Restricted Continuous KNN Queries on Road Networks with Caching Enhancement

Maytham Safar[1], Dariush Ebrahimi[2], Mary Magdalene Jane F.[3], R. Nadarajan[4]
[1,2] Computer Engineering Department
Kuwait University, Kuwait
maytham@mac.com, darebra@yahoo.com

[3,4] Department of Mathematics and Computer Applications
PSG College of Technology
Coimbatore
India
janejayakumar@gmail.com, nadarajan_psg@yahoo.co.in

Journal of Digital Information Management

ABSTRACT: *Using a Global Positioning System (GPS) in the car navigation system enables the driver to perform a wide range of queries, from locating the car position, to finding a route from a source to a destination, or dynamically selecting the best route in real time. With spatial network databases (SNDB), objects are restricted to move on pre-defined paths (e.g., roads) that are specified by an underlying network. In our previous work, we proposed a novel approach, termed Progressive Incremental Network Expansion (PINE), to efficiently support several spatial queries. In this work, we utilize our developed PINE system to efficiently support Restricted Continuous K Nearest Neighbors (RCKNNs) queries. RCKNN continuously finds the K nearest objects to a query point on a given path that are within a specified distance bound. Our solution addresses a new type of query that is plausible to many applications where the answer to the query not only depends on the distances of the nearest neighbors, but also on the user or application needs. By distinguishing between two types of split points, we reduce the number of computations required to retrieve the RCKNNs of a moving object. We further define a caching model to expedite RCKNN query response time.*

Categories and Subject Descriptors
H.2.4 [Systems] Query processing ; H.2.8 [Database Applications]: Spatial databases and GIS ; C.2.3 [Network Operations]

General Terms
K Nearest Neighbors, Global Positioning System

Keywords: navigation system, Network databases, Spatial databases
Received 26 Aug. 2006; Revised and accepted 21 July 2007

1. Introduction and background

Over the last decade, due to the rapid developments in information technology (IT), new applications that go beyond traditional web-based systems were introduced, and hence they come with new challenges for researchers, developers and users. Several types of spatial queries (e.g., nearest neighbor - NN, K nearest neighbors – KNN, continuous nearest neighbor - CNN, continuous k nearest neighbors – CKNN) have been proposed and studied in the context of spatial databases. The most common type is the point KNN query, which is defined as: given a set of spatial objects (or points of interest), and an input query point, retrieve the (K) nearest neighbors to that query point. The NN is the target object with the shortest path from the query point on the route. The efficient implementation of KNN query is of a particular interest in Geographical information systems (GIS). For example, a GPS device in a vehicle gives information of an object's location, which, once located onto a map, serves as a basis to find the K closest restaurants or gas stations with the shortest path to them.

Different variations of KNN queries have been introduced. One variation is the continuous KNNs of any point on a given path. As an example when the GPS device of the vehicle initiates a query to continuously find the 3 nearest gas stations to the vehicle at any point of a path from source to destination. The result is a set of intervals or split points where the KNNs of a moving object on a path will be the same up to these points. The versatility of K nearest neighbors search increases substantially if we consider other variations of it such as the Continuous KNN (CKNN) and restricted CKNN queries (RCKNN.) CKNN query is defined as the K nearest point of interest to every point on a path, and has found applications in the GISs (e.g., find my nearest 3 gas stations at any point during my route from city A to city B.) The result of this type of query is a set of tuples <result, interval> such that the result is the KNN of all points in the corresponding interval ordered by distances to the query point. The interval is defined by two end-points, called split points, which specify where on the path the KNNs of a moving object will change. This means that the KNNs of an object traveling on one interval of the path remain the same all through that interval, until it reaches a split point where its KNNs change. With RCKNN, only the interest points that are within a fixed network distance d around the query object are returned. This would expedite the execution of the CKNN queries due to the reduction of the interest points to be retrieved.

Taken into consideration that mobile devices such as navigators are usually limited on memory resources and have lower computational power, in [16][25] we proposed a novel approach that reduces the problem of distance computation in a very large network, into the problem of distance computation in a number of much smaller networks plus some online "local" network expansion. The main idea behind that approach, termed Progressive Incremental Network Expansion (PINE), is to first partition a large network into smaller/more manageable regions. We achieved this by generating a network Voronoi diagram over the points of interest. Each cell of this Voronoi diagram is centered by one object (e.g., a restaurant) and contains the nodes (e.g., vehicles) that are closest to that object in network distance (and not the Euclidian distance.) Next, we pre-compute the inter distances for each cell. That is, for each cell, we pre-compute the distances across the border points of the adjacent cells. This will reduce the pre-computation time and space by localizing the computation to cells and a handful of neighbor-cell node-pairs. Now, to find the *k* nearest-neighbors of a query object *q*, we first find the first nearest neighbor by simply locating the Voronoi cell that contains *q*. This can be easily achieved by utilizing a spatial index (e.g., R-tree) that is

generated for the Voronoi cells. Then, starting from the query point *q* we perform network expansion on two different scales simultaneously to: 1) compute the distance from *q* to its first nearest neighbor (its Voronoi cell centre point), and 2) explore the objects that are close to *q* (centers of surrounding Voronoi cells) and compute their distances to *q* during the expansion. In solving RCKNN queries, it is important to note that issuing a traditional nearest neighbor query at every point of the line segment is infeasible due to the large number of queries generated and the large overhead. The challenge for this type of query is to efficiently find the location of the split point(s) on the path. Or in other words, where in the path does the KNN change. The main idea behind our approach is that the KNNs of any object on a path between two adjacent nodes (e.g., intersection in road network) can be a subset of any points of interest (e.g., gas stations) on the path. Hence, the solution is based on breaking the entire path to smaller segments (sub-paths), where each segment is surrounded by two adjacent nodes. Our approach is then based on finding the minimum distances 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.

We divide the problem into two cases, depending on the number of neighbors requested by a RCKNN query. 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 PINE model. We show that the split points on the path are simply the intersections of the path with the network Voronoi polygons (NVPs) of the network, which are a subset of the border points of the NVPs. The second case is when more than one neighbor is requested by RCKNN, the main idea behind our approach is that the KNN of any object on a path between two adjacent nodes (e.g., intersection in road system) can only be a subset of any point of interest (e.g., restaurants) on the path, plus the KNNs of the end nodes. Therefore, we need to first find the KNNs of the intersections on the path using PINE, and then find the location of the split points between two adjacent nodes and their associated KNN.

To efficiently find the location of the split point(s) on the path we use a modified version of the IE algorithm proposed by [9]. The solution is based on breaking the entire path into smaller segments, where each segment has two end-points (e.g., adjacent intersections in road network), and finding the KNNs of all nodes in each segment. There is a split point between two adjacent nodes with different KNNs. The location of the split points can be found by first specifying whether each NN is increasing or decreasing, depending on the distances from a query object to the KNNs of the nodes as the objects move, then calculating a split point for each increasing member of the candidate set with every decreasing NN, or vice versa. At every step of the algorithm, we check if the points (interest points, intersection points, or border points) being explored are within a network distance *d* around the query point. Otherwise, we stop expanding in that direction.

In addition, we distinguish between different split points. One type of split points where we replace the element(s) of the KNN list with (a) different one(s) compared to the KNN list of the starting node of a segment is called “Element- SplitPoint” (ESP.) The other type of split points at which we change the order of the elements of the KNN is called “Order- SplitPoint” (OSP.) In some applications, we only care about the k nearest neighbors and not their ordered distances (i.e., being the first nearest neighbor or the second.) For example, suppose that as we are traveling by a car, we issue a query to find the five nearest restaurants to us, however, we would choose to go to the one that serves our favorite cuisine and not necessarily the closest one. The query may return the following five nearest restaurants in ascending order: Indian, Italian, American, Chinese, and Indonesian. Although the Indian restaurant is the closest, we could go to the American restaurant if we like the American cuisine. Here, the choice was not based on pure distances. If the application does not require the order of the KNNs, then we only have to save the ESP points and ignore the OSP. The intuition is that the total count of all ESP is less than OSP.

Contrasting it with [9], our method reduces the number of KNNs queries performed and eliminates the need to update the directions (increasing, decreasing) of the NN as the object moves. We just generate one table to provide the information on where the ESP/ OSP are going to occur, and the hints for the KNN elements at each breakpoint. Hence, reducing the number of computations to retrieve the continuous KNN of a moving object.

Our experimental results in [16][25] showed that VN^3 failed in answering some CKNN queries and provided invalid results. Our analysis of the algorithm identified some flaws in the algorithm, especially in the cases where both end points of a line segment (road link) have a common nearest neighbor. In this case, the algorithm assumes that while moving from one end point to the other, the distance to that common nearest neighbor either increases or decreases throughout the link. However, our investigation and analysis showed that this is not the case. Usually the distance increases until you reach a virtual split point (not real). At this point, the distance gets decreased because the shortest path to the common nearest neighbor would pass through the second end point. Hence, in this paper we provide a modified algorithm to resolve that problem.

Finally, taking into consideration the limitation on memory resources, we propose a new caching mechanism to be used within the navigator system to expedite the query response of the RCKNN queries.

2. Related work

The most common type of query encountered in spatial databases is the point *k* nearest neighbor (KNN) query, which is defined as: given a point query in a multidimensional space, find the *k* closest objects in the database to the query point. This type of query is extensively used in geographical information systems (GIS) and thus was the focus of many researches. There are two groups of algorithms proposed to address the KNN query. Some of the algorithms are based on utilizing the Euclidean distances; other algorithms are based on network distances. The regular KNN queries are the basis for several query variations such as the Restricted Continuous KNN. In this section, we overview previous work related to KNN query, and its variations.

The existing work on the first group considers Cartesian (typically, Euclidean) spaces, where the distance between two objects is determined by their relative position in space. The major disadvantage with the approaches in [15][11][18] is that the shortest path calculations are performed based on Euclidean distances while in practice, objects usually move only on pre-defined roads. This makes the distance calculations depend on the connectivity among these objects. Here, the important measure is the network distance, which renders the algorithms in the first group impractical for SNDB.

The other group of research focuses on solving the KNN for

spatial network databases. In these databases, the underlying network connections are captured and the distance between two objects is the length of the shortest path connecting them. The approaches in [13][8][16] support the exact KNN queries on spatial network databases.

The solution in [13], called the Incremental Network Expansion (INE), introduces an architecture that integrates network and Euclidean information. It is based on creating a search region for the query point that expands from the query that is similar to Dijkstra's algorithm. The advantages of this approach are: 1) it offers a method for finding the exact distance in networks, and 2) the architecture can support other spatial queries like range search and closest pairs. However, this approach suffers from poor performance when the objects (e.g., restaurants) are not densely distributed in the network because this will lead to large portions of the database to be retrieved. This problem happens for large values of k as well.

The Voronoi-based Network Nearest Neighbor (VN^3) approach proposed in [8] is based on the properties of the Network Voronoi Diagrams (NVD). It uses localized pre-computations of the network distances for a very small percentage of neighboring nodes in the network to enhance query response time and reduce disk accesses. In addition, Network Voronoi Polygons (NVPs) of a NVD can directly be used to find the first nearest neighbor *q*. Subsequently, NVP's adjacency information provides a candidate set for other nearest neighbors of *q*. Finally the pre-computed distances are used to 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 VNPs, then the pre-computed distances are used to select "only the next" nearest neighbor of *q*. The advantages of this approach are: 1) it offers a method that finds the exact distances in networks, 2) fast query response time, and 3) progressively returns the *k* nearest neighbors and their distances from the query point. The main disadvantage of this approach is its need for pre-computing and maintaining two different sets of data: 1) query to border computation: computing the network distances from *q* to the border points of its enclosing network Voronoi polygon, and 2) border to border computation: computing the network distances from the border points of NVP of *q* to the border points of any of the other NVPs. Furthermore, this approach suffers in performance with lower density data sets.

We proposed in [16] a novel approach, termed PINE, to efficiently address KNN queries in SNDBs. The main idea behind this approach is to first partition a large network into smaller more manageable regions, then pre-compute distances across the regions. Those two steps can be easily and efficiently implemented using a first order Voronoi diagram, then a computation similar to the INE can be used for the computation of intra-distances. The advantage of PINE is that it has less disk access time and less CPU time than VN^3. In addition, PINE's performance is independent of the density and distribution of the points of interest, and the location of the query object. By performing across-the-network computation for only the border points of the neighboring regions, we avoid global computations later on.

The solutions proposed for regular KNN queries are either directly used or have been adapted to address the variations of KNN queries such as RCKNN queries. Given a predefined route, a continuous query retrieves tuples of the form <result, interval> where each result is accompanied by a future interval, during which it is valid. Despite the importance of continuous queries in SNDBs, the scarce studies in the literature are designed for Euclidean spaces (e.g., [21], which are not applicable to SNDBs.

For example, the approach proposed in [21] uses the R-tree as the underlying data-partition access method. Their algorithm traverses the tree and prunes unnecessary node accesses based on some heuristics that use Euclidean distances. Their goal is to perform one single query of the entire path. The algorithm starts with an initial list of split points (SL) containing only the path starting and ending nodes, and an empty initial list of NNs, and then it incrementally updates the SL during query processing. After, each step, the SL contains the current result with respect to all the data points processed so far. The final result contains each split point that remains in SL after the termination together with its nearest neighbor. The advantage of [21] is the avoidance of multiple database scans by reporting the result with a single traversal of the database. Yet, it still has the major disadvantage of using Euclidean distances that are not applicable to network distances.

Finally, [9] address the problem of CKNN queries in road networks. They proposed two techniques termed: Intersection Examination (IE) and Upper Bound Algorithm (UBA) to find the location and KNN of split point(s) on the path. The first solution, IE, finds KNNs of all the nodes on a path by breaking the path into segments and only examining the KNNs of intersection nodes. There is a split point on the shortest path between two adjacent nodes with different KNNs and the location of that point is calculated. The second approach, UBA, improves the performance of IE by reducing the number of KNN computations by eliminating the computation of KNNs for the nodes that cannot have any split points in between. The intuition of UBA is that when a query object is moved slightly, it is very likely that its KNNs remain the same. UBA proposes a method to find the minimum distance that the object can move without requiring a new KNN to be issued.

There are three shortcomings of [8]: 1) the total number of split points computed using this algorithm is sometimes redundant or useless for some kinds of applications as we explained in section 1, 2) The distance to all the KNN of both end nodes (i.e., the distance to the candidate list of each segment) are updated and ordered at each split point which incurs unnecessary overhead, and 3) The PINE algorithm is more efficient than VN^3 in finding the KNN of a point. (For experimental results see [16]). To the best of our knowledge, [9] is the only approach that uses network distances to find CKNN.

In section 3 we discuss our approach to solve RCKNN using PINE.

3. Restricted continuous *k* nearest neighbors queries (RCKNN)

Restricted continuous nearest neighbor queries are defined as determining the *k* nearest neighbors of any object on a given path that are within a network distance *d* from the query object. An example of this type of query is shown in Figure 1. In this example, a moving object (e.g., a car) is traveling along the path (L_1, L_2, L_3, L_4) (specified by the dashed lines) and we are interested in finding the first 3 closest neighbors (neighbors are specified in the figure by {n_1, …, n_8}) to the object at any given point on the path that are within a network distance *d* from the query object. The result of a restricted continuous KNN 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. The challenge for this type of query is to efficiently find the location of the split point(s) on the path.

In this section we discuss our solution for RCKNN queries in spatial network databases. We first present our approach for

the scenarios when only the first NN is desired (i.e., RCNN), and then for the cases where the RCKNN of any point on a given path is requested.

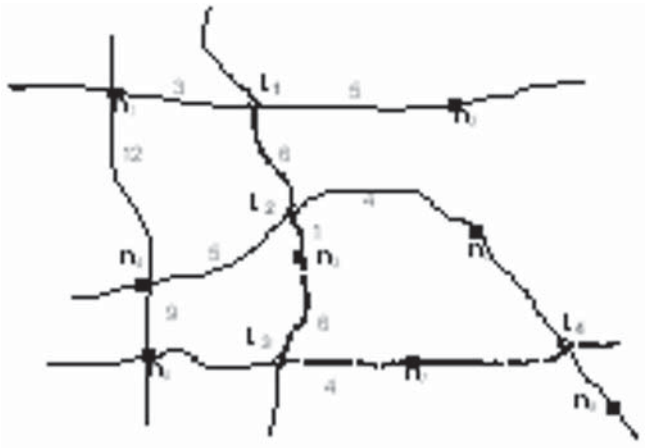

Figure 1. Restricted Continuous K Nearest Neighbors Query

3.1. Restricted continuous 1NN queries (RCNN) using PINE

Our solution for restricted CNN queries is based on our previous work PINE that partitions the network into disjoint first order network Voronoi polygons (NVP) (Safar, 2005) in such a way that the first nearest neighbor of any point inside a polygon is the generator of that polygon. To find the RCNN of a given path, we first find the split points on the path at which the NN changes. By intersecting the path with the NVPs of the network, the points of intersections specify the split points, which in turn, define the path segments inside each polygon. As a result, the first restricted continuous NN for every point in a segment inside a polygon is the generator of that polygon, if it within a network *d* distance from that point. However, this approach cannot be extended to RCKNN queries because the NVD is a first order network diagram that can only specify the first NN.

3.2. Restricted continuous KNN queries (RCKNN) using PINE

Our algorithm for finding the restricted continuous KNN of any point on a path, starts by breaking the path into smaller segments according to some properties, then finding the restricted continuous KNN for each segment, and finally, generating the result set for the entire path by joining the results for all segments. It is shown that there must be a split point on the shortest path between the segments' nodes if the end-nodes have different KNNs [9]. Otherwise, the set of restricted continuous KNN would remain fixed on that segment if the KNNs on that link are within a network distance *d* from the query point. To efficiently find the location of split points, our algorithm performs the following steps:

Step 1: The first step is to break the original path into smaller segments using the technique proposed by [9] such that the end-points of every segment are either an intersection or an interest point.

Step 2: Then, we find the KNNs of the end-nodes of each segment using our KNNs algorithm (PINE) [16][25]. It is shown that the restricted continuous KNNs of each segment are a subset of the union of KNNs of the end-points of that segment; we call this union the candidate list. From this list, we generate a new ordered list of the nearest neighbors for the starting point of the segment. In other words, the list is sorted according to distances to the segment's starting node (L_y). Similar to [9], we also specify the direction of each neighbor (increase/ decrease) according to whether the distance to that neighbor is increasing or decreasing as the query object moves from the starting point of the segment to a split point.

Step 3: Delete from the candidate list the KNN points that are within a network distance larger than *d* from the query point.

Step 4: In this step, we try to find the locations of split points, since we know that if the end-nodes have different KNNs, then there must be one or more split point(s) on the shortest path between the segments' nodes [9]. For each member of the set with increasing direction, compare it with each decreasing direction neighbor to find the location (relative to the starting node of a segment L_y) of all split points in a segment using the following method: Split Point (P) generated from $\uparrow(n_i, d_{n_i})$ and $\downarrow(n_j, d_{n_j})$ which is at a distance of $(d_{n_j} + d_{n_i})/2 - d_{n_i}$ from location L_y (note: d_{n_i} and d_{n_j} are the distances from location L_y.) The total number of split points is always equal to the number of increasing distance neighbors multiplied by the number of decreasing distance neighbors and all must be generated. We will later distinguish between two types of split points.

Step 5: We save the results of the previous step for segment (L_y, L_{y+1}) in a table format sorted incrementally according to distances to L_y. Each row has three entries: (1) split point (P_i), (2) distance between P_i and $L_{y\,(dpi)}$, (3) and split-NN which is a tuple (n_i, n_j) such that the split points are generated from these two neighbors n_i and n_j. For complete pseudo code of the algorithm see Figure 2.

Algorithm modified IE (Path P)

1. Break P to segments such that the end-points of every segment is either an intersection or interest point: P={L_1, L_2, …….L_n}
2. For each segment, start from L_y (y=1): 1) Find kNN (L_y) and kNN(L_{y+1}) using PINE , 2) Find the directions of kNNs of the start of the segment (L_y) , 3) delete the kNNs that are within a network distance larger than *d* from the query object, and 4) Find the location of the split points for the segment (L_y, L_{y+1})

Figure 2. Pseudo code for modified IE algorithm

Given the table, one can easily find the restricted continuous KNN for a moving object in interval L_y, L_{y+1}. Starting with the list of KNN of the beginning node L_y, the KNNs stay the same as the object moves until it reaches the first split point where the KNNs might change according to one of three cases interpreted from the third column entries of the saved table (e.g., Table 1.) At a split point (P), the split-NN(n_i, n_j) could mean (i) neighbors n_i and n_j will change their order in the KNN list if both of them are already in the list (we call these split points OSP), (ii) n_j will replace n_i in the KNN list if n_j is not already in the list (we call these split points ESP), or (iii) nothing is going to change in the KNN list if both n_i and n_j are not in the list. Note that the table lookup process is progressive; each iteration (step), as the query object travels between split points, depends on the result of its previous step.

Split Point	Distance to L_1	Split-NN
P_4	1	(2,3)
P_1	2	(1,3)
P_5	2.5	(2,5)
P_6	3	(2,4)
P_2	3.5	(1,5)
P_3	4	(1,4)

Table 1. Split nodes and Split-NN for segment (L_1,L_2) of Figure 1

In [9], the algorithm keeps track of the candidate list elements and updates their distances to the corresponding split point at each step. We are not updating the distances at all between the split points and the candidate KNN because this incurs unnecessary calculations and wastes storage. In other words, these distances are not valid when the query object is moving between split points, and if required, the distances to the KNNs need to be calculated on-line depending on the current location of the query object.

Table 1 shows the results of an example of applying the above algorithm for the segment (L_1, L_2), where the first split point for this segment is P_4. Hence, the KNNs of any point on (L_1, P_4) interval is equal to the KNNs of L_1 (and P_4), for any point on (P_4, P_1) segment is equal to KNNs of P_4 (and P_1), and so on. Note that the distances from a query object, which is between two split points, to its KNNs can be similarly computed. The results for segments (L_2, n_3), (n_3, L_3) and (L_3, L_4) can be similarly found. Furthermore, all the KNNs shown in the table are within a distance less than the network distance *d* from the query object.

To illustrate our technique, we use the following example: suppose that in Figure 1, we are interested to find the three closest neighbors to any point on the path (L_1, L_2, L_3, L_4) which are within a network distance $d = 8$. We focus on the first segment (L_1, L_2), the other subsequent segments can be treated similarly.

Step 1: The first step is to break the original path (L_1, L_2, L_3, L_4) to smaller segments such that the end- points of every segment are either an intersection or interest points. For the given example, the resulting segments will be (L_1, L_2), (L_2, n_3), (n_3, L_3), (L_3, L_4).

Step 2: Then we determine the KNNs of the two end-nodes of each segment. The three nearest restaurants of L_1 and L_2 with their distances are {(n_1, 3), (n_2, 5), (n_3, 7)} and {(n_3, 1), (n_5, 4), (n_4, 5)}, respectively. Since both end-points of the segments have different (or overlapping) set of KNN, then we know that there must be (a) split point(s) between L_1 and L_2 and that the KNNs of any point on segment (L_1, L_2) is a subset of the candidate list {n_1, n_2, n_3, n_4, n_5}. Next, we generate a new sorted list for L_1 KNNs, specifying whether the NN is increasing or decreasing using ↑and ↓symbols, respectively. The result of this step is {↑(n_1, 3), ↑(n_2, 5), ↓(n_3, 7), ↓(n_5, 10), ↓(n_4, 11)}. Note that the distances for the NN are calculated form L_1.

Step 3: Then we delete all the KNNs of the two end-nodes of each segment that are within a distance larger than 8 from the query points (L_1, and L_2). However, since n_1, n_2, and n_3 are all within a distance less than 8 from L_1 (7 and less) and n_3, n_5, and n_4 are within a distance less than 8 from L_2 (5 and less), then we do not delete them.

Step 4: for each increasing ↑ member of the set, we compare it with each decreasing ↓ one to find the location of the split points. In this example, we have 2 increasing elements (↑ (n_1, 3), ↑ (n_2, 5)) and 3 decreasing elements (↓ (n_3, 7), ↓(n_5, 10), ↓ (n_4, 11)). Therefore, we have to generate a total of 2 * 3 = 6 split points. The first split point (P_1) is generated from ↑(n_1, 3) and ↓ (n_3, 7) and is at a distance of $(7+3)/2$ - 3 = 2 from L_1. The second split point (P_2) is generated from ↑ (n_1, 3) and ↓ (n_5, 10) and is at a distance of $(10+3)/2$ -3 = 3.5 from L_1. Similarly, we calculate the rest of the split points.

Step 5: The split points generated from step 4 are sorted incrementally according to their distances to L_1. In this example, P_4 has the shortest distance to L_1, which is equal to 1, thus it is at the top of Table 1 and first in Figure 3. The third column entries represent the NNs from which the corresponding split point is generated. For example, when we generated P_1 in step 3, we compared n_1 with n_3 hence (1, 3) in the column. Similarly, to generate P_2, we compared n1 with n5, hence (1, 5) in the column. Table 1 shows the results of this step for the segment (L_1, L_2). Using this table we can solve the problem of our example. The problem was to find the continuous three nearest neighbors of a query point moving from L1 to L_4. To solve that, we started with step 1 to get Table 1. Our (modified IE) says starting from L_1 to the first split point (P_4) the 3 NNs are [n_1, n_2, n_3] sorted according to distances to L_1, from the list in step 2. Once we reach P_4, then moving toward P_1 my 3NN will change as follows:

- Look at P_4 entry in Table 1 [P_4 | 1 | (2,3)] the third column entry indicates a change in n_1 and n_3. If these neighbors where already in the list of L_1 3NN, then we change the order of elements only. My new 3NN from $P_4 \rightarrow P_1$ are the same as 3NN form $L_1 \rightarrow P_4$ except for the change of positions n_2 with n_3. The resulting 3 NN for the path $P_4 \rightarrow P_1$ are [n_1, n_3, n_2] sorted according to distances to P_4.
- Then from $P_1 \rightarrow P_5$, we look at P_1 entry in the same table and see (n_1, n_3) in the third column. If the two elements in the tuple were already in the sorted list of P_4's 3NNs, then we change their order as what happened for split point P4 above. The resulting 3NN are the same as 3NN from $L_1 \rightarrow P_4$ with a change in positions of n_1 and n_3 to get this 3NNs [n_3, n_1, n_2].
- Then from $P_5 \rightarrow P_6$,we look at P_5 entry in the same table and see (2, 5) in the third column. If one of the neighbors in the tuple is in the ordered list of P_1's 3NNs, and the other one is not, then we take out one and replace it with the other element. The resulting 3NN are the same as 3NN from $P_1 \rightarrow P_4$ with replacement of a neighbor n_5 with n_2 to get this 3NN [n_3, n_1, n_5], sorted according to distances to P_5.
- Continuing the trip to reach P_6 from P_5, the table entry for P_6 has (n_2, n_4) that are neither in P_5's 3NN list. This means that at this split point there is no change in the nearest neighbors from the ones at the pervious split point and it stays [n_3, n_1, n_5] for the interval]$P_1 \rightarrow P_5 \rightarrow P_6$[. From $P_2 \rightarrow P_3$, we find (1, 5) in Table 1, so the new 3NN are [n_3, n_5, n_1]. Finally, we reach the segment's end L_2 from P_3. Looking at (1, 4) in the table, the new 3NNs for the interval $P_3 \rightarrow L_2$ are [n_3, n_5, n_4].

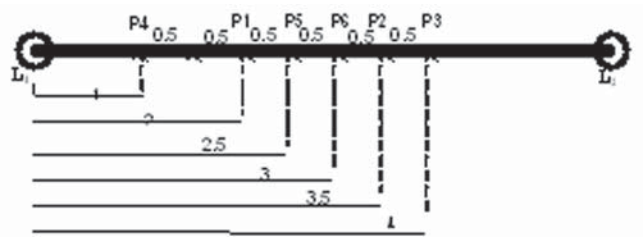

Figure 3. Split points for segment (L_1, L_2) placed on the path ordered according to their distances to L_1

As you notice, at split points P_5 and P_3, we replaced the elements of the 3NN with other elements according to the entries in Table1, thus these points are called (ESP). Furthermore, at split points P_4, P_1, P_6, and P_2 we only changed the order of neighbors as we progressed through the steps, thus these are called (OSP). Figure 4 illustrates these types of split points for the interval (L_1, L_2)

Figure 4. Two types of split points: ESP (P_5 and P_3) and OSP (P_4, P_1, P_6, and P_2) for the segment (L_1, L_2)

3.3. RCKNN extensions and enhancement

Our experimental results in [16][25] showed that VN³ failed in answering some CKNN queries and provided invalid results. Our analysis of the algorithm identified some flaws in the algorithm, especially in the cases where both end points of a line segment (road link) have a common nearest neighbor. In this case, the algorithm assumes that while moving from one end point to the other, the distance to that common nearest neighbor either increases or decreases through out the link. However, our investigation and analysis showed that this is not the case. Usually the distance increases until you reach a virtual split point (not real.) At this point, the distance gets decreased because the shortest path to the common nearest neighbor would pass through the second end point. Hence, in this section we provide a modified algorithm to resolve that problem.

An example of this type of situation is shown in Figure 5, where a moving object (e.g., a car) is traveling along the path (A,B) and we are interested in finding the first 4 closest restaurants 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 KNN. The split points specify the locations on the path where the KNN of the object change. In other words, the KNN of any object on the segment (or interval) between two adjacent split points is the same as the KNN 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.

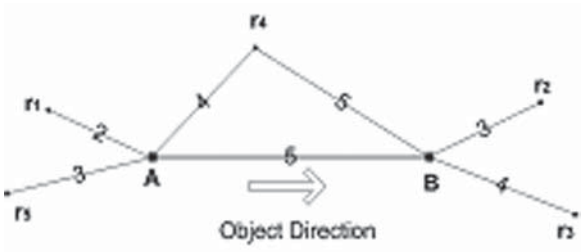

Figure 5. Example with both end points (A,B) having a common NN (r4)

In Figure 5 we have a car that travels from A to B and we want to find the 4 nearest neighbors while it is moving toward B. According to [8] the 4 NN for the car will be a subset of the 4 NN of A and 4 NN of B. Therefore, we follow the following algorithm:

Step 1: First we find the 4 NN for A and for B. 4NN for A = {(r1,2),(r5,3),(r4,4),(r2,8)}, 4NN for B = {(r2,3),(r3,4),(r4,5),(r1,7)} (assume that they are all within a specified network distance d from the query point)

Step 2: Decide which neighbors are common and which are not: Common Neighbors: r1, r4, r2, Uncommon Neighbors: r5 , r3

Step 3: For uncommon points:

- r5 (one of A four nearest neighbors) distance will always increase ↑ as the car moves from A to B
- r3 (one of B four nearest neighbors) distance will always decrease ↓ as the car moves from A to B

Step 4: For common points:

- If the shortest path goes through one of the nodes (A or B) all the time
  - r2 (shortest path will always go through B) → distance will always decrease ↓as the car moves from A to B
  - r1 (shortest path will always go through A) → distance will always increase ↑as the car moves from A to B
- If the shortest path doesn't go through one of the Nodes (A or B) all the time (e.g., at the beginning the shortest path goes through A then the shortest path goes through B). For r4 at the start the distance from the car to r4 will start with an increase ↑ since the shortest path goes through A but at some point the distance from the car to the r4 will ↓ decrease since the shortest path will go through B. This point will be the middle of (r4, A, B, r4) path. To find this point we use the following equation: X=(AC + AB + BC)/2. Hence, the distance from the start point (switch point) = X-AC

Step 5: Form a list that contains A and B 4 NN (add the distance between A and B to the B's neighbors) and add the increase, decrease and switch indicators:{(r1,2)↑, (r5,3)↑, (r4,4)↑3* or 7* ,(r2,8)↓, (r3,9)↓}. 3* means that we will switch the indicator after the car crosses 3 units from A towards B (or when the distance from r4 to the car equals 7 through A).

Step 6: Find the split point and compare it with the switch value. If it is greater than it, then do the switching and then recalculate the split point. For each split point decide whether it is an Order-Split point (OSP) (where only the order of the 4 NN changed), or it is an Element-Split point (ESP)

- Find the split point: the split points will be found when ever we have ↑ followed by ↓
- First split point between r4 and r2 and it will be at 6 and after the car crosses 2 units (which is < switch point). This split point called OSP (the first 4 nearest neighbors wont change)

Step 7: Update the list add 2 unit to pairs that have ↑ indicator and subtract 2 from pairs that have ↓indicator, The new list will be as follows: {(r1,4)↑, (r5,5)↑,(r2,6)↓, (r4,6)↑3*, (r3,7)↓}

Step 8: Go to Step 6 and 7 again

- Between r2,r5 → after .5 unit (2.5 from A)
- Between r4,r3 → after .5 unit (2.5 from A)
- The same split point in this split point one of the 4 nearest neighbors will be changed so this is ESP: {(r1,4.5)↑, (r2,5.5)↓, (r5,5.5)↑, (r3,6.5)↓, (r4,6.5)↑3*}

Step 9: Go to Step 6 and 7 again

- Between r1,r2 → after .5 unit (3 from A which = switch point).
- Between r5,r3 → after .5 unit (3 from A which = switch point).
- This split point called OSP (the first 4 nearest neighbours will not change)
- But its also called switch point where the shortest path

of the common point r4 will go through B rather than going through A and the distance of the car will decrease as the car moves toward B: {(r2,5)↓, (r1,5)↑, (r3,6)↓, (r5,5.5)↑, (r4,7) ↓}

Step 10: Repeat 6 and 7 until the first 4 pairs in the list = 4NN for B = {(r2,3),(r3,4),(r4,5),(r1,7)}, which are r2, r3, r4, r1

4. A Cache mechanism for RCKNN queries

Taken into consideration that mobile devices such as navigators are usually limited on memory resources, in this section we propose a new caching mechanism to be used within the navigator system to expedite the query response of the RCKNN queries.

4.1. Information to be cached

We assume that the following are the different kinds of information that are computed on the run in the car navigator system and thus require caching.

1. Split points for each segment (after the map has been conveniently sub-divided into segments bounded by nodes), which are a finite set of nodes.
2. KNNs for specific points of interest, which are also a finite set of nodes.
3. CKNNs or RCKNNs for a particular segment under consideration, which are also a finite set of nodes.

The pre-computed data stored in the database consists of:

1. Distances between points (to lookup from map), which are of fixed numbers.
2. List of links originating from a particular node, which are a finite set of nodes.

Note that, the terms point and node have been used interchangeably in this paper. We proceed ahead, with the above assumptions. Next we list our observations on the cache design for CKNN or RCKNN problem.

4.2. Cache design

The Cache we propose would be of a fixed-size (fraction of the client's memory), which would be capable to accommodate the heterogeneous information from the above specified categories. The cache memory performance would be hampered if we divide it into fixed/dynamic chunks each for holding a particular category of information. There could be situations in which a particular chunk overflows (thus requiring more memory), while another could lie vacant. To overcome this, we propose to store the different categories of information in the form of pre-defined records, thus eliminating any need to divide the cache. Information of each type is called an info-packet. Different structures of info-packets are used for storing KNNs, SPs and CKNNs/RCKNNs. The cache would thus store these different info-packets (order independent) till its maximum memory capacity is reached. These info-packets would then be subjected to Replacement / Eviction from the cache, based on our Cache Algorithm.

4.3. Structure of the INFO-PACKETS

Each info-packet consists of a header indicating the type of information as seen in Figure 6.

- KNN: The packet stores the INFO_TYPE, Point for which KNN is computed, number of KNNs for the point, total size required by the Packet & the list of KNNs for the point. The SIZE of the packet enables us to decide if a Packet can be accommodated into the cache or not.
- SP: The packet stores the INFO_TYPE, Line segment between which the split points are calculated, number of split points, total packet size and a list of all split points with their distances from Start of Line Segment + Split-NNs.
- CKNN or RCKNN: The storage is similar to KNNs except that instead of a single point, the entire line segment between which the CKNNs or RCKNNs lie, is stored.

INFO TYPE = SP	LINE SEGMENT	No of SPs = n	SIZE
P_1	d_1		(2,3)
P_2	d_2		(1,3)
…	…		(2,5)
P_n	d_n		(2,4)

INFO TYPE = SP	LINE SEGMENT	No of KNNs = n	SIZE
	KNN_1		
	KNN_2		
	…		
	KNN_n		

INFO TYPE = SP	LINE SEGMENT	No of CKNNs = n	SIZE
	$CKNN_1$		
	$CKNN_2$		
	…		
	$CKNN_n$		

Figure 6. Cache packets formats

We assign priorities for the three kinds of information mentioned above as : KNN > SP > CKNN/RCKNN. This is reasonable because CKNN/RCKNNs can be derived from the SPs while SPs can be derived from the KNNs. Our policy thus tries to retain as many relevant KNN packets as possible in the cache and only some fraction of SP, CKNN/RCKNN packets.

4.4. Cache replacement and eviction scenarios

An efficient Cache Replacement Strategy is required to ensure that the cache is always populated with only the relevant info-packets and evicting all obsolete ones. Let us assume that our cache contains considerable amount of info after the client traverses some distance. We discuss the cache contents at various stages.

- *Client is positioned between L_1 and L_2:* At this instant, the cache contains: (a) KNNs of L_1, (b) KNNs of L_2, (c) Split Points between L_1 and L_2, (d) CKNNs/RCKNNs for segment L_1-L_2.
- *Client completes line segment L_1-L_2 and is positioned at L_2:* At this instant the cache contains: (a) KNNs of L_1, (b) KNNs of L_2, (c) Split Points between L_1 and L_2, (d) CKNNs/RCKNNs for segment L_1-L_2. Now we need to update the cache, because the client would take a different segment. The starting point of the new segment would be L_2, but we don't know the other point. At this stage, we make the assumption that from L_2, the client would move towards the NEAREST of all the CKNNs/RCKNNs computed for segment L_1-L_2, say n_6. Based on this neighbor, we could identify the segment (containing the neighbor) and store the KNNs for the other endpoint, say L_3. But there could be situations where the client violates this normal behavior and instead of choosing the NEAREST neighbor, moves towards the next few nearest ones. To solve this problem, we sort the CKNNs/RCKNNs of segment L_1-L_2 in increasing order and pre-fetch the KNNs of as many intersection points as possible based on them. For example, if the sorted CKNNs/RCKNNs are $\{n_6,n_3,n_2\}$, we try to add the KNNs of L_3 (as n_6 lies in segment L_2-L_3) to the cache. If there's some more space available in the cache, we pre-fetch the next KNN-set of L_4 (as n_3 lies in segment L_2-L_4). Thus KNNs for most of the segments possible as pre-fetched into the cache depending on the memory available. Another observation at this point is that, if there's no space to bring in even a single new KNN-set, then the attempts to evict the cache are in the following order :
  - To first evict the CKNNs/RCKNNs stored for L_1-L_2
  - If still space not sufficient, evict the Split Points stored for L_1-L_2
  - If still space not sufficient, evict the KNNs stored for L_1.

This ordering is done based on the priority of these information types discussed earlier. Let us assume here that due to lack of space, only the CKNNs/RCKNNs for L_1-L_2 are evicted.

- *Client chooses Line Segment L_2-L_3 and moves towards L_3:* At this instant the cache contains: (a) KNNs of L_1, L_2, L_3, and those of other intersection points computed. Though the other KNNs are not of immediate use, they are retained if space exists on the cache. (b) Split Points corresponding to L_1-L_2, L_2-L_3, (c) CKNNs/RCKNNs for L_2-L_3. It should be noted at this stage that, if cache doesn't have space to accommodate the newly calculated Split points for L_2-L_3 and CKNNs/RCKNNs for L_2-L_3, then Split-Points for L_1-L_2 are evicted. If eviction still needed, then all unnecessary KNNs of other intersection points are evicted.

This is the basis of our caching algorithm that we outline below:

Caching Algorithm

```
Loop (Until client reaches the destination)
{
   Step 1: Determine client's position.
   Step 2: IF (client positioned between the segment)
                  Goto Step 3
           ELSE  //client positioned at the end point of segment)
                  Goto Step 4

   Step 3 : IF (space not available for storage)
            {
            2.a Evict any CKNNs/RCKNNs stored for other segments
            2.b Evict any Split Points stored for other segments
            2.c Evict any KNNs stored for other intersection points
            }
             ELSE //space available
            {
            2.1 Compute & Store all Split Points for the segment
            2.2 Compute & Store all CKNNs/RCKNNs for the segment
            }

   Step 4 : IF (space not available for storage)
            {
            2.a Evict any CKNNs/RCKNNs stored for other segments
            2.b Evict any Split Points stored for other segments
            2.c Evict any KNNs stored for other intersection points
            }
             ELSE //space available
            {
            2.1 Sort CKNN/RCKNNs list for previous segment
            2.2 Pre-Fetch KNNs for the nearest neighbor in the list
            2.3 IF still space exists, pre-fetch KNNs for other neighbors
            in the increasing order from the list
            }

   Step 5 : Next iteration of Loop.

   } //End of Main Loop
```

5. Experimental results

We conducted several experiments to evaluate the performance of our enhanced algorithms using PINE to solve the Restricted Continuous KNN queries. We used real-world data sets obtained from NavTech Inc., used for navigation and GPS devices installed in cars, and represent a network of approximately 110,000 links and 79,800 nodes of the road system in downtown Los Angeles. The experiments were using Oracle 9 as the database server. We used different sets of points of interest (e.g., restaurants, shopping centers, …etc.)

In the first experiment, we present the average results of 20 runs of restricted continuous K nearest neighbor queries (RCKNN) where K varied from 5 to 20, and changed the network distance *d* from 500 to 16000 meters. We calculated the number of times that the KNN query was issued to answer a RCKNN query, the number of split points on the path, and the execution time in seconds. The traveling paths source

and destination were generated randomly with random lengths, however, we made sure that we do not visit any node more than once (to avoid cycles).

We computed the average results of 20 runs of the enhanced RCKNN queries using PINE for different entities (hospitals, restaurants, …etc.) The entities have different population and cardinality ratio (i.e., the number of entities over the number of links in the network, see Table 2.) For example, in Table 3 we show the query response time with the value of K varying from 5 to 20, and for different values of K (values of {5, 10, 15, 20}) we changed the restriction distance *d* from 500 to 16000 meters. Note that the restriction distances are different for different entities, and it is increasing with lower density entities. This is due to the fact that, the lower the density of an entity the further those entities would spread on the map. Hence, we need to increase the value of the restriction distance *d* just to make sure that we get at least one solution for any query. The first column in the table specifies the restriction distance *d*. From the third column and forward, each table entry has three values (averaged over 20 runs): 1) Number of KNN queries that were issued, 2) Number of split points on the path, and 3) Execution time in seconds.

From Table 3, we conclude that the total number of KNN queries issued is almost constant (on average around 10) regardless of: 1) the density of the entities, 2) the restriction distance *d*, and 3) K value (number of nearest neighbors.) This is because, the number of issued KNN queries depends solely on the way the original path is broken into smaller segments. Those segments' end-points are either an intersection or interest points. The number of intersections in the map depend solely on the roads and not the entities, and hence fixed for all the queries issued. Furthermore, for different entities we have chosen different restriction distances (increases with lower densities), hence the number of interest points are almost fixed for the different path lengths in different entities.

Entities	Qty (density)
Hospital	46 (0.0004)
Shopping Centre	173 (0.0016)
Parks	561 (0.0053)
Schools	1230 (0.015)
Auto Services	2093 (0.0326)
Restaurants	2944 (0.0580)

Table 2. Entities population and cardinality ratio

It is also obvious from Table 3 that, as we increase the restriction distance *d*, the execution time and the number of spilt points increases. This is due to the fact that, as we increase our restriction distance, we allow more intersections and interest points to be explored, and hence increases the execution time and the number of split points. In addition, as we increase the K value, the execution time and the number of split points increases for the same reasons mentioned previously. In Figures 7-9, we show the performance of our RCKNN algorithm on one data entity type (Schools) to illustrate the behavior that we described earlier. Other entities show the same exact behavior.

Furthermore, Table 3 shows that as we decrease the density of the interest points, the execution time increases, however, the number of split points decreases (see Figures 10-12 for an example with *d* = 2000m.) Since, as we decrease the density of the entities, the interest point would spread on a larger area, and hence a larger number of points have to be explored. However, fewer split points are created, because with the decrease of the density fewer interest points would be closer to the query point, and hence fewer changes are made to the list of nearest neighbor.

In the second experiment, we studied the effect of increasing the value of K and the densities of the interest points on the total numbers of ESPs (Element Split Points) and OSPs (Object Split Points.) We present the average results of 20 runs of enhanced continuous K nearest neighbor queries (CKNN) where K varied from 2 to 10 for different entities (hospitals, restaurants, …etc.) The traveling paths from source to destination were generated randomly with random lengths (on average 2 Km.) In Table 4 we show the results for different values of K (values of {2, 4, 6, 8, 10}.) The first column in the table specifies the entity type. From the third column and forward, each table entry has three values (averaged over 20 runs): 1) Total number of Split Points (SPs), 2) Number of Element Split Points (ESPs), and 3) Number of Order Split Points (OSPs).

Table 4 shows that as we decrease the density of the interest points, the number of split points decreases for the same reasons that we discussed earlier in the results of the first experiment. In addition, as we increase the K value for the same entity, the number of split points increases for the same reasons mentioned previously. For small values of K, we get almost the same number of OSPs and ESPs in the SPs. However, we notice that as we increase the K value and the density of the interest points, the percentage of the OSPs increases dramatically over the percentage of the ESPs, and in some cases we have 94% of the SPs are OSPs (see Figures 13 and 14.) On average over all the experiments, the percentage of OSPs was 77.6% and ESPs was 22.4%. Which means that usually, the query answers (interest points) remain fixed, but their distance order to the query points changes.

In the final experiment, we studied the effect of increasing the traveling path length on the total numbers of ESPs and OSPs Points. In Table 5 we show the results for different lengths of traveling paths (values of {2, 4, 6, 8, 10} Km.) The first column in the table specifies the traveling path in kilometers. From the third column and forward, each table entry has three values (averaged over 20 runs): 1) Total number of SPs, 2) Number of ESPs, and 3) Number of OSPs.

Table 5 shows that as we increase the traveling path length, the number of SPs increases. Furthermore, the ratio of OSPs over ESPs increases dramatically (see Figures 15-16) for the same reasons that we discussed earlier in the results of the first experiment. On average over all the experiments, the number of OSPs was 6.5 times more than the ESPs, and on average they constitute of 89% of the SPs.

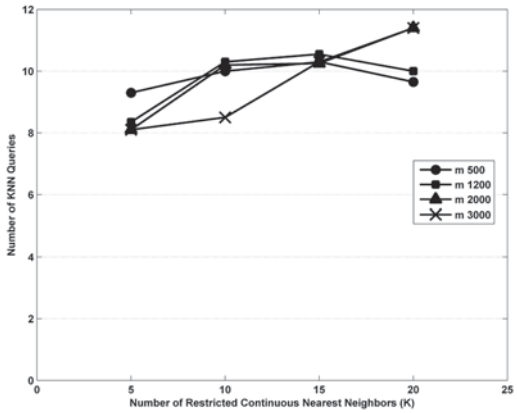

Figure 7. Number of KNN Queries Issued as K increases (Schools Entity)

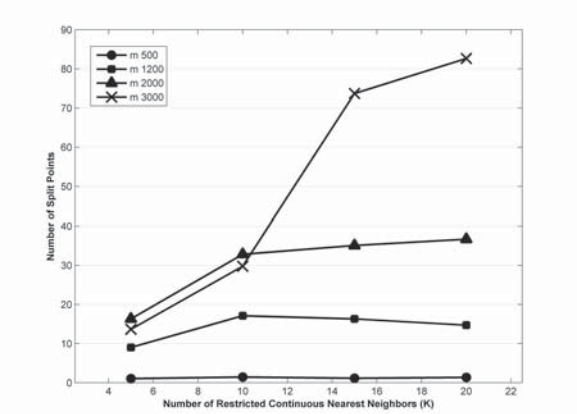

Figure 8. Number of Split Points as K increases (Schools Entity)

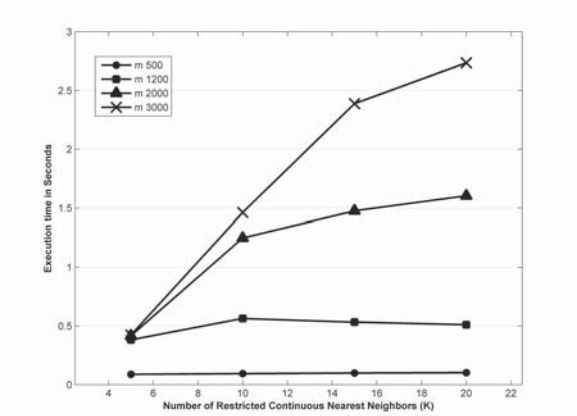

Figure 9. Execution Time in Seconds as K increases (Schools Entity)

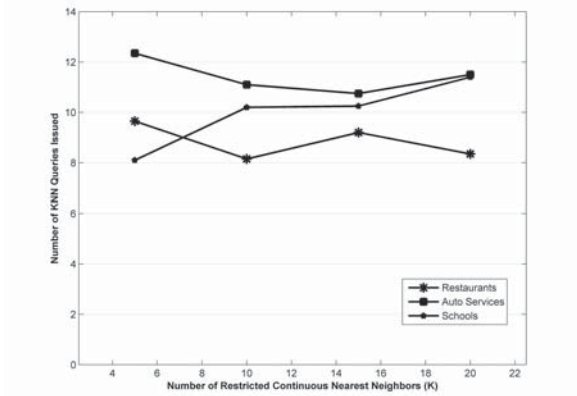

Figure 10. Number of KNN Queries Issued as K increases (three different Entities, with d = 2000m)

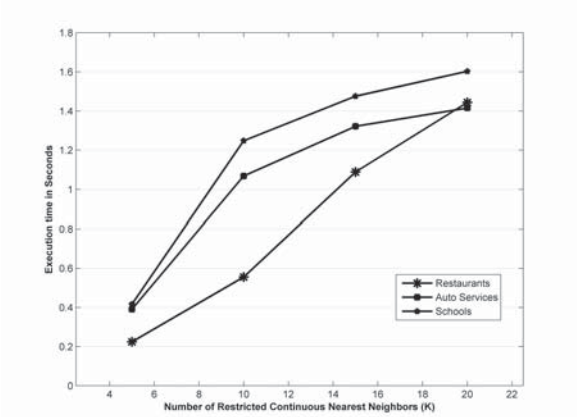

Figure 11. Execution Time (Seconds) as K increases (three different Entities, with d = 2000m)

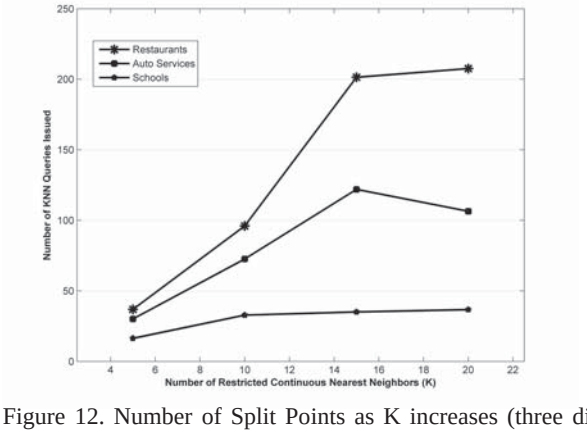

Figure 12. Number of Split Points as K increases (three different Entities, with d = 2000m)

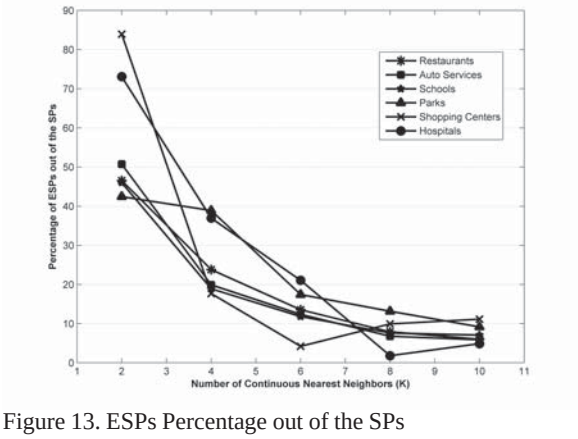

Figure 13. ESPs Percentage out of the SPs

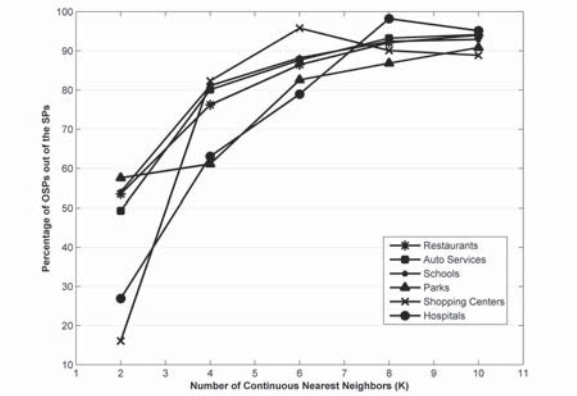

Figure 14. OSPs Percentage out of the SPs

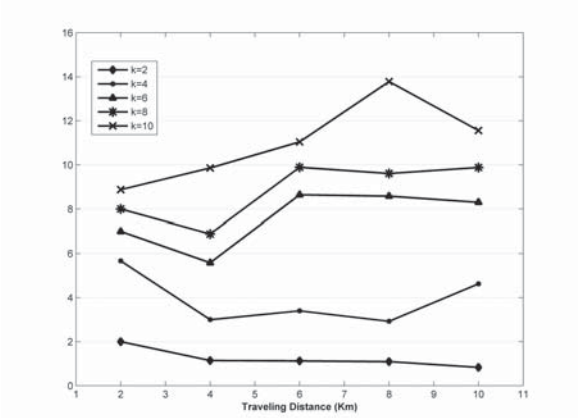

Figure 15. Effect of Increasing the Traveling Distance on the Total SPs

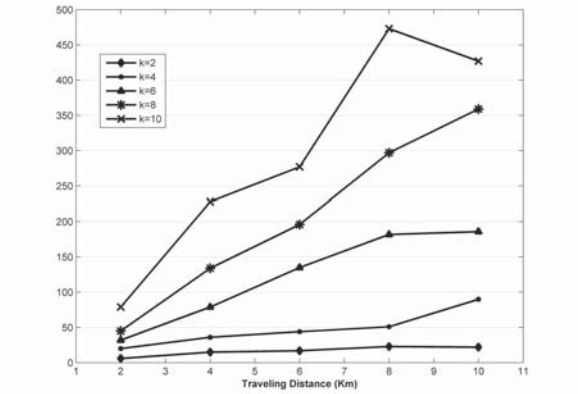

Figure 16. Effect of Increasing the Traveling Distance on the ratio of OSP/ESPs

6. Conclusion and future work

In our previous work, we proposed a novel approach, termed Progressive Incremental Network Expansion (PINE), to efficiently support NN and KNN queries. In this work, we utilize our developed PINE system to efficiently support other spatial queries such as Restricted Continuous Nearest Neighbor (RCKNN) queries. RCKNN is an important type of query that continuously finds the K nearest objects to a query point on a given path within a restricted bounding distance. The result of this type of query is a set of intervals (defined by split points) and their corresponding KNNs. This means that the KNN of an object traveling on one interval of the path remains the same all through that interval, until it reaches a split point where its KNNs change. Existing methods for RCKNN are based on Euclidean distances. In this paper we propose a new algorithm for answering RCKNN in Spatial Network Database (SNDB) where the important measure for the shortest path is network distances rather than Euclidean distances. Our solution addresses a new type of query that is plausible to many applications where the answer to the query not only depends on the distances of the nearest neighbors, but also on the user or application need. By distinguishing between two types of split points, we reduce the number of computations to retrieve the restricted continuous KNN of a moving object. This was accomplished by distinguishing between two types of split points (ESP, OSP), which reduced the number of computations to retrieve the restricted continuous KNN of a moving object. In addition, we defined a caching model to further expedite the response time for RCKNN queries.

This paper shows the road to several interesting and practical directions for future work on different spatial queries using PINE structure. Many works are redirecting the use of such queries from a scientific method to a real commercial application in several fields like telecommunication and location based services. We plan to extend our algorithms and structures to address group KNN, constrained KNN, reverse KNN, continuous RNN and group RNN queries using different caching models.

References

[1] Berchtold, S., et al. (1997). Fast Nearest Neighbor Search in High-Dimensional Space, In: Proceedings of ICDE, Orlando, Florida, USA.

[2] Bozkaya,T. et al. (1997). Distance-Based Indexing for High-Dimensional Metric Spaces, Proceedings of SIGMOD, Tucson, Arizona, USA.

[3] Chiueh, T (1994). Content-Based Image Indexing, Proceedings of VLDB, Santiago de Chile, Chile.

[4] . Ciaccia,P et al. (1997). M-tree: An Efficient Access Method for Similarity Search in Metric Spaces, Proceedings of the VLDB Journal, pp. 426-435.

[5] Corral, Y.et al. (2000). Closest pair queries in spatial databases, IN: Proceedings of ACM SIGMOD International Conference on Management of Data, Dallas, USA.

[6] Hjaltason, G.R.et al. (1999). Distance Browsing in Spatial Databases, In: Proceedings of TODS, No. 2, p. 265-318, vol. 24.

[7] Jung,S. et al. (2002). An Efficient Path Computation Model for Hierarchically Structured Topological Road Maps, IEEE Transaction on Knowledge and Data Engineering.

[8] Kolahdouzan, M.et al. (2004). Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases, Proceedings of VLDB.

[9] Kolahdouzan, M.et al. (2004). Continuous K Nearest Neighbor Queries in Spatial Network Databases, In:

Proceedings of the Second Workshop on SpatioTemporal Database Management STDBM.

[10] Kollios, G. et al. (1999). Nearest Neighbor Queries in a Mobile Environment, Proceedings of the International Workshop on Spatio-Temporal Database Management, p. 119–134.

[11] Korn,F et al. (1996). Fast Nearest Neighbor Search in Medical Image Databases, Proceedings of VLDB, Mumbai, India.

[12] Okabe, A. et al. (2000). Spatial Tessellations, Concepts and Applications of Voronoi Diagrams.John Wiley and Sons Ltd., 2nd edition.

[13] Papadias, D et al. (2003). Query Processing in Spatial Network Databases", Proceedings of VLDB: 802-813.

[14] Rigaux, P.et al. (2002). Spatial Databases with Applications to GIS, Morgan Kaufmann.

[15] Roussopoulos, Net al. (1995).Nearest Neighbor Queries, Proceedings of SIGMOD, San Jose, California.

[16] Safar, M. (2005).K Nearest Neighbor Search in Navigation Systems, Journal of Mobile Information Systems (MIS), IOS Press.

[17] Saltenis, S.et al. (2000). Indexing the Positions of Continuously Moving Objects, Proceedings of ACM SIGMOD.

[18] Seidl,T. et al. (1998). Optimal Multi-Step k-Nearest Neighbor Search", Proceedings of SIGMOD, Seattle, Washington, USA.

[19] Shahabi, C.et al. (2002). A Road Network Embedding Technique for k-Nearest Neighbor Search in Moving Object Databases, Proceedings of ACMGIS, McLean, VA, USA.

[20] Song, Z.et al. (2001). Nearest Neighbor Search for Moving Query Point,In: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases.

[21] Tao, Y.et al. (2002). Continuous Nearest Neighbor Search, Proceedings of VLDB, Hong Kong, China.

[22] Tao, Yet al. (2004).Reverse kNN Search in Arbitrary Dimensionality, Proceedings of 30th Very Large Data Bases (VLDB), p. 744-755, Toronto, Canada, August 29-September 3.

[23] Yiu, M.Let al. (2005).Reverse Nearest Neighbor in Large Graphs,In: Proceedings of ICDE 2005: 186-187.

[24] Yu, Cet al. (2001). Indexing the Distance: An Efficient Method to KNN Processing, Proceedings of the Very Large Data Bases Conference (VLDB).

[25] Safar, M (2006). Enhanced Continuous KNN Queries Using PINE on Road Networks. Proceedings of the 1st International Conference on Digital Information Management (ICDIM).

Entity : Restaurants

Restriction distance *d*		k=5	k=10	k=15	k=20
500 m	# KNN Queries	10.5	8.75	7.65	7.05
	# of split points	19.7	21.9	21.3	20.95
	Execution Time	0.16725	0.1561	0.1522	0.14625
1000 m	# KNN Queries	8.75	8.95	9	8.4
	# of split points	36.2	84.35	107.45	88.25
	Execution Time	0.1953	0.5282	0.61495	0.6102
1500 m	# KNN Queries	8.8	8.15	8.2	7.1
	# of split points	36.7	87.35	160.1	140.6
	Execution Time	0.2032	0.5358	0.87665	0.89225
2000 m	# KNN Queries	9.65	8.15	9.2	8.35
	# of split points	36.85	95.95	201.35	207.55
	Execution Time	0.2242	0.55535	1.0884	1.4436

Entity : Auto Service

Restriction distance d		k=5	k=10	k=15	k=20
500 m	# KNN Queries	10.75	11.35	12.05	11.55
	# of split points	5.95	6.35	5.7	5.5
	Execution Time	0.23985	0.1421	0.153	0.14295
1000 m	# KNN Queries	10.25	10.7	11.6	10.95
	# of split points	24.55	43	49.4	41.25
	Execution Time	0.3039	0.5275	0.5438	0.54455
1500 m	# KNN Queries	11	9.3	10.6	11.35
	# of split points	26.95	56.4	70.45	69.7
	Execution Time	0.34365	0.8126	0.9561	0.97205
2000 m	# KNN Queries	12.35	11.1	10.75	11.5
	# of split points	30.05	72.55	121.9	106.4
	Execution Time	0.39065	1.06805	1.32185	1.4155

Entity : Schools

Restriction distance d		k=5	k=10	k=15	k=20
500 m	# KNN Queries	9.3	10	10.3	9.65
	# of split points	1.05	1.45	1.15	1.35
	Execution Time	0.0889	0.09455	0.09925	0.10245
1200 m	# KNN Queries	8.35	10.3	10.55	10
	# of split points	9	17.1	16.3	14.7
	Execution Time	0.382	0.5639	0.5328	0.511
2000 m	# KNN Queries	8.1	10.2	10.25	11.4
	# of split points	16.3	32.8	35.05	36.65
	Execution Time	0.41625	1.24825	1.47565	1.603
3000 m	# KNN Queries	8.1	8.5	10.3	11.4
	# of split points	13.65	29.75	73.7	82.65
	Execution Time	0.42355	1.46195	2.3875	2.7352

Entity : Parks

Restriction distance d		k=5	k=10	k=15	k=20
1500 m	# KNN Queries	12.15	11.7	11.1	11.7
	# of split points	0.65	0.5	0.75	0.5
	Execution Time	0.6514	0.51165	0.4459	0.47345
2500 m	# KNN Queries	11.5	11.1	12.6	11.55
	# of split points	1.85	1.6	1.65	1.7
	Execution Time	1.1109	1.0429	1.1968	1.01085
4000 m	# KNN Queries	11.4	12.6	12.6	11.45
	# of split points	6.05	27.8	30.15	29.3
	Execution Time	1.4224	3.7874	3.85635	3.46725
5500 m	# KNN Queries	10.8	10.85	12.6	12
	# of split points	6.3	34.45	96.3	115.4
	Execution Time	1.4633	3.4244	6.9976	7.7477

Entity : Shopping Centers

Restriction distance d		k=5	k=10	k=15	k=20
3000 m	# KNN Queries	13.35	11.9	12.25	11.1
	# of split points	0.25	0.7	0.4	0
	Execution Time	0.5366	0.50695	0.485	0.47735
5500 m	# KNN Queries	11	10.8	11.3	10.45
	# of split points	7.65	10.1	11.7	10.95
	Execution Time	2.83665	5.5867	6.0289	5.94985
7500 m	# KNN Queries	10.75	10.8	11	11.6
	# of split points	7.9	23.9	48.2	55.65
	Execution Time	2.60245	9.38455	16.7039	18.1602
9500 m	# KNN Queries	10.35	11.35	11.75	10.85
	# of split points	6.75	25.25	60.2	60.15
	Execution Time	2.7306	9.6265	18.1976	20.8821

Entity : Hospitals

Restriction distance d		k=5	k=10	k=15	k=20
6000 m	# KNN Queries	10.5	12	11.5	12.1
	# of split points	0.5	0.9	1.05	1.45
	Execution Time	1.8438	1.9781	2.0062	2.2835
9000 m	# KNN Queries	11.9	11.45	12.1	12.1
	# of split points	1.85	2.55	2.7	2.55
	Execution Time	16.64745	16.50485	18.1469	18.7539
13000 m	# KNN Queries	11.85	11.3	10.95	11.5
	# of split points	2.2	5.1	6.95	6.15
	Execution Time	16.07205	27.74845	28.5898	29.07655
16000 m	# KNN Queries	12.25	13.4	12.65	11.75
	# of split points	2.7	6.85	18.95	28.2
	Execution Time	16.93755	33.5095	54.21405	120.6689

Table 3. RCKNN query performance using PINE

Entities		**k=2**	**k=4**	**k=6**	**k=8**	**k=10**
Restaurants	# of split points	7.95	33	39.9	60.8	156.9
	# of ESP	3.7	7.85	5.4	4.8	9.4
	# of OSP	4.25	25.15	34.5	56	147.5
Auto Services	# of split points	7	32.15	45.75	76	83.75
	# of ESP	3.55	6.4	5.65	5.15	4.9
	# of OSP	3.45	25.75	40.1	70.85	78.85
Schools	# of split points	5	11.9	22.45	31.65	50.3
	# of ESP	2.3	2.25	2.65	2.4	3.55
	# of OSP	2.7	9.65	19.8	29.25	46.75
Parks	# of split points	1.65	8.1	21.25	42.6	79.65
	# of ESP	0.7	3.15	3.7	5.6	7.3
	# of OSP	0.95	4.95	17.55	37	72.35
Shopping Centers	# of split points	2.8	12.45	17.85	18.15	28.3
	# of ESP	2.35	2.2	0.75	1.8	3.15
	# of OSP	0.45	10.25	17.1	16.35	25.15
Hospitals	# of split points	1.3	2.3	7.6	8.3	11.3
	# of ESP	0.95	0.85	1.6	0.15	0.55
	# of OSP	0.35	1.45	6	8.15	10.75

Table 4. Split Points (SP, ESP, and OSP) using CKNN with PINE

Entities : **Restaurants**

Distance		k=2	k=4	k=6	k=8	k=10
2 Km	# of split points	8	35	80	130	181
	# of ESP	6	10	7	10	14
	# of OSP	2	25	73	120	167
4 Km	# of split points	17	77	165	266	344
	# of ESP	12	19	18	21	24
	# of OSP	5	58	147	245	320
6 Km	# of split points	21	92	232	356	479
	# of ESP	14	23	24	27	29
	# of OSP	7	69	208	329	450
8 Km	# of split points	26	110	275	438	621
	# of ESP	16	27	30	35	39
	# of OSP	10	83	245	403	582
10 Km	# of split points	29	121	297	469	663
	# of ESP	18	30	31	36	41
	# of OSP	11	91	266	433	622

Entities : **Auto Services**

Distance		k=2	k=4	k=6	k=8	k=10
2 Km	# of split points	10	36	80	109	145
	# of ESP	4	5	10	11	11
	# of OSP	6	31	70	98	134
4 Km	# of split points	17	65	120	183	286
	# of ESP	8	12	18	15	23
	# of OSP	9	53	102	168	263
6 Km	# of split points	22	96	211	330	426
	# of ESP	12	16	21	27	30
	# of OSP	10	80	190	303	396
8 Km	# of split points	25	85	195	389	577
	# of ESP	13	19	26	29	37
	# of OSP	12	66	169	360	540
10 Km	# of split points	35	113	210	364	724
	# of ESP	16	22	30	31	53
	# of OSP	19	91	180	333	671

Entities : **Schools**

Distance		k=2	k=4	k=6	k=8	k=10
2 Km	# of split points	6	20	32	45	79
	# of ESP	2	3	4	5	8
	# of OSP	4	17	28	40	71
4 Km	# of split points	15	36	79	134	228
	# of ESP	7	9	12	17	21
	# of OSP	8	27	67	117	207
6 Km	# of split points	17	44	135	196	277
	# of ESP	8	10	14	18	23
	# of OSP	9	34	121	178	254
8 Km	# of split points	23	51	182	297	473
	# of ESP	11	13	19	28	32
	# of OSP	12	38	163	269	441
10 Km	# of split points	22	90	186	359	427
	# of ESP	12	16	20	33	34
	# of OSP	10	74	166	326	393

Entities : **Parks**

Distance		k=2	k=4	k=6	k=8	k=10
2 Km	# of split points	1	8	28	60	83
	# of ESP	0	4	7	7	10
	# of OSP	1	4	21	53	73
4 Km	# of split points	4	11	44	98	124
	# of ESP	3	4	7	9	13
	# of OSP	1	7	37	89	111
6 Km	# of split points	7	35	70	127	215
	# of ESP	4	7	9	12	16
	# of OSP	3	28	61	115	199
8 Km	# of split points	7	38	87	179	251
	# of ESP	4	10	11	16	17
	# of OSP	3	28	76	163	234
10 Km	# of split points	9	60	156	244	281
	# of ESP	6	17	21	24	25
	# of OSP	3	43	135	220	256

Entities : **Shopping Centers**

Distance		k=2	k=4	k=6	k=8	k=10
2 Km	# of split points	4	15	21	25	39
	# of ESP	3	3	0	1	4
	# of OSP	1	12	21	24	35
4 Km	# of split points	6	22	28	29	71
	# of ESP	3	5	3	4	8
	# of OSP	3	17	25	25	63
6 Km	# of split points	7	24	57	65	103
	# of ESP	6	7	3	8	9
	# of OSP	1	17	54	57	94
8 Km	# of split points	9	29	57	108	138
	# of ESP	6	8	6	9	9
	# of OSP	3	21	51	99	129
10 Km	# of split points	9	36	60	107	162
	# of ESP	7	11	8	11	13
	# of OSP	2	25	52	96	149

Entities : **Hospitals**

Distance		k=2	k=4	k=6	k=8	k=10
2 Km	# of split points	1	4	7	13	16
	# of ESP	1	2	1	0	0
	# of OSP	0	2	6	13	16
4 Km	# of split points	2	8	16	22	25
	# of ESP	2	3	3	0	5
	# of OSP	0	5	13	22	20
6 Km	# of split points	4	10	17	23	44
	# of ESP	3	2	3	0	4
	# of OSP	1	8	14	23	40
8 Km	# of split points	5	21	33	67	69
	# of ESP	3	5	5	3	5
	# of OSP	2	16	28	64	64
10 Km	# of split points	8	22	38	69	82
	# of ESP	6	6	6	3	7
	# of OSP	2	16	32	66	75

Table 5. Split Points (SP, ESP, and OSP) using CKNN with PINE for Different Traveling Distances