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
kolahdoz,shahabi@usc.edu

Abstract

Continuous K nearest neighbor queries (C-KNN) are defined as the nearest points of interest to all the points on a path (e.g., continuously finding the three nearest gas stations to a moving car). 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 objects within each interval are the same. The current studies on C-KNN focus on Euclidean spaces. These studies are not applicable to spatial network databases (SNDB) where the distance between two objects is defined as the length of the shortest path between them. In this paper, we propose two techniques to address C-KNN queries in SNDB: IE and UBA. Our empirical experiments show that the UBA approach outperforms IE, specially when the points of interest are sparsely distributed in the network.

1 Introduction

The problem of K nearest neighbor (KNN) queries in spatial databases have been studied by many researchers. This type of query is frequently used in Geographical Information Systems 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. An example of continuous KNN is when the GPS device of the vehicle initiates a query to continuously find the five closest restaurants to the vehicle at any point of a path from a source to a destination. The result of this type of query is a set of intervals, or 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.

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 the objects (e.g., the vehicle and the restaurants) depend on the connectivity of the network rather than the objects' locations. Hence, 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.

We proposed [4] 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 step: 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 several approaches to address continuous KNN queries in SNDB. Depending on the number of neighbors requested by a CNN query,

Proceedings of the Second Workshop on Spatio-Temporal Database Management (STDBM'04), Toronto, Canada, August 30th, 2004.

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 while it is traveling), 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 CNN query (i.e., C-KNN). 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 finding the KNNs of all 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 a new KNN query to be issued. Our empirical experiments show that the second approach outperforms the first solution. To the best of our knowledge, the problem of continuous K nearest neighbors is spatial network databases has not been studied.

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 online 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 proposed 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 depend on their selectivity in the filter step. Roussopoulos et al. in [8] present a branch-and-bound R-tree traversal algorithm to find 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. in [5] 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 et al. in [9] propose an optimal version of this multi-step algorithm by incrementally ranking queries on the index structure. Hjaltason et al. in [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 these approaches that render them impractical for networks is that the filter step of these approaches performs based on Minkowski distance metrics (e.g., Euclidean distance) while the networks are metric space, 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. in [7] 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. Shekar et al. in [10] and Jensen et al. in [3] 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, in [4], 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. in [11] first identify the importance of the continuous nearest neighbors and describe modeling methods and query languages for the expression of these queries, but did not discuss the processing methods. Song et al. in [12] propose the first algorithms for CNN queries. They propose fixed upper bound algorithm that specifies the minimum distance that an object can move without requiring a new KNN to be issued. They also propose a dual buffer search method that can be used when the position of the query can be predicted. Tao et. al in [13] present 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. Finally, Tao et al. in [14] propose a solution for CNN queries based on performing one single query for the entire path. They also extend 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. Finally, Feng et al. in [1] provide 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 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 ([4]) to address KNN queries in spatial network databases. A thorough discussion on Voronoi diagrams is presented in [6].

3.1 Network Voronoi Diagram

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. Consider a set of limited number of points, called *generator points*, 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. The Voronoi polygons that share the same edges are called *adjacent polygons*. A network Voronoi diagram, termed *NVD*, is defined for (directed or undirected) graphs and is a specialization of Voronoi diagrams where the location of objects is restricted to the links that connect the nodes of the graph and distance between objects is defined as their shortest path in the network rather than their Euclidean distance. A network Voronoi polygon $NVP(p_i)$ specifies the links (of the graph), or portions of the links, that p_i is closest point of interest to any point on those links. A *border point* of two NVPs is defined as the location where a link crosses one NVP in to the other NVP.

3.2 Voronoi-Based Network Nearest Neighbor: VN^3

Our proposed approach to find the K nearest neighbor queries in spatial networks [4], termed VN^3, 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. 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. The filter/refinement process in VN^3 is iterative: at each step, first a new set of candidates is generated from the NVPs 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. VN^3 consists of the following major components:

1. Pre-calculation of the solution space: As a major component of the VN^3 filter step, the NVD for the points of interest (e.g., hotels, restaurants,...) in a network must be computed and its corresponding NVPs must be stored in a table.
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. This stage can be expedited by using a spatial index structure generated on the NVPs.
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.

Our empirical experiments shows that VN^3 outperforms the only other proposed approach ([7]) for KNN queries in SNDB by up to one order of magnitude. They also show that the size of the candidate set generated by our proposed filter step is smaller than that of the approaches designed for spatial index structure.

4 Continuous Nearest Neighbor Queries

Continuous nearest neighbor queries are defined as determining the K nearest neighbors of any object on a given path. An example of this type of query is shown in Figure 2 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 $\{r_1, ..., r_8\}$) 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 as the KNNs of the split points. The challenge for this type of query is to efficiently find the location of the split point(s) on the path. 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 is usually defined as their shortest path (or shortest time) which has a computationally complex function. 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 approach 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 CNN queries, when only the first nearest neighbor is requested, is based on the properties of network Voronoi polygons. 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. Subsequently, we can conclude that p_i is the C-NN of the segments of the path that are contained in $NVP(p_i)$. Note that this approach cannot be extended to C-KNN queries since the 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 1 where $\{p_1, ..., p_7\}$ are the points of interest. As depicted in the figure, the path from S to D crosses $NVP(p_1)$, $NVP(p_5)$ and $NVP(p_7)$ 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. Our first solution, IE, is based on examining the KNNs of all the nodes on a path, while our second approach, UBA, eliminates the KNN computation for the nodes that cannot have any split points in between.

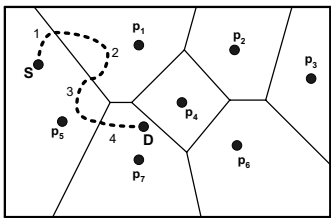

Figure 1: Continuous first nearest neighbor

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 1: 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 n_i and hence $distance(q, o_k) = distance(q, n_i) + distance(n_i, o_k)$. However, we know that $distance(n_i, o_k) > distance(n_i, o_1)$ 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 $distance(q, o_k) > distance(q, o_1)$ which means that o_1 is closer to q than o_k, contradicting our initial assumption.

As an example, this property suggests that in Figure 2, 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 2: Let $p(n_i, n_j)$ be the path between two adjacent nodes n_i and n_j, $O = \{o_1, ..., o_k\}$ and $O' = \{o'_1, ..., o'_k\}$ 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 $\{O \bigcup O'\}$.

Proof: This property is in fact the extension of property 1 and can be similarly proved by contradiction.

As an example, this property suggests that in Figure 2, 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 the nodes will

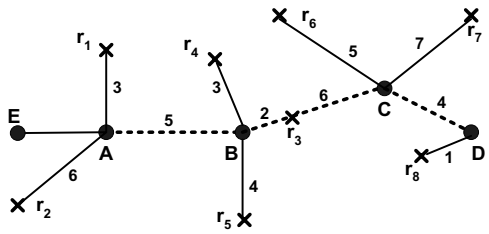

Figure 2: Example of continuous K nearest neighbor query

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 2) are the same and hence, $\{O \bigcup O'\} = O = O'$.

- 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 which includes r_3). We can address this issue in two alternative ways: 1) breaking 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 2 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 as well 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 is negligible. 2) Similar to Properties 1 and 2, we can also 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 $\{r_3 \bigcup \{r_3, r_4, r_5\} \bigcup \{r_6, r_8, r_3\}\}$. For the rest of this paper, we use the first alternative.

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. We can specify whether a point of interest is considered as increasing or decreasing NN using the following property:

Property 3: 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 $O = \{o_1, ..., o_k\}$ 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 formally define increasing/decreasing NNs as:

Definition: A point of interest o is called increasing for the 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 2, r_3 is considered as a decreasing NN for a query object that is traveling from A toward B (since B is on the path between A and r_3), but it is considered an increasing NN when the query is traveling from A toward E.

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 2, we are interested to find the three closest restaurants to any point on the path (A, B, C, D).

Step 1: The first step is to break the original path $(A, B,\ C, D)$ to smaller segments. 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. To illustrate our technique, we focus on the first segment (i.e., AB). The other consequent segments can be treated similarly. The three nearest restaurants to A and B and their distances as $\{(r_1, 3)(r_2, 6)(r_3, 7)\}$ and $\{(r_3, 2)(r_4, 3)(r_5, 4)\}$, respectively. We now know that there must be split point(s) between A and B and the KNNs of any point on segment (A, B) is a subset of the candidate list $\{r_1, r_2, r_3, r_4, r_5\}$.

Step 3: From the above candidate list, we 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 than

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) we then select the split point with 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 above condition 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 will be no split points 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 means there will 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 p_1 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 $np_2 = \frac{6.5+7.5}{2} = 7$, 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. 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. The results for segments (B, r_3), (r_3, C) and (C, D) 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 in Figure **??**, we are interested in the first NN of any point on the traveling path S to D. Clearly, there are only three split points on this path. However, if we utilize IE approach to address this query, the 1-NN query will be issued for all the intersections of the path. We address this issue with our second approach.

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	$\downarrow (r_3, 4)$, $\downarrow (r_4, 5)$, $\downarrow (r_5, 6)$, $\uparrow (r_1, 6)$, $\uparrow (r_2, 9)$

Table 1: Split points for segment (A, B) of Figure 2

4.2.2 Upper Bound Algorithm: UBA

The UBA approach works similar to IE. While IE performs KNN queries for every intersection on the path, the UBA approach postpones the computation of KNN queries to only the locations that is required and hence, provides a better performance by reducing the number of KNN computations. The intuition for UBA is similar to what is discussed in [12]: when a query object is moved slightly, it is very likely that its KNNs remain the same. Song et. al. in [12] define a threshold value as $\delta = \min(d(o_{i+1}, q) - d(o_j, q))$ where q is the query object and $o_{i+1} \in (K)NNs(q)$. The defined δ specifies 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 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 improves the performance of our approach further.

We first discuss the extension of the approach described in [12] to SNDB using the example in Figure 2. 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 that 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 $\delta' = \min(d(o_i, q) - d(o_{i+1}, q))$ 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 remain the same. Otherwise, a new (K+1)NN query must only be issued at the intersection 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 2, if the traveling path is (D, C, B, A) and the point specified by some δ' is between C and B, a new (K+1)NN query must be issued at point C which means that UBA will perform similar to 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. With 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. The value of δ' for this example will then become $9-1=8$, which means that when q starts moving from D to C, as long its distance to D is less than or equal to ($\frac{8}{2}$ =)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. 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 ignore the comparison of the first and second nearest neighbors (if it is necessary) and check any changes in the first nearest neighbor only 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 a higher bound value for δ'.

Figure 3 shows the pseudo code of our IE and UBA approaches.

Function IE(Path P)

1. Break P to segments that satisfy property 2:
$P = \{n_0, ..., n_m\}$
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 2:
$P = \{n_0, ..., n_m\}$
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' *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 3: Pseudo code of IE and UBA

5 Performance Evaluation

We conducted several experiments to compare the performance of the proposed approaches for the continuous KNN queries. The data set 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 the downtown Los Angeles. We performed the experiments using different sets of points of interest (e.g., restaurants, shopping centers) with different densities and distributions. The experiments were performed on an IBM ZPro with dual Pentium III processors, 512MB of RAM, and Oracle 9.2 as the database server. We calculated the number of times that the KNN query must be issued and the required times, for different values of K and different lengths of traveling paths. We present the average results of 100 runs of continuous K nearest neighbor queries.

Table 2 shows the query response time for IE and UBA approaches when the length of the traveling path is 5KM and the value of K varies from 1 to 20. 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 5KM path. As shown in the table, UBA always outperform IE. However, the advantage of UBA over IE is minimal when the points of interest are distributed densely in the network (e.g., restaurants). The reason for this is that in these cases, the value of δ' is relatively close to the average length of the segments. That is, the points determined by the UBA approach on which the next KNN 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. Note that the IE approach requires 34 KNN queries (with different values of K)to be performed in all cases. However, the number of nodes that are

Entities	Qty (density)	K=1	K=3		K=5		K=10		K=20	
			IE	UBA (No. of SP)	IE	UBA (No. of SP)	IE	UBA (No. of SP)	IE	UBA (No. of SP)
Hospital	46 (0.0004)	0.6	153	22.5 (5)	217	82 (13)	476	294 (21)	952	670 (24)
Shopping Centers	173 (0.0016)	0.68	85	20 (8)	110	58 (18)	231	149 (22)	493	377 (26)
Parks	561 (0.0053)	0.7	34	11 (11)	51	16.5 (20)	91.8	62 (23)	187	143 (26)
Schools	1230 (0.015)	0.92	17	7 (14)	23.8	14.7 (21)	48.5	37 (26)	119	94.5 (27)
Auto Services	2093 (0.0326)	1.0	15.3	6.7 (15)	22.1	14.3 (22)	48	38 (27)	85	72 (29)
Restaurants	2944 (0.0580)	1.0	13.6	6.0 (15)	19.8	13.4 (23)	47.6	39.2 (28)	81.6	74.5 (31)

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

specified by UBA approach as the points that the next KNN queries must be performed on (specified inside "()" in the table) are always less than 34.

Also note that the performance of UBA becomes close to IE when the value of K increases. This is also because there are more number of subsequent increasing/decreasing neighbors that must be examined when the value of K increases. This will lead to a smaller value for δ', which will generate more split points on the path. When the value of K is equal to 1, the query response time becomes smaller when the points of interest are sparse. This is because the number of NVPs in the network are less when the points of interest are sparse and hence, the intersection of a line with the NVPs can be determined faster. The experiments for traveling paths of 1, 2, 5, 10, and 20 KM show similar trends.

6 Conclusion

In this paper we presented alternative solutions for continuous K nearest neighbor queries 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 can be simply answered using the properties of our previously proposed VN^3 approach: the split points are the intersection(s) of the path with the network Voronoi polygons of the network. We also proposed two solutions for the cases where continuous K nearest neighbors are requested. 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 the adjacent nodes that have 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 empirical experiments also confirmed that UBA outperforms IE.

7 Acknowledgement

This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0238560 (CAREER), IIS-0324955 (ITR), and in part by a grant from the US Geological Survey (USGS).

References

[1] J. Feng and T. Watanabe. "A Fast Method for Continuous Nearest Target Objects Query on Road Network". In *VSMM'02 pp.182-191 Sept. 2002, Korea.*

[2] G. R. Hjaltason and H. Samet. "Distance Browsing in Spatial Databases". *TODS*, 24(2):265–318, 1999.

[3] C. S. Jensen, J. Kolrvr, T. B. Pedersen, and I. Timko. "Nearest Neighbor Queries in Road Networks". In *Proceedings of the eleventh ACM international symposium on Advances in geographic information systems table of contents, ACM-GIS03, New Orleans, Louisiana, USA, 2003.*

[4] M. Kolahdouzan and C. Shahabi. "Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases". In *VLDB 2004, Toronto, Canada.*

[5] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. "Fast Nearest Neighbor Search in Medical Image Databases". In *VLDB 1996, Mumbai (Bombay), India.*

[6] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *"Spatial Tessellations, Concepts and Applications of Voronoi Diagrams"*. John Wiley and Sons Ltd., 2nd edition, 2000.

[7] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. "Query Processing in Spatial Network Databases". In *VLDB 2003, Berlin, Germany.*

[8] N. Roussopoulos, S. Kelley, and F. Vincent. "'Nearest Neighbor Queries". In *SIGMOD 1995, San Jose, California.*

[9] T. Seidl and H.-P. Kriegel. "Optimal Multi-Step k-Nearest Neighbor Search". In *SIGMOD 1998, Seattle, Washington, USA.*

[10] S. Shekhar and J. S. Yoo.

[11] P. Sistla, O. Wolfson, S. Chamberlain, and D. S. "Modeling and Querying Moving Objects". In *IEEE ICDE 1997.*

[12] Z. Song and N. Roussopoulos. "K-Nearest Neighbor Search for Moving Query Point". In *SSTD 2001, Redondo Beach, CA, USA.*

[13] Y. Tao and D. Papadias. "Time Parameterized Queries in Spatio-Temporal Databases". In *SIGMOD 2002.*

[14] Y. Tao, D. Papadias, and Q. Shen. "Continuous Nearest Neighbor Search". In *VLDB 2002, Hong Kong, China.*