GeoInformatica 9:4, 321–341, 2005

Alternative Solutions for Continuous K Nearest Neighbor Queries in Spatial Network Databases

MOHAMMAD R. KOLAHDOUZAN* AND CYRUS SHAHABI
Department of Computer Science, University of Southern California, Los Angeles, CA 90089, USA
E-mail: {kolahdoz, shahabi}@usc.edu

Received December 5, 2004; Revised February 23, 2005; Accepted April 6, 2005

Abstract

Continuous K nearest neighbor queries (C-KNN) are defined as finding the nearest points of interest along an entire path (e.g., finding the three nearest gas stations to a moving car on any point of a pre-specified path). The result of this type of query is a set of intervals (or split points) and their corresponding KNNs, such that the KNNs of all points within each interval are the same. The current studies on C-KNN focus on vector spaces where the distance between two objects is a function of their spatial attributes (e.g., Euclidean distance metric). These studies are not applicable to spatial network databases (SNDB) where the distance between two objects is a function of the network connectivity (e.g., shortest path between two objects). In this paper, we propose two techniques to address C-KNN queries in SNDB: Intersection Examination (IE) and Upper Bound Algorithm (UBA). With IE, we first find the KNNs of all nodes on a path and then, for those adjacent nodes whose nearest neighbors are different, we find the intermediate *split points*. Finally, we compute the KNNs of the split points using the KNNs of the surrounding nodes. The intuition behind UBA is that the performance of IE can be improved by determining the adjacent nodes that cannot have any split points in between, and consequently eliminating the computation of KNN queries for those nodes. Our empirical experiments show that the UBA approach outperforms IE, specially when the points of interest are sparsely distributed in the network.

Keywords: continuous nearest neighbor queries, spatial network databases

1. Introduction

The problem of K nearest neighbor (KNN) queries in spatial databases has been studied by many researchers. This type of query is frequently used in Geographical Information Systems (GIS) and is defined as: given a set of spatial objects and a query point, find the K closest objects to the query. An example of KNN query is a query initiated by a GPS device in a vehicle to find the five closest restaurants to the vehicle. Different variations of KNN queries are also introduced. One variation is the *continuous KNN* query which is defined as the KNNs of *any* point on a given path (i.e., continuous with respect to space). An example of continuous KNN is when the GPS device of the vehicle initiates a query to find the five closest restaurants to the vehicle at any point of a given path from a source to a destination. The result of this type of query is a set of intervals, or a set of

* Corresponding author.

split points, and their associated KNNs. The split points specify where on the path the KNNs of a moving object will change, and the intervals (bounded by the split points) specify the locations that the KNNs of a moving object remains the same. The challenge in this type of query is to efficiently specify the location and the KNNs of the split points (or intervals).

The majority of the existing work on KNN queries and its variations are aimed at Euclidean spaces, where the path between two objects is the straight line connecting them. These approaches are usually based on utilizing index structures. However, in spatial network databases (SNDB), objects are restricted to move on pre-defined paths (e.g., roads) that are specified by an underlying network. This means that the shortest network path/distance between objects (e.g., a vehicle and the restaurants) depends on the connectivity of the network rather than the objects' coordinates. Index structures that are designed for spaces where the distance between objects is only a function of their spatial attributes (e.g., Euclidean distance), cannot properly approximate the distances in SNDB and hence, the solutions that are based on index structures cannot be extended to SNDB.

In our previous work [5], we introduced a Voronoi based approach, VN^3, to efficiently address regular KNN queries in SNDB. The VN^3 has two major components, network Voronoi polygons (NVP) for each point of interest, and the pre-computed distances between the border points of each polygon to the points inside the polygon. The VN^3 approach provides the result set in an incremental manner and it works in two steps: the filter step uses the first component to generate a candidate set, and the pre-computed component is used in the refinement step to find the distances between the query objects and the candidates, and hence refine the candidates.

In this paper, we propose different approaches to address continuous KNN queries in SNDB. Depending on the number of neighbors requested by a C-KNN query, we divide the problem into two cases. When only the first nearest neighbor is requested (e.g., finding only the closest restaurant to a vehicle on a given path), our solution relies entirely on the properties of VN^3. We show that the split points on the path are simply the intersections of the path with the NVPs of the network, which are a subset of the border points of the NVPs.

We propose two solutions for the cases when more than one neighbor is requested by the continuous KNN query (i.e., $K > 1$). The main idea behind our first approach is that the KNNs of any object on a path between two adjacent nodes (e.g., intersections in road system) can only be a subset of any point(s) of interest (e.g., restaurants) on the path, plus the KNNs of the end nodes. Hence, the first solution is based on breaking the entire path to smaller segments, where each segment is surrounded by two adjacent nodes, and then finding the KNNs of all the segment nodes. We then show that for two adjacent nodes with different KNNs, by specifying whether the distances from a query object to the KNNs of the nodes will be increasing or decreasing as the object moves, we can find the location of the split points between the two nodes.

The intuition behind our second solution is that if an object moves slightly, its KNNs will probably remain the same. Our second approach is then based on finding the minimum distance between two subsequent nearest neighbors of an object, only when the two

neighbors can have a split point between them. This distance specifies the minimum distance that the object can move without requiring the submission of a new KNN query. Our empirical experiments show that the second approach always outperforms the first solution. To the best of our knowledge, the problem of continuous K nearest neighbors in spatial network databases has not been studied before.

The remainder of this paper is organized as follows. We review the related work on regular and continuous nearest neighbor queries in Section 2. We then provide a review of our VN^3 approach that can efficiently address KNN queries in SNDB in Section 3. In Section 4, we discuss our approaches to address continuous KNN queries. Finally, we discuss our experimental results and conclusions in Sections 5 and 6, respectively.

2. Related work

The regular K nearest neighbor queries have been extensively studied and for which numerous algorithms have been proposed. A majority of the algorithms are aimed at *m*-dimensional objects in Euclidean spaces and are based on utilizing one of the variations of multidimensional index structures. There are also other algorithms that are based on computation of the distance from a query object to its nearest neighbors on-line and per query. The regular KNN queries are the basis for several variations of KNNs, e.g., continuous KNN queries. The solutions proposed for regular KNN queries are either directly used, or have been adapted to address the variations of KNN queries. In this section, we review the previous solutions for regular and continuous KNN queries.

The regular KNN algorithms that are based on index structures usually perform in two filter and refinement steps and their performance depends on their selectivity in the filter step. Roussopoulos et al. [10] present a branch-and-bound R-tree traversal algorithm to find the nearest neighbors of a query point. The main disadvantage of this approach is the depth-first traversal of the index that incurs unnecessary disk accesses. Korn et al. [6] present a multi-step *k*-nearest neighbor search algorithm. The disadvantage of this approach is that the number of candidates obtained in the filter step is usually much more than necessary, making the refinement step very expensive. Seidl and Kriegel [11] propose an optimal version of this multi-step algorithm by incrementally ranking queries on the index structure. Hjaltason and Samet [2] propose an incremental nearest neighbor algorithm that is based on utilizing an index structure and a priority queue. Their approach is optimal with respect to the structure of the spatial index but not with respect to the nearest neighbor problem. The major shortage with all these approaches that render them impractical for networks is that the filter step of these solutions performs based on Minkowski distance metrics (e.g., Euclidean distance) while the networks are metric spaces, i.e., the distance between two objects depends on the connectivity of the objects and not their spatial attributes. Hence, the filter step of these approaches cannot be used for, or properly approximate exact distances in networks.

Papadias et al. [9] propose a solution for SNDB which is based on generating a search region for the query point that expands from the query, which performs similar to Dijkstra's algorithm. Shekhar and Yoo [12] and Jensen et al. [4] also propose solutions

for the KNN queries in SNDB. These solutions are based on computing the distance between a query object and its candidate neighbors on-line and per query. Finally, [5], we propose a novel approach to efficiently address KNN queries in SNDB. The solution is based on the first order network Voronoi diagrams and the result set is generated incrementally.

Sistla et al. [13] first identified the importance of the continuous nearest neighbors and described modeling methods and query languages for the expression of these queries, but did not discuss the processing methods. Song and Roussopoulos [14] proposed the first algorithms for CNN queries. Their solution utilizes a fixed upper bound that specifies the minimum distance that an object can move without requiring a new KNN to be issued. They also proposed a dual buffer search method that can be used when the position of the query can be predicted. Tao and Papadias [15], [16] presented a solution that is based on the concept of time parameterized queries. The output of this approach specifies the current result of the CNN query, the expiration period of the result, and the set of objects that will effect the results after the expiration period. This approach provides the complete result set in an incremental manner. Tao and Papadias [15], [16] proposed a solution for CNN queries based on performing one single query for the entire path. They also extended the approach to address C-KNN queries. The main shortcoming of all of these approaches is that they are designed for Euclidean spaces and utilize a spatial index structure, hence they are not appropriate for SNDB.

Iwerks et al. [3] propose a method to maintain the continuous KNN queries for moving objects when the updates are allowed and the motion of the objects is represented as a function of time. Their solution is based on substituting a continuous KNN query with a window query, where the objects that are within a specific distance from the object are filtered. This solution is appropriate when the motion of the objects can be expressed as a function. Li et al. [7] describe a method to continuously monitor the nearest neighbors in the mobile environment by examining only the K-th and $(k+1)$-st nearest neighbor. They propose considering the moving objects in a time-distance space, where each object is represented by a curve, and hence reduce the problem to simply monitoring the location (in time) where the distance of the $(k+1)$-st neighbor becomes less than the distance of the k-th neighbor. The shortcoming of this approach is that it cannot detect when the order of the first K nearest neighbors are changed. Xiong et al. [17] propose a solution to continuously answer a collection of continuous KNN queries based on incremental evaluation and shared execution. As opposed to the solution provided in Iwerks et al. [3], this solution does not make any assumption about the trajectory of the movement of the object(s). However, the solution is based on utilizing methods that are based on R-tree index structures and hence, cannot be applied to spatial network databases.

Finally, Feng and Watanabe [1] provided a solution for C-NN queries in road networks. Their solution is based on finding the locations on a path that a NN query must be performed at. The main shortcoming of this approach is that it only addresses the problem when the first nearest neighbor is requested (i.e., continuous 1-NN) and does not address the problem for continuous K-NN queries. To the best of our knowledge, the problem of continuous K nearest neighbor queries in spatial network database has not been studied.

3. Background: VN^3

Our proposed solutions to address continuous KNN queries utilize the VN^3 approach [5] to efficiently find the KNNs of an object. The VN^3 approach is based on the concept of the *Voronoi diagrams*. In this section, we start with an overview of the principles of the network Voronoi diagrams. We then discuss our VN^3 approach to address KNN queries in spatial network databases. A thorough discussion on Voronoi diagrams and VN^3 are presented in Okabe et al. [8] and Kolahdouzan et al. [5], respectively.

3.1. Network Voronoi diagram

Consider a set of limited number of points, called *generator points* (or points of interest), in the Euclidean plane. We associate all locations in the plane to their closest generator(s). The set of locations assigned to each generator forms a region called *Voronoi polygon* (VP) of that generator. The set of Voronoi polygons associated with all the generators is called the *Voronoi diagram* with respect to the generators set. Consequently, a Voronoi diagram divides a space into disjoint polygons where the nearest neighbor of any point inside a polygon is the generator of the polygon. The Voronoi polygons that share the same edges are called *adjacent polygons*.

A network Voronoi diagram, termed *NVD*, is a specialization of the Voronoi diagrams and is defined for directed and undirected weighted non-planar graphs (e.g., road networks), where the location of objects (e.g., cars) is restricted to be on the links (e.g., streets) that connect the nodes (e.g., intersections) of the graph and the distance between objects is defined as their shortest path in the network rather than their Euclidean distance. A network Voronoi polygon *NV P* for the point of interest p_i (i.e., *NV P*(p_i)) contains (portion of) the links where p_i is the closest point of interest to any point on those links. Note that the points of interest in a spatial network database can only be located on the links of the network. A *border point* of two NVPs is defined as the location where a link crosses from one NVP to another. The edges of the NVPs are generated by connecting the adjacent border points. Unlike a regular Voronoi polygon where edges are straight lines, the edges of an NVP are not straight lines and hence, an NVP usually has a complex shape. However, we can still generate an R-tree index structure for these polygons and utilize the index to locate the polygon that contains a query object. Figure 1 shows an example of a network Voronoi diagram where $\{p_1, p_2, p_3\}$ are the points of interest (or generators), $\{p_3, \ldots, p_{16}\}$ are the nodes of the network, the thick solid lines depict the edges of the NVPs, and $\{b_1, \ldots, b_7\}$ indicate the border points of the NVPs.

Two of the basic properties of the (network) Voronoi diagrams that we exploit in our VN^3 approach are:

Property 1: The Voronoi diagram of a point set P, $V\,D(P)$, is unique.

Property 2: The nearest generator point of p_i (e.g., p_j) is among the generator points whose Voronoi polygons share similar Voronoi edges with $V\,P(p_i)$.

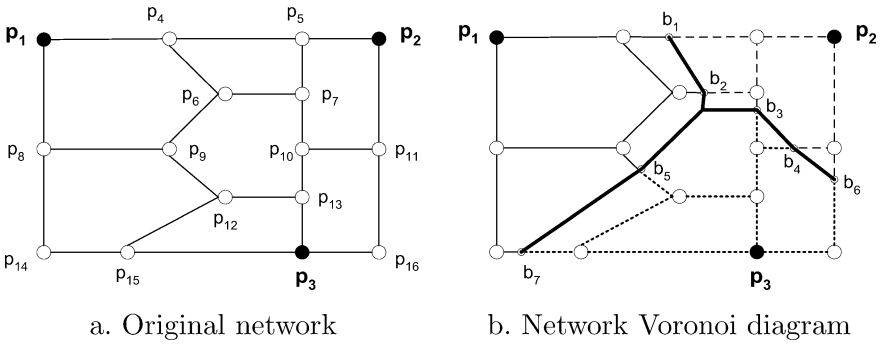

Figure 1. Example of a network Voronoi diagram (NVD).

3.2. *Voronoi-based network nearest neighbor: VN³*

Our proposed approach to find the K nearest neighbor queries in spatial networks [5], termed VN³, is based on the properties of the Network Voronoi diagrams and also *localized* pre-computation of the network distances for a very small percentage of neighboring nodes in the network. The intuition is that the NVPs of an NVD can directly be used to find the first nearest neighbor of a query object q. This can be achieved by utilizing an R-tree index structure generated for the NVPs to locate the polygon that contains q. Subsequently, NVPs' adjacency information can be utilized to provide a candidate set for other nearest neighbors of q. Finally, the pre-computed distances can be used to compute the actual network distances from q to the generators in the candidate set and consequently refine the set. This component of VN³ can also be used to find the actual shortest path from q to its nearest neighbors. The filter/refinement process in VN³ is iterative: at each step, first a new set of candidates is generated from the adjacent generators of the generators that are already selected as the nearest neighbors of q, then the pre-computed distances are used to select "only the next" nearest neighbor of q. VN³ consists of the following major components:

1. Pre-calculation of the solution space: As a major component of the VN³ filter step, the NVD for the points of interest (e.g., hotels, restaurants, ...) in a network must be calculated and its corresponding NVPs must be stored in a table. Note that separate NVDs and set of NVPs must be generated for different types of points of interest.
2. Utilization of an index structure: In the first stage of the filter step, the first nearest neighbor of q is found by locating the NVP that contains q (e.g., using *Contain(q)* function in spatial databases). This stage can be expedited by using a spatial index structure generated on the NVPs. Note that although an NVD is based on the network distance metric, its NVPs are regular polygons and can be indexed using index structures that are designed for the Euclidean distance metric (e.g., R-tree). This means that the *Contain(q)* function invoked on an R-tree index structure generated for NVPs will return the NVP whose generator has the minimum network distance to q.

3. Pre-computation of the exact distances for a very small portion of data: The refinement step of VN^3 requires that for each NVP, the network distances between its border points be pre-computed and stored. These pre-computed distances are used to find the network distances across NVPs, and from the query object to the candidate set generated by the filter step.

3.3. *VN^3 filter step*

The filter step of our VN^3 approach is based on the following two properties. These properties help the filter step to limit its search space to only the adjacent Voronoi polygons.

Property 3: Property 2 (in Section 3.1) suggests that the second nearest generator to "*any location*" inside a Voronoi polygon $V(p_i)$ is among the adjacent generators of p_i.

Property 4: Let $G = \{g_1, \ldots, g_k\} \in P$ be the set of the first k nearest generators of a location q inside $V(g_1)$, then g_k is among the adjacent generators of $\{G \setminus g_k\}$.

The proofs of the above properties are described in Kolahdunzan et al. [5].

Using a hypothetical NVD shown in Figure 2, we now describe the filtering step of VN^3. The figure shows a query point q, a set of generators $(P_1, \ldots, P_{13})$ and their network Voronoi polygons, the border points of the polygons, and the network inside $NV\,P(P_1)$. The definition of NVD requires that the first nearest neighbor of q be P_1 since $V(P_1)$ contains q, hence P_1 can be found by issuing the *Contain(q)* function on an index structure generated for the NVPs of the NVD. Property 3 suggests that the second nearest neighbor of q is among the adjacent generators of P_1, i.e., $C = \{P_2, P_3, P_4, P_5, P_6\}$. This adjacency information is determined and stored during the generation of NVD and can be efficiently retrieved from a lookup table. At this stage, we invoke the refinement step of VN^3 to compute the exact distances between q and all the generators in C to find the second nearest neighbor. Let us assume that the second nearest neighbor of q is P_3. Property 4 then requires that the third nearest neighbor of q be among the adjacent

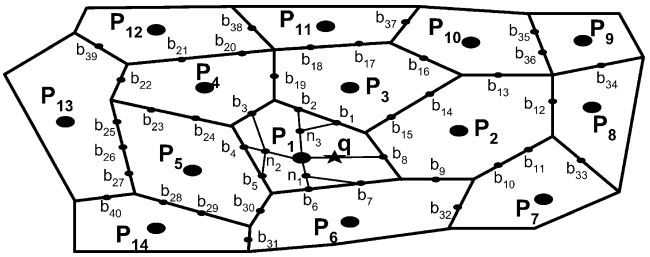

Figure 2. A sample network Voronoi diagram.

generators of $\{P_1, P_3\}$, i.e., $C = \{P_2, P_4, P_5, P_6, P_{10}, P_{11}, P_{12}\}$. Consecutive nearest neighbors of q can then be found using the same iterative approach.

3.4. VN^3 refinement step

Once a nearest neighbor of a query point q is found and the candidate set C is updated, the distances from q to all the elements of C must be computed in order to find the next nearest neighbor. We propose alternative approaches to find the distances between q and the elements of C. These approaches are based on the properties of NVPs and the intuition behind all of them is that in an NVD, all possible paths that can connect an object from outside an NVP to a node inside it, including the polygon's generator, must pass through the border points of the polygon. Hence, our proposed approaches are based on calculating the distance from q to c, a member of the candidate set C, in three steps:

1. From q to the border points of the polygon that contains q (e.g., P_1): we propose two solutions for this step. The first solution is similar to the method proposed in Papadias et al. [9] where by expanding the network around q, we calculate the distances from q to the border points of P_1. The second solution is based on precalculating and storing the distances from the border points of each polygon to all the points (i.e., network nodes) of that polygon in an off-line process. The advantage of this approach is the significant boost in performance by replacing execution of a complex algorithm (e.g., Dijkstra) by a series of simple table lookups.
2. From the border points of p_1 to the border points of c: we also propose two solutions to find these distances. Our first solution works as follows. We assume a network that is made of the border points of the polygons in C. Since the distances from each border point of a polygon to the border points of the same polygon are computed and stored (i.e., third component of VN^3), we can easily compute the distance from one border point to all the border points of the polygons in C. Since the number of border points is a small percentage of the number of points in the network, this can be efficiently achieved in memory using Dijkstra's algorithm. Our second approach improves the performance of the first approach by only performing Dijkstra for a small subset of the border points.
3. From the border points of c to c: these distances are computed during the generation of the NVD and are stored in, and can be efficiently retrieved from a lookup table.

Our empirical experiments shows that VN^3 outperforms the only other proposed approach [9] for KNN queries in SNDB by up to one order of magnitude.

4. Continuous nearest neighbor queries

Continuous nearest neighbor queries are defined as determining the K nearest neighbors of an object on any point of a given path (i.e., continuous with respect to the space). An

example of this type of query is shown in Figure 3 where a moving object (e.g., a car) is traveling along the path (*A*, *B*, *C*, *D*) (specified by the dashed lines) and we are interested in finding the first 3 closest restaurants (restaurants are specified in the figure by $\{r_1, \ldots, r_8\}$) to the object at any given point on the path. The result of a continuous NN query is a set of *split points* and their associated KNNs. The split points specify the locations on the path where the KNNs of the object change. In other words, the KNNs of any object on the segment (or interval) between two adjacent split points is the same, and it is identical to the KNNs of the split points. Note that on a split point, the (K+1)-th NN is identical to the K-th NN (i.e., the definition of switching KNNs). Hence, the KNNs of a split point is identical to the KNNs of any point in its adjacent segments. The challenge for this type of query is to efficiently find the location of the split point(s) on the path and their corresponding KNNs. The current studies on continuous NN queries are focused on spaces where the distance function between two objects is one of the Minkowski distance metrics (e.g., Euclidean). However, the distance function in spatial network databases (e.g., a road network) is usually defined as their shortest path (or shortest time) which is a function of the network connectivity and is computationally complex. This renders the approaches that are designed for Minkowski distance metrics, or the ones that are based on utilization of vector or metric spatial index structures, impractical for SNDB.

In this section, we discuss our solutions for C-KNN queries in spatial network databases.We first present our solutions for the scenarios when only the first NN is desired (i.e., C-NN), and then discuss two solutions for the cases where the KNN of any point on a given path are requested.

4.1. Continuous 1-NN queries

Our solution for C-NN queries, when only the first nearest neighbor is requested, is based on the properties of VN^3. As we showed in Section 3, a network Voronoi polygon of a point of interest p_i specifies all the locations in space (space is limited to the roads in SNDB), where p_i is their nearest neighbor. Hence, in order to specify the C-NN of a given path, we can first specify the intersections of the path with the NVPs of the network. This can be achieved by performing a simple *Intersect*() function on an R-tree

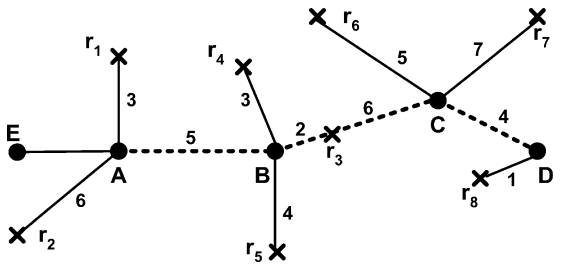

Figure 3. Example of continuous K nearest neighbor query.

index structure that is generated for the NVPs. Subsequently, we can conclude that p_i is the C-NN of the segments of the path that are contained in $NV\ P(p_i)$. Note that this approach cannot be extended to C-KNN queries since the network Voronoi polygons are first order NVPs, i.e., they can only specify the first nearest neighbor of an object.

For example, assume the network Voronoi diagram shown in Figure 4 where $\{p_1, \ldots, p_7\}$ are the points of interest. As depicted in the figure, the path from S to D crosses $NV\ P(p_1)$, $NV\ P(p_5)$ and $NV\ P(p_7)$ and can be divided to 4 segments. We can conclude that p_5, p_1, and p_7 are the first nearest neighbors of any point on segments {1, 3}, {2}, and {4}, respectively.

4.2. Continuous KNN queries

In this section, we discuss two approaches to address continuous KNN queries where both approaches are aimed to find the location of the split point(s) on a given path. Our first solution, IE, is based on examining the KNNs of all the nodes on the path, while our second approach, UBA, eliminates the KNN computation for some of the nodes that cannot have any split points in between.

4.2.1. Intersection examination: IE. Our first approach to address continuous KNN queries in SNDB is based on finding the KNNs of the intersections on the path. We describe the intuition of our IE approach by defining the following properties:

Property 5: Let $p(n_i, n_j)$ be the path between two adjacent nodes n_i and n_j, o_1 and o_1' be the first nearest points of interest (or neighbors) to n_i and n_j, respectively, and assume that $p(n_i, n_j)$ includes no point of interest, then the first nearest point of interest to "*any*" object on $p(n_i, n_j)$ is either o_1 or o_1'.

Proof: This property can be easily proved by contradiction. Assume that the nearest point of interest to a query object q on $p(n_i, n_j)$ is $o_k \notin \{o_1, o_1'\}$. We know that the shortest path from q to o_k, $p(q, o_k)$, must go through either n_i or n_j. Suppose $p(q, o_k)$ goes through

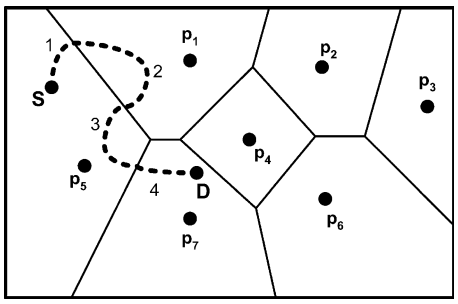

Figure 4. Continuous first nearest neighbor.

n_i and hence *distance* (q, o_k) = *distance* (q, n_i) + *distance* (n_i, o_k). However, we know that *distance* (n_i, ok) > *distance* (n_i, o_1) since o_1 is the first nearest point of interest to n_i and hence, its distance to n_i is smaller than the distance of any other point of interest to n_i. Subsequently, we can conclude that *distance* (q, ok) > *distance* (q, o_1) which means that o_1 is closer to q than ok, contradicting our initial assumption. □

As an example, this property suggests that in Figure 3, the first nearest restaurant to any point between A and B can be either r_1 or r_3 since r_1 and r_3 are the first nearest neighbors of A and B, respectively.

Property 6: Let $p(n_i, n_j)$ be the path between two adjacent nodes n_i and n_j, $O = \{o_1, \ldots, o_k\}$ and $O' = \{o'_1, \ldots, o'_k\}$ be the k nearest points of interest to n_i and n_j, respectively, and assume that $p(n_i, n_j)$ includes no point of interest, then the k nearest points of interest to "*any*" object on $p(n_i, n_j)$ is a subset of $\{O \cup O'\}$.

Proof: This property is in fact the extension of property 5 and can be similarly proved by contradiction. □

As an example, this property suggests that in Figure 3, since the three nearest restaurants to A and B are $\{r_1, r_2, r_3\}$ and $\{r_3, r_4, r_5\}$, respectively, the three nearest restaurants to any object between A and B can only be among $\{r_1, r_2, r_3, r_4, r_5\}$.

From the above properties, we can conclude that:

- If two adjacent nodes have similar KNNs, every object on the path between these nodes will have similar KNNs as the nodes, meaning that there will be no split points on the shortest path between the nodes. That is simply because O and O' (in property 6) are the same and hence, $\{O \cup O'\} = O = O'$.
- In order to find the continuous KNN of any point on a path, we can first break the path into smaller segments where each segment obeys the above properties. We then find the continuous KNN for each segment and finally, the union of the results for all segments generates the result set for the entire path.

The above properties are not valid for a path p if p includes one or more points of interest (e.g., the path between B and C in Figure 3 which includes r_3). We can address this issue by the following two alternative approaches.

1. The first approach is to further break p to smaller segments where each segment does not include any points of interest. For example, we can break the path (B, C) of figure 3 to (B, r_3) and (r_3, C). This will require that in addition to B and C, the KNNs of r_3 be determined, which incurs additional overhead. However, in the real world data sets, the points of interest usually constitute a very small percentage of the nodes in the graph (e.g., in the State of California, restaurants that have a density of less than 5%, are the points of interest with the maximum density). Hence, the incurred overhead by this approach is usually negligible. Note that if

the percentage of the points of interest are unusually high, VN^3 performs substantially faster, as the result of our experiments [5] show that VN^3 performs more efficiently for higher densities of the points of interest, and hence, will compensate for this overhead.

2. The second solution is based on extending Properties 1 and 2 as: we can easily prove that by including points of interest that are on p in the candidate set, the above properties will again become valid for p. For example, this solution suggests that the three closest restaurant to any point on the path (B, C) is a subset of $\{r_3 \cup \{r_3, r_4, r_5\} \cup \{r_6, r_8, r_3\}\}$.

For the rest of this paper, we use the first solution.

Once the KNNs of the nodes in the network are specified, we need to find the location and the KNNs of the split points on each segment (i.e., path between two adjacent nodes). Note that split point(s) only exist on the segments where the nodes of that segment have different KNNs. We divide the KNNs of a node n_i to two increasing and decreasing groups: the NNs that their distances to a query object q, which is traveling from n_i in a specific direction, increases as the distance of q and n_i increases, are called increasing NNs and vice versa. Note that whether a NN is increasing or decreasing depends on the direction that q is traveling, i.e., the increasing NNs become decreasing if q travels in the opposite direction (i.e., q travels toward n_i). We can specify whether a point of interest is considered as increasing or decreasing NN using the following property:

Property 7: Let n_i and n_j be two adjacent nodes, $d(x, y)$ specifies the length of the shortest path between objects x and y, and $O = \{o_1, \ldots, o_k\}$ be the set of points of interest in the network, then the shortest path from n_i to $o_a \in O$ goes through n_j if and only if $d(n_i, o_a) = d(n_i, n_j) + d(n_j, o_a)$.

Proof: This property is self-evident and we omit its proof. □

We now formally define increasing/decreasing NNs as:

Definition: A point of interest o is called an increasing NN for an object that travels on direction $n_i \rightarrow n_j$ if the shortest path from n_i to o does not pass through n_j, and it is called decreasing otherwise.

An example of the above definition and property suggests that in Figure 3, r_3 is considered as a decreasing NN for a query object (i.e., its distance to the query decreases) when the query is traveling from A toward B. This is because the shortest path from A to r_3 passes from B. However, r_3 is considered as an increasing NN for the same query (i.e., its distance to the query increases) when it travels from B toward A.

We can now describe our approach to find the location of the split points between two nodes, and their KNNs, using the following example: suppose that in Figure 3, we are interested to find the three closest restaurants to a query object that is travelling on the

path (A, B, C, D) (i.e., we should find the KNNs of any point on the path) (A, B, C, D)). We follow the following steps:

Step 1: The first step is to break the original path (A, B, C, D) to smaller segments where each segment does not include any point of interest except when a point of interest is a vertex of the segment (i.e., the segment obeys Properties 1 and 2). For the given example, the resulting segments will be (A, B), (B, r_3), (r_3, C) and (C, D).

Step 2: Once the segments are specified, the KNNs of the nodes of each segment must be determined. We use VN^3 approach to efficiently find the KNNs of the nodes. In addition, VN^3 can efficiently find the shortest path between the query and its nearest neighbors and hence, can specify if a NN is an increasing or decreasing NN. To illustrate our technique, we focus on the first segment (i.e., (A, B)). The other subsequent segments are treated similarly. The three nearest restaurants to A and B and their distances are $\{(r_1, 3)(r_2, 6)(r_3, 7)\}$ and $\{(r_3, 2)(r_4, 3)(r_5, 4)\}$, respectively. Since the set of KNNs and A and B are different, we can conclude that (some) split point(s) must exist between A and B. We also conclude that the KNNs of any point on the candidate list $\{r_1, r_2, r_3, r_4, r_5\}$.

Step 3: From the above candidate list, we can easily generate a sorted list of the nearest neighbors for the starting point of the segment, A. We also specify whether each NN is an increasing or decreasing NN using $\uparrow$ and $\downarrow$ symbols, respectively. for the given example, the sorted candidate list for A is $\{\uparrow (r_1, 3), \uparrow (r_2, 6), \downarrow(r_3, 7), \downarrow(r_4, 8), \downarrow(r_5, 9)\}$.

Step 4: We now specify the location of the first split point by:

1. We find the location of the split point for any two subsequent members of the sorted list, o_i and o_{i+1}, where the first and second members have increasing and decreasing distances to A, respectively. The distance of the split point for o_i and o_{i+1} to A can be easily found as: $\frac{d(A,o_{i+1})+d(A,o_i)}{2} - d(A, o_i) = \frac{d(A,o_{i+1})-d(A,o_i)}{2}$. Note that since o_{i+1} is lower that o_i on the sorted candidate list, $d(A, o_{i+1})$ is always greater than $d(A, o_i)$, meaning that the location of the split point is always between A and B.
2. Among the set of possible split points found in step 1, we select the one that has the minimum distance to A as the first split point.
   For the given example, the only two subsequent members of the candidate list that satisfy the condition stated in step 1 are ($\uparrow$ r_2) and ($\downarrow$ r_3) with split point $p_1 = \frac{6+7}{2} = 6.5$ and a distance of (6.5−6=)0.5 to A. Note that we ignore other combinations of any two subsequent members of the sorted list because: a) if two subsequent members both have increasing (or decreasing) distances to A (e.g., $\uparrow(r_1, 3)$ and $\uparrow(r_2, 6)$, or $\downarrow(r_3, 7)$ and $\downarrow(r_4, 8)$), the differences between their distances to a query object moving from A to B will remain constant, meaning that there cannot be any split point(s) between them, and b) if the first member has a decreasing and the second member has an increasing distance to A, when the query object is traveling from A to B, the distance of the query to the first member will be decreased further and the distance to the second member will be increased further, which again means that there can be no split points between the members.

Step 5: We can easily update the sorted candidate list to reflect their distances to the first split point p_1 by adding/subtracting the distance of A and $p1$ to/from the members that have increasing/decreasing distances to A. The sorted candidate list for p_1 will then become $\{\uparrow (r_1, 3.5), \downarrow (r_3, 6.5), \uparrow (r_2, 6.5), \downarrow (r_4, 7.5), \downarrow (r_5, 8.5)\}$.

Step 6: We now treat p_1 as the beginning node of a new segment, (p_1, B), and repeat steps 4 to 6: we first determine the split points for $(\uparrow r_1, \downarrow r_3)$ and $(\uparrow r_2, \downarrow r_4)$ pairs as $np_1 = \frac{3.5+6.5}{2} = 5$ and $np_2 = \frac{6.5+7.5}{2} = 7$ since these are the only combinations of the subsequent neighbors that are increasing and decreasing, respectively. We then find their distances to A as $d(np_1, p_1) = 5 - 3.5 = 1.5$ and $d(np_2, p_1) = 7 - 6.5 = 0.5$, and finally select np_2 as the next split point p_2 since it is closer to A than np_1. We continue executing steps 4 to 6 until the new split point has similar KNNs as B.

Table 1 shows the results of repeating the above steps for the segment (A, B): the KNNs of any point on (A, p_1) interval is equal to the KNNs of A (and p_1), for any point on (p_1, p_2) segment is equal to KNNs of p_1 (and p_2), and so on. Note that the distance from a query object, which is between two split points, to its KNNs can be similarly computed. Subsequently, the results for other segments of the path can be similarly found.

This approach, although provides a precise result set, is conservative and may lead to unnecessary execution of KNN queries. For example, suppose that we are interested in the first NN of any point on the traveling path from S to D in Figure 4. Clearly, there are only three split points on this path. However, if we utilize IE approach to address this query, the 1-NN query must be issued for all the intersections of the path, resulting in unnecessary KNN computations. We address this issue with our second approach.

4.2.2. Upper Bound Algorithm: UBA. While IE performs KNN queries for every node (e.g., intersection) on the path, the UBA approach only performs the computation of KNN queries for a subset of the nodes of the path and hence, provides a better performance by reducing the number of KNN computations. The intuition for UBA is similar to what is discussed in Song and Roussopoulos [14]: when a query object is moved only slightly, it is very likely that its KNNs remain the same. Song and Roussopoulos [14]

Table 1. Split points for segment (A, B) of Figure 3.

Split point	*Distance to A*	*Candidate set*
p_1	0.5	$\uparrow (r_1, 3.5), \downarrow (r_3, 6.5), \uparrow (r_2, 6.5), \downarrow (r_4, 7.5), \downarrow (r_5, 8.5)$
p_2	1.0	$\uparrow (r_1, 4), \downarrow (r_3, 6), \downarrow (r_4, 7), \uparrow (r_2, 7), \downarrow (r_5, 8)$
p_3	1.5	$\uparrow (r_1, 4.5), \downarrow (r_3, 5.5), \downarrow (r_4, 6.5), \downarrow (r_5, 7.5), \uparrow (r_2, 7.5)$
p_4	2.0	$\downarrow (r_3, 5), \uparrow (r_1, 5), \downarrow (r_4, 6), \downarrow (r_5, 7), \uparrow (r_2, 8)$
p_5	2.5	$\downarrow (r_3, 4.5), \downarrow (r_4, 5.5), \uparrow (r_1, 5.5), \downarrow (r_5, 6.5), \uparrow (r_2, 8.5)$
p_6	3	$\downarrow (r_3, 4), \downarrow (r_4, 5), \downarrow (r_5, 6), \uparrow (r_1, 6), \uparrow (r_2, 9)$

define a threshold value as $\delta = \min (d(o_{i+1}, q) - d(o_j, q))$ where q is the query object and $o_{i+1} \in (K+1)$ *NNs* (q). The defined δ is the minimum difference between the distances of any two subsequent KNNs of q. It can be shown that if the movement of q is less than $\frac{\delta}{2}$, the KNNs of q will remain the same. This approach is designed for Euclidean spaces but we apply it to spatial network databases. In addition, we propose a less conservative bound, δ', which further improves the performance of our approach.

We first discuss the extension of the approach described in Song and Roussopoulos [14] to SNDB using the example in Figure 3. Let us assume that a query object q is traveling from D toward C and we are interested in finding the three closest restaurants to q. The above approach suggests to first find the four closest restaurants to D, $\{(r_8, 1), (r_6, 9), (r_3, 10), (r_7, 11)\}$. The value of δ is then computed ($\delta = 1$), and finally it is concluded that while the distance of q and D is less than or equal to $(\frac{\delta}{2} =)0.5$, the 3NNs of q are the same as the 3NNs of D. The next $(3+1)$NN query must then be issued at the point where the distance of q and D becomes 0.5.

As we discussed in Section 4.2.1, depending on the traveling path of a query object q, its KNNs can be divided to two increasing and decreasing groups. We showed that if two subsequent members of the candidate list are both increasing or decreasing, or if the first one is decreasing and the second one is increasing, they cannot generate any split points on the path. This property is in fact the basis of our UBA algorithm.

We define the new threshold value as $\delta' = \min (d(o_{i+1}, q) - d(o_i, q))$ where q is the query object, $o_{i+1} \in (K+1)$*NNs* (q), and o_i and o_{i+1} have increasing and decreasing distances to q, respectively. The reason for this is similar to the discussion presented in the step 4 of Section 4.2.1. Our defined δ' specifies the minimum difference between the distances of *only* the NNs that can generate a split point on the traveling path. If the movement of q is less than $\frac{\delta'}{2}$, the KNNs of q will remain the same. Otherwise, a new (K+1)NN query must only be issued at the point that is immediately before the point that has a distance of $\frac{\delta'}{2}$ to the initial location of q. For example, in Figure 3, if the traveling path is (D, C, B, A) and the point specified by some δ' is between C and B or between D and C, a new (K+1)NN query must be issued at point C and then the split points between C and D are specified similar to IE. In this case, UBA will perform similar to IE and hence, has no advantage over IE. However, if the point specified by some δ' is between B and A, then a new (K+1)NN query must be issued at point B which means UBA eliminates the overhead of computing KNN for C. Note that δ' is always greater than or equal to δ' and hence, provides a better bound for our method.

We now discuss the same example using our UBA approach. The four nearest neighbors of D are $\{\uparrow (r_8, 1), \downarrow (r_6, 9), \downarrow (r_3, 10), \downarrow (r_7, 11)\}$. Note that in addition to specifying the KNNs of an object, the VN^3 approach can also be used to specify the direction (i.e., increasing or decreasing) of the neighbors. This can be achieved by determining the immediate connected node, n, to the object that is on the shortest path from the object to its K-th neighbor, r_k. Consequently, r_k is decreasing if n is on the traveling path. Within our approach, we only examine $\uparrow r_8$ and $\downarrow r_6$ to compute δ' since they are the only subsequent members of the list that satisfy our condition (i.e., are increasing and decreasing, respectively). The value of δ' for this example will then become $9 - 1 = 8$, which means that when q starts moving from D toward C, as long as its distance to

D is less than or equal to $\left(\frac{8}{2}=\right)4$, the 3NNs of q will remain similar to the 3NNs of D. This means that once the $(3+1)$NNs of D are determined, there is no need to compute $(3+1)$NNs of any other point on the $(D \rightarrow C)$ path. Moreover, as we discussed in Section 4.1, the first nearest neighbor of a moving point remains the same as long as the point stays in the same NVP. Hence, we can further improve the performance of UBA by ignoring the comparison of the first and second nearest neighbors (if it is necessary) and only check any changes in the first nearest neighbor by locating the intersection of the path with the NVPs of the network. Consequently, the new value of i in our formula for δ' varies from 2 to K, which may lead to even a higher bound value for δ'.

Figure 5 shows the pseudo code of our IE and UBA approaches. As shown in the figure, the computational complexity of IE and UBA is $O(n^*$ [*complexity of KNN*]) and $O(n^*$ [*complexity of (K + 1)NN*]), respectively, where n is the number of intersections on the path. Note that although UBA seems to be more computationally expensive than IE, our experiments (discussed in Section 5) show that UBA outperforms IE since it yields to a smaller value for n as compared to IE.

5. Performance evaluation

We conducted several experiments with real world data sets to evaluate and compare the performance of the proposed approaches for the continuous KNN queries. The data set

Function IE(Path P)

1. *Break P to segments that satisfy property 6:*
   $P = \{n_0, ..., n_m\}$
2. *Start from* $n_i = n_0$, *for each segment* (n_i, n_{i+1})*:*
2.1 *Find KNN*(n_i) *and KNN*(n_{i+1})
2.2 *Find the directions of KNNs*(n_i)
2.3 *Find the split points of the segment* (n_i, n_{i+1})

Function UBA(Path P)

1. *Break P to segments that satisfy property 6:*
   $P = \{n_0, ..., n_m\}$
2. *Start from* $n_i = n_0$, *while* $n_i \neq n_m$ *:*
2.1 *Find (K+1)NN(* n_i *) and their directions*
2.2 *compute* δ'
2.3 *Find* n_p, n_q *where* δ' *is between* (n_p, n_q)
2.4 *If* $n_q = n_{i+1}$*:*
2.4.1 *IE* (n_i , n_{i+1})
2.5 $n_i = n_{i+1}$

Figure 5. Pseudo code of IE and UBA.

used for the experiments is obtained from NavTeq Inc., used for navigation and GPS devices installed in cars. The data represents a network of approximately 110,000 links and 79,800 nodes of the road system in downtown Los Angeles. We performed the experiments using different sets of points of interest (e.g., restaurants, shopping centers) with different densities and distributions. Moreover, we examined UBA and IE for different values of K and randomly generated paths with different lengths. The experiments were performed on an IBM ZPro with dual Pentium III processors, 512 MB of RAM, and Oracle 9.2 as the database server. We utilized our proposed VN3 approach to perform KNN queries. We calculated the number of times that the KNN query must be issued by the IE and UBA methods and the required times to perform these KNN queries. We present the average results of 100 runs of continuous K nearest neighbor queries for each combination of the parameters.

Table 2 shows the details of the data set and the results of our experiments when the length of the traveling path is (approximately) 5 KM and the value of K varies from 1 to 10. As shown in the table, different entities represent points of interest with different densities (that is the percentage of the number of points of interest over the number of intersections in the network). For example, Restaurants (with the quantity of 2944) represent points of interest with a density that is 150 times more than that of the Hospitals (with the quantity of 46). The table also indicates the query response time (in seconds) for IE and UBA approaches. Moreover, the numbers inside parenthesis in columns under UBA specify the number of nodes a (K + 1)NN query is issued at. Note that for the given data set, the average length of the segments between two adjacent intersections is about 147 meters, meaning that (on average) there are 34 intersections in a 5 KM path and hence IE approach requires 34 KNN queries to be issued.

Our first observation is that for $K = 1$ and regardless of the density of the points of interest, the query response time is almost instantaneous (less than 1 second). This is because this query is transformed to a simple *Intersection*(*Road*, *Polygons*) query that can be efficiently performed by utilizing the R-tree index structure. As shown in the

Table 2. Query processing time (in seconds) of IE vs. UBA, traveling Path = 5KM.

			K = 3		*K = 5*		*K = 10*	
Entities	*Qty (density)*	*K = 1*	*IE*	*UBA (No. of nodes)*	*IE*	*UBA (No. of nodes)*	*IE*	*UBA (No. of nodes)*
Hospital	46 (0.0004)	0.6	153	22.5 (5)	217	82 (13)	476	294 (21)
Shopping Centers	173 (0.0016)	0.68	85	20 (8)	110	58 (18)	231	149 (22)
Parks	561 (0.0053)	0.7	34	11 (11)	51	16.5 (20)	91.8	62 (23)
Schools	1230 (0.015)	0.92	17	7 (14)	23.8	14.7 (21)	48.5	37 (26)
Auto Services	2093 (0.0326)	1.0	15.3	6.7 (15)	22.1	14.3 (22)	48	38 (27)
Restaurants	2944 (0.0580)	1.0	13.6	6.0 (15)	19.8	13.4 (23)	47.6	39.2 (28)

table, this query performs faster for the points of interest with lower densities (e.g., Hospitals). This is because the number of network Voronoi polygons for each entity is equal to the number of points in that entity. This means that the R-tree index structure generated for the points of interest with lower densities will be smaller and hence, faster, as compared to the R-tree index generated for the points of interest with higher densities.

Our next observation is that the query response time of UBA is always less than that of IE. As shown in the table, the number of times a (K + 1)NN query is issued by UBA method is between 5 and 31, which is less than 34, i.e., the number of times a KNN query is issued in IE method. This confirms our intuition for the UBA method. Note that although UBA must find more number of neighbors as compared to IE (i.e., (K+1) as compared to K), but it always outperforms IE because it performs less number of NN queries. The table also shows that the advantage of UBA over IE is minimal when the points of interest are densely distributed in the network. For example, while UBA requires (K + 1)NN queries to be issued for only 5 times when continuous 3 nearest hospitals are requested, it performs 15 queries when continuous 3 nearest restaurants are needed. The reason for this is that when the points of interest are dense (e.g., restaurants), the difference between the distances of two subsequent NNs of a point is small and hence, the value of δ' is relatively close to or smaller than the average length of the segments. That is, the points determined by the UBA approach on which the next (K+1)NN queries must be performed, are (usually) located between two adjacent nodes. Hence, the UBA approach can only eliminate the computation of KNNs for a small number of adjacent nodes. However, for the points of interest that are sparsely distributed in the network (e.g., hospitals), the value of δ' is usually much larger than the average length of the segments. This means that the UBA approach can filter out several adjacent nodes from the computation of KNNs and hence, significantly outperforms IE. The table also shows that the performance of UBA becomes close to IE when the value of K increases. For example, while 8 NN queries are issued by UBA to find continuous 3 nearest Shopping Centers for any point on a 5 KM path, it requires 22 NN queries to find the continuous 10 nearest Shopping Centers for the same path. This is also because for larger values of K, more number of subsequent increasing/decreasing neighbors must be examined. This will lead to a smaller value for δ', which subsequently leads to executing KNN query for more number of nodes.

Our next set of experiments evaluate the behavior of the proposed approaches when the length of the traveling path varies. Figure 6 shows the comparison between performances of UBA and IE for two different entities (i.e., Restaurants and Hospitals) when K = 3 and the length of the traveling path varies from 5 KM to 10 KM. The figure shows that the performance of both IE and UBA approaches increases (almost) linearly when the length of the path increases. As shown in the figure, when the length of the traveling path increases from 5 KM to 10 KM (i.e., doubles), the number of NN queries performed by IE also doubles, i.e., 68.5 to 69 NN queries are performed by IE for Hospitals and Restaurants, respectively. Moreover, the number of NN queries performed by UBA increases to 11.5 and 31 from 5 and 15 for Hospitals and Restaurants, respectively. This shows that the proposed solutions perform linearly with respect to the length

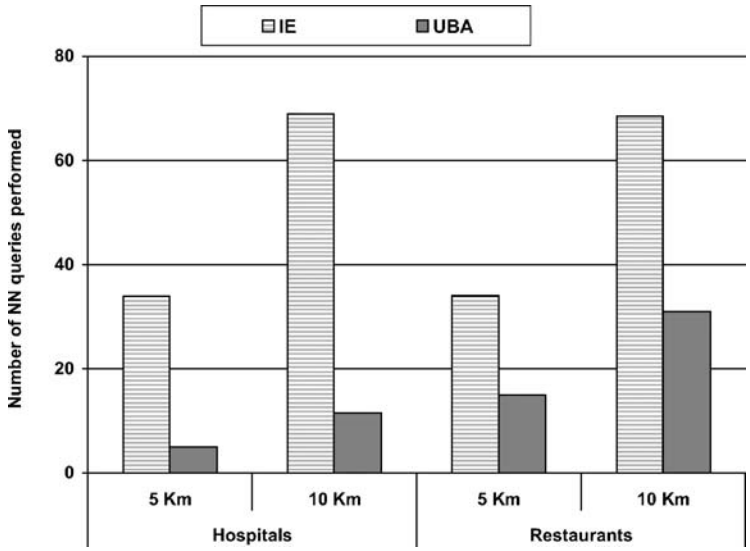

Figure 6. Comparison of UBA and IE for paths with different lengths when K = 3.

of the traveling path. Our experiments for paths with lengths of 1, 2, and 20 KM and different values of K show the same trend.

6. Conclusion

In this paper, we presented alternative solutions for continuous K nearest neighbor queries (C-KNN) in spatial network databases. These solutions efficiently find the location and KNNs of split point(s) on a path. We showed that the continuous 1NN queries (C-NN) can be simply answered using the properties of our previously proposed VN3 approach: the split points are the intersection(s) of the path with the network Voronoi polygons of the network and can be easily located by performing an *Intersection*() function on an R-tree index structure.We also proposed two solutions for the cases where more than one neighbor in a continuous nearest neighbor query is requested (i.e., $K > 1$). With our first solution, IE, we showed that the location of the split points on a path can be determined by first computing the KNNs of all the nodes on the path, and then examining those adjacent nodes with different KNNs. Our second solution, UBA, improves the performance of IE by eliminating the computation of KNNs for the adjacent nodes that cannot have any split points in between. Our experiments also confirmed that UBA outperforms IE, specially when the points of interest are sparse and the value of K is small. The experiments also showed that both IE and UBA perform linearly as to the length of the traveling path increases/decreases.

Acknowledgments

This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0238560 (PECASE), and IIS-0324955 (ITR), and a NASA/JPL grant and unrestricted cash gifts from Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. J. Feng and T. Watanabe. A Fast Method for Continuous Nearest Target Objects Query on Road Network. *VSMM'02, Korea*, pp. 182–191, Sept. 2002.
2. G.R. Hjaltason and H. Samet. "Distance browsing in spatial databases," *TODS*, Vol. 24(2):265–318, 1999.
3. G.S. Iwerks, H. Samet, and K. Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. *VLDB, Berlin, Germany*, pp. 512–523, 2003.
4. C.S. Jensen, J. Kolrvr, Torben Bach Pedersen, and Igor Timko. Nearest Neighbor Queries in Road Networks. *ACM-GIS03, New Orleans, LA, USA*, 2003.
5. M. Kolahdouzan and C. Shahabi. Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases. *VLDB, Toronto, Canada*, 2004.
6. F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast Nearest Neighbor Search in Medical Image Databases. *VLDB, Mumbai (Bombay), India*, 1996.
7. Y. Li, J. Yang, and J. Han. Continuous K-Nearest Neighbor Search for Moving Objects. *SSDBM, Santorini Island, Greece*, 2004.
8. A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu. Spatial Tessellations, Concepts and Applications of Voronoi Diagrams. 0-471-98635-6, 2nd Edition, John Wiley and Sons Ltd: England, 2000.
9. D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query Processing in Spatial Network Databases. *VLDB, Berlin, Germany*, 2003.
10. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. *SIGMOD, San Jose, CA*, 1995.
11. T. Seidl and H.-P. Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. *SIGMOD, Seattle, WA, USA*, 1998.
12. S. Shekhar and J.S. Yoo. Processing in-Route Nearest Neighbor Queries: A Comparison of Alternative Approaches. *ACM-GIS03, New Orleans, LA, USA*, 2003.
13. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. *IEEE ICDE*, 1997.
14. Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. *SSTD, Redondo Beach, CA, USA*, 2001.
15. Y. Tao and D. Papadias. Time Parameterized Queries in Spatio-Temporal Databases. *SIGMOD*, 2002.
16. Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. *VLDB, Hong Kong, China*, 2002.
17. X. Xiong, M.F. Mokbel, and W.G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. *ICDE, Tokyo, Japan*, 2005.

Cyrus Shahabi is currently an Associate Professor and the Director of the Information Laboratory (InfoLAB) at the Computer Science Department and also a Research Area Director at the NSF's Integrated Media Systems Center (IMSC) at the University of Southern California. He received his Ph.D. degree in Computer Science from the University of Southern California in August 1996. He has two books and more than hundred articles, book chapters, and conference papers in the areas of databases and multimedia. Dr. Shahabi's current research interests include Peer-to-Peer Systems, Streaming Architectures, Geospatial Data Integration and Multidimensional Data Analysis. He is currently on the editorial board of ACM Computers in Entertainment magazine and program committee chair of ICDE NetDB 2005 and ACM GIS 2005. He is also serving on many conference program committees such as ICDE 2006, ACM CIKM 2005, SSTD 2005 and ACM SIGMOD 2004. Dr. Shahabi is the recipient of the 2002 National Science Foundation CAREER Award and 2003 Presidential Early Career Awards for Scientists and Engineers (PECASE). In 2001, he also received an award from the Okawa Foundations.

Mohammad R. Kolahdouzan received the B.S. degree in Electrical Engineering from the Sharif University of Technology at Tehran in 1991, M.S. degree in Electrical Engineering in 1997 and Ph.D. in Computer Science in 2004 from the University of Southern California, Los Angeles, CA. He is currently working as a Postdoctoral Research Associate at the Integrated Media Systems Center and Information Sciences Institutes at the University of Southern California working on multidimensional and spatial databases.