Retrieving k-Nearest Neighboring Trajectories by a Set of Point Locations

Lu-An Tang[1,2], Yu Zheng[2], Xing Xie[2], Jing Yuan[3], Xiao Yu[1], Jiawei Han[1]
[1] Computer Science Department, UIUC; [2] Microsoft Research Asia
[3] University of Science and Technology of China
{tang18,xiaoyu1,hanj}@illinois.edu;
{yuzheng, xingx}@microsoft.com; yuanjing@mail.ustc.edu.cn

Abstract. The advance of object tracking technologies leads to huge volumes of spatio-temporal data accumulated in the form of location trajectories. Such data bring us new opportunities and challenges in efficient trajectory retrieval. In this paper, we study a new type of query that finds the *k Nearest Neighboring Trajectories* (*k*-NNT) with the minimum aggregated distance to a set of query points. Such queries, though have a broad range of applications like trip planning and moving object study, cannot be handled by traditional *k*-NN query processing techniques that only find the neighboring points of an object. To facilitate scalable, flexible and effective query execution, we propose a *k*-NN trajectory retrieval algorithm using a candidate-generation-and-verification strategy. The algorithm utilizes a data structure called *global heap* to retrieve candidate trajectories near each individual query point. Then, at the verification step, it refines these trajectory candidates by a lower-bound computed based on the global heap. The global heap guarantees the candidate's *completeness* (*i.e.*, all the *k*-NNTs are included), and reduces the computational overhead of candidate verification. In addition, we propose a *qualifier expectation* measure that ranks partial-matching candidate trajectories to accelerate query processing in the cases of non-uniform trajectory distributions or outlier query locations. Extensive experiments on both real and synthetic trajectory datasets demonstrate the feasibility and effectiveness of proposed methods.

1 Introduction

The technical advances in location-acquisition devices have generated a huge volume of location trajectories recording the movement of people, vehicle, animal and natural phenomena in a variety of applications, such as social networks, transportation systems and scientific studies: In Foursquare [1], the check-in sequence of a user in restaurants and shopping malls can be regarded as a location trajectory. In many GPS-trajectory-sharing websites like Geolife [17, 18, 19], people upload their travel routes for the purpose of memorizing a journey and sharing life experiences with friends. Many taxis in big cities have been embedded with GPS sensors to report their locations. Such reports formulate a large amount of trajectories being used for resource allocation, security management and traffic analysis [8]. Biologists solicit the moving trajectories of animals like migratory birds for their research [2]. Similarly, climatologists are busy collecting the trajectories of natural phenomena such as hurricane and ocean currents [3].

In the above-mentioned applications, people usually expect to retrieve the trajectories passing a set of given point locations. For example, the social network

users want to retrieve their friend's trails of visiting some scenic spots as references for trip planning. The fleet operators expect to analyze the business of their taxis traveling around several hot spots by the GPS traces. The biologists are interested in study the migration trails of birds passing some mountains, lakes and forests. In general, these applications need to efficiently query and access trajectories from large datasets residing on disks by geospatial locations. Note that, the system needs to select the top k trajectories with the minimum aggregated distance to the given locations instead of the trajectory exactly passing those locations, since in most case exact match may lead to no result or not the best results returned. This study aims to provide an efficient method to expedite a novel geospatial query, the *k-Nearest Neighboring Trajectory Query* (*k*-NNT query), in a trajectory database.

Unfortunately, the *k*-NNT query is not efficiently supported in existing systems. Most traditional *k*-NN query processing methods are designed to find point objects [11, 6, 5]. On the other hand, the traditional trajectory search techniques focus on retrieving the results with similar shapes to a sample trajectory [9, 13]. The new problem, searching top-*k* trajectories given a set of geospatial locations, poses the following challenges:

- *Huge size*: Many databases contain large volumes of trajectories. For example, the T-drive system [8] collects the trajectories from over 33,000 taxis for 3 months. The total length of the trajectories is more than 400 million kilometers and the total number of GPS points reaches 790 million. The huge I/O overhead is the major cost in query processing.
- *Distance computation*: The distance computation in *k*-NNT query is more complex than traditional spatial queries. To compute the aggregated distance from a trajectory to a set of query points, the system has to check all the member points of the trajectory, find out the closest one to each individual query point (*i.e.*, shortest matching pairs) and sum up all the matching pairs as the measure. The techniques of point *k*-NN queries, such as *best-first search* [6] and *aggregate k-NN* [5], cannot handle this problem.
- *Non-uniform distribution*: In many real applications, the distributions of trajectories are highly skewed, *e.g.*, taxi trajectories are much denser in downtown than suburban areas. In addition, query points are given by users in an ad-hoc manner and some of them may be far from all the trajectories.

In this study, we propose a robust, systematic and efficient approach to process *k*-NNT queries in the trajectory database. The system employs a data structure called *global heap* to generate candidate trajectories by only accessing a small part of the data and verifies the candidates with the lower-bound derived from global heap. To handle the skewed trajectory data and outlier query locations, a *qualifier expectation* measure is designed to rank the candidates and accelerate query processing.

The rest of the paper is organized as follows. Section 2 provides the background and problem definition, Section 3 describes detailed query processing framework, and Section 4 introduces the qualifier expectation-based algorithm. Section 5 evaluates the approaches by extensive experiments on both real and synthetic datasets. Section 6 discusses related studies. Finally, Section 7 concludes the paper.

2 Problem Formulation

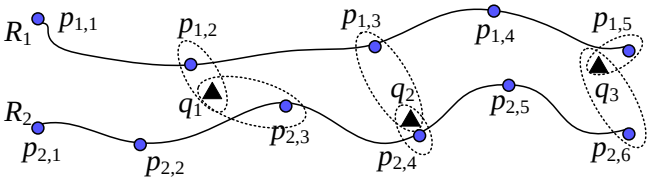
The trajectory data are collected in the form of point sequences. Trajectory R_i can be represented as $R_i = \{p_{i,1}, p_{i,2}, \ldots p_{i,n}\}$, where $p_{i,j}$ is the j-th member point of R_i. The input of k-NNT query Q, according to applications, is specified by a set of point locations, $Q = \{q_1, q_2, \ldots q_m\}$. In the following we first define the distance measures between trajectories and query points.

Definition 1. Let trajectory $R_i = \{p_{i,1}, p_{i,2}, \ldots p_{i,n}\}$ and q be a query point. The *matching pair* of a member point $p_{i,j}$ and q is denoted as $<p_{i,j}, q>$. If $\forall p_{i,k} \neq p_{i,j}$, $dist(p_{i,j}, q) \leq dist(p_{i,k}, q)$, $<p_{i,j}, q>$ is the *shortest matching pair* of R_i and q.

Definition 2. Let trajectory $R_i = \{p_{i,1}, p_{i,2}, \ldots, p_{i,n}\}$ and query $Q = \{q_1, q_2, \ldots, q_m\}$. The distance between R_i and a query point q is the distance of the shortest matching pair $<p_{i,j}, q>$, the *aggregated distance* between R_i and Q is the sum of distances of the shortest matching pairs from R_i to all query points.

$$dist(R_i, Q) = \sum_{q \in Q} dist(R_i, q) = \sum_{q \in Q} dist(p_{i,j}, q)$$

Example 1. Figure 1 shows a matching example of two trajectories and three query points. The query points q_1, q_2 and q_3 are matched with the closest points in R_1 and R_2. The nearest neighboring trajectory is selected by the aggregated distance of all the query points. Even R_2 is more distant to q_1 and q_3, its aggregated distance is still smaller than R_1. So R_2 should be returned as the query result.



dist(R_1, q_1)= *dist*($p_{1,2}$, q_1)= 20 m
dist(R_1, q_1)= *dist*($p_{1,3}$, q_2)= 50 m
dist(R_1, q_1)= *dist*($p_{1,5}$, q_3)= 15 m
dist(R_1, Q)= 85 m

dist(R_2, q_1)= *dist*($p_{2,3}$, q_1)= 30 m
dist(R_2, q_1)= *dist*($p_{2,4}$, q_2)= 5 m
dist(R_2, q_1)= *dist*($p_{2,6}$, q_3)= 40 m
dist(R_2, Q)= 75 m

Figure 1. Aggregated Distance Example

With the distance measures, now we formally describe the task of k-NNT query.

Task Definition: Given the trajectory dataset, D, and a set of query points, Q, the k-*NN*T query retrieves k trajectories K from D, $K = \{R_1, R_2, \ldots, R_k\}$ that for $\forall R_i \in K, \forall R_j \in D - K$, $dist(R_i, Q) \leq dist(R_j, Q)$.

A direct way to process k-NNT query is to scan the whole trajectory dataset, compute all the shortest matching pairs for each trajectory, and then compare their aggregated distances. The I/O overhead is very high since all the trajectories are retrieved.

A candidate-generation-and-verification framework was first proposed by Fagin *et al.* for processing top k aggregation queries for distributed systems and middleware [12]. The Fagin's algorithm generates a candidate set by searching the objects listed at the top in each individual site, and then carries out the detailed verification only on such candidates. The general idea can be employed to solve this problem, since the k-NNTs usually have member points close to some query locations. The system should search for candidate trajectories around each query point, and then verify them for final results. However, since Fagin's algorithm is not designed for spatial query processing, the candidate search operations on individual sites are carried out in parallel (*i.e.*, the system retrieves the top member from each site, then gets the second member of each of the sites, and so on). In the scenarios of k-NNT query, the situations around each query points are different. It is inefficient to search in such parallel manner. Thus the key problem becomes how to coordinate the candidate searching processes. Meanwhile, the algorithm must guarantee that all the k-NNTs are included in the candidate set (*i.e.*, *completeness*). Section 3 will discuss those issues in detail. Table 1 lists the notations used in the following sections.

Table 1. A List of Notations

Notation	Explanation	Notation	Explanation
$\boldsymbol{D}$	the trajectory dataset	$\boldsymbol{K}$	the k-NNT result set
$\boldsymbol{R}$	a trajectory	$\boldsymbol{p_{i,j}, p_i}$	member points of traj.
$\boldsymbol{Q}$	the k-NNT query	$\boldsymbol{q, q_j}$	k-NNT query points
$\boldsymbol{H}$	the individual heap list	$\boldsymbol{h_i}$	an individual heap
$\boldsymbol{G}$	the global heap	$\boldsymbol{N}$	an R-tree node
$\boldsymbol{C}$	the candidate set	$\boldsymbol{\delta}$	the pruning threshold
$\boldsymbol{\mu}$	the full-matching ratio	$\boldsymbol{k}$	a constant given by user

3 Query Processing

3.1 Candidate Generation

The first task of candidate generation is to retrieve the neighboring points around query locations. In this study, we utilize the *best-first* strategy to search for k-NN points [6]. The best-first strategy traverses R-tree index from root node and always visits the node with the least distance to the query point, until it reaches the leaf node and returns it as the result. Based on the best-first search strategy, we construct a data structure of *individual heap* to search k-NN points.

Definition 3. Let q_i be a query point. The *individual heap* h_i is a minimum heap whose elements are the matching pairs of trajectory member point and q_i. The matching pairs are sorted by their distances to q_i.

The individual heap takes q_i as the input and visits R-tree nodes with the shortest distance. If the R-tree node is an inner node, the heap retrieves all its children node and keeps to traverse the tree; if the R-tree node is leaf node (*i.e.*, a trajectory's member point p_j), the heap composes a matching pair of p_j with q_i.

There are two advantages of individual heap: (1) It employs the best-first strategy to traverse R-tree and achieves optimal I/O efficiency [6]. (2) The individual heap does not need to know k in advance, it can pop out the matching pairs incrementally.

For a query $Q = \{q_1, q_2, \ldots, q_m\}$, the system constructs a heap list $H = \{h_1, h_2, \ldots, h_m\}$ and finds out the shortest matching pairs from the m heaps. Since there are multiple heaps, the key problem is to coordinate the searching processes of individual heaps. To this end, we introduce the *global heap*.

Definition 4. Let k-NNT query $Q = \{q_1, q_2, \ldots, q_m\}$ and individual heap list $H = \{h_1, h_2, \ldots, h_m\}$. The *global heap* G consists of m matching pairs, $G = \{<p_1, q_1>, <p_2, q_2>, \ldots, <p_m, q_m>\}$, where $<p_i, q_i>$ is popped from the individual heap h_i. G is a minimum heap that sorts the matching pairs by their distances.

The global heap has two operations, *pop* and *retrieve*. The pop operation simply outputs the shortest matching pair of G. The retrieve operation is carried out immediately after popping a pair $<p_i, q_i>$. The global heap retrieves another matching pair $<p_i', q_i>$ from the corresponding individual heap h_i. In this way, there are always m matching pairs in G.

The popped matching pairs are kept in a candidate set. In the beginning, the candidate trajectories only have a few matching pairs, we call them *partial-matching candidates*. When the global heap pops out more matching pairs, several trajectories will eventually complete all the matching pairs for query Q, they are called *full-matching candidates.* In Figure 2, trajectory R_1 is a full-matching candidate with all shortest matching pairs, and R_2 and R_4 are partial-matching candidates since they miss the pairs of several query points. One may notice that, not all the matching pairs popped out by global heap G are added to the candidate set. For example, the current top element of G is $<p_{1,4}, q_1>$, and there is already a shortest matching pair $<p_{1,2}, q_1>$ in candidate R_1. Since the individual heap h_1 reports the k-NN points in incremental manner, the oldest pair $< p_{1,2}, q_1>$ is guaranteed to be the shortest one from R_1 to q_1. The new pair $<p_{1,4}, q_1>$ is then a *useless pair*. It should be thrown away.

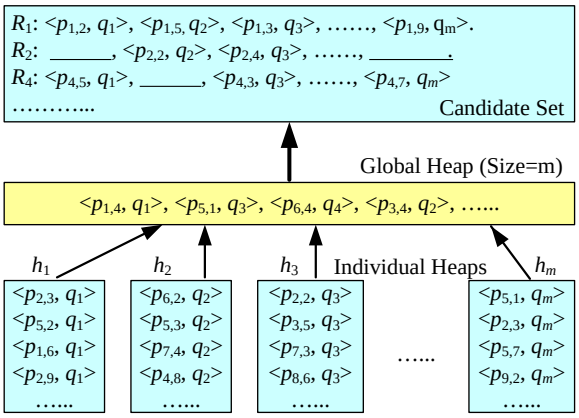


Figure 2. Data Structures for Candidate Generation

The last issue of candidate generation is the stop criterion. The algorithm should not stop unless the candidate set has already contained all the *k*-NNTs.

Property 1. If a candidate set has at least *k* full-matching candidates whose shortest matching pairs are all popped from the global heap, then the candidate set is complete.

Proof: A candidate set *C* is complete if it contains all the *k*-NNTs. That is, for any trajectory $R_i \notin C$, we need to prove that R_i cannot be a *k*-NNT.

For $\forall\, q \in Q$, we denote the shortest matching pair from R_i to q as $<p_i, q>$. Since $R_i \notin C$, $<p_i, q>$ has not been popped out yet. The global heap *G* pops out matching pairs in the order of increasing distance, the distance of $<p_i, q>$ is large than or equal to current matching pair $<p_G, q>$ in *G*, that is: $dist(p_G, q) \leq dist(p_i, q)$.

Let $R_j \in C$ be a full-matching trajectory candidate, whose matching pair for q is $<p_j, q>$. Since $<p_j, q>$ is already popped from *G*, its distance is less than $<p_G, q>$: $dist(p_j, q) < dist(p_G, q)$. Then $dist(p_j, q) < dist(p_i, q)$. Hence we have:

$$dist(R_j, Q) = \sum_{q \in Q} dist(p_j, q) < \sum_{q \in Q} dist(p_i, q) = dist(R_i, Q)$$

There are at least *k* full-matching trajectories in *C*, their aggregated distances are all smaller than R_i, so R_i is not possible to be a *k*-NNT, candidate set *C* is complete. ■

Based on Property 1, we develop the algorithm of *k*-NNT candidate generation. The algorithm first constructs the individual heaps and initializes the global heap and candidate set (Lines 1--3). Each individual heap pops a shortest matching pair to the global heap (Lines 4--5). In this way the global heap has *m* matching pairs. Then the candidate generation process begins. Once the global heap is full with *m* pairs, it pops out the shortest one. The system checks whether the candidate set already contains an old pair with the same trajectory and query point. If there is no such pair, the new popped pair is a shortest matching pair, it is then added to the candidate set (Lines 7--9). After that, the global heap retrieves another pair from the corresponding individual heap (Line 10). This process stops when there are *k* full-matching trajectories in the candidate set (Line 11).

Algorithm 1. *k*-NNT Candidate Generation

Input: Trajectory dataset *D*, Query *Q*, *k*
Output: *k*-NNT Candidate Set *C*

1. **for** each $q_i \in Q$
2. construct the individual heap h_i on *D*;
3. initialize the global heap *G* and candidate set *C*;
4. **for** each individual heap h_i
5. pop a matching pair and push it to *G*;
6. **repeat**
7. pop the shortest pair $<p_j, q_j>$ from *G*;
8. **if** $<p_j, q_j>$ is a shortest matching pair, **then**
9. add $<p_j, q_j>$ to *C*;
10. pop a matching pair from h_j and push it to *G*;
11. **until** *C* contains *k* full-matching candidates
12. **return** *C*;

Example 2. Figure 3 shows an example of the candidate generation algorithm. Suppose k is set to 1. The algorithm first constructs the global heap with matching pairs $<p_{1,4}, q_2>$, $<p_{1,6}, q_3>$ and $<p_{1,2}, q_1>$. In the first iteration the pair $<p_{1,4}, q_2>$ is popped to the candidate list, candidate R_1 is generated. Meanwhile the global heap retrieves the another pair $<p_{5,5}, q_2>$ from q_2's individual heap. In the next three iterations, the global heap pops matching pairs $<p_{1,6}, q_3>$, $<p_{5,5}, q_2>$ and $<p_{4,5}, q_3>$ and generates two partial-matching candidates R_4 and R_5. At the 5th iteration, $<p_{1,2}, q_1>$ is popped out and a full-matching candidate R_1 is generated. The algorithm then stops and outputs the candidate set for further verification.

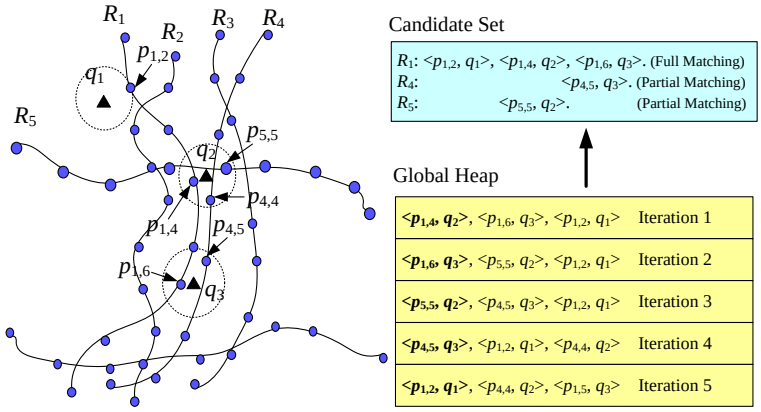


Figure 3. Running Example of Candidate Generation

3.2 Candidate Verification

After generating the candidates, the system needs to verify them to select k-NNTs. For the partial-matching candidates, the algorithm has to make up their missing pairs. The computational and I/O costs are high for this task. Suppose the candidate set's size is l (with k full-matching candidates), each trajectory contains average n member points and the number of query points is m. In the worst case, each partial-matching candidate only has one matching pair, the system has to carry out $(l-k)*n*(m-1)$ times of distance calculation to make up the missing pairs. And it needs to access the $l-k$ partial-matching trajectories. Fortunately the global heap provides a shortcut to enhance the efficiency of candidate verification.

Property 2. Let Q be the query, G be the global heap, R be a partial-matching candidate trajectory and $Q_R \subset Q$ be the subset of query points that are contained in R's matching pairs. Then $LB(R, Q)$, as defined in the following equation, is a lower-bound of R's aggregated distance.

$$LB(R,Q) = \sum_{q_i \in Q_R} dist(p_R, q_i) + \sum_{q_j \in Q - Q_R} dist(p_G, q_j)$$

where $<p_R, q_i>$ is a matching pair in R and $<p_G, q_j>$ is a matching pair in G.

Proof: For $\forall\, q_j \in Q - Q_R$, we denote the shortest matching pair from R to q_j as $<p_R, q_j>$. $<p_R, q_j>$ is a missing pair that has not been popped out from the global

heap G yet. Since G is a minimum heap and pops out matching pairs with increasing distance, $dist(p_R, q_j) \geq dist(p_G, q_j)$. So we have:

$$dist(R,Q) = \sum_{q_i \in Q} dist(p_R, q_i) = \sum_{q_i \in Q_R} dist(p_R, q_i) + \sum_{q_j \in Q - Q_R} dist(p_R, q_j)$$
$$\geq \sum_{q_i \in Q_R} dist(p_R, q_i) + \sum_{q_j \in Q - Q_R} dist(p_G, q_j) = LB(R,Q)$$

Hence $LB(R,Q)$ is a lower-bound of R's aggregated distance. ■

Note that, for $\forall\, q_i \in Q_R$ and $\forall\, q_j \in Q - Q_R$, the distances of $<p_R, q_i>$ and $<p_G, q_j>$ have been already computed. Thus $LB(R, Q)$ is calculated without any I/O overhead.

Algorithm 2 outlines the processing of candidate verification based on Property 2. The algorithm starts by adding all the full-matching trajectories to the result set and obtaining the k-th trajectory's aggregated distance as the pruning threshold (Lines 1--4). Then it computes the lower-bound for each partial-matching candidate. If the lower-bound is larger than the threshold, the trajectory is pruned without further computation (Lines 6--7). Otherwise, the algorithm has to access the trajectory's member points and compute its aggregated distance (Lines 8--9). If the system finds a trajectory with a shorter distance than threshold, it adds the trajectory to the result set and updates the threshold (Lines 9--12). After processing all the partial-candidates, the algorithm outputs the top k trajectories in the result set as k-NNTs (Line 13). This pruning strategy is especially powerful if there are many partial-matching candidates with only one or two matching pairs. The more matching pairs a candidate lacks, the lager lower-bound it will have, and the higher probability it will be pruned.

Algorithm 2. k-NNT Candidate Verification

Input: k-NNT candidate set C, global heap G, query Q.
Output: k-NNTs.

```
1.  initialize result set K;
2.  add all full-matching candidate of C to K;
3.  sort K in the order of increasing distance;
4.  threshold δ ← k-th trajectory's aggregated distance in K;
5.  for each partial-matching candidate R in C
6.      compute LB(R,Q);
7.      if LB(R,Q) ≥ δ then continue;
8.      else
9.          compute dist(R,Q);
10.         if dist(R,Q) < δ then
11.             add R to K;
12.             sort K and update δ;
13. return the top k trajectories in K;
```

Example 3. Figure 4 shows the process to verify the candidates from Example 2. There are one full-matching candidate, R_1, and two partial-matching candidates R_4 and R_5. The algorithm first calculates $LB(R_4, Q)$ and $LB(R_5, Q)$. The calculations are carried out based on the results in the global heap and candidate set. Since $LB(R_4, Q)$

is larger than the threshold, R_4 is pruned directly. The system only accesses the member points of R_5 for further computation. Finally, R_1 is returned as the result.

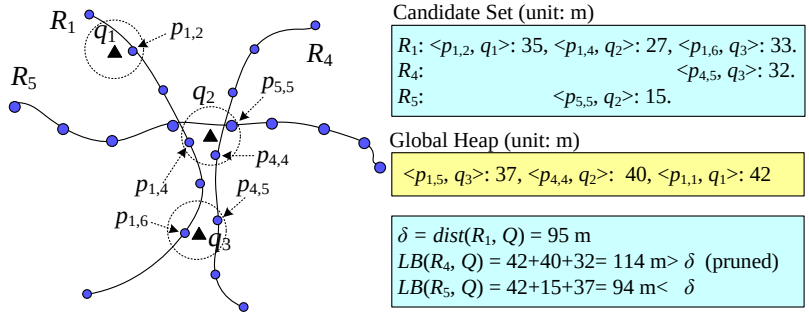


Figure 4. Running Example of Candidate Verification

4 Qualifier Expectation

In *k*-NNT query processing, there are two steps that involve I/O overheads: (1) In candidate generation, the individual heaps traverse the R-tree to pop out matching pairs; (2) In candidate verification, if the lower-bound of a partial-matching candidate is less than the threshold, the system needs to access the trajectory's member points for distance computation. The key to reduce I/O costs is to generate *tight* candidate set, *i.e.*, the number of the candidates should be as small as possible. A tight candidate set costs less time to generate and is easier to be verified since the number of partial-matching trajectories is also smaller.

The major function of global heap is to raise the candidate set's tightness in the premise of guaranteeing completeness. The global heap controls searching processes of different individual heaps, restricts the search regions as equal-radius circles, as illustrated in Figure 5. When the trajectories and query points are all uniformly distributed, the global heap can pop out a similar number of matching pairs to different locations. In this way, the *k* full-matching candidates are soon found and the candidate generation algorithm stops early. However, many real trajectory datasets are skewed: the taxi trajectories are dense in the downtown areas, the animal movements are concentrated around water/food sources. In addition, the query points are ad-hoc. It is possible that a user may provide an *outlier location* that is distant from all the trajectories. In such cases, the matching pairs of outlier locations are much longer, they could be stuck in the global heap and significantly delay query processing.

Example 4. Figure 5 shows an example of outlier location. The query point q_3 is an outlier since it is far from all the trajectories. The distance of its shortest matching pair $<p_{2,7}, q_3>$ is much larger than the pairs from other individual heaps. This pair cannot be popped out from the global heap and no full-matching candidate is found. The global heap has to increase the search radius and keep on popping useless pairs. Finally the algorithm ends with a large candidate set. And the system has to cost even more time in the verification step. The query efficiency is affected seriously due to a single outlier location.

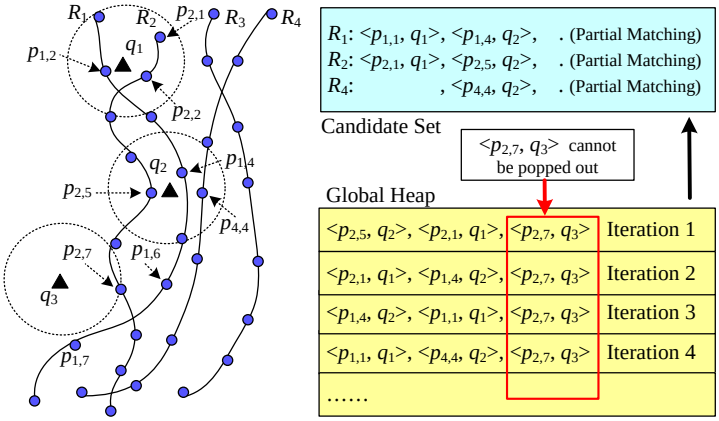


Figure 5. Example of Outlier Point

In the cases of outlier locations, the cost is high to wait global heap to pop k full-matching candidates. *Then can we compute them directly?* The system can retrieve some partial-matching trajectories and make up their missing pairs. The key point is to guarantee the completeness of generated candidates.

Property 3. If a candidate set has at least k full-matching candidates, and their aggregated distances are smaller than the accumulated distance of all the matching pairs in the global heap, then the candidate set is complete.

Proof: A candidate set C is complete if it contains all the k-NNTs. In another word, for $\forall R_i \notin C$, we need to prove that R_i cannot be a k-NNT.

For $\forall\, q \in Q$, we denote the shortest matching pair from R_i to q as <p_i, q> and the current matching pair in G as <p_G, q>. According to the proof of Property 1, we have $dist(p_G, q) \leq dist(p_i, q)$, then

$$Sum(G) = \sum_{q \in Q} dist(p_G, q) \leq \sum_{q \in Q} dist(p_i, q) = dist(R_i, Q)$$

Let R_j denote the full-matching candidate that, $R_j \in C$ and $dist(R_j, Q) < Sum(G)$. Then $dist(R_j, Q) < dist(R_i, Q)$. There are at least k such full-matching candidates in C, then R_i is not possible to be a k-NNT, candidate set C is complete. ■

Based on Property 3, if the candidate set contains k full-matching candidates with smaller distances than the distance sum of global heap's matching pairs, the candidate generation can end safely. We call such full-matching candidates as *qualifiers*. Since the number of partial-matching candidates is usually much larger than k, the system needs to select out the ones that are most likely to become qualifiers to make up. A measure is thus required to represent the expectation of a partial-candidate to be a qualifier. To reveal the essential factors of such measure, let us investigate the following example.

Example 5. Figure 6 lists out the matching pairs in a candidate set. There are three partial-matching candidates R_1, R_2 and R_4 with current aggregated distances as 70m,

80m and 70m. q_3 is an outlier location, the distance of its matching pair is much larger than others. Now if the system has to select a candidate and makes it up, which one is most likely to be a qualifier?

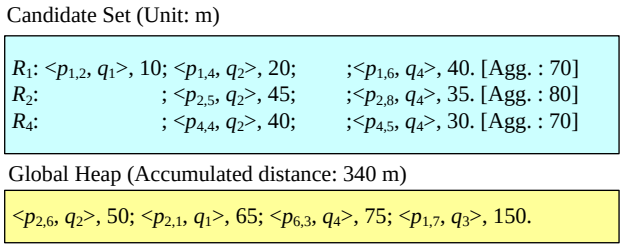


Figure 6. Partial-Matching Candidates

Intuitively, we prefer R_4 to R_2, they have the same number of matching pairs but R_4's aggregated distance is smaller. Furthermore, R_1 is better than R_4, their aggregated distances are the same but R_4 has one more missing matching pair than R_1. To become a qualifier, R_4 needs to make up two matching pairs but R_1 only needs one.

From the above example, we can find out that a candidate's *qualifier expectation* is determined by two factors: the number of missing pairs and the advantage of existing matching pairs over the corresponding ones in global heap.

Let R be a partial-matching candidate trajectory and $Q_R \subset Q$ be the subset of query points that are contained in R's matching pairs. The *qualifier expectation* of R is given in the following equation:

$$Expect(R) = \frac{\sum_{q \in Q_R}(dist(p_G, q) - dist(p_R, q))}{|Q - Q_R|}$$

The qualifier expectation actually denotes an upper-bound of the average distance that R's missing pairs could be larger than the corresponding ones in global heap. The larger this value is, the more likely that R will be a qualifier. Note that, the computation of qualifier expectation can be done with the current matching pairs in the candidate set and global heap, and no more I/O access is needed.

Algorithm 3. Qualifier Expectation-based Generation

Input: Trajectory dataset *D*, Query *Q*, Full-matching ratio μ
Output: *k*-NNT Candidate Set *C*

```
1.    (Lines 1-10 are the same as Algorithm 1) ……
11.     while (|full-matching candidate| / |C|< μ)
12.       compute partial-matching candidate's expectation;
13.       retrieve the candidate R with highest expectation;
14.       make up the matching pairs for R;
15.   until C contains k qualifiers;
16.   return C;
```

With the help of qualifier expectation, we can improve the candidate generation algorithm as shown in Algorithm 3. The first few candidate generating steps are the same as Algorithm 1 (Lines 1--9). The difference is at Line 10, Algorithm 3 controls

the size of full-matching candidates by a ratio parameter μ. Each time the global heap pops out a matching pair to candidate set, the proportion of full-matching candidate is compared with μ. If a full-matching candidate needs to be generated, the algorithm first calculates the qualifier expectations of all partial-matching candidates and picks the one with the highest expectation for making up. (Lines 11--13). The algorithm stops if there are k qualifiers (Lines 14--15).

Example 6. Figure 7 illustrates the qualifier expectation-based method. Suppose k is set as 1 and the full-matching ratio μ is 0.33. At the 5th iteration, the candidate set size is 3 and a full-matching candidate should be generated. The algorithm calculates the qualifier expectations of the three partial-matching candidates. R_1 is the one with the highest expectation. The algorithm then retrieves R_1's member points and makes up the missing pairs. Since dist(R_1, Q) is less than the global heap's accumulated distance, R_1 is a qualifier. The candidate generation ends and R_1, R_2 and R_4 are returned as candidates.

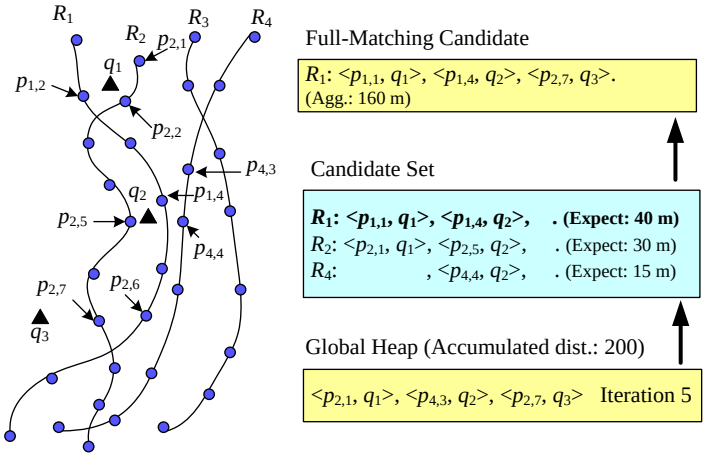


Figure 7. Qualifier Expectation-based Method

5 Performance Evaluation

5.1 Experiment Settings

Datasets: We conduct extensive experiments to evaluate the proposed methods, using both real-world and synthetic trajectory datasets. The real datasets D_3 is retrieved from the Microsoft GeoLife and T-Drive projects [8, 17, 18, 7]. The trajectories are generated from GPS devices with sampling rate from 5 seconds to 10 minutes. Meanwhile, to test the algorithm's scalability, we also generate two synthetic datasets, being comprised of both uniform and skewed trajectory distributions, with a size more than 2 GB.

Environments: The experiments are conducted on a PC with Intel 7500 Dual CPU 2.20G Hz and 3.00 GB RAM. The operating system is Windows 7 Enterprise. All the algorithms are implemented in Java on Eclipse 3.3.1 platform with JDK 1.5.0. The parameter settings are listed in Table 2.

Table 2. Experimental Settings

Dataset	Type	Traj. #	Total Points	File Size
Syn 1 (D_1)	syn., uniform	40,000	$4.0*10^7$	2.0 GB
Syn 2 (D_2)	syn., skewed	40,000	$4.0*10^7$	2.0 GB
Real (D_3)	taxi	12,643	$1.1*10^6$	54 M
The value of k: 4 – 20, default 20				
The query size $\|Q\|$ (number of query points): 2 –10, default 10				
The full matching ratio μ: 20% – 100%, default 40%				

Competitors: The proposed *Global Heap-based algorithm* (GH) and *Qualifier Expectation-based method* (QE) are compared with *Fagin's Algorithm* (FA) [12] and *Threshold Algorithm* (TA) [14].

5.2 Evaluations on Algorithm's Performance

We first evaluate the algorithm using uniform dataset D_1. We start the experiments by tuning different k value. Figure 8 shows the query time and accessed R-tree nodes. Note that the y-axes are in logarithmic scale. GH achieves the best performance in both time and I/O efficiency. Because the trajectories are uniformly distributed in D_1, the global heap does a good job to coordinate the candidate search around query points. No matching pair is stuck in the global heap. It is thus unnecessary to directly make up the partial-matching candidates, which involves higher cost.

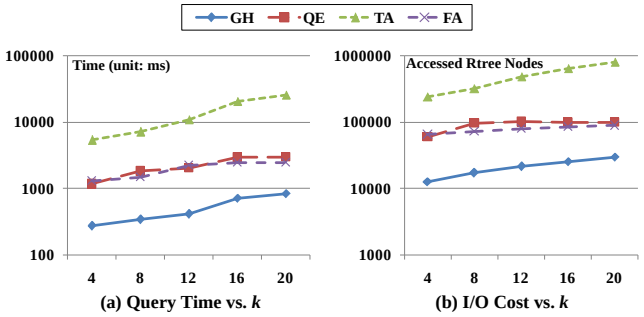


Figure 8. Performances vs. k on D_1

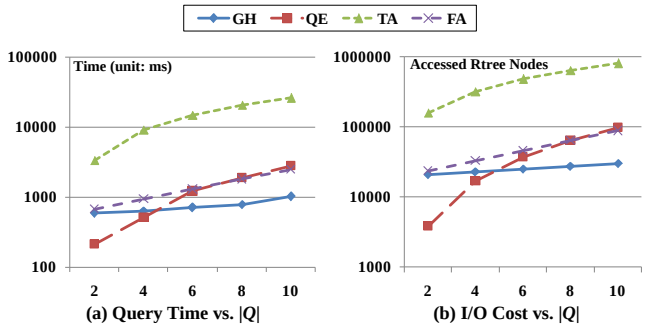
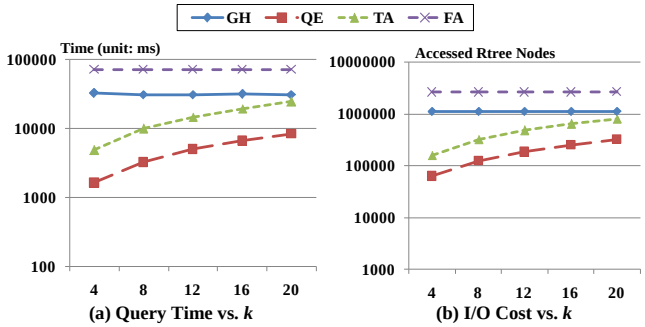


Figure 9. Performances vs. $|Q|$ on D_1

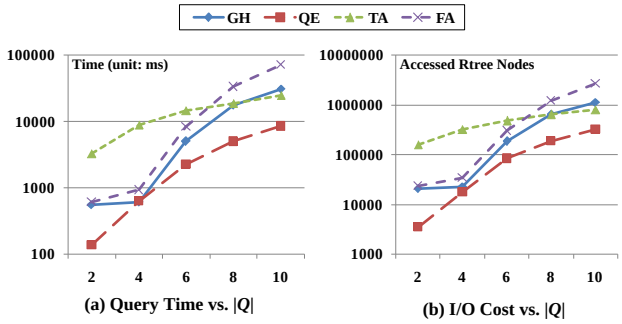
Figure 9 illustrates the influences of query size $|Q|$ in the experiments. When the query size is small, QE has better performance than GH, because in such cases, the partial-matching candidates have higher probability to become qualifiers. Hence the candidate generation ends earlier. When the query size grows larger, GH outperforms other algorithms with the power of coordinated candidate search. It is also more robust than other competitors.

The second part of our experiments is carried out on skewed dataset D_2. Based on default settings, we evaluate the time and I/O costs of the four algorithms with different values of k and $|Q|$. From Figures 10 and 11 one can clearly see the problem of outlier locations on skewed datasets. The best algorithm on D_1, GH, has degenerated to two orders of magnitude slower. FA also has the same problem. Fortunately, QE still achieves a steady performance, it can process the k-NNT query in 10 seconds even with the largest k and $|Q|$ (k = 20, $|Q|$ = 10).



(a) Query Time vs. k (b) I/O Cost vs. k

Figure 10. Performances vs. k on D_2



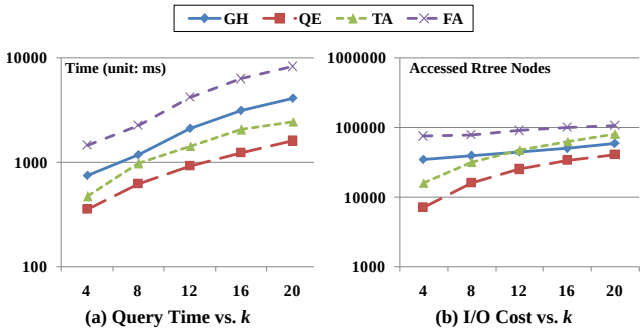
(a) Query Time vs. $|Q|$ (b) I/O Cost vs. $|Q|$

Figure 11. Performances vs. $|Q|$ on D_2

An interesting observation is that, in Figure 10, GH's time and I/O costs are almost not influenced by the number of k, but the costs increase rapidly with $|Q|$ in Figure 11. This phenomenon can be illustrated by the mechanism of global heap. In the case of outlier point locations, the global heap is difficult to pop out the first matching pair, since such query is distant from all the trajectories. The global heap has to wait for a long time before the first full-matching candidate is generated. Once the global heap pops out the first pair of an outlier point location, it may quickly pop out more matching pairs of that outlier, because the search region has already been enlarged to

reach the dense areas of trajectories. As an illustration, please go back to Figure 5, in which GH uses four more iterations to pop out $<p_{2,7}, q_3>$. But once this pair is popped out, other pairs such as $<p_{1,7}, q_3>$ will soon be popped. Hence if GH generates the first full candidate, it can quickly generate more to form a complete candidate set.

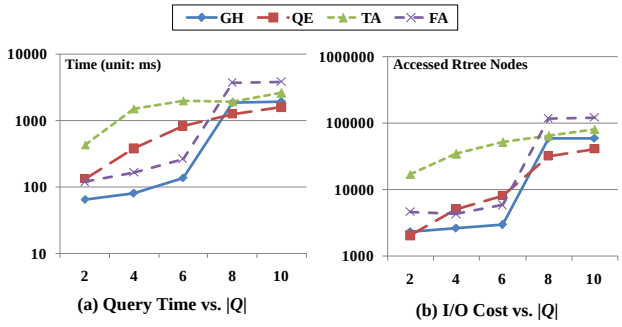
We also conduct experiments on the real dataset D_3. In the experiments of tuning k value, QE is the winner, as shown in Figure 12. Comparing to Figures 8 and 10, the algorithm performances on D_3 are more to close to the ones on skewed dataset D_2, since the distribution of real trajectories is more likely to be skewed.



(a) Query Time vs. k (b) I/O Cost vs. k

Figure 12. Performances vs. k on D_3

In Figure 13, GH has better performance than QE when the query size is less than 6. As query size grows, GH's performance degenerates rapidly and QE will be the most efficient algorithm. With more query points, the probability of outlier location becomes higher. GH suffers from such a problem, but QE is relatively robust.



(a) Query Time vs. $|Q|$ (b) I/O Cost vs. $|Q|$

Figure 13. Performances vs. $|Q|$ on D_3

Finally, we tune the full-matching ratio μ used in QE. The system starts by setting μ as 20% and gradually increases the parameter. The results are recorded in Figure 14. Overall, QE's performance is much better on the real dataset since the data size is much smaller. From the figures one can learn that, when μ is smaller than 40%, the qualifiers are not enough to stop the candidate generation process, but if μ is larger than 60%, the system has to make up too many partial-matching candidates that are unlikely to be qualifiers. QE achieves the best efficiency when μ ranges from 40% to 60%.

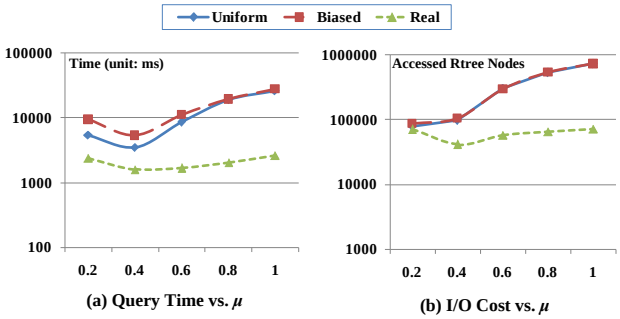


Figure 14. Performances vs. μ

5.3 Discussions

In summary, GH achieves the best efficiency when the trajectories are uniformly distributed, QE is more suitable for the skewed dataset. Although there is no overall winner on real dataset, we suggest QE as the best choice since it is much robust than GH. In addition, QE is better to handle the complex queries with more location points.

When processing *k*-NNT query, we set the upper-bounds of k as 20 and $|Q|$ as 10. Since most users are only interested in the first few results, and it is impractical for them to enter tens of query points as input. From the trends of performances of QE and GH, we can see that the algorithms work without problem with even larger k and $|Q|$. In the experiments, datasets are indexed by R-trees. However, the proposed techniques are flexible to higher dimensions or alternative indexes, such as R* Trees and A-trees. The methods can also be extended to road network. We only need to make minor changes by adjusting the distance computation of individual heap for road network distances.

In the experiments, we use the GPS datasets with high sampling frequency. It is possible that the raw trajectory data collected from other tracking devices are not as ideal as expected. The trajectories could be sparse due to device limitations or users' turning off tracking sensors. On the other hand, if a very long trajectory traverses the entire region, it has higher probability to be selected as *k*-NNT. In such cases, the trajectories should be preprocessed with similar sampling frequency and length.

6 Related Work

A number of algorithms were proposed to process *k*-NN queries for point objects. Roussopoulos *et al.* propose a *depth-first* algorithm for *k*-NN query in 2D Euclidean space [11]. Hjaltason *et al.* improve the algorithm with a *best-first* search strategy [6]. Papadias *et al.* propose the concept of *Aggregate k-NN* (ANN) query and extend the problem to multiple query points [5]. However, those methods search *k*-NN as points, while the objects in *k*-NNT query are trajectories. The distance measures are different. It is difficult to use them for *k*-NNT query processing.

There are many studies about *searching trajectories by sample* based on different similarity measures. Some representative works are: Chen *et al.* propose the *edit distance* [9]; Sherkat *et al.* develop a *uDAE-based MBR approximation* approach [13].

Those methods define the similarity functions on the shape of trajectories, but they do not consider the spatial properties.

There are several studies in the category of searching trajectory by point locations. As a pioneering work, Frentzos *et al.* propose the concepts of *Moving Object Query* (MOQ) to find the trajectories near a single query point [4]. The moving object query can be seen as a special case of *k*-NNT query with a single query point.

The most related work to *k*-NNT query is the *k-Best Connected Trajectory* (*k*-BCT) query [20]. The biggest difference between *k*-BCT and *k*-NNT is the distance measure. In the *k*-BCT query, the similarity function between a trajectory R and query locations Q is defined by an exponential function, where $Sim(\mathrm{R}, Q) = \sum_{q \in Q} \mathrm{e}^{-dist(R,q)}$. With the exponential function, the *k*-BCT query assigns a larger contribution to closer matched query points and trajectories than far away ones. However, the exponential function of *k*-BCT query may not be robust to the distance unit.

For a query with m point locations, the FA algorithm generates candidates in a parallel manner around each point location [12]. Without the coordination of global heap, the FA algorithm costs more time on candidate search and also more time to prune the candidates since the lower-bound cannot be computed to help processing. Fagin *et al.* propose another Threshold Algorithm (TA) for top k aggregation query in distributed systems and middleware [14]. For any generated candidates, TA computes their aggregated distances immediately. Indeed it can be seen as a special case of qualifier expectation-based method when the full candidate ratio μ is set to 100%. In Section 5.2 we study the influence of ratio μ. The results indicate that the best value for μ is in the range of 40% to 60%.

7 Conclusions and Future Work

In this study we present the *k-Nearest Neighboring Trajectory* (*k*-NNT) query to retrieve top *k* trajectories with the minimum aggregated distance to a set of point locations. This *k*-NNT query will facilitate a board range of applications, such as travel recommendation, traffic analysis and biological research. We propose a global heap-based method, which coordinates candidate generation, guarantees the completeness of candidates and offers a lower-bound for candidate verification. Meanwhile, by leveraging the proposed measure of qualifier expectation, our method handles the trajectory dataset with skewed distribution and outlier query locations, which are practical situations we need to face in the real world. We evaluate our methods using both real-world and synthetic trajectory datasets, and compare our methods with the state-of-the-art algorithms. The results demonstrate the feasibility and effectiveness of our proposed methods.

This paper is the first step of our trajectory search study. We plan to evaluate the *k*-NNT queries with different constraints, *e.g.*, temporal constraints and traffic congestions. We are also interested in applying them to advanced spatial applications such as driving route recommendation and traffic management.

Acknowledgements

The work was supported in part by U.S. NSF grants IIS-0905215, CNS-0931975, CCF-0905014, IIS-1017362, the U.S. Army Research Laboratory under Cooperative Agreement No. W911NF-09-2-0053 (NS-CTA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] http://foursquare.com/
[2] http://www.movebank.org
[3] http://weather.unisys.com/hurricane/atlantic
[4] E., Frentzos, K., Gratsias, N., Pelekis, Y., Theodoridis. Nearest Neighbor Search on Moving Object Trajectories. In SSTD 2005.
[5] D., Papadias, Y, Tao, K., Mouratidis, K., Hui. Aggregate Nearest Neighbor Queries in Spatial Databases. ACM TODS, 30(2): 529–576.
[6] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. ACM TODS, 24(2):265–318, 1999.
[7] GeoLife GPS Trajectories Datasets. Released at: http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/default.aspx
[8] J., Yuan, Y., Zheng, C., Zhang, W., Xie, X., Xie, G., Sun, Y., Huang. T-Drive: Driving Directions Based on Taxi Trajectories. ACM SIGSPATIAL GIS 2010.
[9] L. Chen and R. Ng. On the marriage of lp-norms and edit distance. In VLDB 2004.
[10] L. Tang, X. Yu, S. Kim, J. Han, et.al. Tru-Alarm: Trustworthiness Analysis of Sensor Networks in Cyber-Physical Systems. In ICDM 2010.
[11] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In SIGMOD 1995.
[12] R. Fagin. Combining fuzzy information from multiple systems. J. Comput. System Sci., 58:83-89, 1999.
[13] R. Sherkat and D. Rafiei. On efficiently searching trajectories and archival data for historical similarities. In PVLDB, 2008.
[14] R., Fagin, A., Lotem. Optimal aggregation algorithms for middleware. In PODS 2001.
[15] V. W, Zheng, Y., Zheng, X., Xie, Q., Yang. Collaborative location and activity recommendations with GPS history data. In WWW 2010.
[16] Y., Zheng, L., Zhang, X., Xie, W., Y. Ma. Mining interesting locations and travel sequences from GPS trajectories. In WWW 2009.
[17] Y., Zheng, L., Wang, X., Xie, W., Y., Ma. GeoLife: Managing and understanding your past life over maps, In MDM 2008.
[18] Y., Zheng, X., Xie, W., Y., Ma. GeoLife: A Collaborative Social Networking Service among User, location and trajectory. IEEE Data Engineering Bulletin. 33(2): 32-40.
[19] Y., Zheng, Y., Chen, X., Xie, W., Y, Ma. GeoLife2.0: A Location-Based Social Networking Service. In MDM 2009.
[20] Z., Chen, H., T., Shen, X. Zhou, Y., Zheng, X., Xie. Searching trajectories by locations: an efficiency study. In SIGMOD 2010.