of and Bulk Updates on the Data Cube. In: SIGMOD Conference. (1997) 89–99

Efficient Point-based Trajectory Search*

Shuyao Qi[1], Panagiotis Bouros[2], Dimitris Sacharidis[3], and Nikos Mamoulis[1]

[1] Department of Computer Science
The University of Hong Kong
{syqi2,nikos}@cs.hku.hk
[2] Department of Computer Science
Humboldt-Universität zu Berlin, Germany
bourospa@informatik.hu-berlin.de
[3] Faculty of Informatics
Technische Universität Wien, Austria
dimitris@ec.tuwien.ac.at

Abstract. Trajectory data capture the traveling history of moving objects such as people or vehicles. With the proliferation of GPS and tracking technology, huge volumes of trajectories are rapidly generated and collected. Under this, applications such as route recommendation and traveling behavior mining call for efficient trajectory retrieval. In this paper, we first focus on distance-based trajectory search; given a collection of trajectories and a set query points, the goal is to retrieve the top-k trajectories that pass as close as possible to all query points. We advance the state-of-the-art by combining existing approaches to a hybrid method and also proposing an alternative, more efficient range-based approach. Second, we propose and study the practical variant of bounded distance-based search, which takes into account the temporal characteristics of the searched trajectories. Through an extensive experimental analysis with real trajectory data, we show that our range-based approach outperforms previous methods by at least one order of magnitude.

1 Introduction

The proliferation of GPS and tracking technology has brought to availability huge volumes of trajectories from real moving objects such as mobile phone users, vehicles and animals. Searching such a collection of trajectories finds several applications, including route recommendation, behavior mining, and in transportation systems [1, 2]. Different from conventional retrieval tasks which identify similar trajectories to a given one or those crossing a specific spatial region, in this paper we focus on *point-based search*, which retrieves trajectories based on given points. In particular, taking as input a set of query points Q (e.g., a particular set of POIs), the *distance-based trajectory search* studied in [3, 4] retrieves the trajectories that pass as close as possible to all query points.

* Work supported by grant HKU 715413E from Hong Kong RGC, and by the European Social Fund and Greek National Funds through the NSRF Research Program Thales.

Specifically, the distance of a trajectory t to Q is computed by summing up, for each query point $q \in Q$, its distance to the nearest point in t.

Consider for instance a collection of touristic trajectories; a travel agency issues a distance-based query to survey or recommend popular routes that pass close to specific sightseeing attractions. As another example, query set Q could contain traffic congestion points; in this case, the traffic department seeks to discover the causes of the congestion by analyzing the trajectories that pass near the points in Q. In the context of surveillance and security applications, Q may contain locations of crime scenes, and hence the police department issues a distance-based query to investigate the correlation of these crime locations by identifying suspects who moved close to all of them.

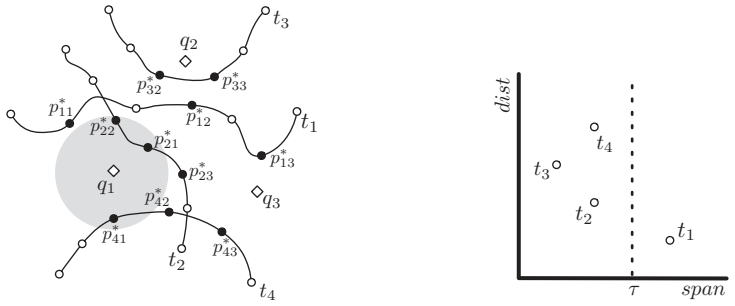
Contributions. This paper tackles two problems under the point-based trajectory search. First, we thoroughly study the efficient evaluation of *distance-based trajectory search*. We review in detail existing algorithms `IKNN` [3] and `GH/QE` [4]. These methods follow a *candidate generation* and *refinement* paradigm, and invoke a nearest neighbor (NN) search centered at each query point to examine the trajectories in ascending order of their distance to Q. By analyzing the pros and cons of these methods, we design a hybrid NN-based algorithm which consistently outperforms `IKNN` and `GH/QE` by over an order of magnitude. Going one step further, we tackle the inherent shortcomings of the NN-based approach itself, namely (a) the increased I/O cost due to independently running multiple NN searches and (b) the increased CPU cost for continuously maintaining a priority queue for each NN search. We propose a novel *spatial range-based* approach, which is up to 2 times faster than our hybrid algorithm.

Second, we observe that the distance-based search ranks trajectories solely on how close they pass to the query points in Q, ignoring however other qualitative characteristics of the retrieved results. To fill this gap, we introduce a practical variant of distance-based trajectory search, which also takes into account the temporal aspect of the trajectories. Specifically, this *bounded distance-based search* filters out non-interesting trajectories, whose points closest to Q span a time interval greater than a user-defined threshold.

Outline. The rest of the paper is organized as follows. Section 2 formally defines the distance-based and bounded distance-based trajectory search while Sections 3 and 4 address their efficient evaluation. Then, Section 5 discusses problem variants where (a) the trajectories are ranked both on their distance to the query points and the time interval they span, and (b) Q is a sequence of query points, instead of a set. Section 6 presents our experimental analysis. Finally, Section 7 outlines related work, while Section 8 concludes the paper.

2 Problem Definition

Let T be a collection of trajectories. A trajectory in T is defined as a sequence of spatio-temporal points $\{p_1, \ldots, p_n\}$, each represented by a $\langle latitude, longitude, timestamp \rangle$ triple. The input of *point-based trajectory search* over collection T is



(a) Trajectory and query points (b) *span-dist* plot of trajectories

Fig. 1. Distance-based trajectory search with 4 trajectories, $T = \{t_1, \ldots, t_4\}$, and 3 query points, $Q = \{q_1, \ldots, q_3\}$; t_1, t_2 is the result to 2-DTS(T, Q), while t_2, t_3 the result to 2-BDTS(T, Q, τ)

a set of m spatial query points $Q = \{q_1, \ldots, q_m\}$. Given a query point $q_j \in Q$ and a trajectory $t_i \in T$, we define the $\langle p^*_{ij}, q_j \rangle$ *matching pair* based on the nearest to q_j point p^*_{ij} of trajectory t_i, i.e., $p^*_{ij} = \arg\min_{p \in t_i} dist(p, q_j)$, where $dist(\cdot, \cdot)$ denotes the distance (e.g., Euclidean) between two points in space. We then define the *distance* of a trajectory to Q based on the matching pairs for every query point q_j as:

$$dist(t_i, Q) = \sum_{q_j \in Q} dist(p^*_{ij}, q_j) \tag{1}$$

Consider the example in Figure 1(a), where query points are represented as diamonds, and trajectory points as circles; filled circles indicate matched points of the trajectory to query points. For trajectory t_1, point p^*_{11} is its closest point to query point q_1, and hence $\langle p^*_{11}, q_1 \rangle$ represents a matching pair. The other matched trajectory points of t_1 are p^*_{12} and p^*_{13}. Note that it is possible for a trajectory point to be matched with multiple query points. This is the case with trajectory t_3, where p^*_{32} is the closest point to both q_1 and q_2, i.e., $p^*_{31} \equiv p^*_{32}$.

We now formally define the *distance-based trajectory search* problem [3, 4].

Problem 1 (Distance-based Trajectory Search) *Given a collection of trajectories T and a set of query points Q, the k-*Distance-based Trajectory Search*, denoted by k-DTS(T, Q), retrieves a subset of k trajectories $R \subseteq T$ such that for each $t \in R$ and $t' \in T \setminus R$, $dist(t, Q) \leq dist(t', Q)$ holds.*

Returning to the example of Figure 1(a), trajectory t_1 has the lowest distance to Q, followed by t_2, t_3 and t_4; hence, the result to 2-DTS(T, Q) is t_1, t_2.

Next, we introduce a novel point-based trajectory search problem by also taking into account the temporal aspect of the trajectories. Let P^*_i be the set of all matching pairs for a trajectory t_i, sorted ascending on the timestamp of the involved trajectory points. We define the *span* of trajectory t_i with respect to Q, denoted by $span(t_i, Q)$, as the length of the time interval between the first

and the last pair in P_i^*, or equivalently:

$$span(t_i, Q) = \max_{q_x, q_y \in Q} (timestamp(p_{ix}^*) - timestamp(p_{iy}^*)) \quad (2)$$

Intuitively, $span(t_i, Q)$ equals the total time needed to reach as close as possible to all query points in Q, following trajectory t_i.

Problem 2 (Bounded Distance-based Trajectory Search) *Given a collection of trajectories T, a set of query points Q and a span threshold τ, the* k-Bounded Distance-based Trajectory Search*, denoted by* k-BDTS(T, Q, τ)*, retrieves the subset of k trajectories $R \subseteq T$ such that:*

- *for each $t \in R$, $span(t, Q) \leq \tau$ holds, and*
- *for each $t' \in T \setminus R$ with $span(t', Q) \leq \tau$, $dist(t, Q) \leq dist(t', Q)$ holds.*

Returning to Figure 1(a), assume for simplicity that trajectory points are reported in fixed time intervals. As a result, the span of a trajectory is proportional to the number of its points from the first to the last matched point (excluding the first). For example, $span(t_1, Q) = 4$ as there are 4 points from p_{11}^* and up to p_{13}^*. Similarly, we obtain the spans of t_2, t_3, t_4 as 2, 1, 2, respectively. Figure 1(b) plots the trajectories in the *span-dist* plane. DTS ignores the span values and simply returns the trajectories with the lowest *dist* coordinate. In contrast, BDTS introduces a threshold, e.g., $\tau = 3$, on the span of the trajectories, depicted as the dashed vertical line. Trajectories to the right of this line, i.e., t_1, do not qualify as BDTS results. Therefore, the result of 2-BDTS is t_2, t_3, i.e., the trajectories with the 2 lowest distances among those left of the line. Notice that BDTS may not return the trajectory with the lowest distance to Q if its span exceeds the threshold; e.g., t_1 in Figure 1.

Depending on the application, one may consider alternative definitions for point-based trajectory search that take into account both the distance and the span metrics. We briefly overview one of them in Section 5, where we also discuss trajectory search given a sequence of query points, instead of a set.

3 Distance-based Trajectory Search

We first discuss trajectory search based on the distance to a set of query points. Section 3.1 revisits existing work, while Sections 3.2 and 3.3 present our NN-based and spatial range-based methods, respectively.

3.1 Existing Methods

Methods `IKNN` [3] and `GH/QE` [4] have previously tackled distance-based trajectory search. Note that in [3] the problem was defined with respect to the similarity of a trajectory t_i to the set of query points Q, defined as $sim(t_i, Q) = \sum_{q_j \in Q} e^{-dist(p_{ij}^*, q_j)}$. In what follows, we describe the straightforward adaptation of the `IKNN` algorithm for the distance metric of Equation (1) (which was also

Algorithm 1: `IKNN`

```
   Input      : collection of trajectories T, set of query points Q, number of results k
   Output     : result set R
   Variables  : candidate set C, k-th distance upper bound UB_k, distance lower bound LB
1  initialize C ← ∅, UB_k ← ∞ and LB ← 0;
2  while UB_k > LB do
3  |  for each q_j ∈ Q do
4  |  |  δ_j-NN(q_j) ← the next δ_j nearest trajectory points to q_j;
5  |  |  update C with δ_j-NN(q_j);
6  |  |  update UB_k and LB;                                  ▷ Equations (3) and (4)
7  R ← Refine_DTS(k, T, Q, C);
8  return R;
```

used in [4]). The adaptation of `GH/QE` and our methods (Sections 3.2 and 3.3) to the similarity metric of [3] is also straightforward and therefore, omitted. Moreover, the relative performance of all methods is identical independent of the metric used.

All existing methods adopt a *candidate generation* and *refinement* evaluation paradigm. During the first phase, a set of candidate trajectories is determined by incrementally retrieving the nearest trajectory points to the query points in Q. For this purpose, the methods utilize a single R-tree to index all trajectory points. A candidate trajectory t is called a *full match* if the matching pairs of t to all query points in Q have been identified; otherwise, t is a *partial* match. As soon as the candidate set is guaranteed to include the final results (even as partial matches), candidate generation is terminated, and the refinement phase is then employed to identify and output the results.

The `IKNN` Algorithm. Note that the `IKNN` algorithm comes in two flavors; in the following, we consider the one based on best-first search, as it was shown in [3] to be both faster and require fewer I/O operations. Algorithm 1 shows the pseudocode of `IKNN`. During candidate generation (Lines 2–6), the algorithm iterates over the points of Q in a round robin manner. For each query point q_j, the (next) batch of nearest to q_j trajectory points is retrieved using the R-tree index, in Line 4. The nearest neighbor search retrieves a different number of trajectory points δ_j per query point q_j, in order to expedite the termination of this first phase (details in [3]). Based on the newly identified matching pairs that involve q_j, the set of candidates C is then updated in Line 5 by either adding new partial matches or filling an empty slot for existing. For each partial match t_i in C, `IKNN` computes an *upper bound* of its distance to Q by setting the distance of t_i to every unmatched query point equal to the diameter of the space (maximum possible distance between two points): [4]

$$\overline{dist}(t_i, Q) = \sum_{q_j \in Q_i} dist(p^*_{ij}, q_j) + |Q \setminus Q_i| \cdot DIAM, \tag{3}$$

where set $Q_i \subseteq Q$ contains all the query points already matched to a point in trajectory t_i. We denote by UB_k the k-th smallest among the distance bounds

[4] Under the similarity-based definition of DTS in [3], `IKNN` sets empty "slots" to 0.

Algorithm 2: GH

```
   Input      : collection of trajectories T, set of query points Q, number of results k
   Output     : result set R
   Variables  : candidate set C, global heap H
1  initialize C ← ∅ and H ← ∅;
2  while C contains less than k full matches do
3  |  pop ⟨p_ij, q_i⟩ from H; ▷ Get the globally nearest trajectory point to some query point
4  |  update C with ⟨p_ij, q_i⟩;
5  |  push to H the next nearest trajectory point to q_i;
6  R ← Refine_DTS(k, T, Q, C);
7  return R;
```

Algorithm 3: QE

```
   Input      : collection of trajectories T, set of query points Q, number of results k
   Output     : result set R
   Variables  : candidate set C, global heap H, distance lower bound LB
1  initialize C ← ∅, H ← ∅ and LB ← 0;
2  while C contains less than k full matches with dist(·, Q) ≥ LB do
3  |  pop ⟨p_ij, q_i⟩ from H; ▷ Get the globally nearest trajectory point to some query point
4  |  update C with ⟨p_ij, q_i⟩;
5  |  push to H the next nearest trajectory point to q_i;
6  |  complete the most promising partial matches in C;                    ▷ Equation (5)
7  |  update LB;                                                           ▷ Equation (6)
8  R ← Refine_DTS(k, T, Q, C);
9  return R;
```

for the trajectories in C. In addition, IKNN computes a *lower bound* LB of the distance to Q for all unseen trajectories (i.e., those not contained in C), by aggregating the distance of the farthest (retrieved so far) trajectory point to each query point in Q. Formally:

$$LB = \sum_{q_j \in Q} dist(p_j^{\delta}, q_j) \tag{4}$$

where p_j^{δ} is the last trajectory point returned by the NN search centered at q_j.

The candidate generation phase of IKNN terminates when $UB_k \leq LB$; in this case, none of the unseen trajectories can have smaller distance to Q compared to the candidates in C. Last, IKNN invokes $\texttt{Refine}_{\texttt{DTS}}$ to produce the results. Briefly, the function examines candidates in ascending order of a lower bound on their distance, retrieving them from disk to compute $dist(\cdot, Q)$ (details in [3]).

The GH/QE Algorithms. Different from IKNN, the methods in [4] retrieve trajectory points in ascending order of the distance to their closest query point. Specifically, a *global heap* H is used to retrieve at each iteration the *globally* nearest trajectory point p_{ij} to some query point q_j, and then, to update candidate set C, accordingly. Algorithm 2 shows the pseudocode of GH. The candidate generation phase of GH is terminated as soon as set C contains k full matches (proof of correctness in [4]). Note that these full matches are not necessary among the final results identified in Line 6 during the refinement phase.

In practice, the order imposed by global heap H cannot guarantee a good performance unless both trajectory and query points are uniformly distributed

in space. For instance, if a particular query point is very close to many trajectories, GH will generate a large number of partial matches with only that slot filled. Consequently, it will take longer to produce the k full matches needed to terminate the generation phase, and at the same time a large number of candidates would have to be refined. A similar problem occurs when a query point is located away from the trajectories.

To address these issues, Tang et al. [4] proposed an extension to GH termed QE, which periodically fills the empty slots for the partially matched trajectories with the highest potential of becoming results. These are then retrieved from disk, and their actual distance is computed. A trajectory has high potential if it has (i) few empty slots and (ii) small distance in each filled slot with respect to the next point to be retrieved for that slot. These factors are captured respectively by the denominator and enumerator of the following equation:

$$potential(t_i) = \frac{\sum_{q_j \in Q_i} \left(dist(p_j^H, q_j) - dist(p_{ij}^*, q_i) \right)}{|Q \setminus Q_i|} \tag{5}$$

where set $Q_i \subseteq Q$ contains all the query points already matched to a point in t_i, p_j^H is the next nearest trajectory point to q_j contained in heap H and p_{ij}^* is the nearest to q_j point in trajectory t_i.

Algorithm 3 shows the pseudocode of QE. The candidate generation phase of QE terminates when candidate set C contains k full matches (similar to GH), provided however that their distance to Q is smaller than the distance of all unseen trajectories (Line 2) (proof of correctness in [4]). To determine this, QE computes in Line 7, a *lower bound* LB of the distance for the unseen trajectories (similar to IKNN) by aggregating the distance of the next nearest trajectory point to every query point, i.e., the contents of heap H:

$$LB = \sum_{q_j \in Q} dist(p_j^H, q_j) \tag{6}$$

3.2 A Hybrid NN-based Approach

The DTS problem can be viewed as a top-k query [5, 6]. For each query point q_j, consider a sorted trajectory list T_j, where each trajectory is ranked according to its distance to the query point. Then, the objective is to determine the top-k trajectories that have the highest aggregate score, i.e., distance, among the lists. However, as these lists are not given in advance and constructing them is costly, the goal is to progressively materialize them, until the result is guaranteed to be among the already seen trajectories.

Following the top-k query processing terminology, a *sorted access* on list T_j corresponds to the retrieval of the next nearest trajectory to query point q_j, which in turn may involve multiple trajectory point NN retrievals. In contrast, a *random access* for trajectory t_i on list T_j corresponds to the retrieval of t_i from disk and the computation of its distance to q_j; in practice, once t_i is retrieved, its distance to all query points can be computed at negligible additional cost.

Methods IKNN, GH and QE employ various ideas from top-k query processing (an overview of this field is presented in Section 7). Particularly, IKNN performs only sorted accesses and prioritizes them in a manner similar to Stream–Combine [7]. Similarly, GH performs only sorted accessses but follows an unconventional strategy for prioritizing them, which explains its poor performance on our tests in Section 6. On the other hand, QE additionally performs random accesses following a strategy similar to the CA algorithm [5] to select which trajectory to retrieve.

In the following, we present the NNA algorithm, which combines the strengths of IKNN and QE. In short, it builds upon the Quick–Combine top-k algorithm [8] performing both sorted and random accesses to generate the candidate set. NNA has the following features. First, similar to IKNN, the algorithm retrieves in a round robin manner, batches of nearest trajectory points to each query point in Q. This addresses the weaknesses of GH when dealing with non-uniformly distributed data. Second, after performing the nearest neighbor search centered at each query point, NNA fills the slots of the trajectories with the highest potential according to Equation (5), similar to QE. Finally, NNA employs the termination condition of IKNN for the candidate generation phase. In practice, NNA extends Algorithm 1 by completing the most promising partial matches in C (similar to QE), between Lines 5 and 6. Hence, it is able to compute tighter bounds compared to IKNN and thus terminate the generation phase earlier. In addition, it produces fewer candidates than IKNN, reducing the cost of the refinement phase.

3.3 A Spatial Range-based Approach

We identify two shortcomings of all the NN-based methods previously described. First, each NN search is implemented independently, which means that R-tree nodes and trajectory points may be accessed multiple (up to $|Q|$) times, which increases the total I/O cost. Second, each NN search is associated with a priority queue, whose continuous maintenance increases the total CPU cost.

Our novel *Spatial Range-based* algorithm, denoted by SRA, addresses both these shortcomings. Similar to the NN-based approaches, it follows a generation and refinement paradigm. However, to generate the candidate set, it issues a spatial range search of expanding radius centered at each query point in Q. All searches operate on a common set N of R-tree nodes, which avoids accessing nodes more than once and hence saves I/O operations. Moreover, set N needs not be sorted according to any distance, eliminating costly priority queue maintenance tasks. The range-based search for each query point q_j is associated with *current radius* r_j, and is also assigned a *maximum radius* θ_j. As the algorithm progresses, current radius r_j increases while maximum radius θ_j decreases. Candidate generation terminates as soon as $r_j > \theta_j$ for some query point q_j.

Algorithm 4 shows the pseudocode of SRA. In Lines 2–4, SRA initializes the current and maximum radius for each query point. For the latter, an upper bound UB_k to the k-th smallest distance to Q is computed. In particular, SRA invokes a sum-aggregate nearest neighbor (sum-ANN) procedure [9] retrieving trajectory points in ascending order of $\sum_{q_j \in Q} dist(\cdot, q_j)$. Assuming that this procedure retrieves point p_i of trajectory t_i, the sum-aggregate value is an upper bound to

Algorithm 4: SRA

Input : collection of trajectories T, set of query points Q, number of results k
Output : top-k list of trajectories R
Variables : candidate set C, k-th distance upper bound UB_k, current r_i and maximum θ_i search radius for each $q_i \in Q$, set of R-tree nodes N

1 initialize $C \leftarrow \emptyset$ and $N \leftarrow$ R-tree root node;
2 compute UB_k invoking a sum-ANN(T, Q);
3 **for** each $q_j \in Q$ **do**
4 initialize $r_j \leftarrow 0$ and $\theta_j \leftarrow UB_k$;
5 **while** $r_j \leq \theta_j$ for all $q_j \in Q$ **do**
6 select current q_c;
7 $r_c \leftarrow r_c + \xi$; ▷ Increase r_c to expand search around q_c
8 expand from N all nodes that intersect with the disc of radius r_c centered at q_c;
9 $S \leftarrow$ trajectory points within spatial range r_c found during expansion;
10 update C with S;
11 update UB_k; ▷ Equation (7)
12 **for** each $q_j \in Q$ **do**
13 update $\theta_j \leftarrow UB_k - \sum_{q_\ell \in Q \setminus \{q_j\}} r_\ell$; ▷ Reduce maximum radius
14 $R \leftarrow \text{Refine}_{\text{DTS}}(k, T, Q, C)$;
15 **return** R;

the distance of t_i, i.e., $dist(t_i) \leq \sum_{q_j \in Q} dist(p_i, q_j)$. Hence, once points from k distinct trajectories have been retrieved, SRA can determine a value for UB_k.

During the candidate generation phase in Lines 5–13, SRA first selects the query point $q_c \in Q$ with the fewest retrieved points so far, and increases its radius by a fixed ξ[5], so that each location retrieves more or less the same number of points. Then, it extends the range search centered at q_c to new radius r_c. In particular, all nodes in N that intersect with the search frontier are expanded, i.e., replaced by their children (Line 8). During the expansion, all trajectory points within the frontier are collected in set S (Line 9). Upon completion of the expansion, set N contains no R-tree node or point within r_c distance to q_c, or with distance to q_c greater than θ_c, and N will be re-used in further iterations.

After the expansion, SRA uses the newly seen trajectory points in S to properly update candidate set C. Note that for each trajectory t_i in C, SRA keeps $|Q|$ slots storing the closest trajectory points $t_i.p_j$ seen so far to each query point q_j. A slot is marked *matched* if the corresponding matching pair has been determined, i.e., when $t_i.p_j \equiv p^*_{ij}$. SRA in Line 10 performs the following tasks for each point p_x in S; let t_i be the trajectory p_x belongs to. For each slot q_j that is not *matched*, SRA checks whether p_x is closer to q_j than $t_i.p_j$, and updates the slot with p_x if true. If the slot for the current query point q_c was among those examined, it is marked as *matched*. The benefits of this update strategy are twofold. First, it guarantees that no matching trajectory point will be missed, even though SRA does not access p_x again (removed from N) for $q_j \neq q_c$. At the same time, it also helps to derive a tighter upper bound for the distance of t_i:

$$\overline{dist}(t_i, Q) = \sum_{q_j \in Q_i} dist(p^*_{ij}, q_j) + \sum_{q_j \in Q \setminus Q_i} dist(t_i.p_j, q_j). \tag{7}$$

[5] In the future, we plan to investigate variable ξ_j values based on current radius r_j and the trajectory point density around q_j, inspired by determining δ_j value in [3].

Compared to Equation (3) utilized by IKNN and NNA, Equation (7) computes a tighter bound on unmatched slots. Based on these bounds, a tighter value for UB_k can be established (Line 11).

To better explain the procedure in Line 10, we use the example of Figure 1(a) for $k = 2$. SRA has just started and thus C is empty. Assume that the current query point is $q_c = q_1$, and let $r_1 = 0 + \xi$ be the radius of the shaded disk depicted in the figure. As a result, set S in Line 9 contains trajectory points $\{p^*_{21}, p^*_{22}, p^*_{41}\}$. Moreover, candidate set C contains t_2 and t_4. For trajectory t_2, p^*_{21} is settled as the matching point to q_1 because $dist(p^*_{21}, q_1) < dist(p^*_{22}, q_1)$ and no unseen point of t_2 can be closer. On the other hand, the matching points to q_2, q_3 cannot be yet determined, but we can use p^*_{21} and p^*_{22} to bound t_2's distances to q_2 and q_3. Therefore, the slots for t_2 become $\langle \mathbf{p^*_{21}}, p^*_{22}, p^*_{21} \rangle$, where bold indicates a *matched* slot. Moreover, an upper bound to the distance of t_2 is determined as $\overline{dist}(t_2, Q) = dist(p^*_{21}, q_1) + dist(p^*_{22}, q_2) + dist(p^*_{21}, q_3)$. Similarly, we obtain the slots for t_4 as $\langle \mathbf{p^*_{41}}, p^*_{41}, p^*_{41} \rangle$.

As a last step, SRA updates the maximum radius for all query points with respect to the new UB_k in Lines 12–13. Observe that SRA's termination condition for candidate generation is essentially identical to that of IKNN. Any trajectory not in the candidate set C must have distance to each q_j at least θ_j, and thus distance at least equal to $LB = \sum_{q_j \in Q} \theta_j$. The termination condition of Line 5, $r_j > \theta_j$ for some q_j, and the update of θ_j, imply that, when candidate generation concludes, $UB_k \leq LB$.

Finally, the performance of SRA can be enhanced following the key idea of QE to further improve the $\overline{dist}(t_j, Q)$ bound and therefore, UB_k. We denote this extension to the SRA algorithm by SRA+. Specifically, in between Lines 10 and 11 in Algorithm 4, SRA+ fills the empty slots of the trajectories in C with the highest potential as computed using Equation (5).

4 Bounded Distance-based Trajectory Search

We next address the bounded distance-based trajectory search. Recall from Section 2 that k-BDTS(T, Q, τ) is equivalent to a k-DTS(T', Q) distance-based query over the subset $T' \subseteq T$ containing only trajectories with $span(\cdot, Q) \leq \tau$. However, as $span(t, Q)$ can be computed only after all the matching pairs of a trajectory t to Q are identified, the major challenge is to limit the number of invalid partial matches generated, i.e., those with the $span(\cdot, Q) > \tau$. In the following, we address this issue in two alternative ways.

The idea behind the incremental approach, denoted as INCREMENTAL, is to progressively construct the result set R by utilizing the generation phase of a DTS method as a "black" box. Algorithm 5 illustrates INCREMENTAL; note that any of the algorithms in Section 3 can be used as the underlying DTS method. At each round, INCREMENTAL asks for the missing $k - |R|$ trajectories to complete the result set R in Lines 3–4. For this purpose, a λ-DTS(T, Q) search is processed, with the λ value been increased at each round by $k - |R|$; during the first round $\lambda = k$. Each time λ is updated in Line 3, the DTS method in Line 4 does not

Algorithm 5: INCREMENTAL

```
Input     : collection of trajectories T, set of query points Q, span threshold τ, number
            of results k
Output    : result set R
Variables : candidate set C, number of intermediate results λ
1 initialize C ← ∅, R ← ∅ and λ ← 0;
2 while |R| < k do
3 |   increase λ by k − |R|;
4 |   C ← next candidate set of λ-DTS(T, Q);
5 |   R ← R ∪ Refine_BDTS(k, T, Q, C, τ);
6 return R;
```

run from scratch. It continues the candidate generation using a new termination condition with respect to the updated λ in order to expand candidate set C. Last, in Line 5, $\texttt{Refine}_{\texttt{BDTS}}$ examines the new candidates to update result set R by computing their $dist(\cdot, Q)$ and eliminating trajectories with $span(\cdot, Q) > \tau$.

Intuitively, INCREMENTAL takes a conservative approach to bounded distance-based trajectory search. As it is unable to predict which partial matches could provide a valid trajectory (full match) with $span(\cdot, Q) \leq \tau$, a refinement phase is needed to "clean" the candidate set. Hence, INCREMENTAL may involve several rounds of generation and refinement phases. To address these issues, we propose the ONE–PASS approach which involves a single generation and refinement round. The idea is again to build upon a DTS method but by extending its candidate generation phase in two ways. First, for each partial match t_i in candidate set C, ONE–PASS computes a lower bound of $span(t_i, Q)$ based on the points of t_i matching the current subset of query points $Q_i \subset Q$, as follows:

$$\underline{span}(t_i, Q) = \begin{cases} 0, & if\ |Q_i| = 1 \\ span(t_i, Q_i), & otherwise \end{cases} \tag{8}$$

Every partial match with $\underline{span}(\cdot, Q) > \tau$ can be safely pruned. Second, the original termination is triggered only after candidate set C contains at least k valid full matches, i.e., with $span(\cdot, Q) \leq \tau$. This is because the k-th upper bound UB_k of existing candidates can be computed only through full matches. For example, candidate generation of ONE–PASS based on SRA+ terminates as soon as at least k valid full matches are identified and $r_j > \theta_j$ holds for some query point q_j.

5 Discussion

We discuss alternative definitions and variants to the point-based search problems introduced in Section 2.

Distance & Span-based Trajectory Search. Although taking into account their temporal span, the bounded distance-based search still ranks the trajectories solely on their distance to the query points in Q. As an alternative, we may

rank the results with respect to a linear combination of the *span-dist* metrics:

$$f(t, Q) = \alpha \cdot dist(t, Q) + (1 - \alpha) \cdot span(t, Q) \tag{9}$$

where α weights the importance of each metric. With Equation (9), we introduce the *k-Distance & Span-based Trajectory Search*, denoted by k-DSTS(T, Q) which returns the subset of k trajectories $R \subseteq T$ with the lowest $f(\cdot, Q)$ value.

All methods discussed in Section 3 can be extended for k-DSTS(T, Q) by replacing $dist(\cdot, Q)$ with $f(\cdot, Q)$. Note that the upper bound $\overline{f}(t, Q)$ of a partial match t can be computed by setting $\overline{span}(t, Q)$ equal to the total duration of the trajectory t. In contrast, as no matching pairs are identified for the unseen trajectories, the lower bound LB or the θ_j values are defined similar to the DTS methods, i.e., essentially setting the lower bound of span to zero. In Section 6.4, we experimentally investigate the efficient evaluation of DSTS.

Order-aware Trajectory Search. Similar to [3], we also consider a variation of the trajectory search when a visiting order is imposed for the query points. In this variation, the matched trajectory point p^*_{ij} to query point q_j, is not necessarily the nearest to q_j point of trajectory t_i. Consider for example trajectory t_2 in Figure 1. The depicted p^*_{22}, p^*_{21}, p^*_{23} for DTS cannot be the matched points in the $q_1 \rightarrow q_2 \rightarrow q_3$ order-aware DTS, as they violate the visiting order. Instead, the matched points that preserve the imposed visiting order are p^*_{22}, p^*_{22}, p^*_{23}, where p^*_{22} is matched with q_1 although $dist(p^*_{22}, q_1) > dist(p^*_{21}, q_1)$. The distance of a trajectory to sequence Q is recursively defined as follows:

$$dist_o(t, Q) = \begin{cases} \min \begin{cases} dist_o(t, T(Q)) + dist(H(t), H(Q)) - DIAM \\ dist_o(T(t), Q) \end{cases} & \text{if } t \neq \varnothing, Q \neq \varnothing \\ |Q| \cdot DIAM & \text{if } t = \varnothing \\ 0 & \text{if } Q = \varnothing \end{cases} \tag{10}$$

where $H(S)$ is the first point (head) in a sequence S, $T(S)$ indicates the tail of S after removing $H(S)$, $\varnothing$ denotes the empty sequence, and $DIAM$ represents the diameter of the space. The distance can be computed by straightforward dynamic programming [3]. To derive an upper bound on a partial matched trajectory t_i, we consider only the subsequence Q_i of Q that contains the matched query points, i.e., $\overline{dist}_o(t_i, Q) = dist_o(t_i, Q_i)$. For order-aware BDTS, distance and its upper bound are the same as in order-aware DTS. Note, however that the lower bound on span (Equation (8)) does not apply as the matching are not yet finalized. For order-aware DSTS evaluation, $f_o(t, Q)$ and its upper bound are defined in a similar manner to order-aware DTS. In Section 6, we experimentally investigate the order-aware variants of all three trajectory search problems.

6 Experimental Analysis

We evaluate our methods for point-based trajectory search. All algorithms were implemented in C++ and the tests run on a machine with Intel Core i7-3770 3.40GHz and 16GB main memory running Ubuntu Linux.

Table 1. POIs in Beijing

category	cardinality
Restaurants	51,971
Hotels	10,620
Pharmacies	6,963
Schools	6,618
Banks	6,057
Police stations	2,509
Supermarkets	2,356
Gas stations	1,916
Post offices	1,125

Table 2. Experimental parameters (default values in bold)

description	parameter	values
Number of results	k	1, 5, **10**, 50, 100
Number of query points	$\|Q\|$	2, 4, **6**, 8, 10
Span threshold ratio	τ/τ_{min}	1, 1.5, **2**, 2.5, 3
Linear combination factor	α	0, 0.25, **0.5**, 0.75, 1

6.1 Setup

We conducted our analysis using real-world trajectories from the GeoLife Project [10–12]. The collection contains 17,166 trajectories with 19m points in Beijing, recording a broad range of outdoor movement. To generate our query sets, we considered around 90k points of interest (POIs) of various types, located inside the same area covered by the trajectories (see Table 1 for details). A query set Q is formed by randomly selecting a combination of $|Q|$ types and a particular POI from each type. We assess the performance of all involved methods measuring their CPU and I/O cost, and the number of candidates they generate over 1,000 distinct query sets Q, while varying (i) the number of returned trajectories k and (ii) the number of query points $|Q|$. In case of BDTS queries, we additionally vary the span threshold via the τ/τ_{min} ratio, where τ_{min} is the minimum possible time required to travel among the query points in Q at a constant velocity of 50km/h. Finally, for DSTS queries, we also vary the weight factor α of Equation (9). Table 2 summarizes all parameters involved in our study.

6.2 Distance-based Trajectory Search

Figure 2 reports the CPU cost, the I/O cost and the number of generated candidates for the DTS methods. As expected the processing cost of all methods goes up as the values of k and $|Q|$ increase. The tests clearly show that `SRA+` is overall the most efficient evaluation method. We also make the following observations.

First, we observe that `IKNN` always outperforms `GH`/`QE`; note that this is the first time the methods from [3, 4] are compared. Naturally, `GH` comes as the least efficient method; due to the examination order imposed by global heap H, the algorithm is unable to cope with the skewed distribution of the real-world data. `QE` manages to overcome the shortcomings of `GH` by completing the empty slots of the most promising candidates. Yet, compared to `IKNN`, `QE` is less efficient due to its weak termination condition for the generation phase; recall that at least k full matches are needed for this purpose which also results in generating a larger number of candidates, as shown in Figures 2(c) and (f). The advantage of `IKNN` over `GH`/`QE` justifies our decision to build the hybrid `NNA` method upon the round robin-based candidate generation of `IKNN` which retrieves nearest neighbor points in batches, and its powerful threshold-based termination condition. `NNA` is indeed the most efficient NN-based method, in fact with an order of magnitude improvement over `IKNN` and `GH`/`QE` on both CPU and I/O cost. Finally, Figure 2

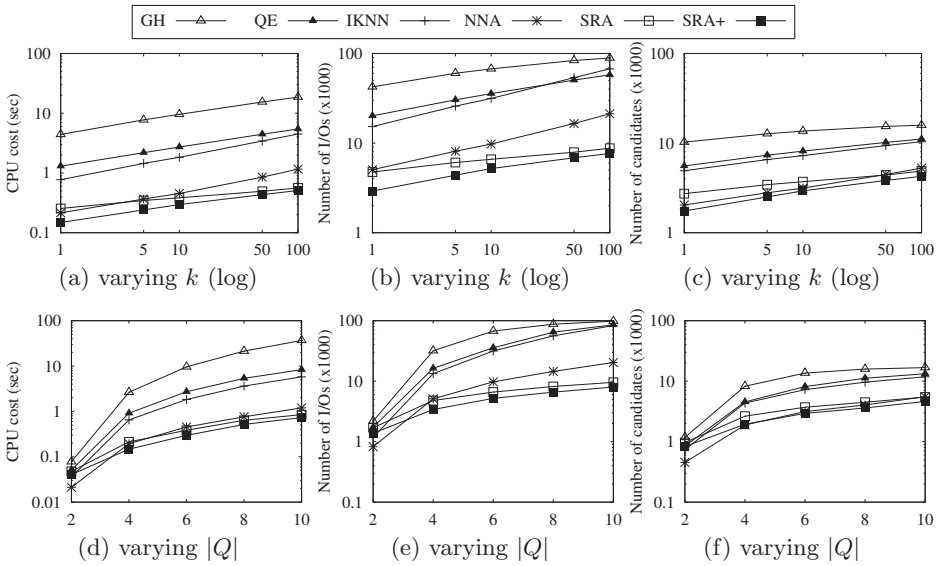


Fig. 2. Performance comparison for Distance-based Trajectory Search

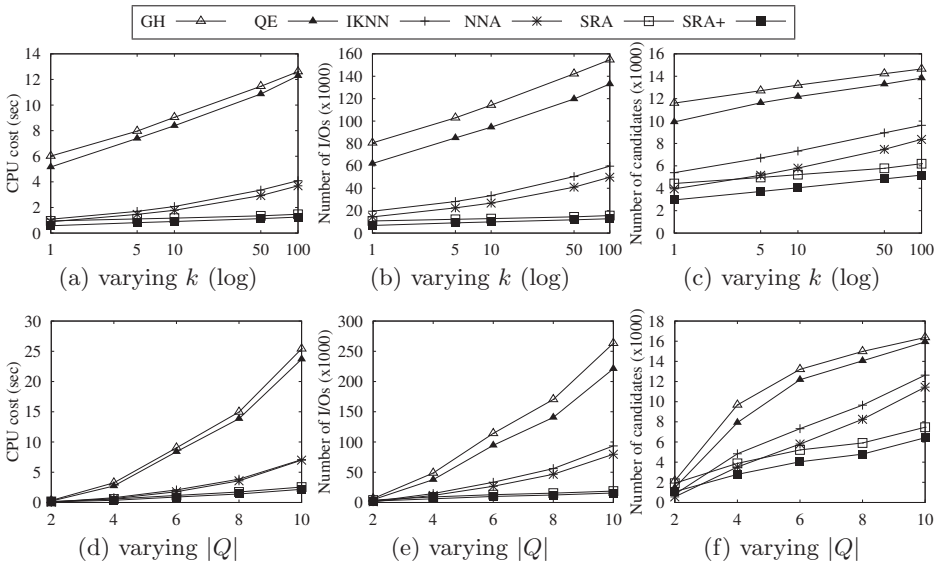


Fig. 3. Performance comparison for Distance-based Trajectory Search (order-aware)

clearly shows the advantage of the spatial range-based evaluation approach over the NN-based one. `SRA` is always faster while incurring fewer disk page accesses than `IKNN`, and in a similar manner, `SRA`+ outperforms `NNA`.

We also experimented with the order-aware variant of DTS. Figure 3 depicts similar results to Figure 2; the spatial range-based evaluation approach is again

superior to the NN-based and overall, SRA+ is the most efficient method. Nevertheless, it is important to notice that the advantage of completing the most proposing candidates is smaller compared to Figure 2, in terms of the CPU cost. Specifically, observe how close is the running time of GH to QE, of IKNN to NNA and of SRA to SRA+, in Figures 3(a) and (d). This is expected as completing partial matches employs dynamic programming to compute $dist_o(\cdot, Q)$.

6.3 Bounded Distance-based Trajectory Search

Next, we investigate the evaluation of BDTS queries while varying the k, $|Q|$ and τ/τ_{min} parameters. Based on the findings of the previous section, we use the SRA+ algorithm as the underlying DTS method. Note that due to lack of space we omit results for the order-aware variant of BDTS; the results however are similar. Figure 4 clearly shows that ONE–PASS outperforms INCREMENTAL in all cases. As expected, the conservative approach of INCREMENTAL generates a larger number of candidates by performing multiple rounds of generation and refinement which results in both higher running time and more disk page accesses. Last, notice that the evaluation of BDTS becomes less expensive for both methods while increasing τ/τ_{min}, as the number of invalid candidates progressively drops.

6.4 Distance & Span-based Trajectory Search

Finally, we study the evaluation of DSTS queries. For this experiment, we extended the most dominant method from [3, 4], i.e., IKNN, and our methods NNA, SRA and SRA+ following the discussion in Section 5. The results in Figure 5 demonstrate, similar to the DTS case, the advantage of both the spatial range-based approach and the SRA+ algorithm which is overall the most efficient evaluation method. Due to lack space, we again omit the figure for the order-aware variant of DSTS as the results are identical to Figure 5.

7 Related Work

Apart from the studies [3, 4] for distance-based search on trajectories detailed in Section 3.1, our work is also related to *top-k* and *nearest neighbor* queries.

Top-k Queries. Consider a collection of objects, each having a number of scoring attributes, e.g., rankings. Given an aggregate function γ (e.g., SUM) on these scoring attributes, a top-k query returns the k objects with the highest aggregated score. To evaluate such a query, a *sorted* list for each attribute a_i organizes the objects in decreasing order of their value to a_i; requests for *random accesses* of an attribute value based on object identifiers may be also possible. Ilyas et al. overviews top-k queries in [6] providing a categorization of the proposed methods. Specifically, when both sorted and random accesses are possible, the TA/CA [5] and Quick–Combine [8] algorithms can be applied. TA retrieves objects from the sorted lists in a round-robin fashion while a priority queue to organizes the

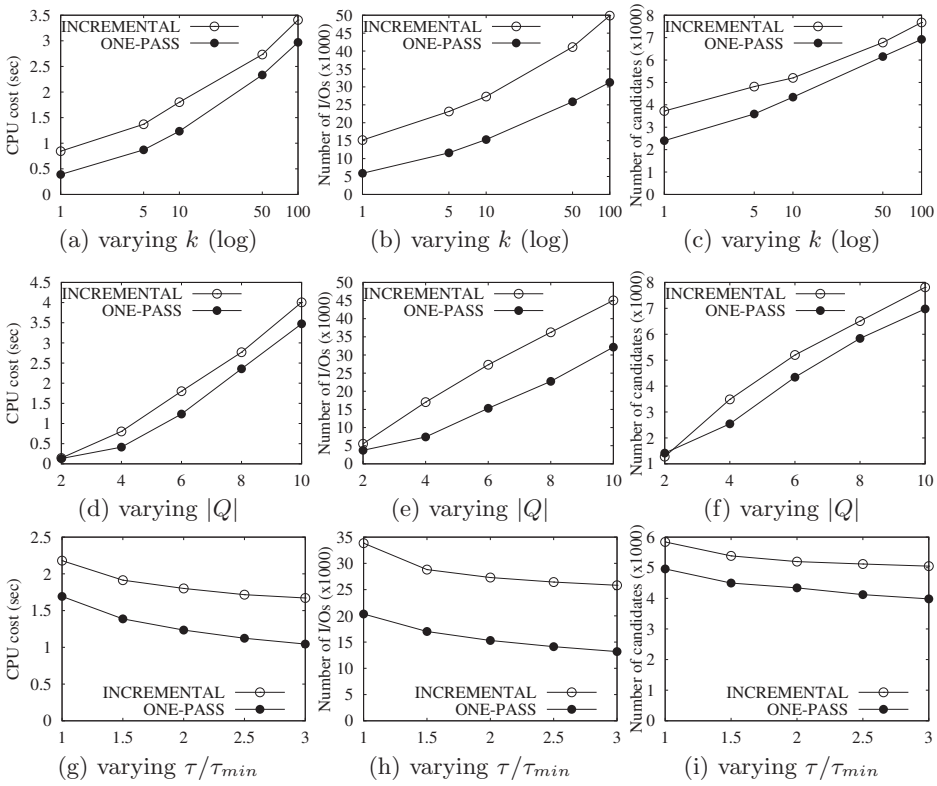


Fig. 4. Performance comparison for Bounded Distance-based Trajectory Search

best k objects so far. Based on the last seen attribute values, the algorithm defines an upper score bound for the unseen objects, and terminates if current k-th highest aggregate score is higher than this threshold. `TA` assumes that the costs of the two different access methods are the same. As an alternative, `CA` defines a ratio between these costs to control the number of random accesses, which in practice are usually more expensive than sorted accesses. Hence, the algorithm periodically performs random accesses to collect unknown values for the most "promising" objects. Last, the idea behind `Quick–Combine` is to favor accesses from the sorted lists of attributes which significantly influence the overall scores and the termination threshold. In contrast, when only sorted accesses are possible, the `NRA` [5] and `Stream–Combine` [7] algorithms can be applied. Intuitively, `Stream–Combine` operates similar to `Quick–Combine` without performing any random accesses. In Section 3.1, we discuss how the methods in [3, 4] build upon previous work on top-k queries to address distance-based search on trajectories.

Nearest Neighbor Queries. There is an enormous amount of work on the *nearest neighbor* (NN) query (also known as similarity search), which returns the object that has the smallest distance to a given query point; k-NN queries output the k nearest objects in ascending distance. Roussopoulos et al. proposed

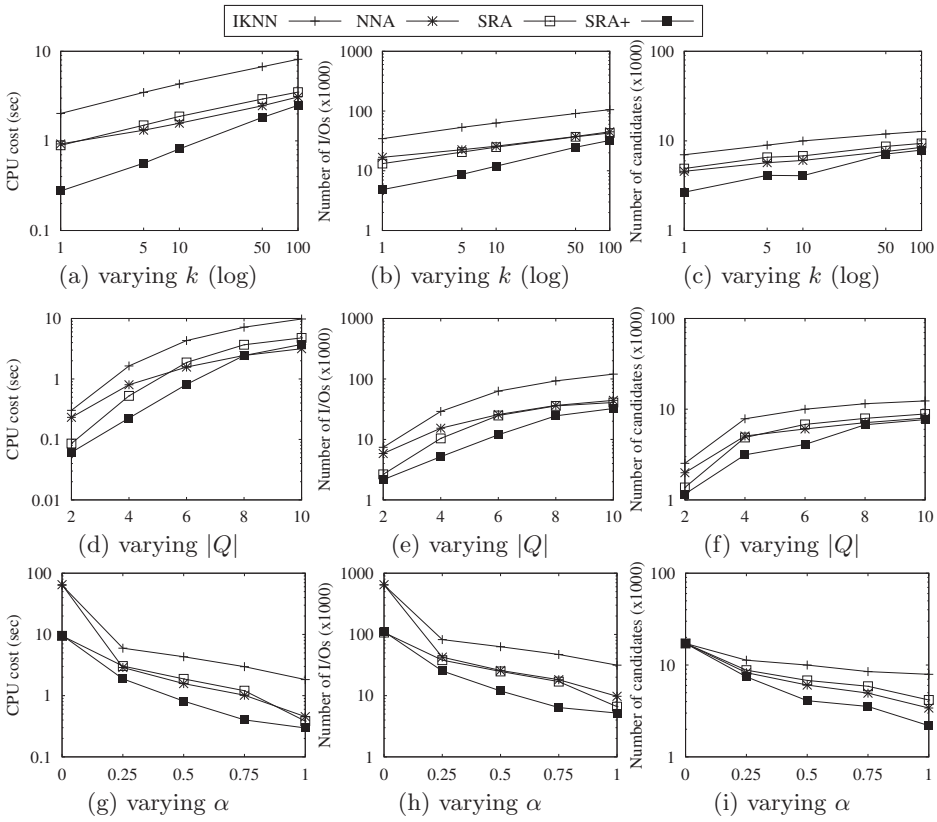


Fig. 5. Performance comparison for Distance & Span-based Trajectory Search

a depth-first approach to k-NN query in [13] while Hjaltason et al. enhanced the evaluation with a best-first search strategy in [14]. An overview of index-based approaches can be found in [15]; efficient methods for metric spaces, e.g., [16], and high-dimensional data, e.g., [17], have also been proposed.

For a set of query points, the *aggregate nearest neighbor* (ANN) query [9] retrieves the object that minimizes an aggregate distance to the query points. As an example, for the MAX aggregate function and assuming that the set of query points are users, and distances represent travel times, ANN outputs the location that minimizes the time necessary for all users to meet. In case of the SUM function and Euclidean distances, the optimal location is also known as the Fermat-Weber point, for which no formula for the coordinates exists.

8 Conclusions

In this paper, we studied the efficient evaluation of point-based trajectory search. After revisiting the existing methods (`IKNN` and `GH/QE`), which examine the trajectories in ascending order of their distance to the queries points, we devised

a hybrid algorithm which outperforms them by a wide margin. Then, we proposed a spatial range-based approach; our experiments on real-world trajectories showed that this approach outperforms any NN-based method. Besides improving the performance of distance-based search, we also introduced and investigated the evaluation of a practical variant for point-based trajectory search, which also takes into account the temporal aspect of the trajectories. As a direction for future work, we plan to consider additional types of annotated data on the trajectories in point-based search, such as textual and social information.

References

1. Zheng, Y.: Trajectory data mining: An overview. ACM Transaction on Intelligent Systems and Technology (September 2015)
2. Zheng, Y., Zhou, X., eds.: Computing with Spatial Trajectories. Springer (2011)
3. Chen, Z., Shen, H.T., Zhou, X., Zheng, Y., Xie, X.: Searching trajectories by locations: an efficiency study. In: SIGMOD. (2010) 255–266
4. Tang, L.A., Zheng, Y., Xie, X., Yuan, J., Yu, X., Han, J.: Retrieving k-nearest neighboring trajectories by a set of point locations. In: SSTD. (2011) 223–241
5. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS. (2001) 102–113
6. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. **40**(4) (2008)
7. Güntzer, U., Balke, W., Kießling, W.: Towards efficient multi-feature queries in heterogeneous environments. In: ITCC. (2001) 622–628
8. Güntzer, U., Balke, W.T., Kießling, W.: Optimizing multi-feature queries for image databases. In: VLDB. (2000) 419–428
9. Papadias, D., Tao, Y., Mouratidis, K., Hui, C.K.: Aggregate nearest neighbor queries in spatial databases. ACM Trans. Database Syst. **30**(2) (2005) 529–576
10. Zheng, Y., Zhang, L., Xie, X., Ma, W.Y.: Mining interesting locations and travel sequences from gps trajectories. In: WWW. (2009) 791–800
11. Zheng, Y., Li, Q., Chen, Y., Xie, X., Ma, W.: Understanding mobility based on GPS data. In: UbiComp 2008: Ubiquitous Computing, 10th International Conference, UbiComp 2008, Seoul, Korea, September 21-24, 2008, Proceedings. (2008) 312–321
12. Zheng, Y., Xie, X., Ma, W.: Geolife: A collaborative social networking service among user, location and trajectory. IEEE Data Eng. Bull. **33**(2) (2010) 32–39
13. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995. (1995) 71–79
14. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Trans. Database Syst. **24**(2) (1999) 265–318
15. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. **33**(3) (2001) 322–373
16. Jagadish, H.V., Ooi, B.C., Tan, K., Yu, C., Zhang, R.: iDistance: An adaptive b^+-tree based indexing method for nearest neighbor search. ACM Trans. Database Syst. **30**(2) (2005) 364–397
17. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Quality and efficiency in high dimensional nearest neighbor search. In: SIGMOD. (2009) 563–576