Journal of Universal Computer Science, vol. 14, no. 4 (2008), 590-611
submitted: 16/10/06, accepted: 18/2/08, appeared: 28/2/08 © J.UCS

Spatial Queries in Road Networks Based on PINE

Maytham Safar
(Kuwait University, Kuwait
maytham@eng.kuniv.edu.kw)

Abstract: Over the last decade, due to the rapid developments in information technology (IT), a new breed of information systems has appeared such as geographic information systems that introduced new challenges for researchers, developers and users. One of its applications is the car navigation system, which allows drivers to receive navigation instructions without taking their eyes off the road. 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. Several types of spatial queries (e.g., nearest neighbour - NN, K nearest neighbours – KNN, continuous k nearest neighbours – CKNN, reverse nearest neighbour – RNN) have been proposed and studied in the context of spatial databases. 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 NN and KNN queries. In this work, we utilize our developed PINE system to efficiently support other spatial queries such as CKNN. The continuous K nearest neighbour (CKNN) query is an important type of query that finds continuously the K nearest objects to a query point on a given path. We focus on moving queries issued on stationary objects in Spatial Network Database (SNDB) (e.g., continuously report the five nearest gas stations while I am driving.) 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 travelling 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 CKNN are based on Euclidean distances. In this paper we propose a new algorithm for answering CKNN in 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 neighbours, 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 continuous KNN of a moving object. We compared our algorithm with CKNN based on VN^3 using IE (Intersection Examination). Our experiments show that our approach has better response time than approaches that are based on IE, and requires fewer shortest distance computations and KNN queries.

Keywords: Nearest Neighbor, Continuous Nearest Neighbor, Road Network, Voronoi, PINE.
Categories: E.1, E.2, H.3.3, H.5.1

1 Introduction

Over the last decade, due to the rapid developments in information technology (IT), particularly communication technologies, a new breed of information systems has appeared such as mobile information systems. Mobility is perhaps the most important market and technological trend within information and communication technology.

Mobile information systems will have to supply and adopt services that go beyond traditional web-based systems, and hence they come with new challenges for researchers, developers and users.

One of the well-known applications that depend on mobility is the car navigation system, which allows drivers to receive navigation instructions without taking their eyes off the road. Using a Global Positioning System (GPS) in the car navigation system enables the driver to perform a wide manner of queries, from locating the car position, to finding a route from A to B, or dynamically selecting the best route in real time.

Several types of spatial queries (e.g., nearest neighbour - NN, K nearest neighbours – KNN, continuous nearest neighbour - CNN, continuous k nearest neighbours – CKNN, reverse nearest neighbour – RNN) 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 neighbours 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 neighbours search increases substantially if we consider other variations of it such as the Continuous KNN (CKNN.) CKNN query is defined as the K nearest point of interest to every point on a path, and has found applications in the GISs. For example, in Figure 1, for Car2 find the nearest 3 restaurants at any point during its route from its location to reach P12. 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 travelling on one interval of the path remain the same all through that interval, until it reaches a split point where its KNNs change.

With 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 distance between objects (e.g., the vehicle and the restaurants) depend on the connectivity of the network rather than the objects' location. Taken also into consideration that Mobile devices are usually limited on memory resources and have lower computational power, efficient algorithms for distance computation are critical for query processing in such real time systems.

In [Safar, 05] we proposed a novel approach (PINE) 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. In this work, we use PINE to efficiently address CKNN and RNN queries

in SNDBs. With RNN, given a set of spatial objects (or points of interest, e.g., restaurants), and a query point (e.g., vehicles' location), find the restaurants that consider that vehicle as their nearest neighbour. For example, in Figure 1, the restaurant P2 has Car3 as its RNN.

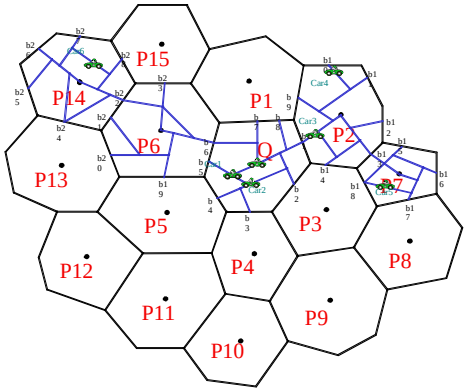

Figure 1: CKNN and RNN Queries

In solving CKNN queries, it is important to note that issuing a traditional nearest neighbour 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 neighbours of an object, only when the two neighbours 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 neighbours requested by a CNN query. When only the first nearest neighbour is requested (e.g., finding only the closest restaurant to a vehicle while it is travelling), 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. In the case when more than one neighbour is requested by CNN query (i.e., CKNN) 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 [Kolahdouzan, 04b]. 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.

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 neighbours and not their ordered distances (i.e., being the first nearest neighbour or the second.) For example, suppose that as we are travelling 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 favourite 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 [Kolahdouzan, 04b], 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 [Safar, 05] 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 neighbour. In this case, the algorithm assumes that while moving from one end point to the other, the distance to that common nearest neighbour 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 neighbour would pass through the second end point. Hence, in this paper we provide a modified algorithm to resolve that problem.

The remainder of this paper is organized as follows. Section 2 provides a related work study. Section 3 provides a background on some definitions and algorithms used in this work. Then, we describe how to answer KNN queries using PINE in section 4. In sections 5 and 6 we discuss our approaches to solve CKNN using PINE, respectively. Section 7 provides our experimental results. Finally, we conclude our work in section 8.

2 Related work

The most common type of query encountered in spatial databases is the point k nearest neighbour (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 Continuous KNN and the Reverse NN. In this section, we overview previous work related to KNN query, and its variations.

Most of the existing work [Roussopoulos, 95][Korn, 96][Seidl, 98] consider Cartesian (typically, Euclidean) spaces, where the distance between two objects is determined by their relative position in space. The current algorithms for computing the distance between a query object q and an object O in a network will automatically lead to the computation of the distance between q and the objects that are (relatively) closer to q than O. The advantage of these approaches is that they explore the objects that are closer to q and compute their distances to q progressively. The major disadvantage with the approaches 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. Hence, they perform poorly when the objects are not densely distributed in the network since then they require a large portion of the network to be retrieved for distance computation. In this work, 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 [Papadias, 03][Kolahdouzan, 04][Safar, 05] support the exact KNN queries on spatial network databases.

The solution in [Papadias, 03], 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 [Kolahdouzan, 04] 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 [Safar, 05] 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 CKNN and RNN 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., [Tao, 02]), which are not applicable to SNDBs.

For example, the approach proposed in [Tao, 02] 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 [Tao, 02] 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, [Kolahdouzan, 04b] 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 [Kolahdouzan, 04]: 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 [Safar, 05]). To the best of our knowledge, [Kolahdouzan, 04b] is the only approach that uses network distances to find CKNN.

3 Background

Our proposed approaches to address the spatial queries are based on PINE algorithm, network Voronoi diagram and Dijkstra's algorithm. 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. Dijkstra's algorithm provides one of the most efficient algorithms that finds shortest paths from the source node to all the other nodes. In [Papadias, 03][Safar, 05] Dijkstra's was preferred over the other famous shortest path algorithm (A*) [Kung, 86] because of the way that it computes the shortest path distance by expanding from the source towards destination. In addition, it uses a queue to store a sorted list of the recently visited nodes instead of applying a heuristics to prune the search space and direct the graph expansion like in A* algorithm.

In this section, we review the principles of the Voronoi diagrams. We start with the Voronoi diagram for 2-dimensional Euclidean space and present only the properties that are used in our approach. We then discuss the network Voronoi diagram where the distance between two objects in space is their shortest path in the network rather than their Euclidean distance and hence can be used for spatial networks. Then, we talk about PINE algorithm. A thorough discussion on Voronoi diagrams is presented in [Okabe, 00][Safar, 05].

3.1 Voronoi diagrams

Imagine you are looking for a school for your kid. Among the criteria to be considered will be the length of the way to school. If you formulate this as a spatial analysis problem, you are looking for the school that is closest to your home, among all schools in your city. The classical approach to solve this spatial analysis problem is the Voronoi diagram. The Voronoi diagram isolates the area that is closest to each school. The Voronoi diagram of a point set P, VD(P), is a unique diagram that consists of a set of collectively exhaustive and mutually exclusive Voronoi polygons (Voronoi cells), VPs. Each Voronoi polygon is associated with a point in P (called generator point) and contains all the locations in the Euclidean plane that are closer to

the generator point of the Voronoi cell than any other generator point in P. The boundaries of the polygons, called Voronoi edges, are the set of locations that can be assigned to more than one generator. The Voronoi polygons that share the same edges are called adjacent polygons and their generators are called adjacent generators. The following property holds for any Voronoi diagram and is used to answer KNN queries: " The nearest generator point of pi (e.g., pj) is among the generator points whose Voronoi polygons share similar Voronoi edges with VP(pi)." (see [Okabe, 00][Kolahdouzan, 04a] for further details). In general, a Voronoi diagram of a set of "sites" (points) is a collection of regions that divide up the plane. Each region corresponds to one of the sites, and all the points in one region are closer to the corresponding site than to any other site.

3.2 Network voronoi diagrams

Sometimes the approach is of limited value, especially if the possibilities to move in space are limited to one or several networks. In this case, the above described method gives only rough estimates and might even be significantly wrong. Several assumptions of the Voronoi diagram are violated in urban areas; distances between two addresses are not Euclidean; they have to be measured along the travel network(s). If your son has to walk around a block of buildings, the way to school can be significantly longer than the Euclidean distance. Thus, we use the Network Voronoi diagram [Okabe, 00]. "A network Voronoi diagram, termed NVD, is defined for 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 the distance between objects is defined as their shortest path in the network rather than their Euclidean distance." [Kolahdouzan, 04b][Roussopoulos, 95]. Spatial networks (e.g., road networks) can be modeled as weighted planar graphs where nodes of the graph represent the intersections and roads are represented by the links connecting the nodes.

The Network Voronoi diagram considers distances only in networks, not in the plane. It divides the network, not the space, into Voronoi cells. A Voronoi cell in a network is the set of nodes and edges that are closer to one Voronoi generator (here, a school) than to any other. For the construction of the Network Voronoi diagram an algorithm is used based on the shortest path algorithm of Dijkstra. Dijkstra's algorithm calculates in a connected network the shortest path from a selected start node to any other node in the network (see [Kolahdouzan, 04b][Roussopoulos, 95] for further details).

4 K nearest neighbor (knn) queries using pine

Taken into consideration that Mobile devices are usually limited on memory resources and have lower computational power, in [Safar, 05] 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 centred 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 neighbour-cell node-pairs. Now, to find the k nearest-neighbours of a query object q, we first find the first nearest neighbour 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 neighbour (its Voronoi cell centre point), and 2) explore the objects that are close to q (centres of surrounding Voronoi cells) and compute their distances to q during the expansion.

At the first scale, a network expansion similar to Incremental Network Expansion (INE) [Papadias, 03] is performed inside the Voronoi cell that contains q (VC(q)) starting from q. To this end, we utilize the actual network links (e.g., roads) and nodes (e.g., restaurants, hospitals) to compute the distance from q (e.g., vehicle) to its first nearest neighbour (the generator point of VC(q)) and the border points of VC(q). When we reach a border point of VC(q), we start a second network expansion at the Voronoi polygons scale. Unlike INE and similar to Voronoi-based Network Nearest Neighbour (VN^3) [Kolahdouzan, 04], the second expansion utilizes the inter-cell pre-computed distances to find the actual network distance from q to the objects in the other Voronoi cells surrounding VC(q). Note that both expansions are performed simultaneously. The first expansion continues until all border points of VC(q) are explored or all KNN are found.

5 Continuous k nearest neighbor (cknn) queries using pine

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. 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 $\{n_1, \ldots, n_8\}$) to the object at any given point on the path. The result of a 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 CKNN queries in spatial network databases. We first present our approach for the scenarios when only the first NN is desired (i.e., CNN), and then for the cases where the CKNN of any point on a given path is requested.

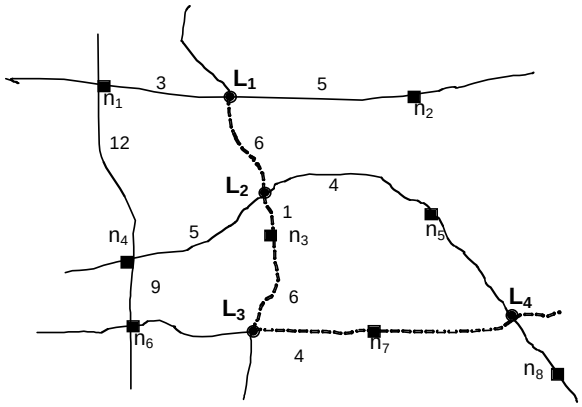

Figure 2: Example of Continuous K Nearest Neighbour Query

5.1 Continuous 1nn queries using pine

Our solution for C-NN queries is based on our previous work PINE that partitions the network into disjoint first order network Voronoi polygons (NVP) [Safar, 05] in such a way that the first nearest neighbor of any point inside a polygon is the generator of that polygon. To find the CNN 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 continuous NN for every point in a segment inside a polygon is the generator of that polygon. However, this approach cannot be extended to CKNN queries because the NVD is a first order network diagram that can only specify the first NN.

5.2 Continuous knn queries using pine

Our algorithm for finding the continuous KNN of any point on a path, starts by breaking the path into smaller segments according to some properties, then finding the continuous KNN for each segment, and finally, generating the result set for the entire path by joining the results for all segments. It has been shown that there must be a split point on the shortest path between the segments' nodes if the end- nodes have different KNNs [Kolahdouzan, 04b]. Otherwise, the set of continuous KNN would remain fixed on that segment. 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 [Kolahdouzan, 04b] 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) [Safar, 05]. It has been shown that the 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 [Kolahdouzan, 04b], 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: 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 [Kolahdouzan, 04b]. 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 4: 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 3.

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):
   - Find *k*NN (L_y) and *k*NN(L_{y+1}) using PINE
   - Find the directions of *k*NNs of the start of the segment (L_y)
   - Find the location of the split points for the segment (L_y, L_{y+1})

Figure 3: Pseudo code for modified IE algorithm

Given the table, one can easily find the 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 KNN stay the same as the object moves until it reaches the first split point where the KNN 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) neighbor 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.

In [Kolahdouzan, 04b], 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.

Split Point	Distance to L1	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 points and Split-NN for segment (L1, L2) of Figure 2

To illustrate our technique, we use the following example: suppose that in Figure 2, we are interested to find the three closest neighbors to any point on the path (L_1, L_2, L_3, L_4). 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: 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 4: The split points generated from step 2 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 4. 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 n_1 with n_5, hence (1, 5) in the column. Table 1 shows the results of this step for the segment (L_1, L_2).

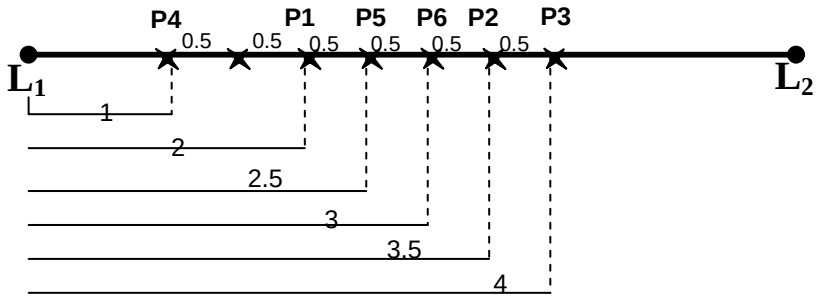

Figure 4: Split points for segment (L1, L2) placed on the path ordered according to their distances to L1

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 L_1 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 → P_1 are the same as 3NN form L_1 → P_4 except for the change of positions n_2 with n_3. The resulting 3 NN for the path P_4 → P_1 are [n_1, n_3, n_2] sorted according to distances to P_4.
- Then from P_1→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 P_4 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].

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 5 illustrates these types of split points for the interval (L_1, L_2).

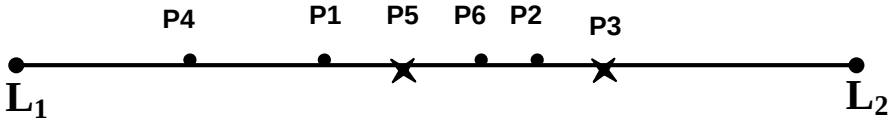

Figure 5: Two types of split points: ESP (P5 and P3) and OSP (P4, P1, P6, and P2) for the interval (L1, L2)

5.3 Cknn extensions and enhancement

Our experimental results in [Safar, 05] 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 neighbour. In this case, the algorithm assumes that while moving from one end point to the other, the distance to that common nearest neighbour 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 neighbour 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 6, where a moving object (e.g., a car) is travelling 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.

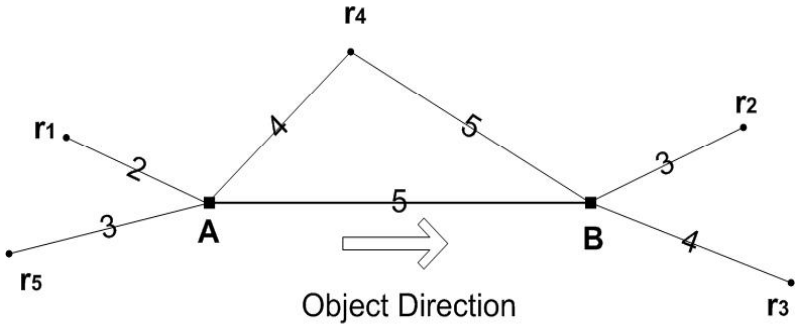

Figure 6: Example with both end point (A,B) having a common NN (r4)

In Figure 6 we have a car that travels from A to B and we want to find the 4 nearest neighbours while it is moving toward B. According to [Kolahdouzan, 04a] 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)}

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

Step 3: For uncommon points:

- r5 (one of A 4 nearest neighbours) distance will always increase ↑ as the car moves from A to B
- r3 (one of B 4 nearest neighbours) 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 neighbours) 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 neighbours 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 neighbours 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

6 Experimental results

We conducted several experiments to compare the performance of our enhanced algorithms using PINE to solve Continuous KNN queries to that of continuous KNN on VN^3 (IE). 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 present the average results of 100 runs of continuous K nearest neighbor queries where K varied from 1 to 20.

6.1 Continuous knn using pine

We conducted several experiments to evaluate the performance of the enhanced Continuous KNN queries with PINE structure using different sets of points of interest (e.g., restaurants, shopping centers, …etc.). We calculated the number of times that the KNN query must be issued and the execution time in seconds, for different values of K and assumed that the length of the travelling paths are equal to 4 km. The traveling paths were generated randomly, however, we made sure that we do not visit any node more than once (to avoid cycles).

In Table 2, we present the average results of 100 runs of the enhanced continuous K nearest neighbor queries using PINE. For example, in the table below we show the query response time when the length of the traveling path is 4 km and the value of K varies from 1 to 20. The first and second columns specify the entities (or points of interest) and their population and cardinality ratio (i.e., the number of entities over the number of links in the network), respectively. From the third column and forward, each table entry has two values (averaged over 100 runs): 1) Number of KNN queries that were issued and 2) Execution time in seconds.

Travelling Path = 4Km Averaged over 100 queries		K=1	K=3	K=5	K=10	K=20
Entities	Qty (density)	#KNN Queries	#KNN Queries	#KNN Queries	#KNN Queries	#KNN Queries
		Execution Time	Execution Time	Execution Time	Execution Time	Execution Time
Hospital	46 (0.0004)	27	26	26	25	24
		16	39	58	102	198
Shopping Centre	173 (0.0016)	26	26	25	25	25
		9	27	34	75	142
Parks	561 (0.0053)	26	25	25	24	24
		4	9	17	26	49
Schools	1230 (0.015)	23	24	23	23	23
		2	5	6	14	23
Auto Services	2093 (0.0326)	23	24	23	23	23
		2	5	6	14	23
Restaurants	2944 (0.0580)	21	22	21	23	22
		1	2	3	6	9

Table 2: Performance of our enhanced algorithm on PINE to solve CKNN queries.

6.2 Continuous knn using vn^3 (IE)

We conducted several experiments to evaluate the performance of the Continuous KNN queries with VN^3 structure (IE) using different sets of points of interest (e.g., restaurants, shopping centers, …etc.) and for different values of K (values of {1, 3, 5, 10, 20}). We used traveling paths of length 4 km. In Table 3, we present the average results of 100 runs of IE for different values of K varying from 1 to 20.

Travelling Path = 4Km Averaged over 100 queries		K=1	K=3	K=5	K=10	K=20
Entities	Qty (density)	#KNN Queries	#KNN Queries	#KNN Queries	#KNN Queries	#KNN Queries
		Execution Time	Execution Time	Execution Time	Execution Time	Execution Time
Hospital	46 (0.0004)	25.9	31.95	33.03	43.0	-
		2.27	197.72	595.35	1939.88	-
Shopping Centre	173 (0.0016)	26.51	32.47	42.22	51.93	61.54
		2	26.93	77.16	418.77	1729.39
Parks	561 (0.0053)	29.44	39.75	48.16	60.39	70.06
		2.36	16.69	32.51	87.48	226.11
Schools	1230 (0.015)	30.96	44.58	50.18	64.61	64.73
		3.55	14.43	26.33	66.72	161.87
Auto Services	2093 (0.0326)	34.64	45.72	50.54	64.99	65.7
		6.56	36.73	73.98	182.23	386.11
Restaurants	2944 (0.0580)	35.44	47.19	54.2	58.9	-
		11.58	54.57	106.14	241.58	-

Table 3: Performance of Continuous KNN queries using VN^3.

6.3 Analysis

From Table 2, Table 3, and Figure 7, we conclude that the total query response time of CKNN (based on PINE) is better than the query response time of CKNN (based on VN^3). On average CKNN-PINE is 9 times faster than CKNN-VN^3. This is because PINE requires less expensive shortest path computations, and as shown in (Safar, 2005) VN^3 requires 8.5 times the number of computations required by PINE to compute any required KNN. As the number of K increases the difference in performance gets even larger. For example, for K = 10, CKNN-VN^3 is almost 40 times slower than CKNN-PINE. In general, as the density increases, the performance of CKNN-VN^3 degrades more relative to PINE. This is because, as the density increases, the number of interest points compared to the total number of links in the network increases. Hence, more intersection points are created and more split points

are expected to appear on the path. From Figure 8, we can conclude that the number of KNN queries issued by CKNN-VN^3 is on average twice more than that of CKNN-PINE (slightly increased by increasing K), and that as the density increases, the number of KNN queries increases also but by a small factor.

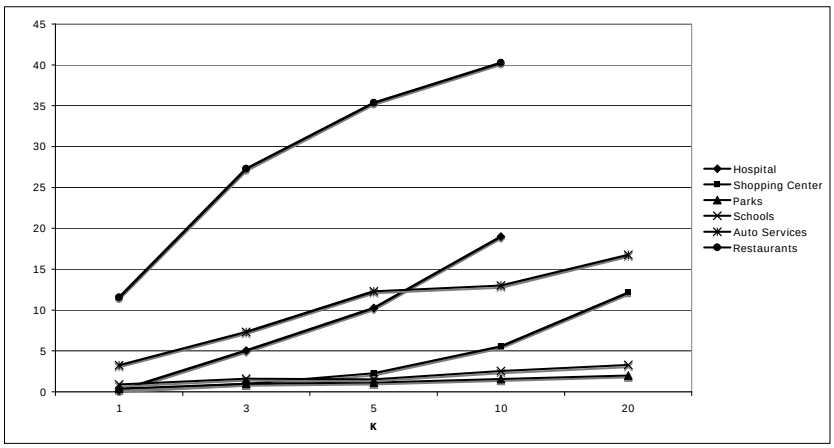

Figure 7: Relative execution time of CKNN using VN^3 vs. PINE.

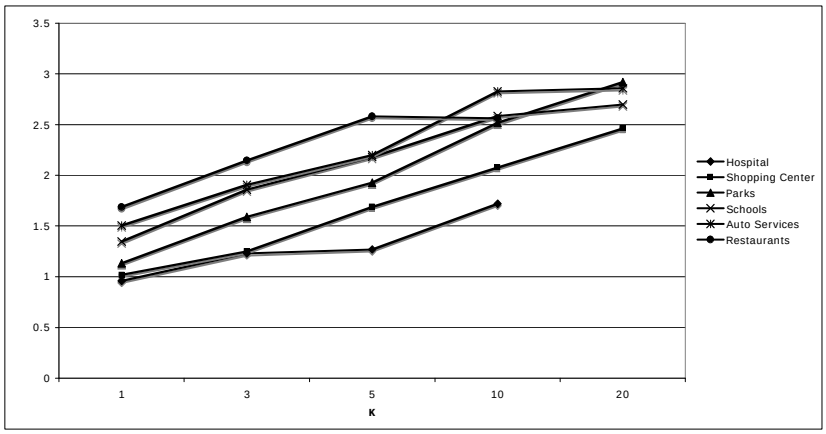

Figure 8: Relative number of KNN queries issued to solve CKNN using VN^3 vs. PINE.

7 Conclusion and future work

In this paper we presented the support of different spatial queries using our PINE structure. We presented a novel approach for continuous *k* nearest neighbor queries (CKNN) in spatial network databases. Our approach, based on PINE, focused on moving queries issued on stationary objects in Spatial Network Database (SNDB). Our solution addressed 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. This was accomplished by distinguishing between two types of split points (ESP, OSP), which reduced the number of computations to retrieve the continuous KNN of a moving object. In addition, we enhanced the CKNN query support using VN^3 by identifying its short comes, especially in the existence of common nearest neighbours between the two end points of a road link.

Our algorithm for continuous K nearest neighbor queries in spatial network databases based on a Progressive incremental network expansion algorithm (PINE), finds the location of split points and the corresponding KNNs on a path. The main features of our algorithm are as follows: 1) CKNN using PINE outperforms CKNN based on VN^3 (in terms of CPU time), one of the few and recently proposed algorithms for CKNN queries in spatial network databases. It outperforms CKNN-VN^3 with a factor of 9 depending on the value of K and the density of the points of interest, 2) CKNN using PINE requires fewer KNN computations as compared to both CKNN using VN^3. CKNN using VN^3 requires a factor of 2 more computations depending on the value of K and the density of the points of interest.

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, constraint KNN, reverse KNN, continuous RNN and group RNN queries.

Acknowledgements

This research was funded by Research Administration at Kuwait University (Project No EO02/04). I would like to thank Fatemah Al-Wazzan, Ahmad Al-Saleh, Ahmad Hammad, Bashar Abdullah, Dariush Ibrahimi, and Khalid Saaifan for their valuable help, feedback, and support.

References

[Berchtold, 97] S. Berchtold, B. Ertl, D.A. Keim, H.P. Kriegel, T. Seidl, Fast Nearest Neighbor Search in High-Dimensional Space, Proceedings of ICDE, 1997, Orlando, Florida, USA.

[Bozkaya, 97] T. Bozkaya, M. Ozsoyoglu, Distance-Based Indexing for High-Dimensional Metric Spaces, Proceedings of SIGMOD, 1997, Tucson, Arizona, USA.

[Chiueh, 94] T. Chiueh, Content-Based Image Indexing, Proceedings of VLDB, 1994, Santiago de Chile, Chile.

[Ciaccia, 97] P. Ciaccia, M. Patella, P. Zezula, M-tree: An Efficient Access Method for Similarity Search in Metric Spaces, Proceedings of the VLDB Journal, 1997, pp. 426-435.

[Corral, 00] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Closest pair queries in spatial databases, Proceedings of ACM SIGMOD International Conference on Management of Data, 2000, Dallas, USA.

[Hjaltason, 99] G.R. Hjaltason, H. Samet, Distance Browsing in Spatial Databases, Proceedings of TODS, 1999, No. 2, pp. 265-318, vol. 24.

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

[Kolahdouzan, 04a] M. Kolahdouzan, C. Shahabi, Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases, Proceedings of VLDB, 2004.

[Kolahdouzan, 04b] M. Kolahdouzan, C. Shahabi, Continuous K Nearest Neighbor Queries in Spatial Network Databases, Proceedings of the Second Workshop on SpatioTemporal Database Management STDBM, 2004.

[Kollios, 99] G. Kollios, D. Gunopulos, V.J. Tsotras,Nearest Neighbor Queries in a Mobile Environment, Proceedings of the International Workshop on Spatio-Temporal Database Management, 1999, pp. 119–134.

[Korn, 96] F. Korn, N. Sidirapoulos, C. Faloutsos, E. Siegel, Z. Protopapas, Fast Nearest Neighbor Search in Medical Image Databases, Proceedings of VLDB, 1996, Mumbai, India.

[Kung, 86] R. Kung, E. Hanson, Y. Ioannidis, T. Sellis, L. Shapiro, M. Stonebraker, Heuristic Search in Data Base Systems, Proceedings of the first international workshop on Expert Database systems, pp. 537 – 548, 1986, South Carolina, United States.

[Okabe, 00] A. Okabe, B. Boots, K. Sugihara, S.N. Chiu, Spatial Tessellations, Concepts and Applications of Voronoi Diagrams, John Wiley and Sons Ltd., 2nd edition, 2000, ISBN 0-471-98635-6.

[Papadias, 03] D. Papadias, J. Zhang, N. Mamoulis, Y. Tao, Query Processing in Spatial Network Databases, Proceedings of VLDB, 2003, pp. 802-813.

[Rigaux, 02] P. Rigaux, M. Scholl, A. Vorsard, Spatial Databases with Applications to GIS, Morgan Kaufmann, 2002.

[Roussopoulos, 95] N. Roussopoulos, S. Kelley, F. Vincent, Nearest Neighbor Queries', Proceedings of SIGMOD, 1995, San Jose, California.

[Safar, 05] M. Safar, K Nearest Neighbor Search in Navigation Systems, Journal of Mobile Information Systems (MIS), 2005, IOS Press.

[Saltenis, 00] S. Saltenis, C.S. Jensen, S.T. Leutenegger, M.A. Lopez, Indexing the Positions of Continuously Moving Objects, Proceedings of ACM SIGMOD, 2000.

[Seidl, 98] T. Seidl, H.P. Kriegel, Optimal Multi-Step k-Nearest Neighbor Search, Proceedings of SIGMOD, 1998, Seattle, Washington, USA.

[Shahabi, 02] C. Shahabi, M. Kolahdouzan, M. Sharifzadeh, A Road Network Embedding Technique for k-Nearest Neighbor Search in Moving Object Databases, Proceedings of ACMGIS, 2002, McLean, VA, USA.

[Song, 01] Z. Song, N. Roussopoulos, K-Nearest Neighbor Search for Moving Query Point, Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases, 2001.

[Tao, 02a] Y. Tao, D. Papadias, Q. Shen, Continuous Nearest Neighbor Search, Proceedings of VLDB, 2002, Hong Kong, China.

[Tao, 02b] Y. Tao, D. Papadias, X. Lian, Reverse kNN Search in Arbitrary Dimensionality, Proceedings of 30th Very Large Data Bases (VLDB), 2002, pp. 744-755, Toronto, Canada.

[Yiu, 05] M.L. Yiu, N. Mamoulis, D. Papadias, Y. Tao, Reverse Nearest Neighbor in Large Graphs, Proceedings of ICDE, 2005, pp. 186-187.

[YU, 01] C. Yu, B.C. Ooi, K.L. Tan, H.V. Jagadish, Indexing the Distance: An Efficient Method to KNN Processing, Proceedings of the Very Large Data Bases Conference (VLDB), 2001.