Mobile Information Systems 1 (2005) 207–224
IOS Press

K nearest neighbor search in navigation systems

Maytham Safar
Computer Engineering Department, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait
Tel.: +965 4987962; Fax: +965 4839461; Mobile: +965 9361062;
E-mail: maytham@eng.kuniv.edu.kw

Abstract. A frequent type of query in a car navigation system is to find the k nearest neighbors (kNN) of a given query object (e.g., car) using the actual road network maps. With road networks (spatial networks), the distances between objects depend on their network connectivity and it is computationally expensive to compute the distances (e.g., shortest paths) between objects. In this paper, we propose a novel approach to efficiently and accurately evaluate kNN queries in a mobile information system that uses spatial network databases. The approach uses first order Voronoi diagram and Dijkstra's algorithm. This approach is based on partitioning a large network to small Voronoi regions, and then pre-computing distances across the regions. By performing across the network computation for only the border points of the neighboring regions, we avoid global pre-computation between every object-pair. Our empirical experiments with real-world data sets show that our proposed solution outperforms approaches that are based on on-line distance computation by up to one order of magnitude. In addition, our approach has better response times than approaches that are based on pre-computation.

Keywords: Navigation system, GPS, shortest path, network distance, Dijsktra's Algorithm, GIS queries, spatial network

1. Introduction

Over the last decade, due the rapid developments in information technology (IT), particularly communication technologies, a new breed of information systems have 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 to adopt services that go beyond traditional web-based systems, and hence it comes with new challenges for researchers, developers and users.

One of 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. One of the frequently used queries in such systems is k nearest neighbor (kNN) queries. This type of query is defined as: given a set of spatial objects (or points of interest, e.g., hospitals), and a query point (e.g., vehicles' location), find the k closest objects to the query. An example of kNN query is a query initiated by a GPS device in a vehicle to find the 5 closest restaurants to the vehicle. 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.

The majority of the existing work on kNN queries is based on either computing the distance between a query and the objects on-line [3,5,9], or utilizing index structures [4,6,8,10,11,13,16–18]. The solution proposed by the first group is based on the fact that 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 computes their distances to q progressively. However, the main disadvantage of these approaches is that they perform poorly when the objects are not densely distributed in the network since then they require large portion of the network to be retrieved for distance computation. The second group of approaches is designed and optimized for metric or vector spatial index structures (e.g., m-tree and r-tree, respectively). The approaches that are based on metric index structures require pre-computations of the distances between the objects and grouping of the objects based on their distances to some reference object (this is more intelligent as compared to a naive approach that pre-computes and stores distances between all the object-pairs in the network). These solutions filter a small subset of possibly large number of objects as the candidates for the closest neighbors of q, and require a refinement step to compute the actual distance between q and the candidates to find the actual nearest neighbors of q. The main drawback of applying these approaches on SNDB is that they do not offer any solution as how to efficiently compute the distances between q and the candidates. Moreover, applying an approach similar to the first group to perform the refinement step in order to compute the distance between q and the candidates will render these approaches, which traverse index structures to provide a candidate set, redundant since the network expansion approach does not require any candidate set to start with. In addition to this drawback, approaches that are based on vector index structures are only appropriate for spaces where the distance between objects is only a function of their spatial attributes (e.g., Euclidean distance) and cannot properly approximate the distances in a network.

A comprehensive solution for spatial queries in SNDB must fulfill these real-world requirements: 1) be able to incorporate the network connectivity to provide exact distances between objects, 2) efficiently answer the queries in real-time in order to support kNN queries for moving objects, 3) be scalable in order to be applicable to usually very large networks, and 4) be independent of the density and distribution of the points of interest.

Taken into consideration that Mobile devices are usually limited on memory resources and have lower computational power, in this paper, we propose a novel approach that fulfills the above requirements by reducing 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 our approach, termed Progressive Incremental Network Expansion (PINE), is to first partition a large network in to smaller/more manageable regions. We achieve 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 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 two different scales simultaneously to: 1) compute the distance from q to its first nearest neighbor (its Voronoi cell center point), and 2) explore the objects

that are close to q (centers of surrounding Voronoi cells) and computes their distances to q during the expansion.

At the first scale, a network expansion similar to INE 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 neighbor (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 VN^3, 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 were found.

To the best of our knowledge, the Incremental Network Expansion (INE) and the Voronoi-based Network Nearest Neighbor (VN^3) approaches presented in [5,9], respectively, are the only other approaches that support the exact kNN queries on spatial network databases. However, VN^3 performance suffers with lower density data sets.

Our empirical experiments with real-world data sets (presented in Section 5) show that VN^3 disk access time tends to be on average twice to four times more than that for $PINE$. In addition, VN^3 CPU time tends to be on average five to ten times more than that for $PINE$. Finally, we show that the required computation for the pre-computation component of VN^3 is on average 13.59 times more than that of $PINE$. INE approach also suffers from poor performance when the objects (e.g., restaurants) are not densely distributed in the network. Our empirical experiments show that INE query processing time is 10 to 12 times more than that of $PINE$, depending on the density of the points of interest. Also, we show that $PINE$'s performance is independent of the density and distribution of the points of interest, and the location of the query object.

The remainder of this paper is organized as follows. We review the related work on k nearest neighbor queries in Section 2. We then provide a review of the Voronoi diagrams and Dijkstra's algorithm, the basis of our proposed $PINE$ approach in Section 3. In Section 4, we discuss our proposed $PINE$ approach. Finally, we discuss our experimental results and conclusions in Sections 5 and 6, respectively.

2. Related work

Numerous algorithms for k-nearest neighbor (kNN) queries are proposed. This type of queries is extensively used in car navigation systems, geographical information systems, shape similarity in image databases, etc. Some of the algorithms are aimed at m-dimensional objects and are based on utilizing one of the variations of multidimensional vector or metric index structures. Other algorithms are based on pre-calculation of the solution space or the computation of the distance from a query object to its nearest neighbors on-line and per query. Finally, there are approaches that support the exact k nearest neighbors' queries on spatial network databases. In this section, we consider each group in turn.

Some of the algorithms are aimed at m-dimensional objects and are based on utilizing one of the variations of multidimensional vector or metric index structures. The algorithms that are based on index structures usually perform in two filter and refinement steps and their performance depend on their selectivity in the filter step. These approaches can be divided in two group: 1) vector index structures [4, 6,8,10,15,18], and 2) metric index structures [11,12,16,17]. Vector index structures are approaches that are designed to utilize spatial index structures and aimed to minimize number of candidates, index nodes and disk accesses required to obtain candidates. There are two major shortages with these approaches

that render them impractical for networks: 1) networks are metric space, i.e., the distance between two objects depends on the connectivity of the objects and not their spatial attributes, however, the filter step of these approaches is based on Minkowski distance metrics (e.g., Euclidean distance). Hence, the filter step of these approaches cannot be used for, or properly approximate exact distances in networks. 2) These approaches do not propose any method to calculate the exact network distance between objects and the query for their refinement step, rather they assume that the distance function can be easily calculated. Metric index structures are approaches that are also based on a filter and refinement process, but as opposed to the vector index structures, they index and filter the objects considering their metric distance. The main disadvantage of these approaches is that they do not offer any solution on how to efficiently compute the distances between the query and the candidates (i.e., the same as the second shortage of the approaches based on vector index) which is required by the refinement step.

There are also other algorithms that are based on pre-calculation of the solution space or the computation of the distance from a query object to its nearest neighbors on-line and per query. Berchtold et al. in [13] suggest pre-calculating, approximating and indexing the solution space for nearest neighbor problem in m dimensional spaces. Pre-calculating the solution space means determining the Voronoi diagram of the data points. The exact Voronoi cells in m dimensional space are usually very complex, hence the authors propose indexing approximation of the Voronoi cells. This approach is only appropriate for first nearest neighbor problem in high-dimensional spaces. Jung et al. in [14] propose an algorithm to find the shortest distance between any two points in a network. Their approach is based on partitioning a large graph into layers of smaller sub graphs and pushing up the pre-computed shortest paths between the borders of the sub graphs in a hierarchical manner to find the shortest path between two points. This approach can potentially be used in conjunction with one of the approaches that are based on metric index, however, the main disadvantage of this approach is its poor performance when multiple shortest path queries from different sources are issued at the same time.

Finally, to the best of our knowledge, there are only two other approaches that support the exact kNN queries on spatial network databases (i.e., Papadias et al. in [5], and Kolahdouzan et al. in [9]). Papadias et al. in [5] propose a solution for nearest neighbor queries in network databases by introducing an architecture that integrates network and Euclidean information and captures pragmatic constraints. Their approach is based on generating a search region for the query point that expands from the query. This approach performs similar to Dijkstra's algorithm and the underlying data structures of the architecture are aimed to minimize number of disk accesses that are required to fetch adjacent links and nodes from the database. The advantages of this approach are: 1) it offers a method that finds the exact distance in networks, and 2) the architecture can support other spatial queries like range search and closest pairs. Since the number of links and nodes that need to be retrieved and examined are inversely proportional to cardinality ratio of entities and number of nodes in the network, the main disadvantage of this approach is a dramatic degradation in performance when the above cardinality ratio is (far) less than 10%, which is the usual case for real world scenarios (e.g., the real data sets representing the road network and different types of entities in the State of California show that the above cardinality ratio is usually between 0.04% and 3%). This is because spatial databases are usually very large and small values for the above cardinality ratio will lead to large portions of the database to be retrieved. This problem happens for large values of k as well.

Kolahdouzan et al. in [9] propose a solution for k nearest neighbor queries in spatial networks, termed VN^3, which is based on the properties of the Network Voronoi diagrams (NVD). In addition, it uses *localized* pre-computation of the network distances for a very small percentage of neighboring nodes in the network to enhance query response time and reduce disk accesses. VN^3 iterative filter/refinement

process is based on the fact that the Network Voronoi Polygons (NVPs) of an NVD can directly be used to find the first nearest neighbor of a query object q. Subsequently, NVPs' adjacency information can be utilized to provide a candidate set for other nearest neighbors of q. Finally, the pre-computed distances can be used to compute the actual network distances from q to the generators in the candidate set and consequently refine the set. The filter/refinement process in VN^3 is iterative: at each step, first a new set of candidates is generated from the NVPs of the generators that are already selected as the nearest neighbors of q, then the pre-computed distances are used to select *"only the next"* nearest neighbor of q. The advantages of this approach are: 1) it offers a method that finds the exact distance in networks, 2) fast query response time, and 3) progressively returns the k nearest neighbors from a query point (i.e., at each iterative step it computes the exact next nearest neighbor and the shortest distance to it). 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.

3. Background

Our proposed approach to address the nearest neighbor queries is based on both the *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 the most efficient algorithm that finds shortest paths from the source node to all the other nodes. 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. A thorough discussion on Voronoi diagrams is presented in [2]. Finally we discuss Dijkstra's algorithm.

3.1. Voronoi diagram

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*. Figure 1 shows an example of a Voronoi diagram [9], its polygons and generators. The following property holds for any Voronoi diagram and will be used later to answer kNN queries: *"The nearest generator point of p_i (e.g., p_j) is among the generator points whose Voronoi polygons share similar Voronoi edges with $VP(p_i)$."* [2,9].

3.2. Network Voronoi diagram

"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 distance between objects is defined as their shortest path in the network rather than their Euclidean

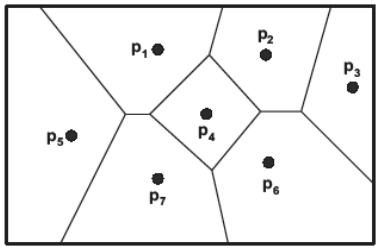

Fig. 1. Example of a Voronoi diagram: Generators and Voronoi polygons.

distance." [2,9]. 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.

Assume a planar graph $G(N, L)$ that consists of a set of nodes $N = \{p_1, \ldots, p_n, p_{n+1}, \ldots, p_o\}$, where the first n elements (i.e., $P = \{p_1, \ldots, p_n\}$) are the generators (e.g., points of interest in a road network), and a set of links $L = \{l_1, \ldots, l_k\}$ that connects the nodes. Also assume that the network distance from a point p on a link in L to p_i in N, $d_n(p, p_i)$, is defined as the shortest network distance from p to p_i. For all $j \in I_n \backslash \{i\}$, we define:

$$Dom(p_i, p_j) = \left\{ p | p \in \bigcup_{o=1}^{k} l_o, d_n(p, p_i) \leqslant d_n(p, p_j) \right\}$$

$$b(p_i, p_j) = \left\{ p | p \in \bigcup_{o=1}^{k} l_o, d_n(p, p_i) = d_n(p, p_j) \right\}$$

The set $Dom(p_i, p_j)$, called the *dominance region* of p_i over p_j on links in L, specifies all points in all links in L that are closer to p_i or of equal distance to p_j. The set $b(p_i, p_j)$, called *bisector* or *border points* between p_i and p_j, specifies all points in all links in L that are equally distanced from p_i and p_j. Consequently, the *Voronoi link set* associated with p_i and *network Voronoi diagram* are defined as following respectively:

$$V_{link}(p_i) = \bigcap_{j \in I_n \backslash \{i\}} Dom(p_i, p_j)$$

$$NVD(P) = \{V_{link}(p_1), \ldots, V_{link}(p_n)\}$$

where $V_{link}(p_i)$ specifies all the points in all the links in L that are closer to p_i than any other generator point in N. Similar to VD defined in Section 3.1, elements of NVD are also collectively exhaustive and mutually exclusive except for their border points. Note that b is a set of points, which unlike Voronoi diagram in Euclidean space, cannot directly generate polygons. However, by properly connecting adjacent border points of a generator g to each other without crossing any of the links, we can generate a bounding polygon, called *network Voronoi polygon*, we term $NVP(g)$, for that generator. Note that generation of $NVP(g)$ only requires local network information, i.e., the links and nodes that are in the area between g and its adjacent generators are used to generate $NVP(g)$.

An example of NVD [9] is shown in Fig. 2, where p_1, p_2, and p_3 are the generators. We can assume that the set of generators is the set of *points of interest* (e.g., hotels, restaurants, ...) and p_4 to p_{16} are the intersections of a road network that are connected to each other by the set of streets L. The NVD of the graph where each line style corresponds to a Voronoi link set of a generator is shown in the same

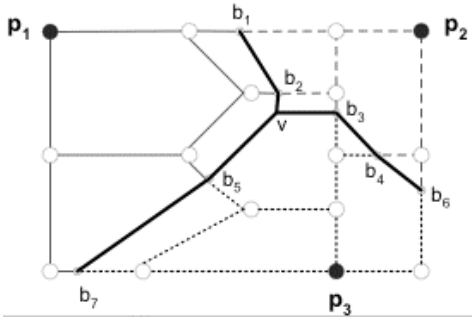

Fig. 2. Example of a network Voronoi diagram (NVD).

figure. Some links are completely contained in V_{link} of a generator (e.g., the link connecting p_6 and p_9 is completely inside $V_{link}(p_1)$), while others are partially contained in different V_{link}s (e.g., the link connecting p_4 and p_5 is divided between and contained in $V_{link}(p_1)$ and $V_{link}(p_2)$. The figure also shows how adjacent border points should be connected to each other: if two adjacent border points are between two similar generators (e.g., b_5 and b_7), they can be connected with an arbitrary line that does not cross any of the members of L. Three or more adjacent border points (e.g., b_2, b_3 and b_5) can be connected to each other through an arbitrary auxiliary point (e.g., v in the figure). By using arbitrary lines and auxiliary points, NVPs will become non-unique. However, since objects in a graph can only be located on links, different NVPs will contain exactly identical Voronoi link sets and hence are unique in this respect. Moreover, as shown in the figure and unlike Voronoi polygons in the Euclidean space, common edges between two NVPs may contain more than two border points and are not necessarily straight lines. Despite this, properties 1 and 2 of Section 3.1.1 are still valid for NVPs.

3.3. Dijkstra's algorithm

In a weighted graph $G = (V, E)$ (otherwise known as a network); where V is a set of vertices, and E is a set of edges; it is frequently desired to find the shortest path between two nodes. Dijkstra's algorithm provides one the most efficient algorithm that finds shortest paths from the source node to all the other nodes. The main idea of the Dijkstra's algorithm is to maintain a set of vertices (S) whose final shortest-path weights from the source (s) have already been calculated, along with a complementary set of vertices ($Q = V - S$) whose shortest-path weights have not yet been determined. The algorithm repeatedly selects the vertex with the minimum current shortest-path estimate among Q. It updates the weight estimates to all vertices adjacent to the currently selected vertex (known as *relaxation*). The vertex is then added to the set S. It continues to do this until all vertices' final shortest-path weights have been calculated (i.e., until the set Q is empty).

kNN problem can be solved by first applying Dijkstra's algorithm to find the distance from the source node to all other nodes in the graph. Next, by sorting the nodes in an ascending order according to their distances from the source node, the kNN nodes can be identified as the top k elements of the sorted list.

4. kNN queries using network Voronoi diagram expansion

In this section we propose a new approach, termed Progressive Incremental Network Expansion ($PINE$), based on Dijkstra's algorithm. $PINE$ is used to find the exact kNN of a query point using

Algorithm: PINE (q, k)

1. $NVP(q)$is the polygon that q belongs to.
2. The generator point ($G_{NVP(q)}$) for $NVP(q)$ is your first nearest neighbor (NN_1).
3. *Cand-NN* is a set of size *k-1* that should contain the next *k-1* candidate (network) nearest interest points sorted in ascending order of their network distance. Initially the set is empty. *Cand-NN* can be implemented as a priority queue.
4. $d_{N\max} = d_{net}(q, Cand - NN_{k-1})$ // if *Cand-NN* is $\varnothing$, $d_{N\max} = \infty$.
5. $PQ = <(n_i, d_{net}(q,n_i))>$; where n_is are the nodes that can be reached from the query point. // sorted in ascending order of their network distance d_{net}.
6. de-queue the node n on top of PQ // with the smallest $d_{net}(q,n)$.
7. while (($d_{net}(q,n) < d_{N\max}$) and (number of nodes in $Cand - NN < k$)
   if (n is n_{Net}) then //n is a network node
   { for each non-visited adjacent node n_x of n
      i. en-queue $(n_x, d_{net}(q,n_x))$
      ii. de-queue the next node n in PQ
   }
   else // n is n_{NVP}: a polygon border point that is on the border of an adjacent polygon with generator point g_x.
   {
      i. update *Cand-NN* from $Cand - NN \cup g_x$. If g_x already belongs to *Cand-NN*, then relax its value, otherwise add it to *Cand-NN*.
      ii. $d_{N\max} = d_{net}(q, Cand - NN_k)$
      iii. for each border point of $NVP(g_x)$($n_{NVP(g_x)}$) en-queue $(n_{NVP(g_x)}, d_{net}(q, n_{NVP(g_x)}))$
      iv. de-queue the next node n in PQ
   }

End PINE

Fig. 3. $PINE$ algorithm.

network Voronoi diagram and network expansion algorithm. It performs network expansion starting from the query point q and examines the interest points (i.e., NNs) in the order they are encountered. This approach is also based on the properties of the network Voronoi diagrams and pre-computation of the network distances for a very small percentage of the nodes in the network.

$PINE$ 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 our approach is to first partition a large network in to smaller/more manageable regions (using $network$ Voronoi diagram). Next, we pre-compute the inter distances for each cell. This will reduce the pre-computation time and space by localizing the computation to cells and handful of neighbor-cell node-pairs. Unlike INE [5], our expansion method utilizes the inter-cell pre-computed

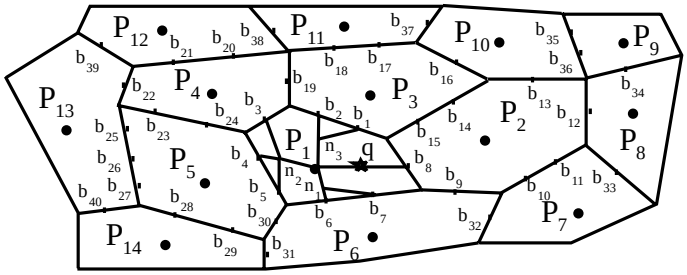

Fig. 4. Sample network Voronoi diagram.

distances to find the actual network distance from the query point to the objects in the surrounding area, hence saves on computation time. VN^3 [9] performance suffers with lower density data sets, because it has to access the pre-computed border-to-border (inter-cell) and query-to-border (intra-cell) distances stored with the polygons. To avoid this problem, with $PINE$, we only need to access border-to-border (inter-cell) distances stored with the polygons.

4.1. Network node types

To explain how $PINE$ works, we need to distinguish between two different types of nodes. First node type ($\boldsymbol{n}_{Net}$) represents the nodes that are inside $NVP(q)$. These nodes are the original network map nodes (e.g., $\boldsymbol{n}_2$ in Fig. 4). The second node type ($\boldsymbol{n}_{NVP}$) represents the border points of a polygon. In the sequel we use $BoP(e)$ to specify the set of border points of an entity e. $\boldsymbol{n}_{NVP}$ can be either $BoP(q)$ or $BoP(NVP)$s of the polygons that we will explore. These nodes are either original network map nodes or nodes generated to create NVD (e.g., $b_3, b_{17}, \ldots$ etc.).

4.2. Border to border computations

Border to border distance (inter-cell) computations are required to find the network distances from $BoP(NVP(q))$ to the border points of the NVP of any generator, $BoP(NVP(g))$. To this end, we pre-compute the point-to-point network distances between the border points of each NVP "separately". For example, this approach suggests that for the NVD shown in Fig. 4, the point-to-point network distances among $\{b_1, \ldots, b_8\}$ (corresponding to $NVP(P_1)$) be pre-computed. It also suggests that the point-to-point network distances among $\{b_1, b_2, b_{14}, \ldots, b_{19}\}$ (corresponding to $NVP(P_3)$) be pre-computed. Note that each border point (e.g., b_1) belongs to at least two NVPs (e.g., $NVP(P_1)$ and $NVP(P_3)$) and hence, its distances to all the border points of two NVPs must be pre-computed. The intuition for this approach is that once the point-to-point network distances among the border points of "each" NVP is computed, these distances can be used to find the network distances between the border points of "any" two NVPs. The other intuition is that this approach has low complexity with respect to both space and computation. The reasons are: 1) The pre-computation is only performed for the border points of each NVP separately, and in real world scenarios (as opposed to the example shown in Fig. 4), the ratio of the total number of the border points to the total number of the nodes in the network is small (see Section 5), and 2) the pre-computation is performed for each NVP separately and not across all NVPs, and the border points of each NVP are fairly close to each other.

4.3. $PINE$ algorithm

$PINE$ works as follows. First, it locates the polygon ($NVP(q)$) that the query point q belongs to using *Contain*() function, hence, retrieving the first nearest neighbor (the generator point of $NVP(q)$). This is based on Voronoi Property 2 mentioned in Section 3.1.1. Note that we do not have the distance information from the query point to that neighbor yet. Then, the links that cover q are located and the nodes of those links are placed on a priority queue PQ according to their network distance to q and are explored later. Next, the node closest to q (i.e., the node on top of PQ), e.g., c, is removed from PQ and nodes that are connected to c are retrieved from the database. Subsequently, the *minimum possible network distance*, d_{mnp}, from q to these nodes are computed and the nodes are placed on PQ (if they are already on PQ, their locations on the queue are updated based on their new distances to q). During this progressive process, when we reach a $\boldsymbol{n}_{Net}$ node we expand from it as INE algorithm using the original network edges/nodes of $NVP(q)$. However, when we reach a $\boldsymbol{n}_{NVP}$ node connecting to a new polygon ($NVP(j)$) adjacent to $NVP(q)$, then we do not need the original network edges/nodes of $NVP(j)$. This is because, for any polygon we know how to reach (shortest path) from any border point to any other border point and how much it costs. Therefore, we consider that all the other border points of $NVP(j)$ as the links that we can reach from $\boldsymbol{n}_{NVP}$ and we add them to PQ to be explored later. In addition, we know the distance from any border point of $NVP(j)$ (including $\boldsymbol{n}_{NVP}$) to its generator point (interest point). Therefore, we compute the d_{mnp} to that generator (not the actual shortest distance) and add the generator point for $NVP(j)$ to our candidate k-1 interest points queue (*Cand-NN*). At any point during the algorithm execution, the nodes inside the *Cand-NN* are only candidate neighbors that we can reach so far and their order in the queue (ordered in ascending order according to distance). The order of the nodes in the queue may change at each execution step, if we were able to find a shorter path to reach the same node that is already in *Cand-NN*. Nodes can also be added or dropped deepening on their updated d_{mnp}. This is because our algorithm depends on Dijkstra's algorithm that updates the distance estimates to all vertices adjacent to the currently selected vertex (known as *relaxation* process). Only at the end of the progressive process, *Cand-NN* would have the next k-1 interest points of q ordered from top to bottom. We recursively apply the above algorithm and terminate it when we have the queue *Cand-NN* filled with k-1 neighbors and the distance from q to any element of PQ is greater than the distance to reach any node in *Cand-NN*. This means, from any point in PQ, we cannot reach another neighbor with a distance shorter than any of the k-1 interest points already discovered. See Fig. 3 for the complete $PINE$ algorithm.

4.4. Analysis

This approach is more suitable for large networks with a small number of points of interest (i.e., a large value for n/mb; m points of interest, n nodes in the network, b connected nodes on one disk block). The approach does not have to retrieve a large portion of the network data (edges/nodes) before the distance from q to $BoP(NVP(q))$ can be computed. It only retrieves the part of the network in the direction that it will explore next, and delays the exploration of the rest of the network, until it is needed. With INE algorithm, if the k interest points cover a large spatial area, then the algorithm would have to explore the exact network edges and nodes for that area (i.e., a large number of nodes and edges). However, in our approach, the search area is divided into a set of network Voronoi polygons. Hence, to find the first interest point, we only need to explore the exact network edges and nodes of the polygon area that contains the query point q ($NVP(q)$). Then, to find the next k-1 interest points, we utilize the

Table 1
Query processing time of $PINE$ vs. INE

Entities	Query processing time (sec.)											
Qty	$k=1$		$k=5$		$k=10$		$k=25$		$k=50$		$k=100$	
(density)	$PINE$ (cpu) disk	INE (cpu) disk	$PINE$ (cpu) disk	INE (cpu) disk	$PINE$ (cpu) disk	INE (cpu) disk	$PINE$ (cpu) disk	INE (cpu) disk	$PINE$ (cpu) disk	INE (cpu) disk	$PINE$ (cpu) disk	INE (cpu) disk
Hospitals	(0.06)	(0.3)	(0.38)	(1.7)	(0.57)	(3.8)	(1.31)	(10.1)	–	–	–	–
46 (0.0004)	1.31	12.4	7.40	78.3	9.12	165.1	34.81	430.2	–	–	–	–
Shopping Centers	(0.02)	(0.09)	(0.12)	(0.5)	(0.23)	(1.1)	(0.45)	(3.1)	–	–	–	–
173 (0.0016)	0.56	3.6	2.32	21.1	4.26	44.0	7.47	118.0	–	–	–	–
Parks	(0.01)	(0.03)	(0.04)	(0.2)	(0.07)	(0.3)	(0.19)	(0.8)	(0.36)	(1.6)	–	–
561 (0.0053)	0.24	1.4	0.72	8.2	1.27	15.3	3.12	36.4	6.01	71.1	–	–
Schools	(0.00)	(0.015)	(0.01)	(0.07)	(0.03)	(0.14)	(0.07)	(0.36)	(0.14)	(0.7)	–	–
1230 (0.0115)	0.12	0.6	0.33	3.5	0.62	6.6	1.40	15.6	2.80	32.2	–	–
Auto Services	(0.00)	(0.013)	(0.01)	(0.05)	(0.01)	(0.09)	(0.03)	(0.23)	(0.07)	(0.44)	(0.14)	(0.87)
2093 (0.0326)	0.11	0.57	0.23	2.43	0.41	4.3	0.87	10.0	1.57	19.4	3.29	38.00
Restaurants	(0.00)	(0.01)	(0.01)	(0.03)	(0.01)	(0.06)	(0.02)	(0.15)	(0.07)	(0.3)	(0.10)	(0.6)
2944 (0.0580)	0.14	0.49	0.24	1.34	0.34	2.7	0.69	6.8	1.55	13.3	2.57	26.0

saved information stored with the polygons (i.e., shortest path between all border points of each polygon) instead of exploring the exact network edges and nodes of the rest of the polygons.

Our preliminary experiments show that the total number of border nodes of an NVP is much smaller than the number of actual network nodes inside NVP. Therefore, $PINE$ would access fewer network nodes and links to explore the same spatial area that INE would explore. Hence, we would have to compute fewer distances online than INE. In addition, it boosts the performance since it eliminates the need for executing complex algorithms for distance computations in the adjacent polygons to $NVP(q)$, rather, the distances can be computed from a lookup table in one disk block access. The disadvantage of this approach is the requirement for an off-line process to pre-calculate and store the above network distances.

5. Performance evaluation

We conducted several experiments to: 1) compare the performance of $PINE$ with its competitor, the INE approach presented in [5], and 2) evaluate the overhead of the pre-computations for $PINE$, and 3) compare the performance of $PINE$ with that of VN^3. We used real-world data set 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 the downtown Los Angeles. The experiments were performed on an IBM ZPro with dual Pentium III processors, 512 MB of RAM, and Oracle 9.2 as the database server. We present the average results of 1000 runs of k nearest neighbor queries where k varied from 1 to 500.

5.1. $PINE$ Vs. INE

Our experiments show that the total query response time of $PINE$ is up to one order of magnitude less than that of INE. Table 1 shows the results of comparing query response time between $PINE$ and INE approach proposed in [5]. The first and second columns specify the entities (or points of interest) and their population and cardinality ratio (i.e., number of entities over number of links in the

Table 2
Overhead of $PINE$ pre-computations

Entities	Points inside each NVP	Average BPs per NVP	Number of Pre-comp. Bor-Bor (inter-cell)
Hospital	1698	52	232,000
Shopping	458	25	225,600
Parks	142	14	239,500
Schools	64	10	246,000
Auto Svc.	38	7	239,900
Restaurants	27	6	243,600

network), respectively. Note that for the given data set, restaurants and hospitals represent the entities with the maximum and minimum cardinality ratios. As shown in the table, when $k = 1$, and regardless of the density of the entities, $PINE$ generates the result set almost instantly. This is because a simple *Contain*() function is enough to find the first NN. However, depending on the density of the entities, INE approach requires between 0.49 to 12.4 seconds to provide the first NN. For all values of k and for different data densities, $PINE$ always outperformed INE in terms of CPU processing times (values inside "( )") (see Fig. 6). INE CPU time tends to be on average 5.42 times more than that for $PINE$, and INE DISK time tends to be on average 11.11 times more than that for $PINE$. This is because INE explores a larger set of the exact network edges and nodes, while $PINE$ utilizes the pre-computed distances stored with the polygons. It is obvious from Table 1 that the time required by the database to retrieve the links from network is the dominant factor and the CPU times are almost negligible. Hence, we can conclude that the total query response time of $PINE$ is up 11.1 times faster than that of INE.

Depending on the density of the entities, the time incurred by INE to retrieve the network from the database is between 10.1 (for high densities and larger ks) and 12.4 (for low densities and higher values of k) times more than that incurred by $PINE$. This is because for lower densities of entities, INE requires larger portion of the network to be retrieved. For example, while there are only 340 links retrieved from the database to find the 10 closest restaurants to a query, 17900 links (equal to 16% of the network) need to be retrieved to find the 10 closets hospital to the same query object. Note that INE does not retrieve the required links in one step, rather, only a small number of links are retrieved from the database at each step. Note that $PINE$ also requires pre-computed values to be retrieved from the database, and the number of required pre-computed values increases for lower densities of the entities and larger values of k. However $PINE$ retrieves the required data in only one step, resulting in much faster data retrieval time.

Table 2 shows the overhead incurred by the pre-computations required by $PINE$. As shown in the table, for entities with higher densities (e.g., restaurants) which generate smaller and more number of NVPs, the average number of nodes inside each NVP and number of border points per NVP are less. This will lead to faster pre-computation process since the pre-computations are performed in smaller size local areas. The third column of the table shows the total number of border-to-border pre-computations, which is almost constant for entities with different densities. This is because when there is more number of NVPs (e.g., restaurants), the average number of border points is smaller and when there is less number of NVPs (e.g., hospital), the average number of border points is larger.

Figure 5 depicts the performance of $PINE$ with respect to the size of the candidate set when kNN queries are performed for different entities. For each value of k (x-axis) we performed 1000 queries where the location of the query point is randomly selected, and we averaged the results. Two observations can be made from the figure. First, the ratio of the size of the candidate set over k (SKS/k) decreases as k increases. For example, while 7 candidates are selected when $k = 3$ (2.33 times the value of k),

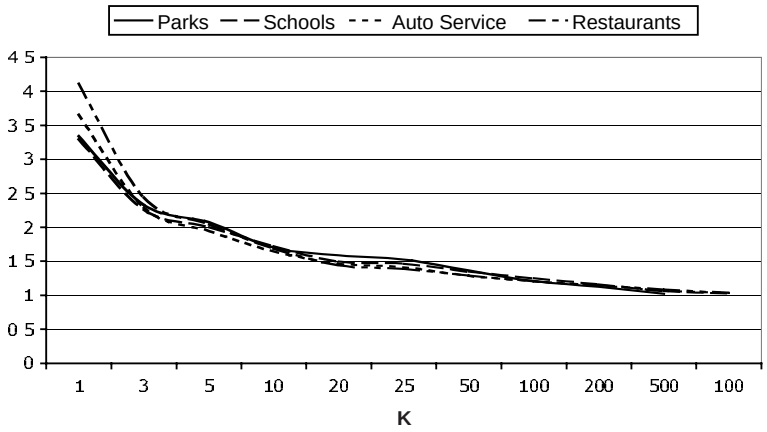

Fig. 5. Performance of $PINE$.

only 31 candidates are selected when $k = 20$ (1.55 times the value of k). The figure also shows that for large values of k, the size of the candidate sets become very close to k. The reason for this is that as k increases, once a generator g is explored, the possibility that some of its adjacent generators have already been explored (already in the candidate NN queue) and no longer need to be examined increases. This is a very important feature of $PINE$ since for large values of k, the average number of points of interest that must be examined significantly decreases. The second observation is that the $PINE$ behaves independently from the density of the points of interest and their distribution in the network. For example, while Restaurants have a cardinality ratio of almost 5 times the Parks, the difference between the corresponding generated candidate sets is only 5.58% (for $k = 500$) to 4.2% (for $k = 3$). This means that whether the points of interest are very dense or sparsely scattered in the network, the performance of $PINE$ does not change. This is because the average number of adjacent generators is "independent" of the density of the points of interest, their distribution, and the underlying network (see [2] for further details).

5.2. $PINE$ Vs. VN^3

Our experiments show that the query response time of $PINE$ is about one half of that of VN^3. Table 3 shows the results of comparing query response time between $PINE$ and VN^3 approach proposed in [9]. As shown in the table, when $k = 1$, and regardless of the density of the entities, both $PINE$ and VN^3 generates the result set almost instantly. This is because a simple *Contain*() function is enough to find the first NN. However, $PINE$ has a slightly larger CPU and disk times. This due to the fact that with VN^3 method the distances from each border point to all the nodes inside the polygons that contain the border point are pre-computed in an off-line process. With $PINE$ the distance *Contain*() returns the generator point of $NVP(q)$ (first NN) but not the distance to it. Hence, Dijkstra's method is applied locally using the local edges and nodes to compute the distance to the first NN. However, for all other values of k and for different data densities, $PINE$ outperforms VN^3 in most cases in terms of CPU processing and disk times (see Fig. 7). VN^3 CPU time tends to be on average 8.25 times more than that for $PINE$, and VN^3 DISK time tends to be on average 2.7 times more than that for $PINE$. VN^3 DISK time is higher because it has to access the pre-computed border-to-border (inter-cell) and query-to-border (intra-cell) distances stored with the polygons, while $PINE$ has to access only border-to-border (inter-cell) distances stored with the polygons. VN^3 CPU time is high because VN^3 updates the distance from the query point to the

Table 3
Query processing time of $PINE$ vs. VN^3

Entities	Query processing time (sec.)											
Qty	$k=1$		$k=5$		$k=10$		$k=25$		$k=50$		$k=100$	
(density)	$PINE$ (cpu) disk	VN^3 (cpu) disk	$PINE$ (cpu) disk	VN^3 (cpu) disk	$PINE$ (cpu) disk	VN^3 (cpu) disk	$PINE$ (cpu) disk	VN^3 (cpu) disk	$PINE$ (cpu) disk	VN^3 (cpu) disk	$PINE$ (cpu) disk	VN^3 (cpu) disk
Hospital	(0.06)	(0)	(0.38)	(1.5)	(0.57)	(4.5)	(1.31)	(15.3)	–	–	–	–
46 (0.0004)	1.31	0.018	7.40	6.5	9.12	14.0	34.81	35.1	–	–	–	–
Shopping Centers	(0.02)	(0)	(0.12)	(0.45)	(0.23)	(1.3)	(0.45)	(3.4)	–	–	–	–
173 (0.0016)	0.56	0.020	2.32	3.3	4.26	6.9	7.47	18.1	–	–	–	–
Parks	(0.01)	(0)	(0.04)	(0.15)	(0.07)	(0.37)	(0.19)	(1.4)	(0.36)	(2.5)	–	–
561 (0.0053)	0.24	0.021	0.72	1.5	1.27	2.8	3.12	6.4	6.01	13.3	–	–
Schools	(0.00)	(0.015)	(0.01)	(0.07)	(0.03)	(0.14)	(0.07)	(0.36)	(0.14)	(0.7)	–	–
1230 (0.0115)	0.12	0.6	0.33	3.5	0.62	6.6	1.40	15.6	2.80	32.2	–	–
Auto Services	(0.00)	(0)	(0.01)	(0.01)	(0.01)	(0.09)	(0.03)	(0.58)	(0.07)	(1.65)	(0.14)	(2.78)
2093 (0.0326)	0.11	0.30	0.23	0.65	0.41	1.4	0.87	2.95	1.57	6.68	3.29	13.1
Restaurants	(0.00)	(0)	(0.01)	(0.01)	(0.01)	(0.04)	(0.02)	(0.26)	(0.07)	(0.8)	(0.10)	(1.85)
2944 (0.0580)	0.14	0.032	0.24	0.57	0.34	1.48	0.69	2.8	1.55	6.1	2.57	12.8

Table 4
Overhead of VN^3 pre-computations

Entities	Points inside each NVP	Average BPs per NVP	Number of Pre-comp. Bor-Bor (inter-cell)	Number of Pre-comp. OPC (intra-cell)
Hospital	1698	52	232,000	8,781,000
Shopping	458	25	225,600	4,653,000
Parks	142	14	239,500	2,630,000
Schools	64	10	246,000	1,787,000
Auto Svc.	38	7	239,900	1,611,000
Restaurants	27	6	243,600	1,348,000

generator points and the border points of the newly explored neighbors (polygons) at each iterative step. While in $PINE$ only the border-to-border distances for the newly explored neighbor (one polygon) is updated in the Candidate queue. It is obvious from Table 3 that the time required by the database to retrieve the links from network is the dominant factor and the CPU times are almost negligible. Hence, we can conclude that the total query response time of $PINE$ is up 2.7 times faster than that of INE.

Table 4 shows the overhead incurred by the pre-computations required by VN^3. The third column of the table shows the total number of border-to-border pre-computations (inter-cell computation), which is exactly the same as for $PINE$ (shown in Table 2). The fourth column shows the extra computations (intra-cell computations) that are required for VN^3.

Using Tables 2 and 4, we compare the total required pre-computations for both $PINE$ and VN^3. One observation can be made is that VN^3 requires on average a larger number of pre-computations than $PINE$ (due to the need for the extra intra-cell computations), however, this ratio reduces as the density of the data set increases. This is because for entities with higher densities (e.g., restaurants), which generate smaller and more number of NVPs, the average number of nodes inside each NVP and number of border points per NVP are less. This will lead to faster pre-computation process since the pre-computations are performed in smaller size local areas. For example, for the hospital data set, VN^3 requires 38.8 times the number of computations required by $PINE$. While VN^3 requires only 6.5 times the number of computations required $PINE$ for the restaurant data set (a higher density set).

The distance pre-computations are usually performed offline, hence it should not affect the overall performance of VN^3. However as shown in Table 3, the time required by the database to retrieve the

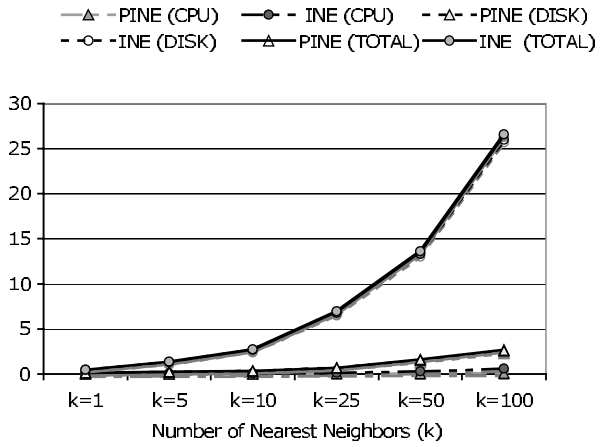

Fig. 6. Performance of $PINE$ VS. INE for restaurants data.

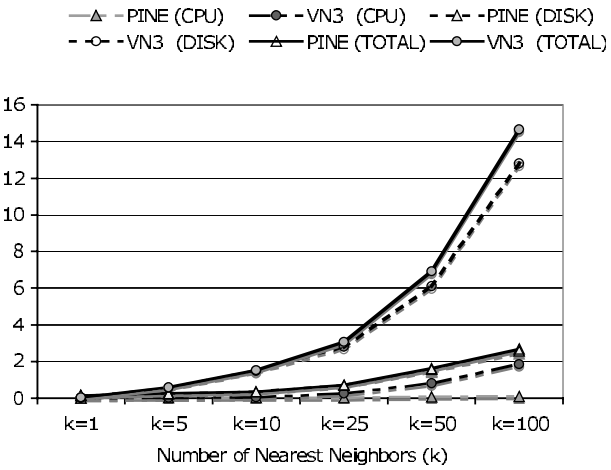

Fig. 7. Performance of $PINE$ VS. VN^3 for restaurants data.

links from network online is the dominant factor and the CPU times are almost negligible. Since VN^3 requires larger amount of pre-computed values to be retrieved from the database than $PINE$, therefore $PINE$ outperforms VN^3.

See Fig. 8 for a complete comparison between $PINE$, INE, and VN^3 in terms of CPU, Disk and total query processing time for Restaurants data.

6. Conclusion

In this paper we presented a novel approach for k nearest neighbor queries in spatial network databases. Our approach, $PINE$, is based on: pre-calculating the network Voronoi polygons (NVP), pre-computing some network distances, and Dijkstra's algorithm. We showed how NVPs could immediately be used to find the first nearest neighbor of a query object. We also showed how the pre-computed distance expedites the process of finding the other nearest neighbors. The main features of $PINE$ are as follow:

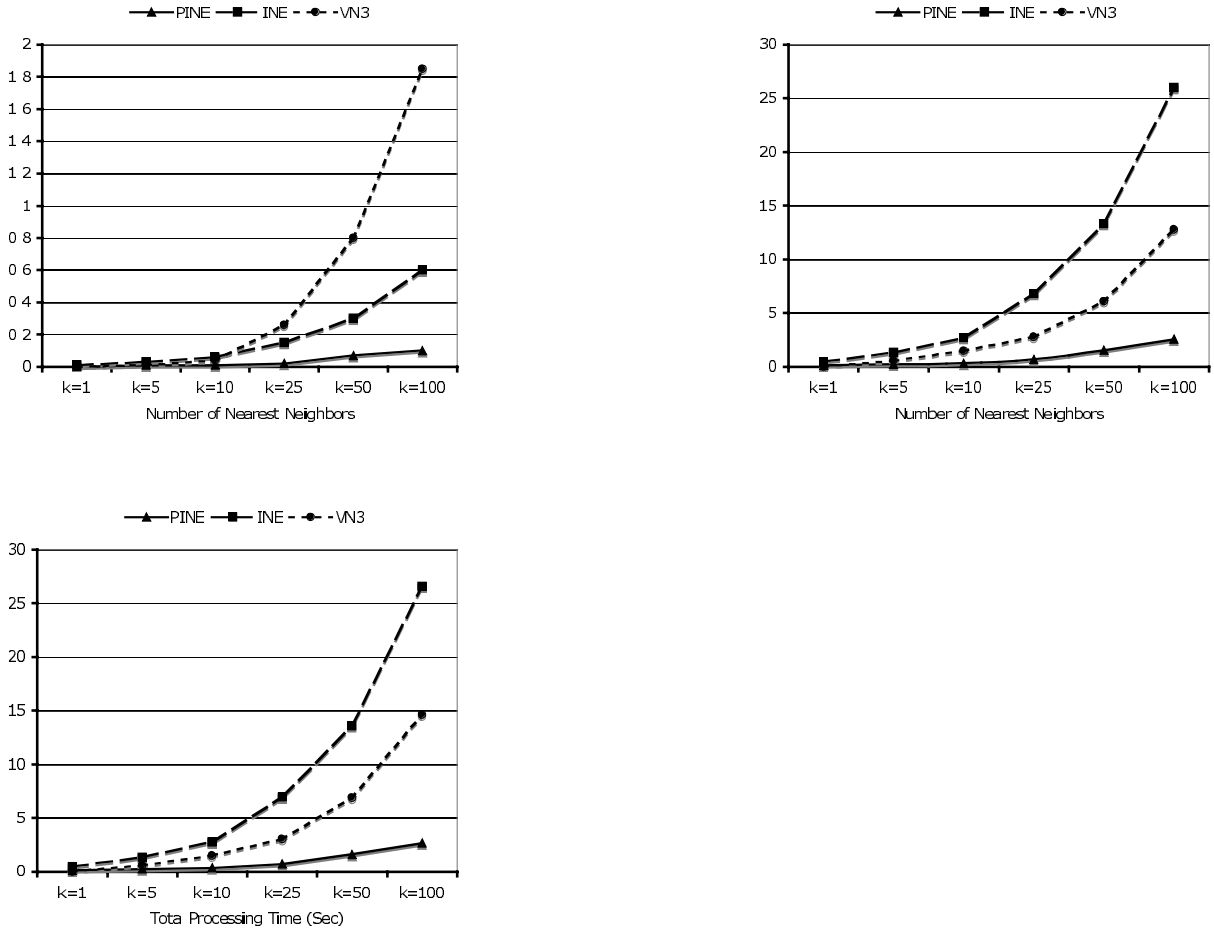

Fig. 8. Performance comparison of $PINE$, INE, and VN^3.

1) $PINE$'s CPU time outperforms both INE and VN^3, the only other approaches proposed for kNN queries in spatial network databases. It outperforms INE with a factor of 5.41, and VN^3 with a factor of 8.25 depending on the value of k and density of the points of interest, 2) $PINE$'s DISK time outperforms both INE and VN^3. It outperforms INE with a factor of 11.1, and VN^3 with a factor of 2.7 depending on the value of k and density of the points of interest.

The time required by the database to retrieve the links from network is the dominant factor and the CPU times are almost negligible. Hence, we can conclude that the total query response time of $PINE$ is up to 11.1 times faster than that of INE, and up to 2.7 times faster than that of VN^3.

$PINE$'s algorithm results in up to 2.33 times less number of candidates as compared to that of the traditional approaches. In addition, the size of $PINE$'s candidate set has less variance across different query point locations and densities of the points of interest. Consequently, the query response time becomes more deterministic, which is an important feature for many real-time kNN query applications.

As with VN^3, the pre-computation required by $PINE$ has low computation and space complexities due to performing the pre-computations in local areas as opposed to across the entire network. VN^3 and INE progressively return the k nearest neighbors from a query point (i.e., at each iterative step it computes the exact next nearest neighbor and the shortest distance to it), which is vital for an interactive

real time system such as navigation system. While with $PINE$, the k nearest neighbors is returned at the end of the searching algorithm. However, during each step it provides some candidates of k nearest neighbors, which is useful for systems where the exact NNs are not required. We plan to extend $PINE$ to address similar kNN queries such as group kNN, constraint kNN, and finding the actual shortest path between a query and its closest neighbors, as our future work.

Acknowledgements

This research was funded by Research Administration at Kuwait University (Project No EO03/04). The author would like to thank C. Shahabi, M. Kolahdouzan, and D. Poravanthattil for their valuable feedback in writing section 3.2 on network Voronoi diagram, and their help in setting up the experiments.

References

[1] A. Corral, Y. Manolopoulos, Y. Theodoridis and M. Vassilakopoulos, *Closest pair queries in spatial databases, Proceedings of ACM SIGMOD International Conference on Management of Data*, Dallas, USA, 2000.
[2] A. Okabe, B. Boots, K. Sugihara and S. Nok Chiu, *Spatial Tessellations, Concepts and Applications of Voronoi Diagrams*, (2nd edition), John Wiley and Sons Ltd., ISBN 0-471-98635-6, 2000.
[3] C. Shahabi, M. Kolahdouzan and M. Sharifzadeh, *A Road Network Embedding Technique for k-Nearest Neighbor Search in Moving Object Databases*, ACMGIS, McLean, VA, USA, 2002.
[4] C. Yu, B.C. Ooi, K.L. Tan and H.V. Jagadish, *Indexing the Distance: An Efficient Method to KNN Processing*, Proceedings of the Very Large Data Bases Conference (VLDB), 2001.
[5] D. Papadias, J. Zhang, N. Mamoulis and Y. Tao, *Query Processing in Spatial Network Databases*, VLDB, Berlin, Germany, 2003.
[6] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel and Z. Protopapas, *Fast Nearest Neighbor Search in Medical Image Databases*, VLDB, Mumbai (Bombay), India, 1996.
[7] G. Kollios, D. Gunopulos and V.J. Tsotras, *Nearest Neighbor Queries in a Mobile Environment*, Proceedings of the International Workshop on Spatio-Temporal Database Management, 1999, 119–134.
[8] G.R. Hjaltason and H. Samet, Distance Browsing in Spatial Databases, *TODS* **24**(2) (1999), 265–318.
[9] M. Kolahdouzan and C. Shahabi, *Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases*, VLDB, Toronto, Canada, 2004.
[10] N. Roussopoulos, S. Kelley and F. Vincent, *Nearest Neighbor Queries*, SIGMOD, San Jose, California, 1995.
[11] P. Ciaccia, M. Patella and P. Zezula, M-tree: An Efficient Access Method for Similarity Search in Metric Spaces, *The VLDB Journal* (1997), 426–435.
[12] P. Rigaux, M. Scholl and A. Voisard, *Spatial Databases with Applications to GIS*, Morgan Kaufmann, 2002.
[13] S. Berchtold, B. Ertl, D.A. Keim, H.P. Kriegel and T. Seidl, *Fast Nearest Neighbor Search in High-Dimensional Space*, ICDE, Orlando, Florida, USA, 1998.
[14] S. Jung and S. Pramanik, An Efficient Path Computation Model for Hierarchically Structured Topological Road Maps, *IEEE Transaction on Knowledge and Data Engineering* (2002).
[15] S. Saltenis, C. Jensen, S. Leutenegger and M. Lopez, *Indexing the Positions of Continuously Moving Objects*, ACM SIGMOD, 2000.
[16] T. Bozkaya and Z. Meral Ozsoyoglu, *Distance-Based Indexing for High-Dimensional Metric Spaces*, SIGMOD, Tucson, Arizona, USA, 1997.
[17] T. Chiueh, *Content-Based Image Indexing*, VLDB, Santiago de Chile, Chile, 1994.
[18] T. Seidl and H.P. Kriegel, *Optimal Multi-Step k-Nearest Neighbor Search*, SIGMOD, Seattle, Washington, USA, 1998.
[19] Y. Tao, D. Papadias and Q. Shen, *Continuous Nearest Neighbor Search*, Proceedings of the Very Large Data Bases Conference (VLDB), Hong Kong, China, 2002.
[20] Z. Song and N. Roussopoulos, *K-Nearest Neighbor Search for Moving Query Point*, Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases, 2001.

Maytham Safar is currently an Assistant Professor at the Computer Engineering Department at Kuwait University. He received his Ph.D. degree in Computer Science from the University of Southern California in 2000. He has one book and more than 20

articles, book chapters, and conference/journal papers in the areas of databases and multimedia. Dr. Safar's current research interests include Peer-to-Peer Networks, Spatial Databases, Multidimensional Databases, and Geographic Information Systems. He served on many conferences as a reviewer and/or a scientific program committee member such as ICDCS, EURASIA-ICT, ICWI, ICME, AINA, WEBIST, IPSI, and iiWAS. He also served as a member on the editorial board or a reviewer for many journals such as IEEE Transactions on Multimedia Journal, ACM Computing Reviews, Journal of Digital Information Management (JDIM), Multimedia Tools and Applications Journal (MTAP), and Euro-Asia Journal of Applied Sciences.