Under consideration for publication in Knowledge and Information Systems

Efficient Location-based Search of Trajectories with Location Importance

Da Yan[†1], James Cheng[§2], Zhou Zhao[†3] and Wilfred Ng[†4]

[†]*Department of Computer Science and Engineering, Hong Kong University of Science and Technology*
{[1]yanda, [3]zhaozhou, [4]wilfred}@cse.ust.hk
[§]*Department of Computer Science and Engineering, The Chinese University of Hong Kong*
[2]jcheng@cse.cuhk.edu.hk

Abstract. Given a database of trajectories and a set of query locations, location-based trajectory search finds trajectories in the database that are close to all the query locations. Location-based trajectory search has many applications such as providing reference routes for travelers who are planning a trip to multiple places of interest. However, previous studies only consider the spatial aspect of trajectories, which is inadequate for real applications. For example, one may obtain the reference route of a tourist who just passed by a place of interest without paying a visit. We propose the k *Important Connected Trajectories* (k-*ICT*) query by associating trajectories with *location importance*. For any query location, the result trajectories should contain an *important* point close to it. We describe an effective method to infer the importance of trajectory points from the temporal information. We also propose efficient R-tree based and grid-based algorithms to answer k-ICT queries, and verify the efficiency of our algorithms through extensive experiments on both real and synthetic datasets.

1. Introduction

With the popularity of location-acquisition technology, huge amounts of trajectory data are being generated at an unprecedented scale. We differentiate two types of trajectory data. The first type is simply a sequence of time-stamped locations, usually generated by mobile devices such as cell phones and GPS receivers at a relatively high sampling rate. The sample points in such trajectories have very little or no semantics, and many recorded locations are not important. The second type of trajectory is a sequence of locations with semantics, where each recorded location is usually important. One example of such a trajectory is a sequence of geo-tagged photos taken by a traveler in a trip. Numerous such trajectories can be obtained from photo-sharing websites such as Flickr (www.flickr.com), and people usually take photos at locations they like. Another example of such a trajectory is a sequence of check-in records of some traveler at the places he/she cares. Such trajectories are available from location-based social network services such as FourSquare (foursquare.com).

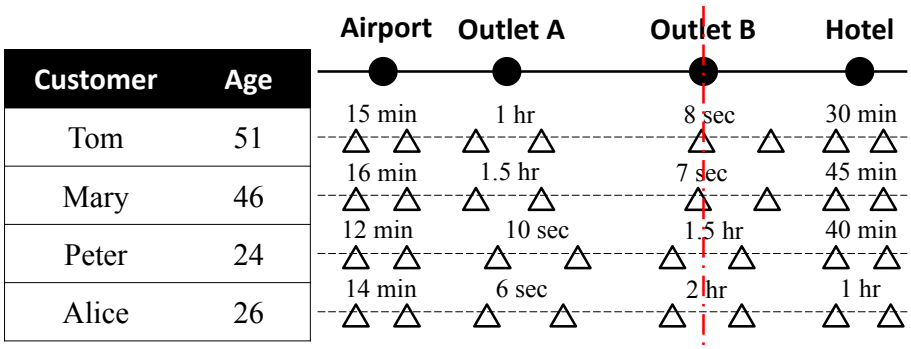


Fig. 1. Illustration of the weakness of the k-BCT query

The proliferation of trajectory data has spawned many novel applications. One example is searching trajectories by locations [1, 6, 7]. Location-based trajectory search was first proposed in [1] as the k *Best-Connected Trajectories* (k-BCT) query. Given a few query locations, a k-BCT query finds k trajectories that are close to all query points from a trajectory database. Location-based trajectory search can benefit users in many real life applications. For example, it can help travelers who are planning a trip to multiple places of interest in an unfamiliar city, by providing similar routes traveled by other people for reference. Location-based trajectory search is also useful in human behavior analysis, where the query locations can be tourist attractions (specified by a travel agency) or the stops of a new metro line (specified by the transport department).

The k-BCT query, however, considers only the spatial aspect of trajectories, which is inadequate for many real applications. Consider a travel agency that queries a database of tourist trajectories for market analysis. Figure 1 shows a database with four trajectories, each belonging to a different tourist. For simplicity, we assume the data space to be 1D rather than 2D, and we only mark the relevant trajectory samples using $\triangle$. For example, Tom spent 15 minutes at the airport (for check-out), 1 hour at Outlet A (for shopping), 8 seconds at Outlet B (just passing by), and 30 minutes at the hotel (for check-in and taking a rest). From Figure 1, we can see that young people (e.g., Peter and Alice) may usually go shopping at Outlet B on their way from the airport to the hotel, while middle-aged people (e.g., Tom and Mary) would prefer to go shopping at Outlet A. Unfortunately, a 2-BCT query over the database with query locations, $\{$Airport, Outlet B, Hotel$\}$, would return the trajectories of Tom and Mary (who actually went shopping at Outlet A), since the 5-th sample in the trajectories of Tom and Mary is closer to Outlet B than any of the samples in the trajectories of Peter and Alice. As a result, the travel agency may make a wrong arrangement: when a tourist bus picks up a group of middle-aged tourists at the airport and goes to the hotel, it would stop at Outlet B for the tourists to go shopping.

This example demonstrates that it is necessary to take location importance into consideration. Although Tom and Mary have a trajectory sample close to Outlet B, the importance of the sample with respect to the whole trajectory is low since Tom and Mary just passed by Outlet B. On the contrary, Peter and Alice went shopping at Outlet B, though their trajectory samples are farther away from Outlet B (the samples were probably recorded at a car park nearby).

In this paper, we propose a new type of location-based trajectory search called the k *Important Connected Trajectories* (k-ICT) query, over a database of trajectories asso-

ciated with location importance. We discuss how to derive the importance of trajectory sample points from their timestamps, and develop efficient algorithms for answering k-ICT queries.

The main contributions of this paper are summarized as follows:

- We propose the k-ICT query over a database of trajectories with location importance, which returns trajectories of much higher utility compared with the k-BCT query [1].
- We design a practical method for deriving the importance of trajectory sample points from their timestamps.
- We propose two R-tree based algorithms for answering k-ICT queries, founded on two variants of *Threshold Algorithm* (TA) for top-k queries.
- We further develop two grid-based algorithms, which process k-ICT queries using our grid index built from the *Multiplicatively Weighted Voronoi Diagram* (MWVD) of trajectories. The grid-based algorithms address the drawbacks of the R-tree based algorithms. Experiments show that the grid-based algorithms are more efficient in terms of both time and space.

The rest of this paper is organized as follows. Section 2 reviews the related work. In Section 3, we formulate the k-ICT query. Section 4 discusses how to derive trajectory location importance from raw GPS data. We present our R-tree based algorithms in Section 5, and describe the grid-based algorithms in Section 6. We report experimental results in Section 7 and conclude the paper in Section 8.

2. Related Work

Conventional Trajectory Search. Given a query trajectory, conventional trajectory search finds k trajectories with the shortest distances to the query trajectory. Definitions of the distance function include [2, 3, 4, 5]. However, these definitions ignore the time dimension of the trajectory samples, and thus may overrate insignificant trajectory samples.

Trajectory Search by Locations. Location-based trajectory search was first proposed by [1], where the query input is a set of locations. Compared with searching trajectories by a complete query trajectory, it is more practical to search trajectories by locations of interest. Consider the example where a traveler is planning a trip to an unfamiliar city. He/she can easily specify the places he/she intends to visit as the query points, by clicking them on a digital map. On the other hand, it is difficult for a new comer to specify a preferred route as the query trajectory. Recent research starts to enhance location-based trajectory search with keywords [6, 7]. However, location importance has not been considered and thus queries may easily overrate insignificant trajectory samples.

Mining Important Locations from Trajectories. There are studies on how to mine important locations from trajectory data, such as raw GPS data [8] and Flickr data [9]. These works measure location importance from all the trajectories. Another work [10] finds important locations from a single trajectory. The work models a trajectory by *stops* and *moves*, where a stop is a semantically important part of the trajectory. They proposed the IB-SMoT algorithm to generate stops: given a database of geographic objects, if a part of trajectory intersects with the object, and the time span of the sub-trajectory is above a minimum time threshold, then the sub-trajectory is identified as a stop. Later work uses density based clustering of the trajectory samples to find stops,

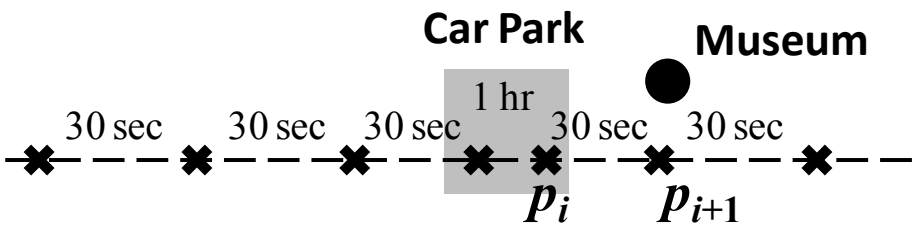


Fig. 2. Intuition behind distance function in Eq. (2)

such as CB-SMoT [11] and DB-SMoT [12]. Conceptually, the samples of a stop are important, while the samples of a move are immaterial. However, these methods do not provide a concrete importance score for the samples (or stops), and thus it is impossible to compare the importance of different samples (or stops).

3. Problem Formulation

We now formally define the k-ICT query. Let D be a database of trajectories, where each trajectory $T \in D$ is a sequence of points $(p_1, p_2, \ldots, p_\ell)$. We assume that each point p_i is associated with a score $w(p_i) \geq 0$, which corresponds to the importance of p_i in trajectory T. For raw GPS data, we can derive the importance score using the time stamps of the trajectory samples, which we will further discuss in Section 4. For trajectories obtained from Flickr photos, the location importance of a photo can be derived using the number of page visits; the score can also be manually set by the photo owner.

A *k*-ICT query, Q, is represented by a set of m locations (or points): $Q = \{q_1, q_2, \ldots, q_m\}$. We first introduce the distance functions that define how a k-ICT query is to be evaluated.

Distance Related to One Query Point. Let us first focus on a specific query point $q_i \in Q$. We define the weighted distance between query point q_i and a trajectory point p_j as follows:

$$d(q_i, p_j) = \frac{\|q_i p_j\|}{w(p_j)}, \tag{1}$$

where we use $\|pq\|$ to denote the Euclidean distance between two points p and q. Note that a larger importance score $w(p_j)$ makes p_j closer to q_i (since $d(q_i, p_j)$ is smaller).

We define the weighted distance between query point q_i and a trajectory $T = (p_1, p_2, \ldots, p_\ell)$ as the weighted distance between q_i and its closest trajectory point in T:

$$d(q_i, T) = \min_{p_j \in T} \{d(q_i, p_j)\}. \tag{2}$$

We now illustrate the intuition behind the distance function in Equation (2). Consider the vehicle GPS trajectory fragment shown in Figure 2, which is generated as follows. A traveler rented a GPS-equipped car to travel around a city. He drove to a car park near a museum, parked his car, stayed in the museum for an hour, and then drove to the next destination. Since the car was turned off when it was parked, the on-board GPS device was also off. As a result, no sample was generated during that one hour

when the car was parked, and $p_i \in T$ is the first sample after the traveler drove the car away from the car park.

In this example, the query point q in question is the museum. Now assume that $w(p)$ is proportional to the time the car stopped at point p. Although $\|qp_{i+1}\| < \|qp_i\|$, it is obvious that $w(p_{i+1}) \ll w(p_i)$ and thus $d(q, p_{i+1}) > d(q, p_i)$ according to Equation (1), i.e. p_i is closer to q than p_{i+1}. Therefore, $d(q_i, T) = d(q, p_i)$ by Equation (2). Note that $d(q, p_i)$ correctly estimates the confidence that the traveler of T visited the museum, since he may instead visit an aquarium nearby after parking his car. In the latter case, $d(q, p_{i+1})$ overestimates the confidence that the traveler visited the museum since he actually visited the aquarium nearby the point p_{i+1}. Thus, $d(q, p_i)$ presents an accurate estimate in this case.

Overall Distance Function. We now define the weighted distance between a query Q and a trajectory T, by aggregating $d(q_i, T)$ for all query points $q_i \in Q$. Since we want to find trajectories close to all query points in Q, we define the overall weighted distance as:

$$d(Q, T) = \sum_{i=1}^{m} d(q_i, T). \tag{3}$$

Intuitively, $d(Q, T)$ is the total distance of traveling from the closest position of T to q_i for all $q_i \in Q$.

In addition to the physical meaning described above, Equation (3) is also meaningful from the probabilistic point of view, which we discuss next. Let us denote p_{n_i} to be the trajectory point of T closest to q_i (in terms of weighted distance), then $d(q_i, T) = d(q_i, p_{n_i})$. We also denote $p(q_i, T)$ to be the probability that the owner of the trajectory T visited q_i, and a reasonable assumption is that $p(q_i, T)$ decays exponentially as $d(q_i, p_{n_i})$ increases. Using the PDF (Probability Density Function) of the exponential distribution, we have $p(q_i, T) = \lambda e^{-\lambda \cdot d(q_i, T)}$. Since we have no preference of one query point over another, we use the same λ for all $q_i \in Q$. Since we want a result trajectory to be close to all query points, the probability that the owner of trajectory T visited all $q_i \in Q$ is:

$$\prod_{i=1}^{m} p(q_i, T) \quad \propto \quad e^{-\lambda \sum_{i=1}^{m} d(q_i, T)}, \tag{4}$$

where we assume that "whether the owner visited one query location" is independent of "whether he visited another query location".

Since we want to maximize the probability value of Equation (4), it is equivalent to minimize $d(Q, T) = \sum_{i=1}^{m} d(q_i, T)$.

The k-BCT query [1] adopts a similarity function $sim(Q, T) = \sum_{i=1}^{m} e^{-d(q_i, T)}$. If we fix $\lambda = 1$, then $sim(Q, T) = \sum_{i=1}^{m} p(q_i, T)$. Compared with Equation (4), this similarity function is undesirable, since the similarity value is high as long as one query point is close to T, even if all other query points are far from T. Similar observation is mentioned in [17], which proposes to use a sum-of-Euclidean-distance measure. In this paper, we use the sum-of-weighted-distance measure to incorporate object importance.

We define k-ICT querying as follows.

Definition 1 (k-ICT Querying). Given a database of trajectories $D = \{T_1, \ldots, T_n\}$ ($n \geq k$), a set of query locations Q, a k-ICT query is to find a set of k trajectories, $R \subseteq D$,

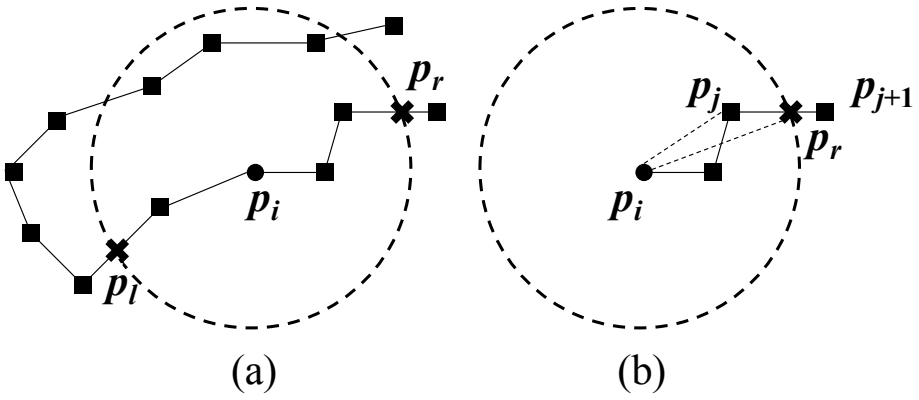


Fig. 3. Illustration of the evaluation of location importance

such that

$$d(Q,T) \leq d(Q,T'),\ \forall T \in R,\ \forall T' \in D - R.$$

4. Location Importance

In this section, we discuss how to compute the importance of trajectory samples from raw GPS data.

Formulation. A GPS reading can be represented by a triplet $(latitude, longitude, timestamp)$. In order to manipulate the data in Euclidean space, we map the coordinates of all sample points from the GPS coordinates $(latitude, longitude)$ to Universal Transverse Mercator (UTM) coordinates $(easting, northing)$, or simply (x, y).

Given a trajectory $T = (p_1, p_2, \ldots, p_\ell)$, where each sample point $p_i = (p_i.x, p_i.y, t(p_i))$, we want to compute the importance $w(p_i)$ for all the sample points $p_i \in T$.

We define the neighborhood of a sample point $p_i \in T$, denoted by $Cir(p_i)$, to be a circle centered at p_i with radius r, where r is a user-specified parameter. Figure 3(a) shows the circle $Cir(p_i)$ and the trajectory T. Let p_l (or respectively, p_r) be the first location on T reaching the boundary of $Cir(p_i)$ when going backward (or respectively, forward) from p_i along T. Note that p_l and p_r may not be an existing trajectory sample, but rather the intersection point between $Cir(p_i)$ and a segment $p_j p_{j+1}$ as shown in Figure 3(b). In this case, we use linear interpolation to compute the location and time stamp of p_r (or p_l). Another extreme case is that p_l (or respectively, p_r) may be the first (or respectively, last) trajectory sample that is inside $Cir(p_i)$, since in this case we cannot go backward (or respectively, forward) from p_i along T.

We define the following measure using p_l and p_r:

$$\Delta t(p_i) = \max\{t(p_r) - t(p_i), t(p_i) - t(p_l)\}. \quad (5)$$

Here, $(t(p_r) - t(p_i))$ is the time spent before the traveler left $Cir(p_i)$ from p_i in the forward direction, while $(t(p_i) - t(p_l))$ is the time spent before the traveler left $Cir(p_i)$ from p_i in the backward direction (or more intuitively, the time spent from when the traveler stepped in $Cir(p_i)$ until he reached p_i). Intuitively, $\Delta t(p_i)$ is defined

such that as long as the traveler stopped near p_i (no matter in the forward or backward direction), the importance of p_i is promoted. The greater $\Delta t(p_i)$ is, the more important the trajectory sample p_i is.

This definition of $\Delta t(p_i)$ has two benefits. First, even when the traveler is in an important location (e.g., a marketplace), he may still be walking around and the accumulated distance can be large. Using a neighborhood circle to cover the marketplace, we can correctly identify that the locations in the marketplace are important. Second, when a GPS-equipped car is turned off, so is the GPS device. Thus, we can only consider the last few samples before the car stops, or the first few samples after the car starts as important (these locations may still be inside the car park), which are better covered using the neighborhood circle.

The next issue is how to compute $w(p_i)$ using $\Delta t(p_i)$. Obviously, $w(p_i)$ should increase fast with $\Delta t(p_i)$ when $\Delta t(p_i)$ is small, but increase slowly when $\Delta t(p_i)$ is large. For example, a 1-minute stop may not be important since it may be due to a red traffic light, while a 10-minute stop is more likely to be important. On the other hand, a 1-hour stop quite certainly implies an important location, and the score should not increase too much even if $\Delta t(p_i)$ becomes 2 hours.

When $\Delta t(p_i) = 0$, we want the importance $w(p_i) = 0$. Furthermore, we want $w(p_i)$ to be within $[0, 1]$ so as to carry a probability meaning: the confidence that the traveler of trajectory T stops at p_i. As a result, we define our importance score as follows:

$$w(p_i) = 1 - e^{-\alpha \cdot \Delta t(p_i)}, \tag{6}$$

where α controls how fast $w(p_i)$ increases with $\Delta t(p_i)$.

Computation Details. Next, we discuss the details of computing $\Delta t(p_i)$. We first describe how we compute p_r (the computation of p_l is similar) for the scenario in Figure 3(b), where $p_j \in T$ is inside $Cir(p_i)$ and the next sample p_{j+1} is outside of $Cir(p_i)$.

Instead of directly computing p_r, we first compute segment length $\|p_j p_r\|$. According to the Cosine Law, we can compute $\cos \angle p_i p_j p_r$ as follows:

$$\cos \angle p_i p_j p_r = \frac{\|p_i p_j\|^2 + \|p_j p_{j+1}\|^2 - \|p_i p_{j+1}\|^2}{2 \cdot \|p_i p_j\| \cdot \|p_j p_{j+1}\|},$$

where $\|p_i p_j\|$, $\|p_j p_{j+1}\|$ and $\|p_i p_{j+1}\|$ can be easily computed from the coordinates of p_i, p_j and p_{j+1}.

Then, according to the Cosine Law, $\|p_j p_r\|$ can be obtained by solving the following quadratic equation:

$$\begin{aligned}\|p_i p_r\|^2 = \|p_i p_j\|^2 + \|p_j p_r\|^2 - \\ 2 \cdot \|p_i p_j\| \cdot \|p_j p_r\| \cdot \cos \angle p_i p_j p_r,\end{aligned} \tag{7}$$

where $\|p_i p_j\|$ is computed from the coordinates of p_i and p_j, and $\|p_i p_r\| = r$. The roots of Equation (7) are:

$$\begin{aligned}\|p_j p_r\|^{(1)} = \|p_i p_j\| \cdot \cos \angle p_i p_j p_r + \\ \sqrt{\|p_i p_j\|^2 \cdot \cos^2 \angle p_i p_j p_r - \|p_i p_j\|^2 + r^2},\end{aligned} \tag{8}$$

$$\begin{aligned}\|p_j p_r\|^{(2)} = \|p_i p_j\| \cdot \cos \angle p_i p_j p_r - \\ \sqrt{\|p_i p_j\|^2 \cdot \cos^2 \angle p_i p_j p_r - \|p_i p_j\|^2 + r^2}.\end{aligned} \tag{9}$$

However, the second root $\|p_j p_r\|^{(2)}$ can be discarded since its value is negative. To see this, recall that p_j is inside $Cir(p_i)$, and hence $\|p_i p_j\| < r$. Thus, we have

$$\begin{aligned}
&\sqrt{\|p_i p_j\|^2 \cdot \cos^2 \angle p_i p_j p_r - \|p_i p_j\|^2 + r^2} \\
> &\sqrt{\|p_i p_j\|^2 \cdot \cos^2 \angle p_i p_j p_r - r^2 + r^2} \\
> &\|p_i p_j\| \cdot \cos \angle p_i p_j p_r.
\end{aligned}$$

Therefore, $\|p_j p_r\|^{(2)} < 0$ according to Equation (9), and we conclude that the value of $\|p_j p_r\|$ is given by Equation (8).

Algorithm 1 Computing $(t(p_r) - t(p_i))$

Input: Trajectory $T = (p_1, p_2, \ldots, p_\ell)$
Output: $\Delta t = t(p_r) - t(p_i)$

1: $\Delta t \leftarrow 0$;
2: **for** $p_j := p_i$ **to** $p_{\ell-1}$ **do**
3: **if** p_{j+1} is inside $Cir(p_i)$ **then**
4: $\Delta t \leftarrow \Delta t + (t(p_{j+1}) - t(p_j))$;
5: **else**
6: Compute $\|p_j p_r\|$ by Equation (8);
7: $\Delta t \leftarrow \Delta t + \frac{\|p_j p_r\|}{\|p_j p_{j+1}\|}(t(p_{j+1}) - t(p_j))$;
8: **return** Δt;
9: **return** Δt;

Now we discuss how to compute $(t(p_r) - t(p_i))$. The value of $(t(p_i) - t(p_l))$ can be computed similarly, and both of them are then used to compute $\Delta t(p_i)$ according to Equation (5). The algorithm is described in Algorithm 1. We check samples forward along T starting from p_i (Line 2). If the next sample p_{j+1} is inside $Cir(p_i)$, then the whole segment $p_j p_{j+1}$ is inside $Cir(p_i)$ (due to the convexity of circles), and we accumulate the time spent on $p_j p_{j+1}$ to the result (Lines 3-4). Otherwise, we compute p_r and accumulate the time spent on $p_j p_r$ to the result (Lines 5-7). Note that in the latter case, we already reach the boundary of $Cir(p_i)$ and thus the accumulated time is directly returned (Line 8).

Parameter Setting. We have two parameters: (1) radius r of $Cir(p_i)$, and (2) the decay rate α in Equation (6). Typically, r is set as the diagonal length of a market place, or the distance between a car park and the intended destination. While the parameter choice is application-dependent, our experiments on several vehicle GPS datasets show that our method always provides reasonable importance score (judged by human) when $r = 50$m and $\alpha = 0.002$. The details are omitted due to the space limitation.

5. R-Tree Based Algorithms

In this section, we introduce two R-tree based algorithms for answering k-ICT queries. Before we present our algorithms, we first describe the *Threshold Algorithm* (TA) [13], since our algorithms adopt the TA framework for top-k query processing.

5.1. Threshold Algorithm and Its Variants

TA [13] has been widely adopted for processing top-k queries, including the k-BCT querying algorithm [1] and the keyword-aware variants [6, 7]. In the setting of [13], we are given a database table D of n tuples, where the schema of the table is $(A_1, A_2, \ldots, A_m)$. For each attribute A_i, a list L_i is built by sorting all the tuples in non-decreasing order of the values of attribute A_i, and stored on disk. Each entry in L_i is a pair (id, val), where id is the id of the corresponding tuple, and val is the value of attribute A_i for the tuple. We describe two algorithms that use the lists to find the top-k tuples, where the ranking score of a tuple equals the summation of the values of all its m attributes (a smaller score is preferred).

Fagin's Algorithm (FA). FA finds the top-k tuples in three steps. *Step 1*: read a (id, val) pair from each list in a round-robin manner, until there are k tuples whose id's have been seen from all the m lists. *Step 2*: for each tuple id seen (from any list), retrieve the tuple from the table D if any of its attribute values are missing (random access to D is needed). *Step 3*: compute the ranking score by summing the attribute values for each tuple whose id has been seen, and return the k tuples with the smallest summation values.

Threshold Algorithm (TA). Unlike the *filter-and-refine* framework of FA where random access to D is only used in the refinement step, TA adopts a more aggressive approach. TA also reads a (id, val) pair from each list in a round-robin manner, but for each tuple id seen, TA immediately retrieves the tuple from D by random access, computes the ranking score, and updates the current top-k tuples. Meanwhile, for each list L_i, TA maintains a variable τ_i equal to val of the last (id, val) pair read from L_i. The round-robin operation stops when the ranking score of the current top k-th tuple is equal to or smaller than $\sum_{i=1}^{m} \tau_i$.

In general, TA reads less pairs from the lists than FA, but performs more random accesses to D than FA. While we focus on FA and TA when introducing our algorithms, other variants of TA may also be adopted by our algorithm.

5.2. R-Tree Based Algorithms

We now present our R-tree based algorithms. We first describe a key operator used by our algorithms: the *incremental weighted nearest-neighbor* (*NN*) algorithm.

Incremental Weighted NN Algorithm. Unlike [1], in our problem, each trajectory point p_i is associated with an importance score $w(p_i)$, and its distance to a query point q is evaluated as $\frac{\|qp_i\|}{w(p_i)}$ using Equation (1). Thus, R-tree is no longer sufficient for solving our problem. Instead, we index the trajectory points by an *aggregate R-tree* (*aR-tree*) [14] with aggregate function MAX, called *MAX R-tree*.

Compared with a traditional R-tree, each node entry e of a MAX R-tree maintains the maximum importance score among all points indexed under e (i.e., indexed in the subtree rooted at the node pointed to by e). Given an R-tree node entry e, we denote its MAX aggregate value by $w(e)$. We also denote the *Minimum Bounding Rectangle* (*MBR*) of e by $mbr(e)$. Then, for any trajectory point p indexed under e, its weighted distance to query point q is given by:

$$d(q,p) = \frac{\|qp\|}{w(p)} \geq \frac{mindist(q, mbr(e))}{w(e)}, \tag{10}$$

where $mindist(q, mbr(e))$ is the distance from q to its closest point in $mbr(e)$. The inequality holds as $mindist(q, mbr(e)) \leq \|qp\|$ and $w(e) \geq w(p)$.

For simplicity, given an R-tree node entry e and a query point q, we define:

$$LB(q, e) = \frac{mindist(q, mbr(e))}{w(e)}. \quad (11)$$

According to Equation (10), $LB(q, e)$ lower bounds the weighted distance from any point indexed under e to q.

Algorithm 2 Computing the Next Weighted NN of q_i

Input: query location q_i, priority queue *min-heap*, Max R-tree $tree$
Output: $(d(q_i, p), p)$ where p is the next NN of q_i

```
1:  while min-heap is not empty do
2:    (LB(q_i, e), e) ← min-heap.dequeue();
3:    if e is a leaf node entry then
4:      p ← the trajectory point pointed to by e;
5:      return (LB(q_i, e), p);
6:    else
7:      node ← the R-tree node pointed to by e;
8:      for each entry e' of node do
9:        Compute LB(q_i, e');
10:       min-heap.enqueue(LB(q_i, e'), e');
```

Algorithm 2 describes our incremental weighted NN algorithm. When processing a k-ICT query, we maintain a priority queue of R-tree node entries for each query point q_i, so that the next NN of q_i can be incrementally obtained using Algorithm 2. Initially, the priority queue *min-heap* contains only the root node of the Max R-tree $tree$, and each call of Algorithm 2 updates *min-heap* and retrieves the next NN of q_i.

We now explain Algorithm 2 in details. In each round, the entry e with the smallest $LB(q_i, e)$ is dequeued from *min-heap* (Line 2). If e is an entry of a leaf node, then it points to a trajectory point p and $LB(q_i, e) = d(q_i, p)$. In this case, we can conclude that p is the next NN (Line 5), since any unseen trajectory point p' is indexed under some node entry en in *min-heap*, and $d(q_i, p') \geq LB(q_i, en) \geq LB(q_i, e)$. Otherwise, e is an entry of a non-leaf node $node$, and we enqueue all the entries of $node$ into *min-heap* (Lines 7-10).

R-Tree based FA and TA. We now introduce our two R-tree based algorithms for answering k-ICT queries, one based on FA and the other based on TA. Both algorithms use Algorithm 2 for sequentially accessing the next NN of each query point q_i.

Algorithm 3 presents the R-tree based FA for answering k-ICT queries. Similar to FA, Algorithm 3 has two phases: the filtering phase (Lines 3-15) and the refinement phase (Lines 16-20).

In each round of the filtering phase, Algorithm 3 obtains the next NN of each q_i for processing (Line 4), i.e., the NNs of the query points are processed in a round-robin manner. Since there are N trajectory points indexed by $tree$, there are at most N rounds (Line 3). All the seen trajectories are maintained using a hash table $table$, where the hash key is the trajectory id. If the trajectory of the obtained point p has not been seen yet, we know that $d(q_i, T) = d(q_i, p)$, and thus we insert T into $table$ and record $d(q_i, p)$ as the value of the i-th attribute, denoted by $T[i]$ (Lines 7-9). Otherwise, T is already in $table$,

Algorithm 3 R-tree based FA for Answering k-ICT Queries

Input: k, query set Q, trajectory database D, Max R-tree $tree$
Output: k-ICT (the top-k trajectories)

1: $table \leftarrow \emptyset, d \leftarrow 1$;
2: $N \leftarrow$ number of trajectory points in D;
3: **while** $d \leq N$ **do**
4: **for each** $q_i \in Q$ **do**
5: Retrieve the d-th NN p of q_i, together with the weighted distance $d(q_i, p)$, using Algorithm 2;
6: $T \leftarrow$ the trajectory that p belongs to;
7: **if** $T \notin table$ **then**
8: Insert T into $table$;
9: $T[i] \leftarrow d(q_i, p)$; $/* \, T[i]$ represents $d(q_i, T) \, */$
10: **else**
11: **if** $T[i]$ is not yet assigned **then**
12: $T[i] \leftarrow d(q_i, p)$;
13: **if** there are k trajectories in $table$ whose attribute values are all assigned **then**
14: **goto** Line 16;
15: $d \leftarrow d + 1$;
16: **for each** $T \in table$ **do**
17: Read T;
18: For any $T[i]$ not yet assigned: $T[i] \leftarrow d(q_i, T)$;
19: Compute $d(Q, T) = \sum_{i=1}^{m} T[i]$;
20: Update the top-k trajectories;
21: **return** the top-k trajectories;

and we check whether $T[i]$ has been assigned a value (Line 11). If $T[i]$ has already been assigned a value, we ignore the obtained point p since the point in T that is closest to q_i has already been processed before. Otherwise, p is the point in T closest to q_i, and we set $T[i]$ to be $d(q_i, p)$. The filtering phase terminates once k tuples are seen with the value of $T[i]$ assigned for all $q_i \in Q$, which is similar to the traditional FA.

In the refinement phase, we first compute $d(q_i, T)$ for any $T[i]$ whose value has not yet been assigned (a more efficient method of obtaining $T[i]$ is actually used, which we will discuss in Section 6.2). Then, for all the seen trajectories $T \in table$, $d(Q, T)$ is computed and the k tuples with the smallest values of $d(Q, T)$ are returned.

Algorithm 4 presents the R-tree based TA for answering k-ICT queries. Recall that the conventional TA maintains a variable τ_i for each list L_i, whose value equals the attribute value of the last accessed entry. TA stops when the ranking score of the current top k-th tuple is equal to or smaller than $\sum_{i=1}^{m} \tau_i$. In our problem, $\tau_i = d(q_i, p)$ where p is last accessed NN of q_i. We set τ to 0 at the beginning of a round-robin processing round (Line 5), and add $\tau_i = d(q_i, p)$ to τ for each query point q_i (Line 16). Therefore, at the end of the round-robin processing round, $\tau = \sum_{i=1}^{m} \tau_i$ is exactly the pruning threshold, which is then compared with the top k-th trajectory in Line 17 to determine the stopping condition.

In Lines 9-15, we only process the trajectory T of the current point p if T is not in $table$, by accessing T to assign $T[i]$ (a more efficient method discussed in Section 6.2 is actually used here), computing $d(Q, T)$ and updating the top-k results. Note that if T

Algorithm 4 R-tree based TA for Answering k-ICT Queries

Input: k, query set Q, trajectory database D, Max R-tree $tree$
Output: k-ICT (the top-k trajectories)

1: $table \leftarrow \emptyset, d \leftarrow 1$;
2: $N \leftarrow$ number of trajectory points in D;
3: *max-heap*$\leftarrow \emptyset$;
4: **while** $d \leq N$ **do**
5: $\tau \leftarrow 0$;
6: **for each** $q_i \in Q$ **do**
7: Retrieve the d-th NN p of q_i, together with the weighted distance $d(q_i, p)$, using Algorithm 2;
8: $T \leftarrow$ the trajectory that p belongs to;
9: **if** $T \notin table$ **then**
10: $T[i] \leftarrow d(q_i, p)$;
11: $\forall j \neq i$, compute $T[j] = d(q_j, T)$ by accessing T;
12: Insert T into $table$;
13: Compute $d(Q, T) = \sum_{i=1}^{m} T[i]$;
14: *max-heap*$.enqueue(d(Q, T), T)$;
15: If *max-heap*$.size() > k$: *max-heap*$.dequeue()$;
16: $\tau \leftarrow \tau + d(q_i, p)$;
17: **if** *max-heap*$.size() = k$ **and** *max-heap*$.top() \leq \tau$ **then**
18: **goto** Line 20;
19: $d \leftarrow d + 1$;
20: **return** the k trajectories in *max-heap*;

is in $table$, then $T[i]$ must have been assigned for all $i = 1, \ldots, m$ (Lines 10-12), and hence we can ignore T.

Finally, we note that the correctness of both Algorithms 3 and 4 is easy to see by following the correctness of FA and TA [13]. We thus omit the details here.

Limitations of R-Tree Based Algorithms. We identify the following limitations of using an R-tree index built over all the trajectory points in the database, which motivates our grid-based algorithm to be introduced in Section 6.

Firstly, the incremental NN search for each query q_i is done over an R-tree that contains all the trajectory samples. However, if we know q_i beforehand, then only one sample per trajectory requires examining (i.e., the sample with the shortest weighted distance to q_i), and there are totally $n = |D|$ such samples, much less than the number of all samples in D. Therefore, there is huge room for improvement in terms of sample candidate pruning.

Secondly, much of the computation done by the R-tree based algorithms could be wasteful. This is because consecutive samples of a trajectory are close in space, and are very likely indexed under the same R-tree node. As a result, in consecutive calls of Algorithm 2 for retrieving the NNs of a query point q_i, many returned NNs may come from the same trajectory.

Finally, we use the maximum importance $w(e)$ of an R-tree node entry to compute the lower bound in Equation (11), which is not tight. As long as there is one point indexed under e with a large weight, the whole entry e has to be accessed early even if all the other points have very low weight, resulting in the addition of all its child nodes into the priority queue.

6. Grid-Based Algorithms

In this section, we present the grid-based algorithms.

Overview. We first give an overview of how our grid-based algorithms address all the three drawbacks of the R-tree based algorithms mentioned in Section 5.2.

Firstly, to avoid doing NN search over all trajectory points, we divide the data space by a grid, so that each grid cell covers a small region. We observe that only a small fraction of samples per trajectory have the chance to be the NN of some location in a cell. Thus, if a query point locates in a grid cell, we only need to check the samples relevant to the cell.

Secondly, to avoid checking a lot of samples of a trajectory that do not contribute to the top-k answers, we propose to pre-compute the *Multiplicatively Weighted Voronoi Diagram* (*MWVD*) of the points of each trajectory. Note that a sample p_i is the weighted NN of q if and only if q locates inside the Voronoi cell of p_i.

Finally, to avoid the interference of samples from different trajectories, we treat trajectories as the first-class citizen (while the R-tree index treats the trajectory points as the first-class citizen). Given a grid cell, we group all its relevant samples by trajectories, and the NN search is done in the unit of trajectories rather than trajectory points.

We discuss these ideas in details in the following subsections.

6.1. Trajectory Preprocessing by MWVD

For each trajectory $T = (p_1, p_2, \ldots, p_\ell)$, we pre-compute the MWVD [15] of its points, which is then used to build our grid index. We first briefly review the MWVD and then show how we use it in our solution.

Let U be the data domain. Given two samples p and p', the *dominant region* of p over p' is defined as:

$$R_{p|p'} = \{q \in U \mid d(q,p) \leq d(q,p')\}.$$

We now consider the shape of $R_{p|p'}$. Let us first assume that $w(p) < w(p')$, then $R_{p|p'}$ is characterized by the region within circle $C_{p|p'}$, whose center c and radius r are given as follows:

$$c = \left(\frac{w^2(p') \cdot p.x - w^2(p) \cdot p'.x}{w^2(p') - w^2(p)}, \frac{w^2(p') \cdot p.y - w^2(p) \cdot p'.y}{w^2(p') - w^2(p)} \right)$$

$$r = \frac{w(p) \cdot w(p') \cdot \|pp'\|}{w^2(p') - w^2(p)}.$$

Figure 4 illustrates the concept of dominant region with circle $C_{p|p'}$. In fact, $C_{p|p'}$ is an Apollonius circle, since for any point x on its boundary, $\frac{\|px\|}{\|p'x\|} = \frac{w(p)}{w(p')}$.

When $w(p) > w(p')$, $R_{p|p'}$ is characterized by the region outside of circle $C_{p|p'}$. For example, in Figure 4 where $w(p') > w(p)$, $R_{p'|p} = U - C_{p|p'}$. Finally, when $w(p) = w(p')$, the perpendicular bisector of pp' divides the space into two half planes, and $R_{p|p'}$ corresponds to the half plane that contains p, denoted by $H_{p|p'}$.

The Voronoi cell of a trajectory point $p \in T$ is given by:

$$VC(p) = \bigcap_{p' \in T - \{p\}} R_{p|p'}, \tag{12}$$

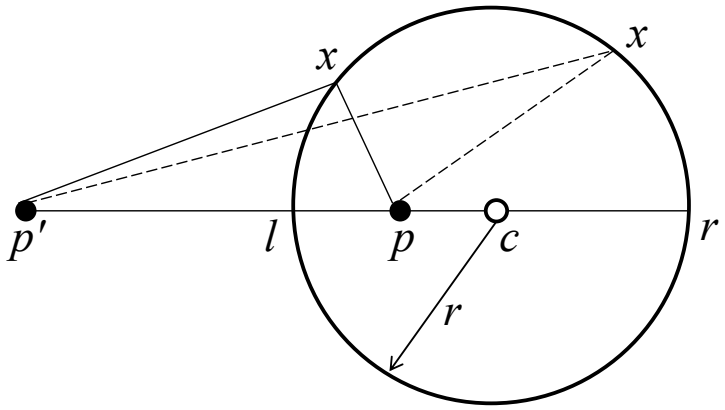


Fig. 4. Illustration of dominant regions

since any point in $VC(p)$ should be in $R_{p|p'}$ for any $p' \in T - \{p\}$. Given a sample $p \in T$, we divide the other samples in T into three sets: T^+ contains all samples p' with $w(p') > w(p)$, T^- contains all samples p' with $w(p') < w(p)$, and T^0 contains all samples p' with $w(p') = w(p)$.

Equation (12) implies that $VC(p)$ may be represented by $\ell - 1$ circles or lines in the worst case. In fact, not all circles/lines contribute to the final shape of $VC(p)$ and many of them can be pruned by the six pruning rules presented in [16]. We adopt the best-first search algorithm of [16] for MWVD computation, but the computation is done in memory since the number of points in each trajectory is usually not large.

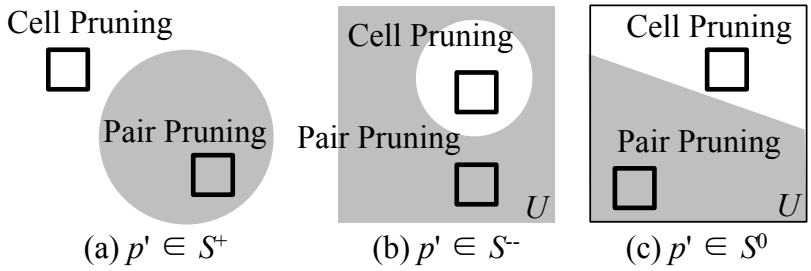
6.2. Grid Index

Next, we describe two indices used in our grid-based algorithms. In our problem, we assume that there exists a rectangular data space U, such that all trajectory points and query points locate inside U. For example, U can be the bounding box of a city region. Our grid-based approach divides U by an $N \times N$ grid, denoted by G.

For each trajectory T, we build a *random access* index, denoted by $RAI[T]$, which returns $d(q, T)$ given a query point q; while for each grid cell $G[i, j]$, we build a *sequential access* index, denoted by $SAI[i, j]$, which returns trajectories in non-decreasing order of $d(q, T)$ for a query point q falling in $G[i, j]$.

Random Access Index (RAI). We now describe how we build $RAI[T]$. First, we compute $VC(p)$ for all $p \in T$, where $VC(p)$ is represented by a set of pairs $(p', R_{p|p'})$. We say that p' is *related to* $VP(p)$ if $(p', R_{p|p'}) \in VC(p)$. Then, for each grid cell $G[i, j]$, we compute the set of Voronoi cells overlapping with the rectangular region R that $G[i, j]$ represents. We denote the set by $S(R) = \{VC_R(p_{i_1}), VC_R(p_{i_2}), \ldots, VC_R(p_{i_s}))\}$. Note that a Voronoi cell $VC_R(p)$ may contain less pairs of $(p', R_{p|p'})$ than the original $VC(p)$, since we only need to characterize its shape within R. If $VC(p)$ does not overlap with R, p cannot be the weighted NN of any location in R, and is thus pruned.

We now consider how to compute $VC_R(p)$ from the original $VC(p)$. We divide the trajectory points p' related to $VC(p)$ into three sets S^+, S^- and S^0, according to



Condition 1	Condition 2	Action
$p' \in S^+$	$G[i, j]$ is outside of $C_{p \mid p'}$	Prune $VC(p)$
$p' \in S^+$	$C_{p \mid p'}$ contains $G[i, j]$	Prune $(p', R_{p \mid p'})$
$p' \in S^-$	$C_{p' \mid p}$ contains $G[i, j]$	Prune $VC(p)$
$p' \in S^-$	$G[i, j]$ is outside of $C_{p' \mid p}$	Prune $(p', R_{p \mid p'})$
$p' \in S^0$	$H_{p' \mid p}$ contains $G[i, j]$	Prune $VC(p)$
$p' \in S^0$	$H_{p \mid p'}$ contains $G[i, j]$	Prune $(p', R_{p \mid p'})$

Fig. 5. Cell Pruning & Pair Pruning

whether p' belongs to T^+, T^- and T^0, respectively. We check each $(p', R_{p|p'}) \in VC(p)$ in turn for the following:

– *Cell Pruning:* if $R_{p|p'}$ does not overlap with R, we prune $VC_R(p)$ immediately since $VC(p) \cap R_{p|p'} = \emptyset$;
– *Pair Pruning:* if $R_{p|p'}$ contains R, then p' has no contribution to the shape of $VC_R(p)$, and thus $(p', R_{p|p'})$ is not included in $VC_R(p)$;
– Otherwise, $(p', R_{p|p'})$ is added to $VC_R(p)$.

Figure 5 lists the conditions for Cell Pruning and Pair Pruning when $p' \in S^+$, S^- and S^0.

In our implementation, we do not compute $S(R)$ for each grid $G[i, j]$ with region R directly from the original Voronoi set. Instead, we perform the computation by building a quadtree $qtree$ whose leaf nodes correspond to the grid cells. By specifying the height of the quadtree as h, we obtain a $2^h \times 2^h$ grid (i.e., $N = 2^h$).

Each quadtree node, $node$, is associated with a region $node.R$ and a set of the Voronoi cells overlapping with $node.R$, i.e., $S(node.R)$. Algorithm 5 shows how we compute $S(node.R)$ for each quadtree node $node$ in a recursive manner. Let the quadtree root be $root$ with $root.R = U$ and $S(root.R) = \{VC(p_1), \cdots, VC(p_\ell)\}$, the recursion is initiated over each child node of $root$ with $level = 1$. For each node, we compute its Voronoi cell set only from that of its parent (Line 3). If the set contains only one Voronoi cell $V_{node.R}(p)$, then for any location in $node.R$, p is its weighted NN. We stop recursion in that case (Line 5). Otherwise, if the current level is not the leaf level, we continue to split $node$ and construct its four children (Lines 6-8).

Algorithm 5 Computing Quadtree Node $node$

Input: Current node $node$, Parent node par, current level $level$

1: $S(node.R) \leftarrow \emptyset$;
2: **for each** $VC_{par.R}(p) \in S(par.R)$ **do**
3: Compute $VC_{node.R}(p)$ by checking the pairs in $VC_{par.R}(p)$, and do the pruning listed in Figure 5;
4: If $VC_{node.R}(p)$ is not pruned, add it to $S(node.R)$;
5: **if** $level < h$ and $|S(node.R)| > 1$ **then**
6: Split $node.R$ into four equal quadrants, $R_i,\ i = 1, 2, 3, 4$;
7: Create child nodes $ch_i,\ i = 1, 2, 3, 4$ with $ch_i.R = R_i$;
8: Recurse over each child node;

After the quadtree $qtree$ is constructed, for all its nodes $node$, $S(node.R)$ is already computed. Then, for each grid cell $G[i, j]$ with region R, we compute the set of trajectory points whose Voronoi cells overlap with R, denoted by $C_T[i, j]$. We compute $C_T[i, j]$ by finding the leaf node, $leaf$, that contains the center of R using $qtree$; and for each $VC_{leaf.R}(p) \in S(leaf.R)$, we add the corresponding trajectory point p into $C_T[i, j]$.

It is easy to see that, for any query location in R, its weighted NN must be some trajectory point in $C_T[i, j]$. We call $C_T[i, j]$ as the candidate set of $G[i, j]$ from now on. For each trajectory T, we store C_T, which is an $N \times N$ array of trajectory point lists, on disk as the random access index $RAI[T]$.

Given a query point q, we identify the grid cell $G[i, j]$ that q locates in, load the list $C_T[i, j]$ into memory, and compute $d(q, T)$ as follows:

$$d(q, T) = \min_{p \in C_T[i,j]} \{d(q, p)\}. \tag{13}$$

Compared with loading the whole trajectory T in memory, it is more efficient to obtain $d(q, T)$ using this random access index, since $|C_T[i, j]|$ is much smaller than the trajectory length ℓ. Therefore, in our implementation, we use this index to compute $d(q, T)$ instead of accessing T directly (recall Lines 17-18 of Algorithm 3 and Line 11 of Algorithm 4).

Sequential Access Index (SAI). For each grid cell $G[i, j]$, we also build a list $L[i, j]$ for retrieving trajectories T in non-decreasing order of $d(q, T)$, where query point q locates in $G[i, j]$. Since q can be any location in $G[i, j]$, the value of $d(q, T)$ is not fixed beforehand. We compute the lower bound of $d(q, T)$ instead, denoted by $LB(q, T)$, which is given by:

$$LB(q, T) = \min_{p \in C_T[i,j]} \left\{ \frac{mindist(p, R)}{w(p)} \right\}, \tag{14}$$

where R is the region of $G[i, j]$.

Each trajectory T has an entry in $L[i, j]$, represented by $en(T) = (T, C_T[i, j], LB(q, T))$. The list $L[i, j]$ is constructed by sorting the entries in non-decreasing order of $LB(q, T)$. We store the $N \times N$ list array L on disk.

Given a query point q that falls in $G[i, j]$, in order to retrieve trajectories in non-decreasing order of $d(q, T)$ using $L[i, j]$, we maintain a priority queue *min-heap* in main memory. We get the next trajectory T with the smallest value of $d(q, T)$ in two steps:

– We read the next entry $en(T)$ from $L[i,j]$, evaluate $d(q,T)$ using Equation (13), and add $(T, d(q,T))$ into *min-heap*. The process is repeated until the value $d(q,T')$, where $T' =$ *min-heap*$.top()$, is smaller than the $LB(q,T)$ of the last accessed entry $en(T)$. Note that all subsequent entries have lower bound values larger than $d(q,T')$.
– We return $T' =$ *min-heap*$.top()$ as the next NN, and remove it from *min-heap*.

The priority queue *min-heap* is a memory buffer that reorders the trajectories in $L[i,j]$ by $d(q,T)$, and we call it as the sequential access index of $G[i,j]$, denoted by $SAI[i,j]$.

Grid-based Algorithms. Our two grid-based algorithms also follow the FA and TA frameworks, respectively, but use the grid index (i.e., the RAI and SAI) in replace of the R-tree index.

The grid-based FA differs from Algorithm 3 in the following aspects:

– Line 2 now becomes "$N \leftarrow n$", where $n = |D|$;
– Line 5 is now replaced by "retrieve the d-th NN of q_i using $SAI[j,k]$, where q_i falls in $G[j,k]$";
– Line 6 is no longer necessary since $SAI[j,k]$ directly returns the trajectory T along with $d(q_i,T)$;
– Lines 9 and 12 now become "$T[i] \leftarrow d(q_i,T)$";
– We no longer need to do the checking in Line 11, since each T will be accessed only once for each query point q_i.

The grid-based TA differs from Algorithm 4 in the following aspects:

– Line 2 now becomes "$N \leftarrow n$";
– Line 7 is now replaced by "retrieve the d-th NN of q_i using $SAI[j,k]$, where q_i falls in $G[j,k]$";
– Line 8 is no longer necessary;
– Line 10 now becomes "$T[i] \leftarrow d(q_i,T)$";

Extension to Skewed Trajectory Distribution. Our current algorithm uses a uniform grid to partition the rectangular data space U. Our experiments show that our algorithm works quite well on the datasets with trajectories relatively uniformly distributed over U. However, it is not the best choice when the trajectory distribution is skewed.

Although the road network of most regions occupies the majority of the region's bounding box U (e.g., Colorado), it is not always true. For example, in the bounding box of Florida, most regions correspond to the ocean where no trajectory can exist, and it is meaningless to divide such regions into grid cells. Furthermore, there are usually much more trajectory points in city centers than in outskirts, and thus dense regions should be divided into finer granularity.

We proposes a heuristics to handle data skewness. Specifically, we first build a linear quadtree index over all the trajectory points. Then, we build our RAI and SAI indices over the leaf nodes of the linear quadtree. We have conducted experiments to compare the performance of using uniform grid with that of using linear quadtree over skewed trajectory data, and found that the latter is an order of magnitude faster than the former, and achieves similar performance compared with using uniform grid over relatively uniform trajectory data.

7. Experimental Results

In this section, we evaluate the performance of our algorithms: *RTree-TA*, *RTree-FA*, *Grid-TA*, and *Grid-FA*. We implemented our algorithms in JAVA. All the experiments were run on a public Linux server with eight 3GHz Intel CPU and 32GB memory.

7.1. Datasets and Query-sets

We first describe the datasets and query-sets used in our experiments.

Datasets. We use two following two real trajectory datasets:

– *Trucks*[1]: This dataset consists of 276 trajectories of 50 trucks delivering concrete to several construction places around Athens metropolitan area in Greece for 33 distinct days.
– *SchoolBuses*[2]: This dataset consists of 145 trajectories of 2 school buses collecting (and delivering) students around Athens metropolitan area in Greece for 108 distinct days.

For both datasets, the length of the trajectories is in the order of hundreds. We choose these datasets since there exists some important locations in their underlying applications, such as construction places and schools.

To further study the scalability of our algorithms when the number of trajectories increases, we generate synthetic datasets based on the *Trucks* dataset. Specifically, to generate a dataset with n trajectories, we repeat the following operations n times: (1) randomly pick a trajectory from the *Trucks* dataset; (2) shift it in a random direction by a small randomly generated distance (within 200m); (3) insert the new trajectory into the synthetic dataset.

We generate synthetic datasets from a real dataset since we want the generated trajectories to exhibit the properties of real trajectories.

Query-sets. We do not generate query locations randomly, since vehicle trajectories usually follow the underlying road network. Moreover, a query location in a sparse region not covered by the road network is meaningless in real applications.

We generate a meaningful query-set containing m query locations in the following way: (1) randomly pick a trajectory from the trajectory dataset to query over; (2) pick the top-10% points of the trajectory in terms of importance; (3) randomly select m locations from these points without replacement; (4) shift these locations in a random direction by a small randomly generated distance (within 200m), and add them to the query-set.

In this way, we are generating meaningful query locations which are important and correlated for at least one trajectory in the dataset.

7.2. Evaluation Measures

The k-ICT query has two query parameters: (1) the number of query points, m; and (2) the number of trajectories, k, that the user wants the query to return. These parameters

[1] http://www.chorochronos.org/?q=node/5
[2] http://www.chorochronos.org/?q=node/6

Table 1. Top-5 Query Answers

Metrics	top-1	top-2	top-3	top-4	top-5
Weighted Distance	(40, 81189.4)	(33, 85939.5)	(232, 86572.9)	(224, 87504.1)	(123, 87625.2)
Euclidean Distance	(221, 95943.7)	(40, 81189.4)	(198, 126783.7)	(232, 86572.9)	(28, 94666.5)

are usually small in real applications. We also have a parameter for the dataset, which is the number of trajectories, n.

We measure the following four costs of our algorithms when the above parameters change: (1) CPU time; (2) number of blocks accessed by sequential index (the Max R-tree, or the grid index SAI); (3) number of blocks accessed by random index (the grid index RAI); (4) number of priority queue entries in main memory.

Since our algorithms are I/O bound, the number of blocks accessed by sequential/random indices are the most important performance criteria. When using the grid index $SAI[i, j]$ for a query point locating in $G[i, j]$, we maintain a main memory buffer of one block which is refilled from $L[i, j]$ whenever it is used up. Therefore, we can use the number of blocks accessed to evaluate the I/O cost. As for R-tree, the nodes are loaded in blocks, and thus the number of blocks accessed can be measured.

The smaller memory a query requires, the more queries a server can handle simultaneously. Therefore, we also measure the memory cost of our algorithms. For the R-tree based algorithms, the memory cost is dominated by the priority queue *min-heap* used for NN search (see Algorithm 2), while for grid-based ones, the memory cost is dominated by the priority queue of $SAI[i, j]$ for reordering $L[i, j]$ (see Section 6.2). The total number of memory entries equals the sum of the entries in the priority queue for each query point q_i, and we report the maximum number among all the round-robin iterations. Throughout the experiments, we fix $r = 50$ m and $\alpha = 0.002$ when generating sample importance using the method discussed in Section 4, and fix the size of a block as 512 bytes. We generate 1000 queries in each experiment, and all results are averaged over the 1000 runs.

7.3. Effect of Query Parameters

To study the performance of our algorithms with respect to the query parameters m and k, we build grid indices over the *Trucks* and *SchoolBuses* datasets, by constructing a quadtree of height $h = 5$. Accordingly, the grid we use is of size 32×32.

To study the effect of m, we fix k as 5 and process queries with $m = 1, 2, \ldots, 10$. On the other hand, to study the effect of k, we fix m as 3 and process queries with $k = 1, 2, \ldots, 10$.

Figure 6 reports the performance of our algorithms for processing k-ICT queries over the *Trucks* dataset when $k = 5$ and the number of query points m increases from 1 to 10.

Figure 6(a) shows that the CPU time of *RTree-FA* is much larger than the other three algorithms, while the grid-based algorithms record the shortest CPU time.

Since all our algorithms are I/O bound, the results reported in Figure 6(b) and (c) dominate the overall performance of query processing. According to Figure 6(b), *RTree-FA* requires reading a lot of blocks (or R-tree nodes) for the incremental NN search, and both of the R-tree based algorithms read significantly more blocks for sequential access than the grid-based algorithms. For example, when $m = 5$, *RTree-FA* reads over 1844 blocks while *Grid-FA* reads only 87 blocks. For random access, Figure 6(c) shows that *Grid-FA* (or respectively, *Grid-TA*) also reads fewer blocks than *RTree-FA* (or

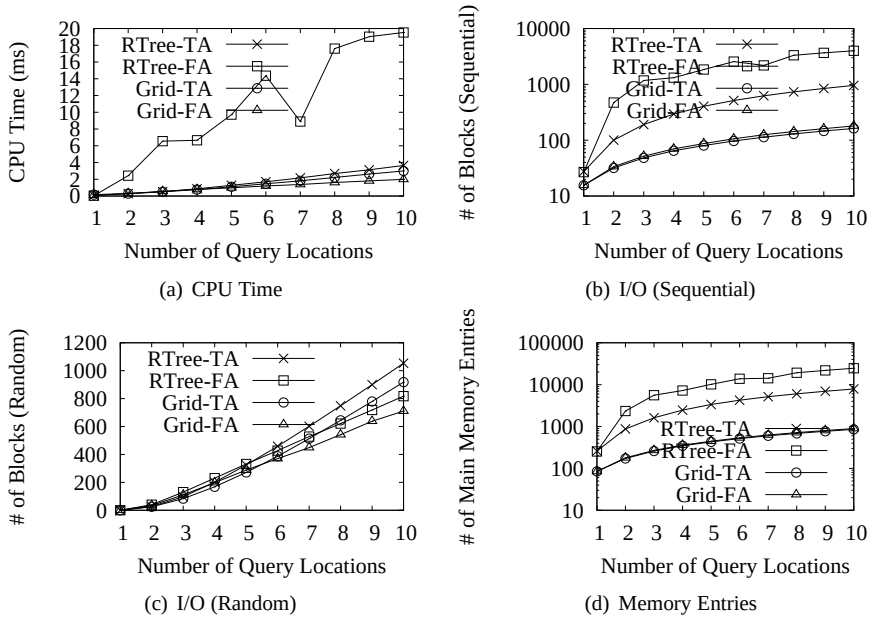


Fig. 6. Effect of m using the Trucks dataset

respectively, *RTree-TA*), though the difference is not as big as in the case of sequential access.

Overall, *Grid-TA* is slightly faster than *Grid-FA*, several times faster than *RTree-TA*, and an order of magnitude faster than *RTree-FA*.

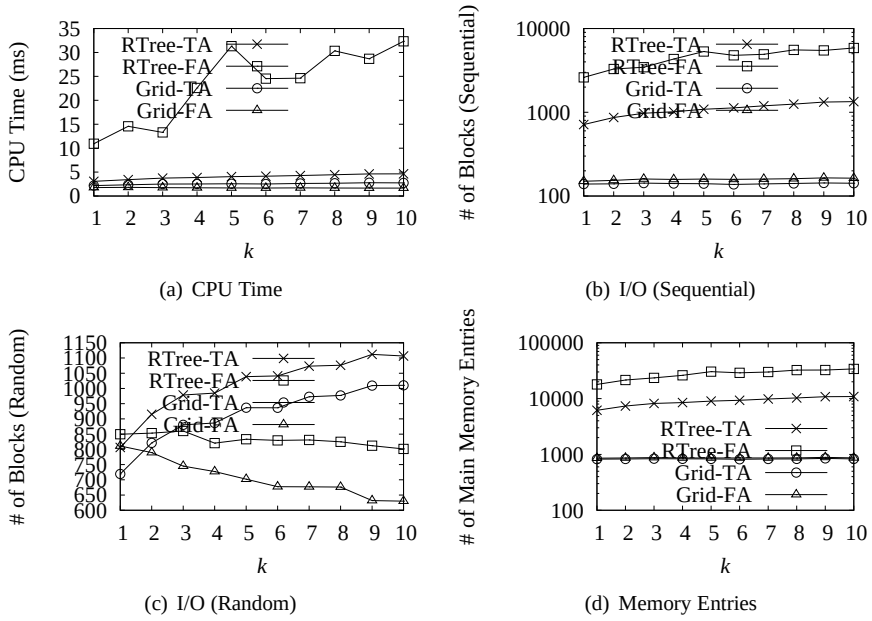
Figure 6(d) shows that the number of data entries maintained in memory by *RTree-TA* and by *RTree-FA* is from several times to tens of times larger than that by both of the grid-based algorithms. Given the fact that the size of an entry maintained by the grid index is much smaller than an R-tree node entry (which contains MBR and weight besides the node pointer), the grid-based algorithms are much more memory-efficient than the R-tree based ones.

Figure 7 reports the performance of our algorithms over the *Trucks* dataset when $m = 3$ and k increases from 1 to 10. The results are similar to that of increasing m we just discussed, except for the I/O cost of random access. As shown in Figure 7(c), the two FA-based algorithms, *RTree-FA* and *Grid-FA*, read fewer blocks when m increases, while the two TA-based algorithms, *RTree-TA* and *Grid-TA*, read more blocks when m increases.

As for the *SchoolBuses* dataset, Figure 8 reports the performance of our algorithms when m changes, and Figure 9 reports the performance of our algorithms when k changes. It can be observed that the performance trend of the algorithms is similar to that for the Trucks dataset discussed above (we thus omit the details).

7.4. Results of Scalability Test

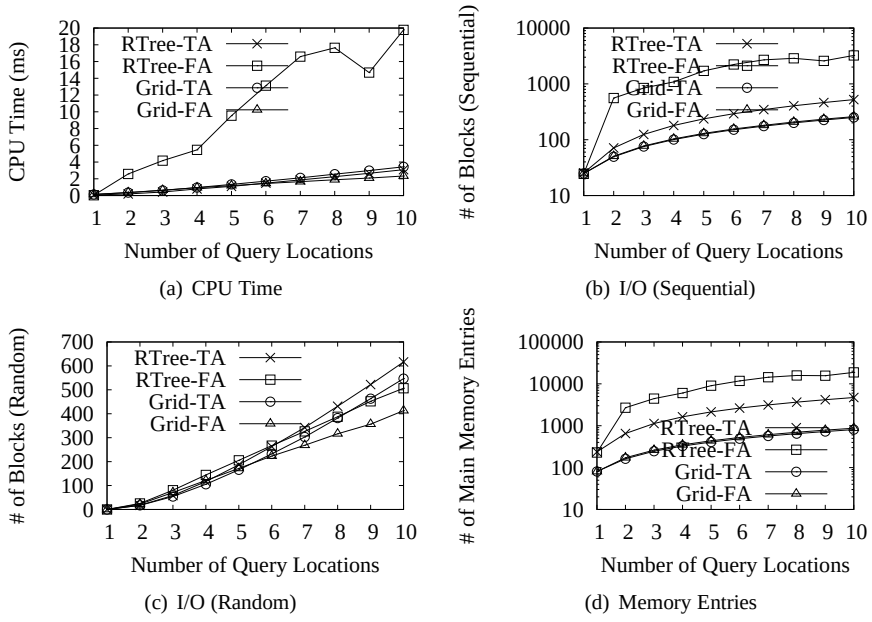
To study the scalability of our algorithms, we generate synthetic datasets D with $|D| = 1k, 2k, \cdots, 10k$, and process queries with m=3 and k=5. The grid indices are built by constructing a quadtree of height h=6, and accordingly, the grid is of size 64×64.



(a) CPU Time (b) I/O (Sequential)

(c) I/O (Random) (d) Memory Entries

Fig. 7. Effect of k using the Trucks dataset



(a) CPU Time (b) I/O (Sequential)

(c) I/O (Random) (d) Memory Entries

Fig. 8. Effect of m using the SchoolBuses dataset

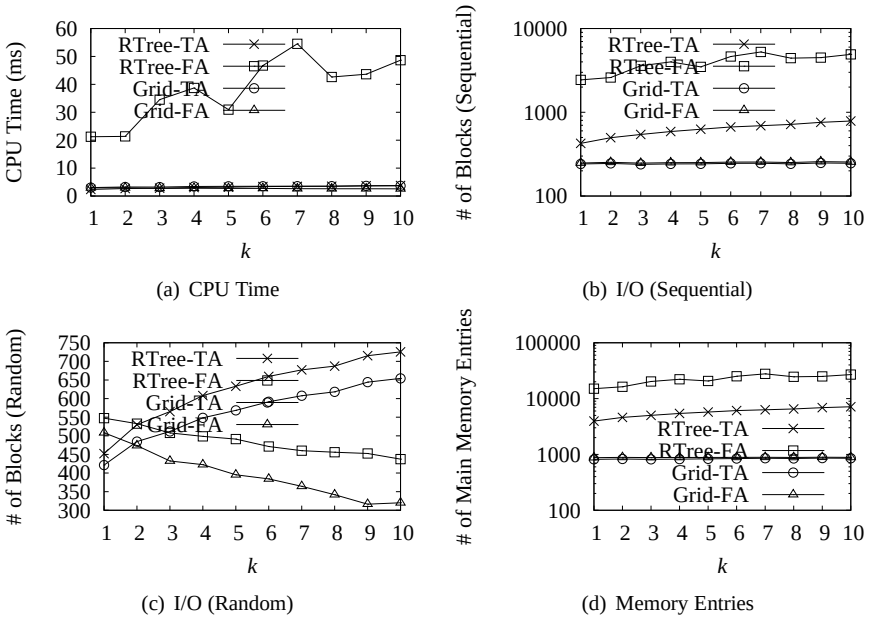


Fig. 9. Effect of k using the SchoolBuses dataset

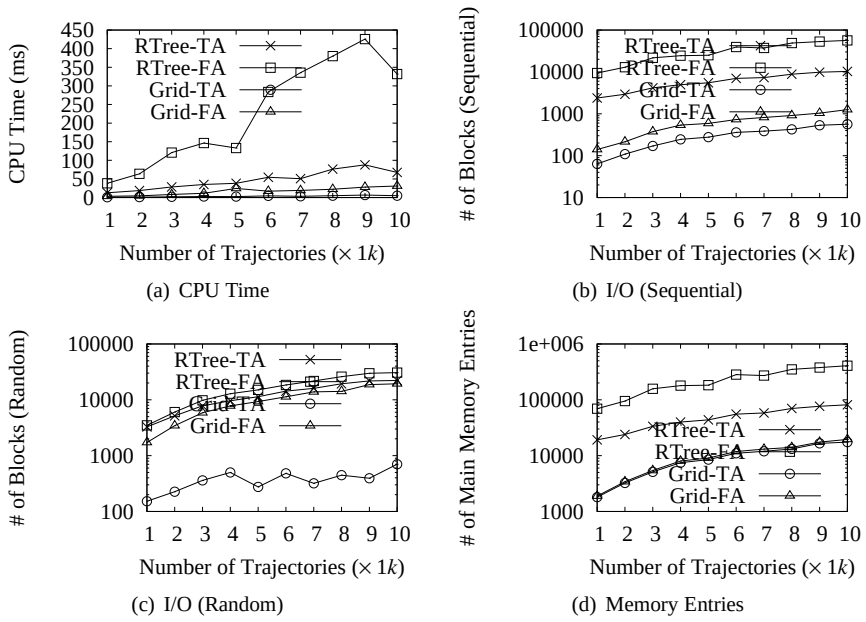


Fig. 10. Scalability results

Figure 10 shows the scalability of our algorithms when the number of trajectories increases. We can see from Figure 10(c) that for a large database, the number of blocks read by the algorithms by random access is quite different when compared with the results for the smaller datasets, as *Grid-TA* reads significantly fewer blocks than the other three algorithms. In other respects, the performance trend is quite consistent with the results previously reported for changing query parameters in Section 7.3.

Overall, the grid-based algorithms are more efficient than the R-tree based ones, and *Grid-TA* is now over an order of magnitude faster than *Grid-FA*.

7.5. Quality of Trajectory Answers

Till now, we have only studied the performance of our algorithms. In this subsection, we compare the quality of the trajectories found by our sum-of-weighted-distance measure with that of the traditional sum-of-Euclidean-distance measure. We randomly generate $k = 5$ query points over the Trucks dataset, and compute the top-5 trajectories using both measures. A representative query answer is given in Table 1, where each cell in the table corresponds to a (trajectory ID, sum-of-weighted-distance) pair. After examining these trajectories, we find that Trajectories 28, 198 and 221, returned by the sum-of-Euclidean-distance measure, do not even stop in some of the query points and are thus of low quality. On the other hand, the trajectories found by our sum-of-weighted-distance measure match and are close to all the query points.

7.6. Summary of Experimental Results

To sum up, we have the following observations: (1) the grid-based algorithms are significantly more efficient than the R-tree based algorithms; (2) the TA-based algorithms are more efficient than the FA-based algorithms; and (3) *Grid-TA* is much faster than the other three algorithms on large datasets.

8. Conclusions

We proposed the new problem of k *Important Connected Trajectories* (k-ICT) query processing over trajectories with location importance. We designed effective methods to infer the importance of trajectory locations from the temporal information, and developed four algorithms to answer the queries: two based on the R-tree index, and the other two based on an efficient grid index. The R-tree index based algorithms are adaptations of the algorithms in [1] to querying trajectories with location importance. However, the R-tree index only captures the spatial aspects of the trajectory points, and location weights are only considered during R-tree querying. On the other hand, our grid index includes the location weights as first-class citizen, and is thus more suitable for querying trajectories with location importance.

We showed by experiments on both real and synthetic datasets that our algorithms are efficient for answering k-ICT queries. The grid index based algorithms are especially efficient in terms of both time and space: they incur one to two orders of magnitude less sequential IO cost and computational overhead compared with R-tree index based algorithms, due to the more effective pruning power of the grid index. As for trajectory traversal, TA is more effective than FA since the aggressive strategy of TA

tightens the pruning threshold much faster. Overall, the combination of TA with grid index offers the best performance.

References

[1] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng and X. Xie. “Searching Trajectories by Locations - An Efficiency Study”. In *SIGMOD*, 2010.
[2] B.-K. Yi, H. Jagadish and C. Faloutsos. “Efficient Retrieval of Similar Time Sequences under Time Warping”. In *ICDE*, 1998.
[3] M. Vlachos, G. Kollios and D. Gunopulos. “Discovering Similar Multidimensional Trajectories”. In *ICDE*, 2002.
[4] L. Chen and R. Ng. “On the Marriage of Lp-norms and Edit Distance”. In *VLDB*, 2004.
[5] L. Chen, M. T. Özsu and V. Oria. “Robust and Fast Similarity Search for Moving Object Trajectroies”. In *SIGMOD*, 2005.
[6] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng and P. Kalnis. “User Oriented Trajectory Search for Trip Recommendation”. In *EDBT*, 2012.
[7] K. Zheng, S. Shang, N. J. Yuan and Y. Yang. “Towards Efficient Search for Activity Trajectories”. In *ICDE*, 2013.
[8] X. Cao, G. Cong and C. S. Jensen. “Mining Significant Semantic Locations from GPS Data”. In *VLDB*, 2010.
[9] Y. Yang, Z. Gong, L. H. U. “Identifying Points of Interest by Self-Tuning Clustering”. In *SIGIR*, 2011.
[10] S. Spaccapietra, C. Parent, M. L. Damiani, J. A. de Macedo, F. Porto and C. Vangenot. “A Conceptual View on Trajectories”. *Data & Knowledge Engineering*, vol. 65, no. 1, pp. 126–146, Elsevier, 2008.
[11] A. Tietbohl, V. Bogorny, B. Kuijpers and L. O. Alvares. “A Clustering-Based Approach for Discovering Interesting Places in Trajectories”. In *SAC* 2008.
[12] J. A. M. R. Rocha, G. Oliveira and V. Bogorny. “DB-SMoT: A Direction-Based Spatio-Temporal Clustering Method”. In Intelligent Systems, 2010.
[13] R. Fagin, A. Lotem and M. Naor. “Optimal aggregation algorithms for middleware”. In *PODS*, 2001.
[14] I. Lazaridis and S. Mehrotra. “Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure”. In *SIGMOD*, 2001.
[15] A. OKabe, B. Boots, K. Sugihara and S. Chiu. “Spatial Tessellations, Concepts and Applications of Voronoi Diagrams”. *Wiley*, 2000.
[16] D. Wu, M. L. Yiu, C. S. Jensen and G. Cong. “Efficient Continuously Moving Top-k Spatial Keyword Query Processing”. In *ICDE*, 2011.
[17] L. A. Tang, Y. Zheng, X. Xie, J. Yuan, X. Yu and J. Han. “Retrieving k-Nearest Neighboring Trajectories by a Set of Point Locations”. In *SSTD*, 2011