Open access • Proceedings Article • DOI:10.1109/MDM.2010.40

Efficient Evaluation of k-Range Nearest Neighbor Queries in Road Networks

Jie Bao, Chi-Yin Chow, Mohamed F. Mokbel, Wei-Shinn Ku

Institutions: University of Minnesota, Auburn University

Published on: 23 May 2010 - Mobile Data Management

Topics: Query optimization, Query expansion, Web query classification, Online aggregation and Sargable

Efficient Evaluation of k-Range Nearest Neighbor Queries in Road Networks

*Jie Bao *Chi-Yin Chow *Mohamed F. Mokbel †Wei-Shinn Ku
* Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, USA
Email: {baojie, cchow, mokbel}@cs.umn.edu
† Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849, USA
Email: weishinn@auburn.edu

***Abstract*—A k-Range Nearest Neighbor (or kRNN for short) query in road networks finds the k nearest neighbors of every point on the road segments within a given query region based on the network distance. The kRNN query is significantly important for location-based applications in many realistic scenarios. For example, (1) the user's location is uncertain, i.e., user's location is modeled by a spatial region, and (2) the user is not willing to reveal her exact location to preserve her privacy, i.e., her location is blurred into a spatial region. However, the existing solutions for kRNN queries simply apply the traditional k-nearest neighbor query processing algorithm multiple times, which poses a huge redundant searching overhead. To this end, we propose an efficient kRNN query processing algorithm in this paper. Our algorithm (1) employs a shared execution approach to eliminate the redundant searching overhead, and (2) provides a parameter that can be tuned to achieve a tradeoff between the query processing performance and the storage overhead, while guaranteeing the user's exact k-nearest neighbors are included in the query answers. The experimental results show that our algorithm always outperforms the existing solution in terms of query response time, and the introduced tuning parameter is an effective way to achieve the tradeoff between the query response time and the storage overhead.**

I. Introduction

k-Nearest Neighbor (or kNN for short) query is one of the most popular query types in location-based services [1], [2], where a user issues a kNN query to the service provider for the k-nearest objects of interest to her current location. With the advances in spatial databases, the kNN query has been extended from the Euclidean space to the road network environment [3], [4], [5], where the user can issue the kNN query to find her k-nearest objects of interest based on the network distance. The kNN query result over the network distance or the travel time is more meaningful to the user, since the user is usually traveling on the road network,. Recently, the kNN query has been further extended to k-range nearest neighbor (or kRNN for short) query in the road network. The main idea of the kRNN query is to find the k-nearest objects of interest to every point on the road segments within a query region given by the user. Figure 1 gives an example of a kRNN query in a road network, where each line represents a road segment, each circle represents an intersection of road segments, and the dotted rectangle is the given query region. In this example, the query answer includes the k-nearest objects of interest to every point on the road segments, $\overline{BE}$, $\overline{DE}$, and $\overline{EF}$, which are covered by the query region.

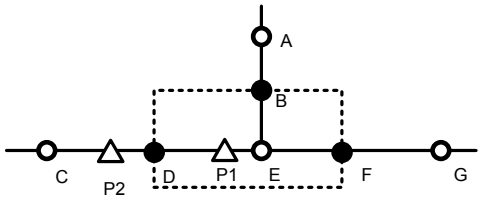

Fig. 1: An example of a k-range nearest neighbor query in road networks.

Many research efforts have shown that the kRNN query is significantly important for many location-based applications:

- **Uncertain location.** The uncertain location information is posed by either the imprecision of the positioning techniques or the discontinuous location updates [6], [7], [8]. With the imprecise positioning techniques, e.g., 3G cellular services and Wi-Fi, the user is not able to acquire her exact location. On the other hand, the discontinuousness of location update is caused by the agreement between the user and the service provider to reduce the location-update frequency in order to reduce the energy consumption and communication overhead (e.g., [9], [10]).
- **Privacy-aware applications.** Due to the possibility of privacy leakages with a potentially untrusted location-based server, the user may not be willing to expose her exact location to the service provider. Many existing privacy-preserving techniques have been proposed to blur the user's exact location into a spatial region (e.g. [11], [12], [13], [14], [15], [16], [17]).

In these two realistic scenarios, the user's location is modeled by a spatial region. The location-based database server only knows that the user is anywhere within the query region, rather than an exact location point. Thus, the server has to find the k-nearest objects of interest to every point on the road segments within the query region in order to guarantee that the exact kRNN query answer is included in the query result returned to the user. It has been proved that the exact answer of the

This research was supported in part by NSF grants IIS-0811998, IIS-0811935, CNS-0708604, CNS-0831502, CNS-0855251, and by a Microsoft Research Gift.

kRNN query includes (1) the objects within the query region and (2) the k-nearest objects of each intersection point, i.e., termed a *boundary point*, of the query region and the road segments in the underlying road network [14].

Unfortunately, existing approaches for kRNN queries that apply a traditional kNN query processing algorithm in the road network, i.e., the incremental network expansion (INE) algorithm [3], to every *boundary point* incur a huge redundant searching overhead [11], [12], [14]. We will illustrate the existing approach based on the INE algorithm using the example given in Figure 1, where the *boundary points* of the query are at the intersections B, D, and F. The existing approach executes a range query to select the objects on the road segments within the query region (represented by a dotted rectangle), i.e., the road segments $\overline{DE}$, $\overline{BE}$, and $\overline{EF}$, to an answer set; and hence, the answer set includes the object P_1. Then, it executes the INE algorithm for each *boundary point*. The kNN query processing for the *boundary point* D searches the road segments $\overline{CD}$ and $\overline{DE}$. The kNN query processing for B searches the road segments $\overline{AB}$, $\overline{BE}$, $\overline{DE}$, $\overline{CD}$, $\overline{EF}$, and $\overline{FG}$. Finally, the kNN query processing for F searches the road segments $\overline{AB}$, $\overline{BE}$, $\overline{CD}$, $\overline{DE}$, $\overline{EF}$, and $\overline{FG}$. The final answer set includes two objects P_1 and P_2. As a result, the total number of road segments processed by the traditional approach is 17. However, an optimal solution searches only six road segments, i.e., $\overline{AB}$, $\overline{BE}$ $\overline{CD}$, $\overline{DE}$, $\overline{EF}$ and $\overline{FG}$, and each of these road segment is processed once. Therefore, the redundant searching overhead of the existing approach is $(17-6)/6\times100\% = 183\%$. The redundant searching overhead could become even much worse if the query region contains more *boundary points* or the objects are sparsely distributed in the road networks.

To avoid the redundant searching overhead in the traditional approach, we propose an efficient algorithm to process kRNN queries in the road network. The main idea of our algorithm is to share the execution among the searching process for each *boundary point* of the query. Our shared execution paradigm requires the shortest network distance from each *boundary point* to a certain set of objects in order to find the query answer. Such shortest distances can be either pre-computed and stored in the main memory or computed on-the-fly during the query processing. Although pre-computing all possible required shortest network distances can reduce query processing time, it incurs very high storage overhead. To this end, our algorithm also introduces a system parameter that controls the amount of space for storing the pre-computed shortest network distance. A larger parameter value achieves better query processing performance, but it incurs higher storage overhead. Thus, this parameter can be tuned to achieve a tradeoff between the query processing performance and the storage overhead. Our kRNN query processing algorithm is evaluated through simulated experiments. The experimental results show that (1) our algorithm always outperforms the existing solution based on the INE algorithm in terms of both query processing time and query response time, and (2) the tuning parameter is an effective way to provide a trade off between the query response time and the storage overhead.

The rest of the paper is organized as follows: Section II highlights the related works. Section III gives our system model. Our proposed kRNN query processing algorithm is presented in Section IV. The experimental results are given in Section V. Finally, Section VI concludes the paper.

II. Related Work

In location-based services, processing k-nearest neighbor (kNN) queries in the road network has been well studied (e.g., [3], [4], [5], [18], [19], [20], [21], [22], [23]). Among the existing solutions for kNN queries, the simplest one is the incremental network expansion (INE) algorithm [3]. The basic idea of the INE algorithm is to incrementally search the road segments from the query point until the k-nearest objects of interest of the query point are found. Since the INE algorithm does not need any pre-computed shortest distance information, it incurs low storage overhead. However, the limitation of the INE algorithm is that it cannot take the advantage of the optimization based on the pre-computed shortest distance. To overcome this limitation, there are many kNN query processing algorithms that utilize the pre-computed network distance to optimize the query processing (e.g., [4], [5], [18], [19], [20], [21], [22], [23], [24], [25]). Although these optimized algorithms perform faster than the INE algorithm, they incur higher storage overhead. In the worst case, the storage overhead of the algorithm requiring the pre-computed shortest distance is $O(n^2)$, where n is the number of intersections of the road segments in the underlying road network .

To our best knowledge, the k-range nearest neighbor (kRNN) query has been only studied in the context of privacy-preserving location-based services where the exact location of a kNN query issuer is blurred into a spatial region [11], [12], [14], [13]. The basic idea of the existing solutions for the kRNN query is to execute a spatial range query to retrieve the objects on the road segments within the query region, and use the INE algorithm to find the k-nearest objects of each intersection point, termed a *boundary point*, of the query region boundary and the road segments in the underlying road network. As illustrated in Figure 1, simply applying the INE algorithm multiple times for processing kRNN queries incurs a huge redundant searching overhead.

Our kRNN query processing algorithm can distinguish itself from the existing solutions for the kRNN query, as it (1) employs a shared execution paradigm to share the execution of the searching process of each *boundary point* of the query to eliminate redundant computational overhead, and (2) introduces a new tuning parameter to control the amount of space for storing the pre-computed network distance in order to achieve a tradeoff between the query processing performance and the storage overhead. It is important to note that our algorithm always returns the exact k-nearest objects of interest to the user within the answer set, regardless of the actual user location within the query region and the value of the tuning parameter.

III. System Model

In this section, we describe the road network and system model, define the k-range nearest neighbor (kRNN) query and its answer, and present the formal definition of our problem.

Road network and system model. We model the underlying road network as a weighted undirected graph $G = (V, E)$ where E is an edge set of the road segments in the road network, V is a vertex set of the intersection points of the road segments, and each edge is given the length of its corresponding road segment as a weight. In this work, we consider our system with a mobile environment, in which the mobile user is able to communicate with the service provider through wireless communication infrastructure, e.g., 3G cellular services and Wi-Fi.

Definitions. A kRNN query is defined in a form (*ID*, *Region*, *ObjectofInterest*, k), where *ID* is the user's unique identity, *Region* is the query region, *ObjectofInterest* defines the type of objects of interest of the query, and k is the required number of nearest objects of interest. The *Region* of the kRNN query covers a set of road segments that is referred to as *inside road segments*, while each intersection point of the *Region* boundary and the road segments in the underlying road network is referred to as a *boundary point*. To guarantee that the kRNN query answer includes the exact k-nearest objects to the user, regardless of the user's actual location within the *Region*, the correct answer for a kRNN query must include (a) the objects within the *inside road segments*, and (b) the k-nearest objects of each of *boundary point* [14].

Problem definition. We now give the formal definition of our problem. Given a k-range nearest neighbor query Q with a query region *Region*, the underlying road network G, and a set of objects of interest of the query O, we want to find the k-nearest objects in O to every point on the road segments in G within Q.*Region*. The key objectives of our algorithm are to (1) eliminate the redundant searching overhead in the traditional solution to improve query processing performance, and (2) design a tuning parameter that controls the amount of space dedicated to store the pre-computed shortest distance information to achieve a tradeoff between the query processing performance and the storage overhead.

IV. Efficient kRNN Query Algorithm

In this section, we first present the key data structures and main idea of our k-range nearest neighbor (kRNN) query processing algorithm. Then, we describe our algorithm with a detailed example, and the correctness of our algorithm is proofed at the end of this section.

A. Data Structures

In general, our algorithm has five key data structures.

Shortest distance table. This table stores the pre-computed shortest network distance between two vertices in the road network. Each entry in this table is in a form ($< x, y >$, $d(x, y)$), where $< x, y >$ is a key, and $d(x, y)$ is the shortest distance from x to y. We assume that the shortest distance is symmetric, i.e., $d(x, y) = d(y, x)$. In addition, we have a system tuning parameter that controls the size of the shortest distance table in order to achieve a tradeoff between the query processing performance and the storage overhead. The table stores the most frequently accessed shortest distance information based on historical statistics, and it is updated periodically, e.g., hourly or daily.

Answer set table. This table contains the data objects that will be returned to the user. Each entry in this table contains the data object identity along with its distance to each *boundary point* of the query region.

Searching queue. We construct a *searching queue* for each *boundary point* of the query to store the road segments that will be searched by our algorithm. In the *searching queue* Q_i, the road segments are sorted by their shortest distance to the corresponding *boundary point*, i, in an increasing order. Each *searching queue* is associated with two parameters.

- *Searching bound.* This parameter records the network distance from the corresponding *boundary point* to its k-th nearest object in the *answer set table*.
- *Searched distance.* This parameter records the distance from the corresponding *boundary point* to the first item in its *searching queue*, which also indicates the network distance this *searching queue* has covered.

Search collision point table. This table maintains the vertices of the road segments that are searched by the searching process of more than one *boundary points*. Each entry in this table contains the vertex ID and its distance to each *boundary point* of the query.

Searched segment set. This set contains the road segments that have been searched by the algorithm.

B. Main Ideas

We will discuss the main ideas of our algorithm.

Shared execution. In our algorithm, we share the result of the searching process of each *boundary point* of the query to avoid the redundant searching overhead in the traditional solution. With this shared execution paradigm, each *boundary point* is able to know its distance to the data objects that are found by the searching process of the other *boundary points*.

Searching bound. The *searching bound* is the value to tell the algorithm to terminate the searching process of a *boundary point*. To determine the *searching bound* of a *searching queue*, we have to know the distance from the corresponding *boundary point* to each data object that is found by the algorithm. In the algorithm, such a distance can be either retrieved from the *shortest distance table* or calculated based on the *search collision point table* on-the-fly along with the query processing.

Approach 1: Using the shortest distance table. Given a *boundary point* b on a road segment v_iv_j and a data object o on a road segment v_pv_q, if the shortest distances of the keys $< v_i, v_p >$, $< v_i, v_q >$, $< v_j, v_p >$, and $< v_j, v_q >$ are stored in the *shortest distance table*, we can easily calculate the shortest distance from the *boundary point* to the data object, i.e., $min(d(b, v_i) + d(v_i, v_p) + d(o, v_p), d(v_i, v_p) + d(b, v_i) + d(o, v_p), d(v_j, v_p) + d(b, v_j) + d(o, v_p), d(v_j, v_q) +$

$d(b, v_j) + d(o, v_p)$). Although using the *shortest distance table* can improve the query processing performance, storing the shortest distance of every pair of vertices in the road network may incur very large storage overhead. To this end, we introduce a tuning parameter to specify the size of the *shortest distance table*; and therefore, tuning this parameter can achieve a tradeoff between the query processing performance and the storage overhead. When we need to calculate a shortest distance, we first check if the *shortest distance table* has enough information for computing the shortest distance. If this is the case, we simply retrieve the four relevant entries to compute the shortest distance. Otherwise, we calculate the shortest distance as using the *Approach 2*.

Approach 2: Using the search collision point table. A search collision takes place, when the searching process of a *boundary point* attempts to search a road segment that has been searched by the searching process of some other *boundary points*. Then, two vertices of such a road segment that are referred to as *search collision points* are inserted to the *search collision point table* along with its distance to each *boundary point* of the query, which is calculated based on the information of its ancestral *search collision point* which this road segment is expanded from. If a data object is found by the algorithm, we identify its ancestral *search collision point* from which the algorithm finds the object, and update the distance from each *boundary point* of the query to the object based on the distance information of this ancestral *search collision point* in the *search collision point table*.

C. Algorithm

Initially, we construct a *searching queue* for each *boundary point* of the query, and then set the *searched distance* to zero and the *searching bound* to ∞ for each *searching queue*. In general, our kRNN query processing algorithm has two main steps.

Step 1. Inside road segment search step. In this step, we find the data objects within the *inside road segments*, and insert them into the *answer set table*. Then, we calculate the shortest distance for every pair of *boundary points* of the query, and insert each *boundary point* to the *search collision point table* along with its shortest distance to each of the other *boundary points*. This step is depicted in Lines 6 to 8 in Algorithm 1. It is important to note that when the algorithm needs to compute a shortest distance, it first checks if it can use the first approach described in Section IV-B to determine the shortest distance; otherwise, it uses the second approach to do so.

Step 2. Boundary point expansion step. The main purpose of this step is to search beyond the *inside road segments* by iterations. In each iteration, we select the *searching queue* with the minimum *searched distance* to process. If the *answer set table* has at least k objects, the *searching bound* is set to the distance from the corresponding *boundary point* to its k-th nearest data object in the *answer set table* (Line 13 in Algorithm 1). Then, we check for the termination condition, i.e., (1) the *searched distance* of the selected *searching queue* is equal to or larger than the *searching bound* for all the other

Algorithm 1 Efficient kRNN Query Processing in Road Networks.

Input: *Boundary Point Set* B, *Inside Road Segment Set* R, and *Integer* k.

```
1: Initialize the data structures
2: for Each boundary point b_i in B do
3:    Create a searching queue Q_i for b_i
4: end for
5: //Step 1: Inside road segment search step
6: Insert the data objects on the road segments in R to the answer set table
7: Find the shortest distance of every pair of boundary points in B
8: Insert each boundary point in B to the search collision point table
9: //Step 2: Boundary point expansion step
10: while Not all Q_1, Q_2, ..., Q_|B| are terminated do
11:    Select Q_i with the minimum searched distance
12:    if The number of data objects in the answer set table ≥ k then
13:       Set the searched distance of Q_i to the shortest distance from b_i to its k-th nearest object in the answer set table
14:    end if
15:    if Q_i meets the termination condition, i.e., its searched distance ≥ all searching bound or Q_i finds k data objects by itself then
16:       Terminate Q_i
17:    else
18:       while Q_i searched distance is not changed do
19:          S ← the top road segment in Q_i
20:          if S is in the searched segment set then
21:             if S contains data objects then
22:                Update the answer set table
23:             end if
24:             Update the search collision point table
25:             Remove S from Q_i
26:          else
27:             Insert the data objects on S to the answer set table
28:             Insert S to the searched segment set
29:             Insert the adjacent road segments of S into Q_i
30:          end if
31:       end while
32:    end if
33: end while
34: Return the data objects in the answer set table to the user
```

searching queues or (2) no less than k data objects has been found by selected *searching queue* itself. If the termination condition takes place, the searching on the selected *searching queue* is terminated and will not be selected to process in the future executions. Otherwise, the algorithm continues to process the road segment in the selected *searching queue* based on two cases.

Case 1: The road segment is in the searched segment set. In this case, a search collision takes place, i.e., there are different shortest pathes from the query region to this road segment. If there are some data objects on this segment, for each data object, we update the shortest distance from each *boundary point* to the data object in the *answer set table* if necessary (Line 22). The two vertices of the segment are inserted into the *search collision point table* along with their distance to each *boundary point* (Line 24). Then, the road segment is removed

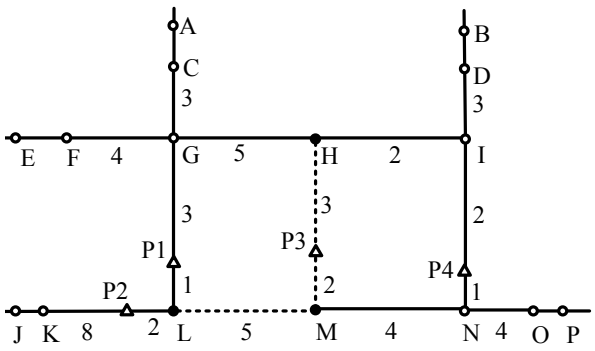

Fig. 2: An example of road networks.

from the queue without considering its adjacent road segments because these adjacent road segments will be searched by the searching process of some other *boundary point* (Line 25).

Case 2: The road segment is NOT in the searched segment set. The data objects found on the road segment are inserted into the *answer set table* along with its shortest distance to each *boundary point*, and the road segment is inserted into the *searched segment set* (Lines 27 to 28). The adjacent road segments of the processed road segment are inserted into the *searching queue* for further processing (Line 29).

Our algorithm repeats this step until all the *searching queues* meet the termination condition. After the algorithm terminates, the objects stored in the *answer set table* are returned to the user as a query result.

D. Example for efficient kRNN query processing algorithm

In this section, we give a detailed example to illustrate our kRNN query processing algorithm. Figure 2 depicts a road network, where the road segments are represented by lines, the intersection of the road segments are represented by circles, and the data objects are represented by triangles. In this example, the user issues a 2-RNN query with a query region that contains two road segments $\overline{HM}$ and $\overline{LM}$, which are represented by dotted lines. The *boundary points* of this query are H, L, and M, which are represented by black circles. In our example, we assume that the tuning parameter for the *shortest distance table* is zero, i.e., the *shortest distance table* does not have any shortest distance information. Thus, all shortest distance information required by the algorithm is calculated based on the *search collision point table* on-the-fly.

Figure 3 gives the status of the *searching queues* for each iteration. In each *searching queue*, the number under the ID of the road segment is the shortest distance from the road segment to the corresponding *boundary point*. The arrow over the *searching queue* indicates the active *searching queue*, which has the minimum *searched distance*, for the current iteration. Table I depicts the value changes in the *searching bound* and *searched distance* for each *searching queue*, the *searched segment set*, the *answer set table* and the *search collision point table* during the query processing.

Initial step. The initial step (Figure 3a) for the algorithm is to construct the *searching queue* for each *boundary point*, Q_H, Q_L and Q_M. During the *inside road segment search step*, data object P_3 is found. P_3 is inserted into the *answer set table* with the shortest distance to all the *boundary points*, as depicted in the *initial* column in Table I. The *inside road segments* $\overline{HM}$ and $\overline{LM}$ are inserted into the *searched segment set*, while all the *boundary points* are inserted into the the *search collision point table* with the shortest distance information to the other *boundary points*. For each *boundary point*, its adjacent road segments are inserted into its *searching queue*, as depicted in Figure 3a. Since we only found one data object, which is less than the k value, the *searching bound* of each *searching queue* remains ∞.

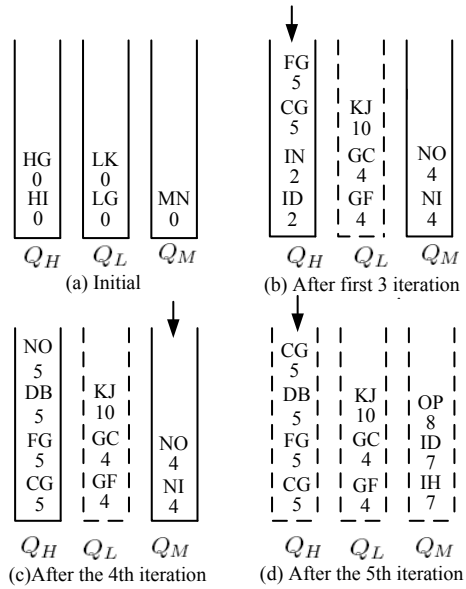

Fig. 3: An example of query procedure in each of searching queue.

The first 3 iterations. Since the *searched distance* of all the *searching queues* is initially set to be zero, we can arbitrarily select any *searching queue* to start with. For instance, we start the query processing with Q_H. **In the first iteration**, $\overline{HI}$ and $\overline{HG}$ are processed, but no data object is found. Our algorithm only inserts their adjacent segments into the Q_H. **In the second iteration**, we select Q_L to process road segments $\overline{LG}$ and $\overline{KL}$. During this iteration, we find P_2 and P_1 on $\overline{KL}$ and $\overline{LG}$, respectively. Since $\overline{KL}$ and $\overline{LG}$ are not in the *searched segment set*, the distance information of P_1 and P_2 is calculated based on the searching collision point that they expanded from, which is L. The distance information is calculated by using the sum of the distance information of L stored in the *collision points table* and the distance from P_1 to L. The distance from P_1 to all the *boundary points* is updated in the *answer set table*, i.e., the distance from P_1 to L is 1, the distance from P_1 to H is $10 + 1 = 11$ and the distance from P_1 to M is $5+1 = 6$. Similarly, the distance information from P_2 to all *boundary points* are updated in the *answer set table* accordingly. Since Q_L finds two data objects by itself, it is terminated, as its *searching queue* is marked by dotted lines. **In the third iteration**, Q_M processes $\overline{MN}$. Since we

TABLE I: An example of query procedure in each data structure.

		Initial values			After first 3 iterations			After the 4th iteration			After the 5th iteration		
		Q_H	Q_L	Q_M	Q_H	Q_L	Q_M	Q_H	Q_L	Q_M	Q_H	Q_L	Q_M
Searching bound		∞	∞	∞	11	2	6	4	2	6	4	2	5
Searched distance		0	0	0	2	4	4	5	4	4	5	4	7
Searched segments set		$\overline{LM}$ $\overline{HM}$			$\overline{LM}$ $\overline{HM}$ $\overline{LG}$ $\overline{LK}$ $\overline{MN}$ $\overline{IH}$ $\overline{HG}$			$\overline{ID}$ $\overline{IN}$ $\overline{LM}$ $\overline{HM}$ $\overline{LG}$ $\overline{LK}$ $\overline{MN}$ $\overline{IH}$ $\overline{HG}$			$\overline{ID}$ $\overline{IN}$ $\overline{LM}$ $\overline{HM}$ $\overline{LG}$ $\overline{LK}$ $\overline{MN}$ $\overline{IH}$ $\overline{HG}$ $\overline{NO}$		
Answer set	P_1				11	1	6	11	1	6	11	1	6
	P_2				12	2	7	12	2	7	12	2	7
	P_3	3	7	2	3	7	2	3	7	2	3	7	2
	P_4							4	14	9	4	10	5
Search collision points	H	0	10	5	0	10	5	0	10	5	0	10	5
	L	10	0	5	10	0	5	10	0	5	10	0	5
	M	5	5	0	5	5	0	5	5	0	5	5	0
	I										2	12	7
	N										5	9	4

do not find any data object on road segment $\overline{MN}$, we insert its adjacent road segments to the *searching queue*. After the first three iterations, the *searched distance* (denoted as SD) of each *searching queue* is updated, i.e., $Q_H.SD$=2, $Q_L.SD$=4 and $Q_M.SD$=4. Moreover, we have more than k data objects in the *answer set table*, the *searching bound* (denoted as SB) of each *searching queue* can be updated, i.e., $Q_H.SB$=11, $Q_L.SB$=2 and $Q_M.SB$=6.

The 4th iteration. In this iteration, Q_H is selected to be processed, because it has the smallest *searched distance*, as depicted in Figure 3b. Q_H searches $\overline{ID}$ and $\overline{IN}$, and finds P_4 on $\overline{IN}$. The distance information for P_4 is calculated based on its ancestral *search collision point* H. As a result, the distance from P_4 to H is $0 + 4 = 4$, the distance from P_4 to L is $10 + 4 = 14$ and the distance from P_4 to M is $5 + 4 = 9$. The *searching bound* is updated for Q_H and Q_M to be 4 and 6, respectively. The $Q_H.SD$ is updated to be 5 and $\overline{ID}$ and $\overline{IN}$ are inserted into the *searched segment set*. After this iteration, $Q_H.SD$ is greater than $Q_H.SB$, but $Q_H.SD$ is smaller than $Q_M.SB$; and therefore, Q_H does not meet the termination condition.

The 5th iteration. In the fifth iteration, Q_M is selected to be processed, as it has the smallest *searched distance*. $\overline{NO}$ is processed first without finding any data object. After that, Q_M processes $\overline{IN}$. However, $\overline{IN}$ is already in the *searched segment set*, which generates a search collision and makes I and N *search collision points*. Both of them are inserted into the *search collision points table* and the distance from the point to every *boundary point* is calculated. The distance from I to all the *boundary points* is first calculated based on the *search collision point* H, where it is expanded from. The distances from to I to H, L and M are updated to be 2, 12 and 7, respectively. The distance from N to all the *boundary points* is calculated based on its ancestral *search collision point*, M, in the same way, which makes the distances to H, L and M to be 9, 9 and 4, respectively. Then I updates its distances to each *boundary point* based on the length of $\overline{IN}$ and the distance information of point N in the *search collision point table*. The distances calculated through N for I to *boundary points* H, L and M are 12, 12 and 7, respectively. Because all the calculated results are greater than I's original value, no update will be made for I. On the other hand, the distance information for N calculated though I to the *boundary points* H, L and M is 5, 15 and 10, respectively. The distance from N to H is updated to 5, because the new calculated value is smaller. Moreover, we find P_4 is in $\overline{IN}$. As the result, the distance from P_4 to M is updated to be $4 + 1 = 5$. The distance from P_4 to L is updated to be $9 + 1 = 10$. The distance from P_4 to H remains 4 because it is smaller than $9+1 = 10$. Then, I and N are inserted into the *search collision point table* with the distance information, as shown in Table I . The *searched distance* for Q_M is updated to 7, which is greater than $Q_M.SB$ and $Q_H.SB$. Q_M is terminated after this iteration as the *searching bounds* of all the *searching queues* are less than its *searched distance*. We mark the queue in dotted lines, shown in Figure 3c.

The 6th iteration. There is only one active *searching queue* Q_H left. Since $Q_H.SD$ is greater than the *searching bound* of all the other *searching queues*, it is terminated; as Q_H is marked by dotted lines (Figure 3d). As all the *searching queues* are done, our algorithm is terminated. The data objects in the *answer set table*, i.e., P_1, P_2, P_3 and P_4, are returned to the user.

E. Proof of correctness

In this section, we prove that our kRNN query processing algorithm always returns the exact k-nearest objects to the user within the answer set, regardless of the actual user location within the query region.

Theorem 1. *Given a kRNN query with a query region* $Region$ *issued by a user residing in* $Region$ *and a set of objects of interest* $\mathcal{O}$*, the answer set* $\mathcal{A}$ *returned by our query processing*

TABLE II: Experiment Parameter Settings.

Parameter	Default Value	Range
Requested data object number (K value)	10	1 to 20
Total data object number	600	200 to 1000
Query region size (ratio over total space)	0.018	0.002 to 0.050

algorithm always includes the exact k-nearest objects to the user.

Proof: Suppose that an object $O \in \mathcal{O}$ is one of the k-nearest objects to the user, but $O \notin \mathcal{A}$. Since our algorithm selects all the objects within $Region$ to $\mathcal{A}$, O must be outside $Region$. Thus, if O is one of the k-nearest objects to the user, O has to be one of the k-nearest objects to a *boundary point* of the query. However, for each *boundary point* P of the query, our algorithm searches the road segments from P with a range of at least the distance from P to its k-th object, i.e., the termination condition for searching queue P in our algorithm, and selects all the objects within this range to $\mathcal{A}$. Since $O \notin \mathcal{A}$, O is not one of the k-nearest object of any *boundary point*. As a result, it contradicts to the assumption that O is one of the k-nearest objects to the user. ■

V. Performance Evaluation

In this section, we evaluate the performance of our kRNN query processing algorithm by comparing with the traditional solution as a baseline. We also investigate the query performance with the impact of the tuning parameter that controls the size of the *shortest distance table* and the communication bandwidth between the user and the database server.

A. Experiment Settings

The road map of Hennepin county, Minnesota, USA, which contains 39,513 nodes and 54,444 road segments, is used as the road network in our experiment. We generate 100 kRNN queries in the road network and use the mean value as the result for each experiment. Table II summarizes the parameters we used in the experiment. All the experiments are performed on a Sun Ultra 27 server with a Quad-core 3.2 GHz Intel Xeon 3570 processor and 6 GB RAM.

In our experiment, we evaluate the performance of our algorithm in terms of three major measures: (1) *query processing time*, (2) *size of the answer set*, and (3) *overall response time*. The query processing time is the average time consumed for the algorithm to finish the query processing. The size of the answer set is the average number of the data objects returned to the user. Since our algorithm may return more data objects, it is used to measure the quality of the answer set. The overall response time is the average of the sum of the query processing time and the transmission time of sending the answer to the user.

The section is organized as follows: First, we compare the query processing performance of our algorithm with the traditional solution. Then, we investigate the impact for different tuning parameters in our algorithm. Finally, we evaluate the impact of the communication bandwidth for our algorithm.

TABLE III: Relationship of area size and the boundary points.

	0.002	0.008	0.018	0.032	0.050
Boundary points	4	5	9	12	16
Inside segments	7	9	21	32	52

B. Comparison with the traditional solution

In this section, we present the results of our algorithm with the traditional solution with respect to four different settings: (1) different required numbers of the data objects (different k values) in the query, (2) different numbers of data objects in the system, (3) different sizes of query regions, and (4) different distributions of the data objects in the road network. We use two extreme cases of our algorithm to compare with the traditional approach: (a) query processing with all the shortest distance information, referred as *KRNN-F*, and (b) query processing without any shortest distance information, referred as *KRNN-E*.

1) Impact of the number of requested data objects: Figure 4a depicts the performance of our algorithm with different requested numbers of data objects (k value). As shown in the result, the query processing time of all the approaches increases, because a larger k value usually leads to a larger searching area. Moreover, we notice that the query processing time of the traditional algorithm increases rapidly as the k value gets larger. On the other hand, the query processing time of our algorithm increases relatively slower than the traditional solution in both extreme cases. The reason of the significant difference is that a larger k indicates a larger searching area, and introduces more redundant searching overhead between the searching process of each *boundary point* for the traditional approach. Our algorithm does not have this problem because the redundant searching overhead is eliminated by our shared execution paradigm.

2) Impact of the total number of data objects: Figure 4(b) shows the query processing time of our algorithm with different total numbers of data objects in the road network. The distribution of these data objects is based on the uniform distribution. As shown in the experimental result, the query processing time of all the algorithms decreases as the total number of data objects gets larger. Because the more data objects in the system, the smaller searching area the query processing algorithm needs to cover. Moreover, the traditional algorithm covers a larger search area for each *boundary point* in the area of low density data objects, which incurs more redundant searching overhead.

3) Impact of size of the query region: The query regions are modeled by circles in our experiment, whose center a location point that is randomly picked in the road network and the radius is uniformly selected with a range varied from 1 to 5 times of the average length of the road segments in the road network, i.e., the ratio of the query region size to the total system space is varied from 0.20% to 5.00%. The relationship

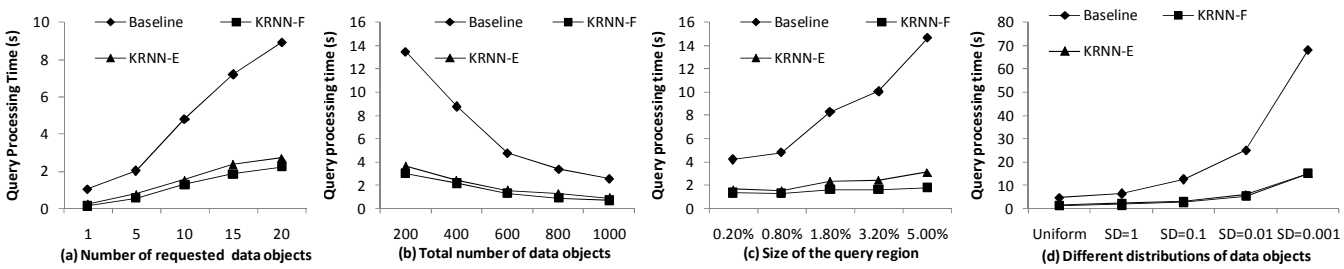

Fig. 4: Comparison with the traditional solutions.

between the size of the query region and the average number of insider road segments and the *boundary points* is depicted in Table III.

Figure 4(c) illustrates the query processing time with different sizes of query regions. The query processing time of all the algorithms increases as the size of the query region gets larger. Because with a larger query region, more *boundary points* will be introduced. It is especially a drawback for the traditional solution, because the more *boundary points* lead to more overlaps in the searching space which introduces more redundant searching overhead. The processing time of the traditional solution grows linearly with the number of *boundary points*. On the other hand, query processing time of our proposed algorithm grows relatively slower than the traditional solution. Thus, the proposed algorithm is more scalable in terms of the query region size. The significant deviation of two extreme cases of our algorithm is a result of the approximation in the calculation. The approach without the pre-computed shortest distance information is likely to generate a larger searching bound and consume more processing time.

4) Impact of distributions of data objects: Figure 4d depicts the impact of different distributions of the data objects over the road network for the query processing. We evaluate our algorithm and the baseline solution using two different distributions: (1) Uniform distribution which means the data objects in that road network is evenly deployed and no hot spot in the map. (2) Gaussian distribution with different standard deviations which means the data objects are deployed densely in a certain range of location. We use Gaussian distribution to simulate the hot spot in the road networks, i.e., downtown areas in the city. The parameter *SD* on Figure 4d is the standard deviation value of the Gaussian distribution used for the data objects deployment over the road network. The smaller value of *SD* indicates that the data objects are more likely to be placed densely in a smaller area.

The result in Figure 4d indicates that the traditional algorithm has a huge impact on the distribution of the data objects. If the data objects are unevenly distributed in the road network, the performance of the traditional solution is degraded significantly due to the redundant searching overhead. Because in the Gaussian distribution, the data objects are distributed densely in a small area. As a result, the searching process for each *boundary point* needs to cover that particular area to get the its k-nearest neighbors which incurs much more redundant searching overhead. On the other hand, different distributions of data objects do not have that significant impact on the query processing time of our algorithm as shown in Figure 4d.

5) Summary: The above experimental results show that our algorithm always outperforms the traditional solution significantly in terms of the query processing time, which is more efficient for the kRNN queries. In most cases, our algorithm gets over 100% performance gain. Moreover, our algorithm is more adaptive to the changes in the experimental parameters than the traditional solution, which indicates our algorithm is more scalable.

C. Tradeoff between the storage and performance

In this section, we present the results of our algorithm with different tuning parameters P, which indicates the different size of the *shortest distance table*. The tuning parameter P is set as a percentage value, where 100% indicates that the *shortest distance table* contains the shortest distance information for any two vertexes and 0% indicates that the shortest distance table does not exist that the all distance information should be calculated by the *search collision points table* on-the-fly. With the settings in our experiment, it costs 980 MB memory to materialize the full shortest distance table. The entries in the shortest distance table are ranked by the access frequency during a warm up process. During the warm up process, we issues 1000 kRNN queries and ranks the entries in the table by the number of accesses. Tuning parameter P in the system indicates that top P percents of the *shortest distance table* is materialized.

We evaluate the impact of the tuning parameter with three different parameter settings: (1) different numbers of the data objects (different k values) in the query, (2) different numbers of total data objects in the road network, and (3) different sizes of query regions. For each of the experiment, we evaluate not only the query processing time but also the size of the *answer set* that returns to user as the quality of the query answers.

1) Impact of the number of requested data objects: Figure 5a depicts the query processing time for the different values of the tuning parameter P with different requested number of data objects (k value). As shown in the figure, the larger shortest distance table we have, the less query processing time the algorithm consumes to get the answer set. Figure 6a illustrates the size of the answer set for the different tuning parameters with different k values. Observed

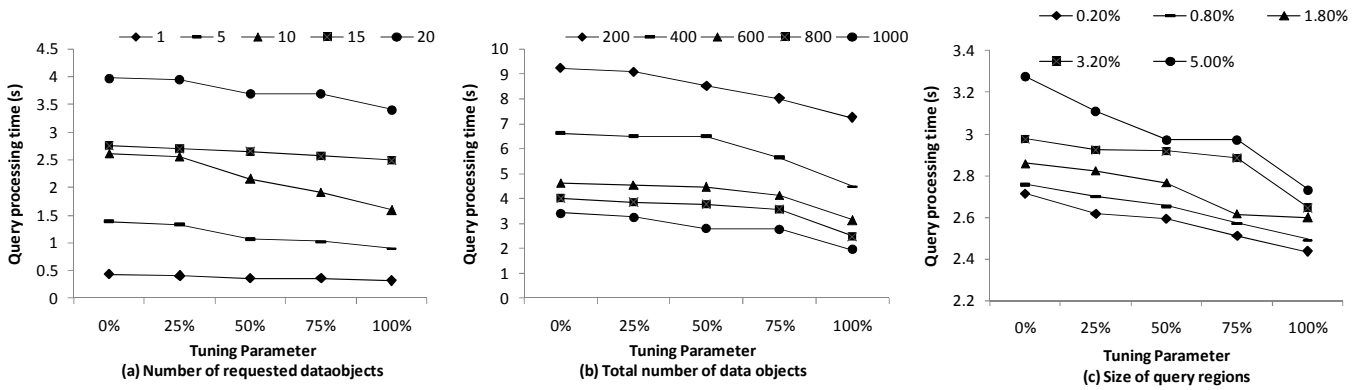

Fig. 5: Query processing time with respect to various tuning parameter values.

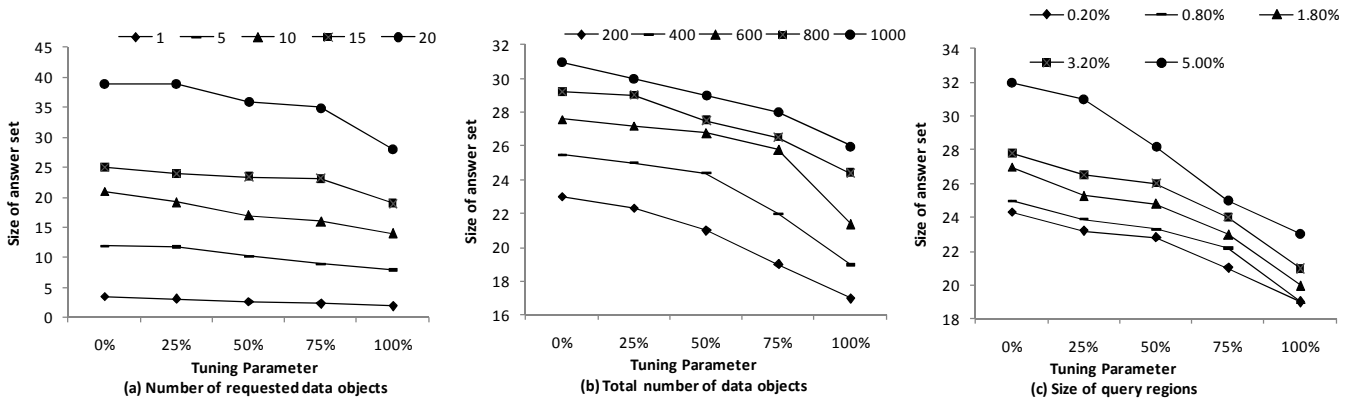

Fig. 6: Answer set size with respect to various tuning parameter values.

from the figure, larger value of the tuning parameter P leads to a smaller size of the answer set for user. The smaller size of answer set means a better quality of query answer.

2) Impact of the number of total data objects: Figure 5b depicts the query processing time for the different values of the tuning parameter P with different total number of data objects over the road network. The data objects are uniformly distributed over the road network in this experiment. As shown in the figure, with a larger tuning parameter, the query processing time decreases for all cases. Figure 6b illustrates the size of the answer set for the different value of tuning parameters with different total number of data objects over the road network. The figure confirms that a larger value of the tuning parameter leads to a smaller size of the answer set and a better quality of answers for the user.

3) Impact of the query region size: Figure 5c depicts the query processing time for the different values of the tuning parameter P with different size of query regions. The relationship of the query region size and the boundary points is illustrated in Table III. As shown in the figure, with the larger *shortest distance table* in our algorithm, the less query processing time our algorithm consumes. Figure 6c illustrates the size of the answer set for the different tuning parameters with different size of query regions. From the figure, it is obvious that the larger value of the tuning parameter leads to a smaller size of the answer set and a better quality of answers for the user.

4) Summary: As shown in the above experimental results, the size of the tuning parameter P provides a tradeoff between the query processing performance (both in the query processing time and the quality of the answer set) and the storage overhead. The larger value of the tuning parameter we choose, the larger *shortest distance table* we need to maintain. However, with the larger *shortest distance table*, the query performance is improved with less query processing time and better quality of the answer set.

D. Impact of different communication bandwidth

Due to the approximate result by our *search collision point table* based calculation method for the shortest distance information, the algorithm is likely to return extra data objects to the user comparing with the optimal solution. We introduce the overall response time to determine the impact of the extra number data objects we return to the user. The query response time is consisted of two parts: (1) the query processing time, and (2) transmission time for the answer set. The overall response time is an end-to-end performance metric to measure the performance of our algorithm. If we return more data objects to the user, the transmission time increases. In this experiment, we use the traditional algorithm as the baseline to compare with the two extreme cases of our algorithm:

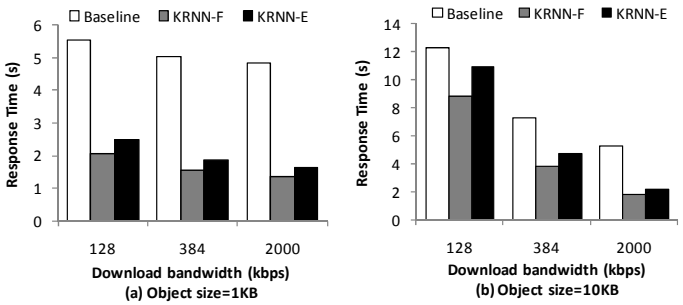

Fig. 7: Object size and bandwidth.

(a) query processing with all the shortest distance information, referred as *KRNN-F*, and (b) query processing without any shortest distance information, referred as *KRNN-E*.

In our experiment, we consider that there are two different sizes of each data object, 1 KB and 10 KB. We assume that the user connects with the service provider through 3G mobile networks. The download bandwidth is different with different user mobility speeds, e.g., 128 kbps (e.g., kilo bits per second) when the user is in a moving vehicle, 384 kbps when the user is walking and 2 Mbps when the user is staying at the same place or moving in a very slow speed.

Figure 7 illustrates the overall response time of our algorithm and the traditional solution with different object sizes and communication speeds. In all the scenarios, although some extra data objects are given in the answer set by our algorithm, the overall response time of our algorithm still outperforms the traditional solution. The closest case in this experiment is in the scenario when the users are moving very fast and they ask for the data objects with a large size. However, the performance of all the test cases is suffered. Because the transmission time of sending the answer set to the user dominates the overall response time, which indicates the system bottleneck is no longer at the query processing time in the database but the transmission time in the communication channel.

VI. Conclusion

In this paper, we proposed an efficient query processing algorithm for k-range nearest neighbor (kRNN) queries in road networks. Our algorithm distinguishes itself from the existing solution for the kRNN query, as it (1) designs a shared execution paradigm to eliminate the redundant searching overhead in the existing solution to improve query processing performance, and (2) introduces a system tuning parameter that controls the amount of space dedicated to store the shortest network distance information for the query processing to achieve a tradeoff between the query processing performance and the storage overhead. We evaluate our algorithm extensively through simulated experiments. The experimental results show that our algorithm outperforms the existing solution in terms of query processing time and overall response time, and the tuning parameter, which specifies the size of the pre-computed shortest network distance table, is an effective way to achieve the tradeoff between the query processing performance and the storage overhead.

References

[1] C.-Y. Chow, M. F. Mokbel, J. Naps, and S. Nath, "Approximate evaluation of range nearest neighbor queries with quality guarantee," in *SSTD*, 2009, pp. 283–301.

[2] B. Zheng, J. Xu, W.-C. Lee, and D. L. Lee, "Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services," *VLDB J.*, vol. 15, no. 1, pp. 21–39, 2006.

[3] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *VLDB*, 2003, pp. 802–813.

[4] U. Demiryurek, F. B. Kashani, and C. Shahabi, "Efficient continuous nearest neighbor query in spatial networks using euclidean restriction," in *SSTD*, 2009, pp. 25–43.

[5] H. Hu, D. L. Lee, and J. Xu, "Fast nearest neighbor search on road networks," in *EDBT*, 2006, pp. 186–203.

[6] V. T. de Almeida and R. H. Güting, "Supporting uncertainty in moving objects in network databases," in *GIS*, 2005, pp. 31–40.

[7] D. Pfoser and C. S. Jensen, "Capturing the uncertainty of moving-object representations," in *SSD*, 1999, pp. 111–132.

[8] M. L. Yiu, N. Mamoulis, X. Dai, Y. Tao, and M. Vaitis, "Efficient evaluation of probabilistic advanced spatial queries on existentially uncertain data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 1, pp. 108–122, 2009.

[9] Y. Cai and T. Xu, "Design, analysis, and implementation of a large-scale real-time location-based information sharing system," in *MobiSys*, 2008, pp. 106–117.

[10] O. Wolfson, H. Cao, H. Lin, G. Trajcevski, F. Zhang, and N. Rishe, "Management of dynamic location information in domino," in *EDBT*, 2002, pp. 769–771.

[11] K. Mouratidis and M. L. Yiu, "Anonymous query processing in road networks," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 2–15, 2010.

[12] W.-S. Ku, Y. Chen, and R. Zimmermann, "Privacy protected spatial query processing for advanced location based services," *Wireless Personal Communications.*, vol. 51, no. 1, pp. 53–65, 2009.

[13] J. Bao, H. Chen, and W.-S. Ku, "Pros: a peer-to-peer system for location privacy protection on road networks," in *GIS*, 2009, pp. 552–553.

[14] T. Wang and L. Liu, "Privacy-aware mobile services over road networks," *PVLDB*, vol. 2, no. 1, pp. 1042–1053, 2009.

[15] M. F. Mokbel, C.-Y. Chow, and W. G. Aref, "The new casper: Query processing for location services without compromising privacy," in *VLDB*, 2006, pp. 763–774.

[16] C.-Y. Chow, M. F. Mokbel, and X. Liu, "A peer-to-peer spatial cloaking algorithm for anonymous location-based service," in *GIS*, 2006, pp. 171–178.

[17] P. Kalnis, G. Ghinita, K. Mouratidis, and D. Papadias, "Preventing location-based identity inference in anonymous spatial queries," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 12, pp. 1719–1733, 2007.

[18] M. R. Kolahdouzan and C. Shahabi, "Voronoi-based k nearest neighbor search for spatial network databases," in *VLDB*, 2004, pp. 840–851.

[19] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases," in *SIGMOD Conference*, 2008, pp. 43–54.

[20] K. C. K. Lee, W.-C. Lee, and B. Zheng, "Fast object search on road networks," in *EDBT*, 2009, pp. 1018–1029.

[21] H. Hu, D. L. Lee, and V. C. S. Lee, "Distance indexing on road networks," in *VLDB*, 2006, pp. 894–905.

[22] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko, "Nearest neighbor queries in road networks," in *GIS*, 2003, pp. 1–8.

[23] M. L. Yiu, N. Mamoulis, and D. Papadias, "Aggregate nearest neighbor queries in road networks," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 6, pp. 820–833, 2005.

[24] J. Sankaranarayanan and H. Samet, "Distance oracles for spatial networks," in *ICDE*, 2009, pp. 652–663.

[25] X. Huang, C. S. Jensen, and S. Saltenis, "Multiple k nearest neighbor query processing in spatial network databases," in *ADBIS*, 2006, pp. 266–281.