An Efficient and Scalable Approach to CNN Queries in a Road Network

Hyung-Ju Cho Chin-Wan Chung

Dept. of Electrical Engineering & Computer Science
Korea Advanced Institute of Science and Technology
373-1 Kusong-dong, Yusong-gu, Taejon 305-701, Korea
{hjcho, chungcw}@islab.kaist.ac.kr

Abstract

A continuous search in a road network retrieves the objects which satisfy a query condition at any point on a path. For example, return the three nearest restaurants from all locations on my route from point s to point e. In this paper, we deal with NN queries as well as continuous NN queries in the context of moving objects databases. The performance of existing approaches based on the network distance such as the shortest path length depends largely on the density of objects of interest. To overcome this problem, we propose UNICONS (a *uni*que *con*tinuous *s*earch algorithm) for NN queries and CNN queries performed on a network. We incorporate the use of precomputed NN lists into Dijkstra's algorithm for NN queries. A mathematical rationale is employed to produce the final results of CNN queries. Experimental results for real-life datasets of various sizes show that UNICONS outperforms its competitors by up to 3.5 times for NN queries and 5 times for CNN queries depending on the density of objects and the number of NNs required.

1 Introduction

Due to the advances in mobile communication and database technologies, diverse innovative mobile computing applications are emerging. The ability to support continuous queries from mobile clients on a road network is essential for a class of mobile applications. In this paper, we investigate continuous nearest neighbor (CNN) searches under the following two conditions: (i) Moving objects such as cars or people run on a road network and static objects such as gas stations or restaurants are located on the road network. (ii) The distance measure is defined as the shortest path length (network distance) on the network.

CNN searches on a road network are essential for emerging location-based services and many real-life GIS applications. Cars move according to a given path on a road. In addition, with the development of mobile devices such as PDAs and cellular phones, it is practically possible to track cars and people in real-time. Such location-aware devices enable location-based services that provide users with a variety of useful information based on their current positions. Furthermore, CNN searches on a road network constitute interesting and intuitive problems from the practical as well as theoretical point of view. Nevertheless, there is limited previous work in the literature.

We present the following example of the CNN query with real-life semantic on the map in Figure 1, where it is assumed that a, b, c, d, and e are gas stations : *Find the 2 closest gas stations from all points on the path P from n_1 to n_6 (i.e., $P = \{n_1, n_2, n_3, n_4, n_5, n_6\}$).*

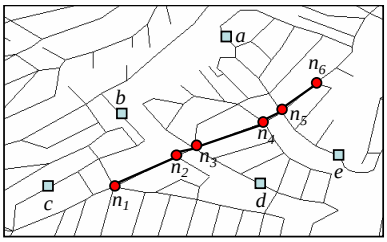


Figure 1: Example continuous search

Unlike the output at a query point, the result of a continuous search for a given path P contains a set of (I, R_I) tuples, where $I \subseteq P$ is a valid interval over which the same query result is generated and R_I is

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

a set of objects satisfying the query condition at any point on I. Two key requirements for good continuous search algorithms are as follows: (i) Use as few static queries required to answer the continuous search as possible. (ii) Reduce the computational overhead in determining valid intervals. The valid intervals specify the locations that the k NNs of a moving query object remains the same.

Existing approaches are largely affected by the density of objects of interest. Voronoi-based Network Nearest Neighbor (VN^3) [7] and Upper Bound Algorithm (UBA) [8] developed by Kolahdouzan et al. for NN queries and CNN queries, respectively, are efficient for the case where objects are sparsely distributed in the network. Hence, their approaches are negatively affected by the increase in the density of objects. Conversely, Incremental Network Expansion (INE) [9] presented by Papadias et al. for NN queries suffers from poor performance when objects are not densely distributed. To overcome this problem, we propose an efficient continuous search algorithm called UNICONS which deals with NN queries as well as CNN queries. We improve the performance of NN queries with the introduction of precomputed NN lists in using Dijkstra's shortest path algorithm [3]. The technique directly contributes to the reduction of the NN query cost of UNICONS. For CNN queries, the advantage of UNICONS over UBA is that only two NN queries are performed between adjacent nodes, independent of the density of objects and the number of NNs required. The mathematical rationale is presented to show how we produce the final result (i.e., a set of (I, R_I) tuples). An advantage of UNICONS is the ease of implementing it with an existing spatial access method, such as the R*-tree [1], and the adjacency list structure to represent the graph structure both of which are disk-based. Our contributions are summarized as follows:

- We propose an efficient continuous search algorithm called UNICONS for NN queries as well as CNN queries in a road network.
- We incorporate the precomputed NN lists in using Dijkstra's algorithm [3] for NN queries and introduce techniques which avoid unnecessary disk I/Os. UNICONS outperforms VN^3 [7], which is currently regarded as the best approach, by up to 3.5 times in a realistic experimental environment.
- UNICONS significantly reduces the number of disk I/Os in processing CNN queries since static queries are issued only at the intersection points on a query path. UNICONS outperforms its competitor, called UBA [8], in query response time by up to 5 times depending on the number of NNs required and the density of objects of interest.

The rest of the paper is structured as follows: Section 2 gives the survey of previous methods for NN and CNN queries. Section 3 presents the improved algorithms for NN queries based on the network distance. Section 4 presents algorithms for the CNN search. Section 5 evaluates the proposed techniques with comprehensive experiments. Finally, Section 6 concludes the paper.

2 Related Work

Several algorithms have been developed using the network distance. Shahabi et al. proposed an embedding technique to transform a road network into a higher dimensional space in order to utilize computationally simple metrics [10]. The main disadvantage of this method is that it provides only an approximation of the actual distance. Jensen et al. [6] propose a data model and definition of abstract functionality required for NN queries in spatial network databases. They use algorithms similar to Dijkstra's algorithm in order to perform online calculations of the shortest distance from a query point to an object. Shekhar et al. [11] present four alternative techniques for finding the first nearest neighbor to a moving query object on a given path.

Papadias et al. [9] presented a solution called INE for NN queries in spatial network databases by introducing an architecture that integrates network and Euclidean information and captures pragmatic constraints. Since the number of links and nodes that need to be retrieved and examined is inversely proportional to the cardinality ratio of objects, the main disadvantage of this approach is a dramatic degradation in performance when the above cardinality ratio is very small, which is a usual case for real world scenarios. In addition, it does not optimize primitive operations to facilitate an efficient network search since it is designed to support both conventional spatial queries based on the Euclidean distance and queries based on the network distance.

Kolahdouzan et al. proposed a new approach for NN queries in spatial network databases [7]. Their approach, called VN^3, precalculates the network Voronoi polygons (NVPs) and some network distances. VN^3 is based on the properties of the Network Voronoi diagrams. Their experiments with several real-world data sets showed that VN^3 outperforms INE [9] by up to an order of magnitude. The intuition is that the NVPs can directly be used to find the first nearest neighbor of a query object q. Subsequently, the adjacency information of NVPs can be utilized to provide a candidate set for other nearest neighbors of q. However, in the case where the number of NNs required increases and the number of objects of interest increases, VN^3 suffers from the computational overhead of precalculating NVPs. Consequently, the performance of VN^3 degenerates considerably for high densities of objects.

Feng et al. [4, 5] provide a solution for CNN queries in road networks. Their solution is based on finding

the locations on a path where a NN query must be performed. The main shortcoming of this approach is that it only addresses the problem when the first nearest neighbor is requested (i.e., continuous 1-NN) and does not address the problem for continuous k NN queries. Finally, Kolahdouzan et al. presented a solution called UBA for CNN queries in spatial network databases [8]. UBA restricts the computation of NN queries to only the locations where they are required and hence, provides better performance by reducing the number of NN computations. UBA takes advantage of VN^3 for obtaining (k+1) NNs at a static location. For this goal, UBA retrieves (k+1) NNs of a query object q to compute the minimum distance during which the query object can move without requiring a new NN query to be issued. However, UBA still requires a large number of NN queries to find split points. The split points are the locations on the path at which the k NNs of a moving query object change. Consequently, the execution time increases sharply in proportion to the number of NNs required and the density of objects.

3 NN search

These algorithms enhance previous research results and they are the basis of our CNN search algorithms. To clarify the meanings of the terms used, we formally define an edge, a node, a path, a query point, and an object.

Definition 3.1 *A road network consists of a set of nodes, together with a set of edges, each of which directly connects two nodes. A path is a sequence of successively neighboring edges where the terminating points are distinct. A query point (e.g., the current position of a user) is a location of interest on the road network. An object (e.g., school or gas station) is a point of interest which is located in the road network.*

We briefly explain the network distance used in a road network. Edges provide the connectivity of the original road network. An edge connecting two nodes, n_i and n_j, has the network distance $d(n_i, n_j)$. If n_i and n_j are not adjacent nodes, $d(n_i, n_j)$ denotes the shortest path length from n_i to n_j. The same concept is applied to the network distance between a node and an object and the distance between two objects. Sometimes, travel time may be used as the network distance, which is very useful in many cases. Suppose that there exists an edge between n_i and n_j. The travel time from n_i to n_j following a narrow street may be larger than that of a path going through some intermediate nodes. In addition, the travel time may dynamically change depending on traffic congestion and road conditions.

Network distances between objects depend on their network connectivity and are computationally expensive to calculate. Therefore, our approach for NN searches introduces the utilization of precomputed NNs. Lemma 1 states a simple but important fact for our NN searches.

Lemma 1 *Consider a query point q on an edge $\overline{n_i n_j}$. Let $\mathcal{R}_q$ be the set of objects satisfying the query condition, $\mathcal{O}_{\overline{n_i n_j}}$ be the set of objects on $\overline{n_i n_j}$, and $\mathcal{R}_{n_i}$ and $\mathcal{R}_{n_j}$ be the sets of objects satisfying the same query condition at n_i and n_j, respectively. Then,*

$$\mathcal{R}_q \subseteq (\mathcal{O}_{\overline{n_i n_j}} \cup \mathcal{R}_{n_i} \cup \mathcal{R}_{n_j})$$

Proof) Lemma 1 is self-evident, so its proof is omitted. ■

If each node on a graph has k precomputed NNs of its own, a k NN query result at any point on the graph can be immediately obtained from objects on an edge containing the query point and the k NNs for the two nodes associated with the corresponding edge. However, it is very difficult to maintain the k NNs for all nodes when the value of k is very large or the number of nodes is very large. For this reason, our approach maintains precomputed NNs for only a small portion of the nodes. We formally define intersections and condensing points as follows:

Definition 3.2 *A node where three or more edges meet is called an intersection point and an intersection point which maintains precomputed NNs is called a condensing point.*

Each condensing point stores m (> 1) NNs where the value of m is provided as a parameter. The performance of our approach improves with an increase in the number of condensing points and m, the size of the precomputed NN lists. Naturally, our approach shows the best performance if the number of condensing points points equals that of intersection points and the value of m is greater than or equal to the value of k, where k is the number of NNs required in a query. In case $k > m$, we can dynamically compute k NNs starting from already precomputed m NNs. Query results can be obtained from a combination of the precomputed information and the Dijkstra's search method. Particularly, in the case where objects are sparsely distributed, our approach is expected to show much better performance than INE [9] which is simply based on Dijkstra's algorithm. This is because our approach can compute the result of an NN query without visiting the NNs directly. In addition, our algorithms can avoid unnecessary disk I/Os by introducing two additional data structures, $E_{visited}$ and E_{empty} as seen in Figures 3 and 5 respectively, which keep useful information from previously visited edges.

Figure 2 shows the algorithm for the NN search. If the query location q on a network is given, our NN search algorithm starts with find_edges (q) on line 4 to discover the set E_q of edges containing q. For example, in the case where q is a node, adjacent edges of

the node belong to E_q. Originally, our guess for the network distance *kth_dist* from q to the kth nearest object o_{kth} is infinity. Next, we explore the objects on the edge $\overline{n_{q1}n_{q2}}$ to retrieve the qualifying objects whose network distances from q are within *kth_dist*, which is performed by explore $(\overline{n_{q1}n_{q2}}, q, kth\_dist)$ on line 6. Objects on the edges which contain q are first investigated in explore $(\overline{n_{q1}n_{q2}}, q, kth\_dist)$ and their distances from q are computed in this function. For a network expansion, we push $(n_{q1}, d(q, n_{q1}))$ and $(n_{q2}, d(q, n_{q2}))$ to the priority queue PQ. If PQ is empty or the network distance $d(q, n_{top})$ of the top element in PQ is not less than *kth_dist*, the algorithm breaks out of the while loop and terminates. Otherwise, the adjacent edges of n_{top} are explored repeatedly. Note that PQ is a priority queue, so $d(q, n_{top})$ on line 12, which is popped from PQ is less than or equal to the distance from q to any other node in PQ.

```
proc NN_Search (q, k, kth_dist)
/* q is a query point, k is the number of nearest
neighbors to be retrieved, and
kth_dist is an initial distance
between the kth object and q. */
begin
1.  /* R keeps objects satisfying the query predicate */
2.  R := ∅
3.  /* find the set E_q of edges containing q */
4.  E_q := find_edges (q)
5.  for each n_q1n_q2 ∈ E_q do
6.     explore (n_q1n_q2, q, kth_dist)
7.     PQ.push (n_q1, d(q, n_q1))
8.     PQ.push (n_q2, d(q, n_q2))
9.  end-for
10. while PQ is not empty do
11.    (n_top, d(q, n_top)) := PQ.pop ()
12.    if d(q, n_top) ≤ kth_dist then
13.       Search_Node (n_top, d(q, n_top), kth_dist)
14.    else
15.       exit while loop
16. end-while
end
```

Figure 2: NN search algorithm

The Search_Node algorithm shown in Figure 3 explores the precomputed NNs and the adjacent edges of node n_{top} with an offset $d(q, n_{top})$. If node n_{top} is a condensing point, the algorithm first scrutinizes each $o_{n_{top}}^{i-th}$ of n_{top} where $o_{n_{top}}^{i-th}$ is the i-th NN of n_{top}. If $d(q, n_{top}) + d(n_{top}, o_{n_{top}}^{i-th})$ is less than *kth_dist*, a tuple $(o_{n_{top}}^{i-th}, d(q, n_{top}) + d(n_{top}, o_{n_{top}}^{i-th}))$ is added to $\mathcal{R}$ and *kth_dist* is updated to the network distance $d(q, o_{kth})$ of the kth nearest object in $\mathcal{R}$. If node n_{top} is a condensing point and $d(q, n_{top}) + d(n_{top}, o_{n_{top}}^{m-th})$ is greater than or equal to *kth_dist*, the Search_Node ends without searching adjacent edges. Otherwise, the algorithm should scrutinize each adjacent edge $\overline{n_{top}n_i}$ of node n_{top}. To avoid unnecessary visit of the same edge, $E_{visited}$ of line 11 maintains a set of $(\overline{n_{v1}n_{v2}}, d^{max}(q, \overline{n_{v1}n_{v2}}))$ tuples where $\overline{n_{v1}n_{v2}}$ is a previously visited edge and $d^{max}(q, \overline{n_{v1}n_{v2}}) = min\{d(q, n_{v1}) + d(n_{v1}, n_{v2}),\ d(q, n_{v2}) + d(n_{v2}, n_{v1})\}$.

```
proc Search_Node (n_top, d(q, n_top), kth_dist)
/* n_top is a node popped from PQ, d(q, n_top) and
kth_dist are as previously defined */
begin
1.  /* let P_n_top = {o_n_top^1st, ..., o_n_top^m-th}
2.  be the set of precomputed NNs of n_top */
3.  if n_top is a condensing point then
4.     for each o_n_top^i-th ∈ P_n_top do
5.        if d(q, n_top) + d(n_top, o_n_top^i-th) < kth_dist then
6.           R := R ∪ {(o_n_top^i-th, d(q, n_top) + d(n_top, o_n_top^i-th))}
7.           kth_dist := d(q, o_kth) where o_kth ∈ R
8.     end-for
9.
10. if n_top is not a condensing point or
11.    d(q, n_top) + d(n_top, o_n_top^m-th) < kth_dist then
12.    for each adjacent edge n_topn_i of n_top do
13.       if (n_topn_i ∈ E_visited and d(q, n_top) ≥ d^max(q, n_topn_i)) then
14.          skip the visit of this edge
15.       else
16.          Search_Edge (n_topn_i, d(q, n_top), kth_dist)
17.          PQ.push (n_i, d(q, n_top) + d(n_top, n_i))
18.          E_visited := E_visited ∪ {(n_topn_i, d(q, n_top) + d(n_top, n_i))}
19.    end-for
end
```

Figure 3: Search_Node algorithm

Figure 4 shows that $E_{visited}$ plays an important role in avoiding unnecessary duplicate accesses to previously visited edges. Given a query point q, which is node n_1, we first explore adjacent edges, $\overline{n_1n_2}$ and $\overline{n_1n_3}$. Then, $E_{visited} = \{(\overline{n_1n_2}, 3),\ (\overline{n_1n_3}, 4)\}$. Next, a search is executed with $(n_2, 3)$ popped from PQ=$\{(n_2, 3),\ (n_3, 4)\}$. Hence, $\overline{n_2n_1}$ and $\overline{n_2n_3}$ will be visited. However, $\overline{n_2n_1}$ is not visited since $d(q, n_2) = 3$ is not less than $d^{max}(q, \overline{n_1n_2}) = 3$ in $E_{visited}$. Edge $\overline{n_2n_3}$ is visited since this edge does not belong to $E_{visited}$, and then $(\overline{n_2n_3}, 8)$ is added to $E_{visited}$. As a result, $E_{visited} = \{(\overline{n_1n_2}, 3), (\overline{n_1n_3}, 4), (\overline{n_2n_3}, 8)\}$, $d(q, a) = d(q, n_2) + d(n_2, a) = 4$ and $d(q, b) = d(q, n_2) + d(q_2, b) = 7$. Finally, n_3 is explored. $\overline{n_3n_1}$ is not visited due to the same reason that $\overline{n_2n_1}$ was not visited. However, $\overline{n_3n_2}$ should be visited because $d(q, n_3) = 4$ is less than $d^{max}(q, \overline{n_2n_3}) = 8$ in $E_{visited}$. Therefore, $d(q, b) = 7$ is updated to $d(q, b) = d(q, n_3) + d(n_3, b) = 5$.

The Search_Edge algorithm shown in Figure 5 inspects objects on the edge $\overline{n_{top}n_i}$, where $d(q, n_i) = d(q, n_{top}) + d(n_{top}, n_i)$. To avoid duplicate accesses to edges where no objects are located, E_{empty} keeps the

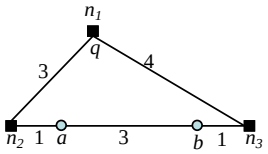


Figure 4: $d(q,a) = 4$ and $d(q,b) = 5$

set of the previously visited edges without objects. It is expected that E_{empty} is very effective when the objects are sparsely located. If $d(q,o_i) < kth\_dist$, a tuple $(o_i, d(q,o_i))$ is added to $\mathcal{R}$ and kth_dist is updated to the network distance $d(q,o_{kth})$ of the kth nearest object in $\mathcal{R}$. When the NN search algorithm ends, the k NNs in $\mathcal{R}$ are returned.

proc Search_Edge ($\overline{n_{top}n_i}$, $d(q,n_{top})$, kth_dist)
/* $\overline{n_{top}n_i}$ is an edge to be visited, $d(q,n_{top})$ and kth_dist are as previously defined. */
begin
1. **if** $\overline{n_{top}n_i} \in E_{empty}$ **then**
2. **return**
3. /* let $O_{\overline{n_{top}n_i}}$ be the set of objects on the edge $\overline{n_{top}n_i}$ */
4. **if** $O_{\overline{n_{top}n_i}}$ is empty **then**
5. $E_{empty} := E_{empty} \cup \{\overline{n_{top}n_i}\}$
6. **return**
7. **for** each object $o_i \in O_{\overline{n_{top}n_i}}$
8. **if** $d(q,n_{top}) + d(n_{top},o_i) < kth\_dist$ **then**
9. $\mathcal{R} := \mathcal{R} \cup \{(o_i, d(q,n_{top}) + d(n_{top},o_i))\}$
10. $kth\_dist := d(q,o_{kth})$ where $o_{kth} \in \mathcal{R}$
11. **end-for**
end

Figure 5: Search_Edge algorithm

4 CNN search

We start with two basic concepts for continuous search in Section 4.1. Section 4.2 presents algorithms for CNN search.

4.1 Basic Ideas

We extend Lemma 1 to Lemma 2. Lemma 2 states that the union of the set of objects on the query path and the sets of objects satisfying the query predicate at nodes on the query path is equivalent to the union of sets of query results at all points on the query path. This is the first basic concept of UNICONS.

Lemma 2 *To perform a continuous search along a path=$\{n_i, n_{i+1}, \ldots, n_j\}$, it is sufficient to retrieve objects on the query path and to run a static query at each node n_k ($i \le k \le j$). Let $\mathcal{R}_{path}$ be the set of objects satisfying the continuous query condition at some point on the query path, $\mathcal{O}_{path}$ be the set of objects on the query path, and $\mathcal{R}_{n_k}$ ($i \le k \le j$) be the set of objects satisfying the query condition at n_k ($i \le k \le j$). Then,*

$$\mathcal{R}_{path} = \mathcal{O}_{path} \cup \mathcal{R}_{n_i} \cup \mathcal{R}_{n_{i+1}} \cdots \cup \mathcal{R}_{n_j}$$

Proof) For any query point $q \in path$, $\mathcal{R}_q$ is defined as the set of objects satisfying the query condition at q. Without loss of generality, we can assume that q belongs to an edge $\overline{n_k n_{k+1}}$ ($i \le k \le j-1$). Thus, $\mathcal{R}_q$ can be represented by Lemma 1 as follows:

$$\mathcal{R}_q \subseteq \mathcal{O}_{\overline{n_k n_{k+1}}} \cup \mathcal{R}_{n_k} \cup \mathcal{R}_{n_{k+1}}$$

Thus,

$$\mathcal{R}_{path} = \bigcup_{q \in path} \mathcal{R}_q \subseteq \mathcal{O}_{path} \cup \mathcal{R}_{n_i} \cup \mathcal{R}_{n_{i+1}} \cdots \cup \mathcal{R}_{n_j} \quad (1)$$

The following inverse formula which exchanges the left side with the right of the equation (1) can be trivially proved using the contradiction method.

$$\mathcal{O}_{path} \cup \mathcal{R}_{n_i} \cup \mathcal{R}_{n_{i+1}} \cdots \cup \mathcal{R}_{n_j} \subseteq \mathcal{R}_{path} \quad (2)$$

From the above equations (1) and (2),

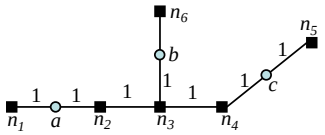
$$\mathcal{R}_{path} = \mathcal{O}_{path} \cup \mathcal{R}_{n_i} \cup \mathcal{R}_{n_{i+1}} \cdots \cup \mathcal{R}_{n_j}$$

■

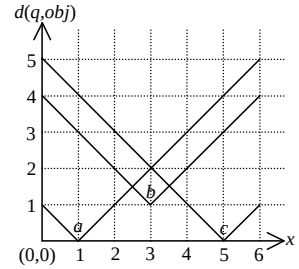
Lemma 2 provides an insight into how we can compute the continuous query result by combining objects on the query path and static query results at nodes on the query path.

Figure 6 shows the change in network distance between a dynamic query point q and three static objects a, b, and c when q moves on the query path $P = \{n_1, n_2, n_3, n_4, n_5\}$. Let x be the total movement length of q while q moves along the query path. For instance, in Figure 6(a), $x = 0$ when q is located at n_1, $x = 1$ when q reaches a, and so forth. The path can be viewed as a line segment with a length of 6. As q moves from n_1 to n_5 along the query path, x changes from 0 to 6. Then, for $x \in [0,6]$, $d(q,a) = |x-1|$ as depicted in Figure 6(b). q reaches b via n_3. Therefore, $d(q,b) = d(q,n_3) + d(n_3,b) = |x-3|+1$. $d(q,c)$ can be computed in the same way as $d(q,a)$ in that both a and c are on the query path. Hence, $d(q,c) = |x-5|$. In this way, the change in the network distance $d(q,obj)$ between a moving query point q and a static object obj on a network can be expressed as a piecewise linear equation. This is the second basic concept of our continuous search algorithms. Therefore, without relying on issuing queries, mathematical analysis can be used in determining NNs.

We can extend the second concept to paths with one way traffic. Suppose that the same path (i.e., $\{n_1, n_2, n_3, n_4, n_5\}$) in Figure 6(a) is given for the network of Figure 7(a). The difference from Figure 6(a) is that the two edges $\overline{n_2 n_3}$ and $\overline{n_3 n_4}$ can be traversed

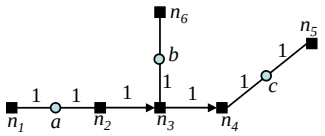


(a) path = $\{n_1, n_2, n_3, n_4, n_5\}$

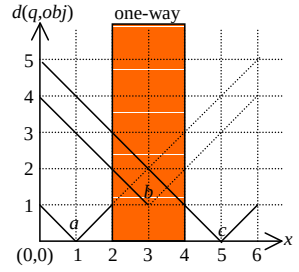


(b) $d(q,a)$=$|x-1|$, $d(q,b)$=$|x-3|$+1, $d(q,c)$=$|x-5|$

Figure 6: $d(q,a)$, $d(q,b)$, and $d(q,c)$



(a) Path = $\{n_1, n_2, n_3, n_4, n_5\}$



(b) $d(q,a)$, $d(q,b)$, and $d(q,c)$

Figure 7: $d(q, obj)$ for the query path with one-way traffic

in only one direction. That is, $d(n_2, n_3) = d(n_3, n_4) = 1$ and $d(n_3, n_2) = d(n_4, n_3) = \infty$.

Figure 7(b) illustrates the impact of the query path with one-way traffic on $d(q,a)$, $d(q,b)$, and $d(q,c)$. Unlike the query path with bi-directional traffic in Figure 6, a path with one-way traffic should be managed carefully since the direction of movement is fixed in a one-way road. For instance, there is no way to move from n_3 to a and from n_4 to b. Thus, $d(n_3, a) = \infty$ and $d(n_4, b) = \infty$ in Figure 7(a). In the case object a is located before the beginning of a one-way road, it is not possible for q to return to a after q has passed node n_2. That is, $d(q,a) = \infty$ for $x \in$ [2,6]. Similarly, after q passes n_3, there is no way for q to reach b. Thus, $d(q,b) = \infty$ for $x \in$ [3,6]. Since c is located after the one-way edges, it is not affected by $\overline{n_2 n_3}$ and $\overline{n_3 n_4}$. After q goes past c, it is possible to return to c. As shown in Figure 7(b), $d(q,a) = |x-1|$ for $x \in$ [0,2], $d(q,a) = \infty$ for $x \in$ (2,6], $d(q,b) = |x-3|+1$ for $x \in$ [0,3], $d(q,b) = \infty$ for $x \in$ (3,6], and $d(q,c) = |x-5|$ for $x \in$ [0,6].

4.2 Algorithms

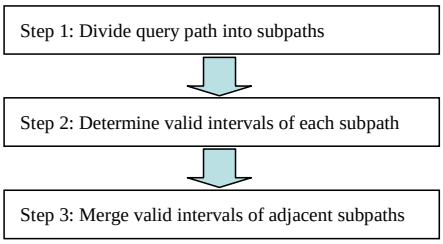


Figure 8: Sketch of CNN Search algorithm

As shown in Figure 8, our CNN search algorithm consists primarily of three subtasks. In Step 1, we divide a query path into multiple subpaths, where intersection points on the query path become the start and end points of each subpath. In Step 2, the algorithm computes valid intervals for the subpaths obtained in Step 1. The details are explained in Figure 9. Finally, in Step 3, the algorithm combines the valid intervals and the query results for their adjacent subpaths to obtain the final result. Since Step 1 and Step 3 can be conducted without difficulty, we focus on Step 2. Among intersection points, the points with larger fanouts are selected and used as condensing points. The reason for this is because, as the fanouts increase, the number of edges to be visited also increases.

Let s_{SP} and e_{SP} be the start and the end points of subpath SP, respectively. Let $\mathcal{S}_{SP}$ and $\mathcal{E}_{SP}$ be the sets of k NNs at the two points s_{SP} and e_{SP}, respectively. Let $\mathcal{O}_{SP}$ be the set of objects on subpath SP. The use of $(k+1)$ NNs at s_{SP} makes it possible to quickly determine whether $\mathcal{S}_{SP}$ is equal to $\mathcal{E}_{SP}$. Lemma 3 shows this fact.

Lemma 3 *Let o_k and o_{k+1} be the k-th and (k+1)-th NNs from point s_{SP}, respectively. Let L_{SP} be the length of the subpath SP. Then, the following is satisfied.*
$d(s_{SP}, o_{k+1}) - d(s_{SP}, o_k) \geq 2 \cdot L_{SP} \rightarrow \mathcal{S}_{SP} = \mathcal{E}_{SP}$ *and* $\mathcal{O}_{SP} \subset \mathcal{S}_{SP}$

Proof) Let o_i $(1 \leq i \leq N)$ be the i-th NN from point s_{SP} where N is the total number of objects. Then, $\mathcal{S}_{SP} = \{o_1, o_2, \ldots, o_k\}$. Let MIN($d(a,b)$) and MAX($d(a,b)$) be the minimum and maximum values of $d(a,b)$, respectively. For $d(e_{SP}, o_i)$, MIN($d(e_{SP}, o_i)$) is $d(s_{SP}, o_i) - L_{SP}$ and MAX($d(e_{SP}, o_i)$) is $d(s_{SP}, o_i) + L_{SP}$. We first show that $\mathcal{S}_{SP} = \mathcal{E}_{SP}$. It suffices to show that MIN($d(e_{SP}, o_{k+1})$) $\geq$ MAX($d(e_{SP}, o_k)$) due to the fact that MIN($d(e_{SP}, o_i)$) $\leq$ MAX($d(e_{SP}, o_i)$), MIN($d(e_{SP}, o_i)$) $\leq$ MIN($d(e_{SP}, o_{i+1})$), and

$\text{MAX}(d(e_{SP}, o_i)) \leq \text{MAX}(d(e_{SP}, o_{i+1}))$. $d(s_{SP}, o_{k+1}) - d(s_{SP}, o_k) - 2 \cdot L_{SP} \geq 0 \leftrightarrow \{d(s_{SP}, o_{k+1}) - L_{SP})\} - \{L_{SP} + d(s_{SP}, o_k)\} \geq 0 \leftrightarrow \text{MIN}(d(e_{SP}, o_{k+1})) - \text{MAX}(d(e_{SP}, o_k)) \geq 0$. Therefore, $\text{MIN}(d(e_{SP}, o_{k+1})) \geq \text{MAX}(d(e_{SP}, o_k))$. Consequently, $\mathcal{E}_{SP} = \{o_1, o_2, \ldots, o_k\}$. Next, we show that $\mathcal{O}_{SP} \subset \mathcal{S}_{SP}$. Since o_j $(k+1 \leq j \leq N)$ cannot be on SP under the given condition, $\mathcal{O}_{SP}$ is a subset of $\mathcal{S}_{SP}$. ■

Lemma 4 enables the algorithm to determine whether a subpath has two or more valid intervals.

Lemma 4 *$\mathcal{S}_{SP}$ is equal to $\mathcal{E}_{SP}$ and $\mathcal{O}_{SP}$ is a subset of $\mathcal{S}_{SP}$ if and only if there is no split point in SP.*

Proof) Consider a query point q on subpath SP. According to Lemma 1,

$$\mathcal{R}_q \subseteq (\mathcal{S}_{SP} \cup \mathcal{O}_{SP} \cup \mathcal{E}_{SP})$$

If $\mathcal{S}_{SP}$ is equal to $\mathcal{E}_{SP}$ and $\mathcal{O}_{SP}$ is a subset of $\mathcal{S}_{SP}$, the union of $\mathcal{S}_{SP}$, $\mathcal{O}_{SP}$, and $\mathcal{E}_{SP}$ has k elements. Therefore, for any point $q \in SP$, $\mathcal{R}_q$ is forced to have the same k NNs. That is, there is no split point in SP. Conversely, if there is no split point in SP, $\mathcal{R}_q$ has the same k NNs for all points on SP. That is, the union of $\mathcal{S}_{SP}$, $\mathcal{O}_{SP}$, and $\mathcal{E}_{SP}$ has k elements. To satisfy this condition, $\mathcal{S}_{SP}$ equals $\mathcal{E}_{SP}$ and $\mathcal{O}_{SP}$ is a subset of $\mathcal{S}_{SP}$. ■

Lemmas 3 and 4 are expected to be more effective when objects are populated sparsely in the network. This is because the distances among objects in a low density data set are longer than those in a high density data set.

Unlike NN queries, CNN queries require the execution of subsequent NN queries at adjacent locations on the query path. The k NNs obtained from s_{SP} can be used as initial candidate NNs at e_{SP}. Let o_i $(1 \leq i \leq k)$ be the i-th NN at s_{SP}. Suppose that $d(s_{SP}, o_i)$ is specified. Then, $d(e_{SP}, o_i)$ is determined as follows: If o_i is on the subpath SP, $d(e_{SP}, o_i)$ is $L_{SP} - d(s_{SP}, o_i)$. If the shortest path from s_{SP} to o_i goes through e_{SP}, $d(e_{SP}, o_i)$ is $d(s_{SP}, o_i) - L_{SP}$. Otherwise, $d(e_{SP}, o_i)$ is $d(s_{SP}, o_i) + L_{SP}$. Other candidates can be found by expanding from e_{SP}

Figure 9 presents the Step 2 of the CNN search algorithm in Figure 8 which determines valid intervals for a subpath. The inputs to the Step 2 of the CNN search algorithm are k and *subpath*, where the names are self descriptive. Algorithms for Filter_Tuples and Find_Split_Points are described in detail in Figures 11 and 12, respectively.

The result $\mathcal{R}$ in this algorithm is the set of (obj, x, y) tuples where *obj* is a qualifying object for the continuous search. Then, x and y are calculated to determine the points of a chart such as the one shown in Figure 6 according to the following conditions:

```
proc CNN_Search (k, subpath SP)
/* k is the number of NNs requested and
SP is a query subpath */
begin
1.  /* R keeps objects satisfying the NN query */
2.  /* Step 2.1. scanning the subpath */
3.  O_SP := Scan_Subpath (SP)
4.  R := O_SP
5.
6.  /* Step 2.2. issuing two NN queries */
7.  S_SP := NN_Search (s_SP, k, kth_dist)
8.  /* Lemma 3 */
9.  if d(s_SP, o_{k+1}) − d(s_SP, o_k) ≥ 2 · L_SP then
10.     return /* no split point is on SP */
11. else
12.     E_SP := NN_Search (e_SP, k, kth_dist)
13.     R := R ∪ S_SP ∪ E_SP
14.
15. /* Lemma 4 */
16. if S_SP = E_SP and O_SP ⊂ E_SP then
17.     return /* no split point is on SP */
18.
19. /* Step 2.3. filtering tuples */
20. Filter_Tuples (k, R)
21.
22. /* Step 2.4. finding split points on SP */
23. Find_Split_Points (k, R)
end
```

Figure 9: CNN search algorithm for the subpath

1. If *obj* is an object on a given subpath, $x := d(s_{SP}, obj)$ and $y := 0$.

2. If *obj* is an object satisfying the query predicate at s_{SP}, $x := 0$ and $y := d(s_{SP}, obj)$.

3. If *obj* is an object satisfying the query predicate at e_{SP}, $x := L_{SP}$ and $y := d(e_{SP}, obj)$.

In the following, we provide further details of the four steps of the CNN search algorithm for a subpath.

Step 2.1. Scanning Subpath from s_{SP} toward e_{SP}

In the first step, the algorithm retrieves the objects on the subpath and adds their corresponding tuples in the form of $(o_i, x_{o_i}, 0)$ to $\mathcal{R}$ where o_i is an object on the query path, and $x_{o_i} := d(s_{SP}, obj)$.

Step 2.2. Issuing two NN Queries at s_{SP} and e_{SP}

In the second step, the CNN search algorithm issues two NN queries at s_{SP} and e_{SP} on the subpath. This step is conducted using the NN search algorithm of Figure 2 and Lemmas 3 and 4. In line 13, the k NNs obtained at s_{SP} and e_{SP} are added to $\mathcal{R}$. In the case of the k NNs obtained at s_{SP}, a tuple

$(o_s, 0, d(s_{SP}, o_s))$ for o_s is added to $\mathcal{R}$. For the k NNs obtained at e_{SP}, a tuple $(o_e, L_{SP}, d(e_{SP}, o_e))$ for o_e is added to $\mathcal{R}$. The k NNs obtained from s_{SP} are used as candidate NNs of a current NN query. Note that e_{SP} of the current subpath corresponds to s_{SP} of the next subpath. If $\mathcal{S}_{SP}$ is equal to $\mathcal{E}_{SP}$ and $\mathcal{O}_{SP}$ is a subset of $\mathcal{E}_{SP}$, the subpath is simply a valid interval and it has the same k NNs by Lemma 4.

Step 2.3. Filtering Tuples

In the third step, among the tuples obtained in the first and second steps, some tuples with the same object in $\mathcal{R}$ are removed using the *cover* relationship.

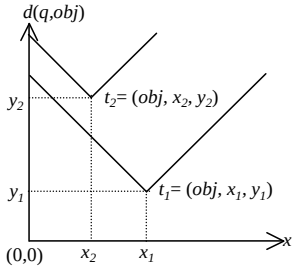


Figure 10: t_1 covers t_2 iff $y_2 \geq |x_2 - x_1| + y_1$

For two tuples t_1 and t_2, let $d(q, t_1.obj)$ and $d(q, t_2.obj)$ be the network distances from q to the object obj in t_1 and the object obj in t_2, respectively. However, note $t_1.obj$ and $t_2.obj$ denote the same object.

Definition 4.1 *For two tuples t_1 and t_2 containing the same object obj, t_1 is said to cover t_2 if $d(q, t_1.obj) \leq d(q, t_2.obj)$.*

For two tuples $t_1 = (obj, x_1, y_1)$ and $t_2 = (obj, x_2, y_2)$ in $\mathcal{R}$, t_1 covers t_2 if and only if $y_2 \geq |x_2 - x_1| + y_1$. Using the second concept in Section 4.1, the network distance from q to $t_1.obj$ is $d(q, t_1.obj) = |x - x_1| + y_1$ and the distance to $t_2.obj$ is $d(q, t_2.obj) = |x - x_2| + y_2$ as shown in Figure 10. To satisfy the cover condition, a point (x_2, y_2) should satisfy the inequality $y \geq |x - x_1| + y_1$ as shown in Figure 10. That is, $y_2 \geq |x_2 - x_1| + y_1$. Since $d(q, t_1.obj)$ and $d(q, t_2.obj)$ denote the different network distances from q to obj, if $d(q, t_1.obj) \leq d(q, t_2.obj)$, the tuple t_2 is found to be unnecessary and removed. Figure 11 presents the algorithm which filters redundant tuples from $\mathcal{R}$.

Step 2.4. Dividing the subpath into valid intervals

In the fourth step, we first divide the subpath into valid intervals. This is based on the divide and conquer method. Next, to retrieve the k NNs for each valid interval, all we have to do is to simply determine the k smallest $d(q, t.obj)$ values at a point in the corresponding interval, where $d(q, t.obj)$ denotes the

proc Filter_Tuples (k, $\mathcal{R}$)
begin
1. **while** there are distinct tuples $t_i, t_j \in \mathcal{R}$ **do**
2. **if** $t_i.obj = t_j.obj$ **and** $d(q, t_i.obj) \leq d(q, t_j.obj)$ **then**
3. $\mathcal{R} := \mathcal{R} - \{t_j\}$
4. **end-while**

end

Figure 11: Filter_Tuples algorithm

network distance from q to obj in tuple t. Lemma 1 shows that query result $\mathcal{R}_q$ at any point $q \in SP$ is a subset of the union of $\mathcal{O}_{SP}$, $\mathcal{S}_{SP}$, and $\mathcal{E}_{SP}$. Consequently, valid intervals of SP can be determined using objects which belong to $\mathcal{O}_{SP}$, $\mathcal{S}_{SP}$, and $\mathcal{E}_{SP}$. To divide a subpath into valid intervals, the algorithm computes all cross points of line segments drawn for $d(q, t.obj)$s where $t.obj$ belongs to the union of $\mathcal{O}_{SP}$, $\mathcal{S}_{SP}$, and $\mathcal{E}_{SP}$. Then, it adds their x values to $\mathcal{X}$, the set of x values for the cross points. The x values in $\mathcal{X}$ are sorted in ascending order. Each pair of neighboring values in $\mathcal{X}$, x_k and x_{k+1}, is used to identify a valid interval. Note that k NN queries are not issued to retrieve the k NNs for each pair of neighboring values. In each valid interval, the k NNs with the k smallest $d(q, t.obj)$s are the query result. If adjacent valid intervals have the same query result, they are merged into a single valid interval. Figure 12 presents the algorithm which determines the split points on the subpath.

Example 1: For an effective explanation, we proceed with an exemplary CNN query which is "for static objects a to e on the road network of Figure 13, continuously display the two closest objects from any position on the query path = $\{n_3, n_5, n_7, n_8\}$."

Step 1 First, we divide the entire query path into multiple subpaths on the basis of intersection points of the query path. Then, we obtain two subpaths SP_1 = $\{n_3, n_5, n_7\}$ and $SP_2 = \{n_7, n_8\}$.

Step 2 We determine the valid intervals for the two subpaths. For simplicity, we focus on computing valid intervals for $SP_1 = \{n_3, n_5, n_7\}$.

Step 2.1 We scan subpath SP_1 to collect the objects on SP_1. Let $\mathcal{O}_{SP_1}$ be the set of (obj, x, y) tuples containing the objects on SP_1, where obj, x, and y are defined previously. Then, $\mathcal{O}_{SP_1} = \{(c, 2, 0)\}$. This means that the distance to c from the start point n_3 of SP_1 is 2.

Step 2.2 Let $\mathcal{S}_{SP}$ and $\mathcal{E}_{SP}$ be the sets of (obj, x, y) tuples for the objects satisfying the query condition at the start point and the end point of subpath SP respectively, where obj, x, and y are defined previously.

Two NN queries are executed at the start point n_3 and the end point n_7 of SP_1. Then, $\mathcal{S}_{SP_1}$ = $\{(a, 0, 1), (b, 0, 1)\}$ and $\mathcal{E}_{SP_1} = \{(e, 4, 1), (c, 4, 2)\}$. Note that in $\mathcal{E}_{SP_1}$, the tuple $(c, 4, 2)$ may be replaced with

proc Find_Split_Points (k, $\mathcal{R}$)
begin
1. **while** there are distinct tuples $t_i, t_j \in \mathcal{R}$ **do**
2. (x_+, y_+) := Compute_Cross_Points(t_i, t_j)
3. /* let $\mathcal{X}$ be the set of x values of cross points */
4. $\mathcal{X} := \mathcal{X} \cup \{x_+\}$
5. **end-while**
6.
7. $\mathcal{R}_{SP} = \emptyset$ /* $\mathcal{R}_{SP}$ is a set of (I, R_I) tuples */
8. /* obtain k NNs for each valid interval */
9. /* let x_i and x_{i+1} be two adjacent x values
10. and $R_{[a,b]}$ be the set of k NNs in $[a, b]$ */
11. **for** x_i, x_{i+1} ($x_i, x_{i+1} \in \mathcal{X}$ and $x_i \leq x_{i+1}$) **do**
12. $\{([x_i, x_{i+1}], R_{[x_i,x_{i+1}]})\}$
13. := Compute_k_NN_Objects ($\mathcal{R}, [x_i, x_{i+1}]$)
14. $\mathcal{R}_{SP} = \mathcal{R}_{SP} \cup \{([x_i, x_{i+1}], R_{[x_i,x_{i+1}]})\}$
15. **end-for**
16.
17. **while** adjacent intervals have the same k NNs **do**
18. **if** $R_{[x_i,x_j]} = R_{[x_j,x_k]}$ **then**
19. $R_{[x_i,x_k]} := R_{[x_i,x_j]}$
20. $\mathcal{R}_{SP} = \mathcal{R}_{SP} - \{([x_i, x_j], R_{[x_i,x_j]}), ([x_j, x_k], R_{[x_j,x_k]})\}$
21. $\mathcal{R}_{SP} = \mathcal{R}_{SP} \cup \{([x_i, x_k], R_{[x_i,x_k]})\}$
22. **end-while**
23. **return** $\mathcal{R}_{SP}$
end

Figure 12: Find_Split_Points algorithm

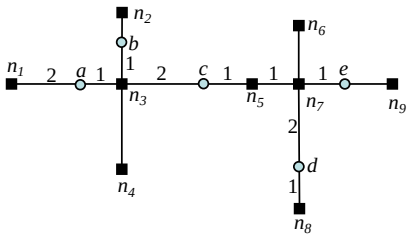


Figure 13: Example road network

the tuple $(d, 4, 2)$ since $d(n_7, c)$ is equal to $d(n_7, d)$ and object d can also be contained in the query result at n_7 instead of object c. As a result, $\mathcal{R}$ becomes the union of $\mathcal{O}_{SP_1}$, $\mathcal{S}_{SP_1}$, and $\mathcal{E}_{SP_1}$ as follows: $\mathcal{R}$ = {(obj, x, y)|(c,2,0),(a,0,1),(b,0,1),(c,4,2),(e,4,1)}. Figure 14 shows the result of mapping 5 tuples in $\mathcal{R}$ on 5 points where the second and third attributes of tuples are used as x and y coordinates of the points on a 2-dimensional chart. The x-axis represents the movement length of q from the start position n_3 of the CNN search. The y-axis represents the network distance from q on the query path to all objects a, b, c, d, and e in $\mathcal{R}$. This can be explained by the second basic concept of Section 4.1. For instance, when $x = 0$, $d(q, a) = d(q, b) = 1$, $d(q, c) = 2$, $d(q, c) = 6$ and $d(q, e) = 5$, when $x = 2$, $d(q, a) = d(q, b) = 3$, $d(q, c) = 0$, $d(q, c) = 4$ and $d(q, e) = 3$, etc.

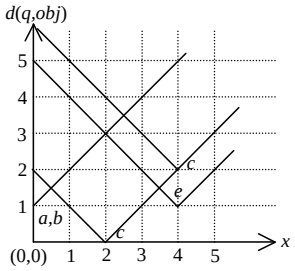


Figure 14: Plotting tuples in $\mathcal{R}$ on the chart

Step 2.3 Some tuples in Figure 14 are removed by using the *cover* relationship. Figure 15 shows the result after removing a redundant tuple $\{(c, 4, 2)\}$ in Figure 14.

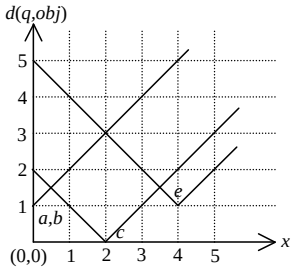


Figure 15: Filtering tuples

Step 2.4 As stated previously, a tuple (obj, x_1, y_1) on the chart gives the network distance from q to obj as follows: $d(q, obj) = |x - x_1| + y_1$ for $x \in [0, 4]$ where 4 is the length of the subpath. In Figure 15, since $d(q, a) = d(q, b) = |x| + 1$, $d(q, c) = |x - 2|$, and $d(q, e) = |x - 4| + 1$, Two NNs are a and b when $x = 0$, two NNs are c and e when $x = 3$, and so forth. Therefore, to find two NNs at any point on the query path, we simply examine the chart to identify the k smallest $d(q, obj)$ values at the corresponding point. This is similar to a well-known skyline problem [2] such as drawing the skyline of a city given the locations of the buildings in the city. The first skyline to be drawn with the smallest $d(q, obj)$ values at each point becomes the set CNN_{1st} of the closest neighbors from any point on the query path. That is, CNN_{1st} = {(I, R_I)|([0, $\frac{1}{2}$], {a or b)}), ([$\frac{1}{2}$, 3$\frac{1}{2}$], {c}), ([3$\frac{1}{2}$, 4], {e})}.

Based on the divide and conquer method, the subpath is broken into valid intervals which are determined by cross points of line segments on the chart. Cross points are easily obtained using linear equations. As shown in Figure 16, the query path is divided into 4 valid intervals as follows: I_1=[0, $\frac{1}{2}$], I_2=[$\frac{1}{2}$, 2], I_3=[2, 3$\frac{1}{2}$], and I_4=[3$\frac{1}{2}$, 4]. In each valid interval, we just find the k objects with the k smallest $d(q, obj)$ values. For example, for $I_1 = [0, \frac{1}{2}]$, a and b are the 2 nearest neighbors and their network distances from q are $d(q, a) = d(q, b) = |x| + 1$. Similarly, for I_2 = [$\frac{1}{2}$,2],

c and a are the two NNs and their network distances are $d(q,c) = |x-2|$ and $d(q,a) = |x|+1$. For all valid intervals, their query results are shown in Figure 17(a) and the final CNN search result is shown in Figure 17(b) since I_3 and I_4 are merged into $I'_3 = [2, 4]$ since they all have the same qualifying objects c and e. Hence, $\mathcal{R}_{SP_1} = \{(I, R_I)|\ ([0, \frac{1}{2}], \{a, b\}),\ ([\frac{1}{2}, 2], \{a, c\}),\ ([2, 4], \{c, e\})\}$.

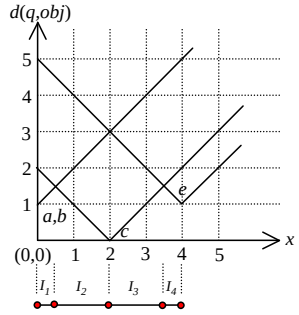


Figure 16: Dividing the query path into valid intervals

I	R_I
I_1=[0, $\frac{1}{2}$]	{a, b}
I_2=[$\frac{1}{2}$, 2]	{c, a}
I_3=[2, 3$\frac{1}{2}$]	{c, e}
I_4=[3$\frac{1}{2}$, 4]	{c, e}

(a) Initial result

I	R_I
I_1=[0, $\frac{1}{2}$]	{a, b}
I_2=[$\frac{1}{2}$, 2]	{a, c}
I'_3=[2, 4]	{c, e}

(b) Final result

Figure 17: CNN search result for $SP_1 = \{n_3, n_5, n_7\}$

Step 3 In Step 2, we obtained the CNN query result for $SP_1 = \{n_3, n_5, n_7\}$. That is, $\mathcal{R}_{SP_1} = \{(I, R_I)|\ ([0, \frac{1}{2}], \{a, b\}),\ ([\frac{1}{2}, 2], \{a, c\}),\ ([2, 4], \{c, e\})\}$. In the same way, if we compute the CNN query result for $SP_2 = \{n_7, n_8\}$, $\mathcal{R}_{SP_2} = \{(I, R_I)|\ ([4, 7], \{d, e\})\}$. In Step 3, we simply merge $\mathcal{R}_{SP_1}$ and $\mathcal{R}_{SP_2}$ for SP_1 and SP_2, respectively, in order to the final query result for the entire query path $P = \{n_3, n_5, n_7, n_8\}$. Consequently, $\mathcal{R}_P = \{(I, R_I)|\ ([0, \frac{1}{2}], \{a, b\}),\ ([\frac{1}{2}, 2], \{a, c\}),\ ([2, 4], \{c, e\}),\ ([4, 7], \{d, e\})\}$.

5 Performance Study

In Section 5.1, we experimentally compare our algorithms and other algorithms for NN queries in terms of I/O cost using a system running Windows on a 2.7 GHz processor and 512 MB memory. In Section 5.2, we explore the performance of CNN queries in terms of disk I/O and execution time using the same system. In most cases, the costs of NN queries are disk I/O bound and the computational cost is considered to be trivial in comparison to the I/O cost while the computational cost is non-trivial for CNN queries.

To represent a road network, we use real road data for Wisconsin in the United States from Tiger/Line data [12]. We set the page size to 4 KB and employ an LRU buffer of 16 MB which accommodates approximately 10% of the road data. The road data description is as follows: $|N|$ = 1,469,468 and $|E|$ = 1,594,867, where N is the set of nodes and E is the set of edges. The working space is fixed to the two-dimensional unit space $[0,1)^2$. For simplicity, we consider bidirectional edges. However, this does not affect the interpretability and value of the results. The number of intersection points is 223,569. Among them, we have chosen 64,748 (about 4.4% of total number of nodes) nodes as condensing points where four or more edges meet. Each condensing point maintains the NN list which consists of 10 precomputed NNs. Naturally, if the size of the precomputed NN list is larger, the query cost decreases while the maintenance cost increases. Due to the high maintenance cost, we do not employ more than 10 precomputed NNs.

We use real-world data sets also from Tiger/Line data that represent shopping centers, campgrounds, parks, schools, and lakes or ponds in Wisconsin. The sets contain 178, 423, 1154, 2979, and 5177 objects, respectively. In order to simulate a large dataset such as that of restaurants, which is not available in Tiger/Line data, we make a composite data set which consists of the points from the sets for shopping centers, campgrounds, parks, schools, and lakes or ponds. According to the response of the Wisconsin State Government to our inquiry, there are 11,215 restaurants in Wisconsin. To investigate the performance of NN queries, we execute workloads of 200 queries whose locations are randomly selected on the network. Similarly, we also perform workloads of 100 CNN queries whose initial locations are also randomly distributed in a road network and next edges are selected with even probability.

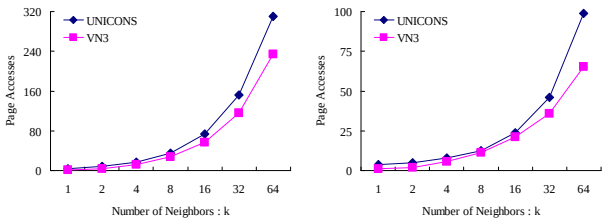
5.1 NN queries

We compare query costs of UNICONS for NN queries with those of VN^3 [7], which is currently regarded as the best approach for NN queries on the road network.

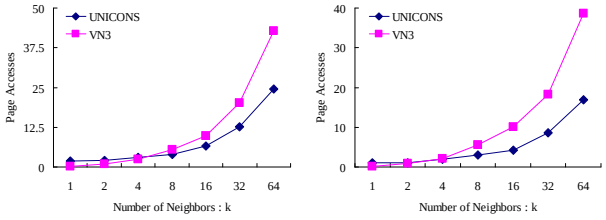
Figure 18 shows the number of disk I/Os incurred by each of the two methods for the value of k ranging from 1 to 64. As the dataset cardinality increases, the number of page accesses drops quickly. When the value of k is 1, VN^3 generates the result set with constant cost, regardless of the density of objects. This is expected because the implementation of VN^3 is based on the Voronoi diagram which is efficient to find the first NN. UNICONS shows performance as good as VN^3 for NN queries which require less than 10 NNs because it maintains the NN lists of precomputed 10 NNs at the condensing points. As shown in Figures 18(a) and 18(b), VN^3 is more efficient than UNICONS since UNICONS requires a larger portion of network to be retrieved due to the very low density of objects. VN^3 also requires precomputed values to be retrieved from

the database and the number of these precomputed values required increases for lower densities and larger values of k. However, UNICONS suffers more than VN^3 from the lower densities due to the increase in the search space.

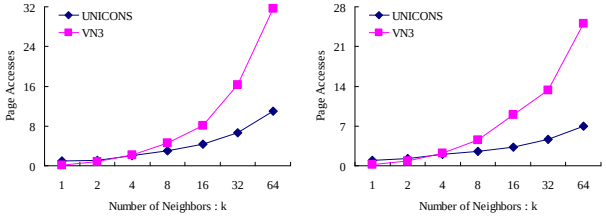
On the other hand, as shown in Figures 18(c), 18(d), 18(e), and 18(f), UNICONS outperforms VN^3 with the increase in the density of objects. Since both methods process queries by using precomputed information, the performance gap between UNICONS and VN^3 is closely associated with the density of objects. In the case where objects (e.g., shopping centers) are distributed sparsely, VN^3 has an advantage over UNICONS since the number of NVPs is as small as that of objects. Conversely, in the case where objects (e.g., composite data) are distributed densely, UNICONS is in a stronger position than VN^3 due to the reduced search space and the help of condensing points. Additionally, in UNICONS, objects sharing the same edge have the same search key and therefore, fewer disk I/Os are required. As a whole, the experimental results indicate that except for the particular cases with lower object densities, UNICONS outperforms VN^3 by up to a factor of 3.5 and the performance difference between the two approaches increases with the density of objects and the value of k.



(a) Shopping centers (178) (b) Campgrounds (423)



(c) Parks (1154) (d) Schools (2979)

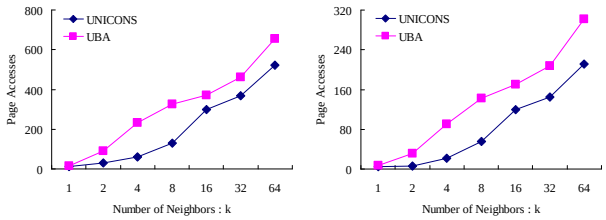


(e) Lakes or ponds (5177) (f) Composite data (9911)

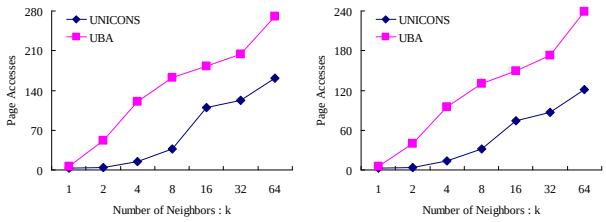
Figure 18: Performance comparison for NN queries

5.2 CNN queries

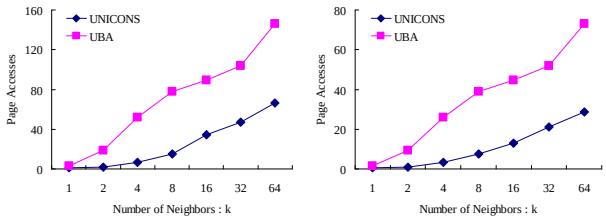
We conducted several experiments to compare the performance of UNICONS with its competitor, the UBA approach presented in [8]. We calculated the number of page accesses and the required times for different values of k.



(a) Shopping centers (178) (b) Campgrounds (423)



(c) Parks (1154) (d) Schools (2979)

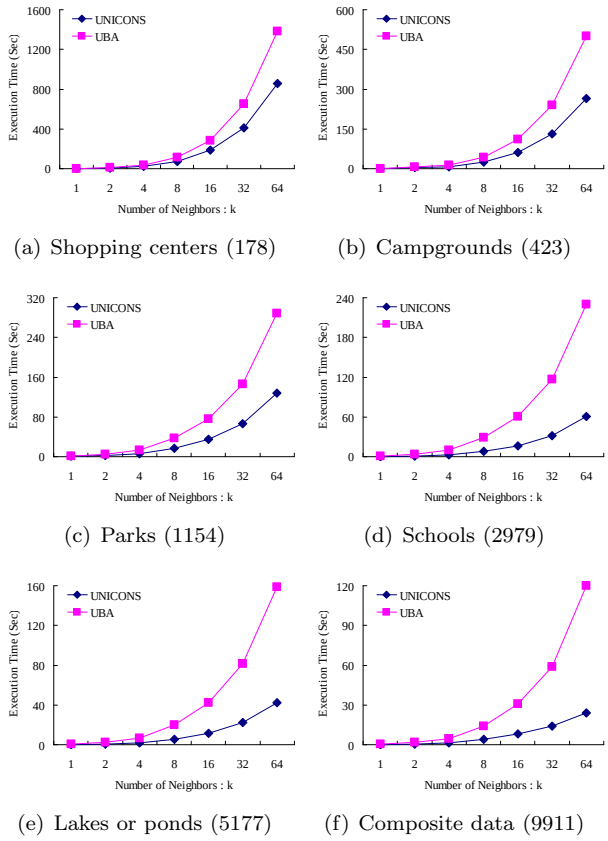


(e) Lakes or ponds (5177) (f) Composite data (9911)

Figure 19: Page accesses for CNN queries

Figure 19 depicts the number of page accesses for UNICONS and UBA approaches when the length of query paths is fixed to 0.05 and the value of k varies from 1 to 64. As shown in Figure 19, UNICONS always outperforms UBA. When the objects of interest are distributed densely in the network (e.g., composite data), the performance of UNICONS is up to 5 times better than that of UBA. The reason for this is that UBA requires a large number of NN queries. Such a trend is striking, particularly when the value of k is large. The advantage of UNICONS over UBA is minimal when the objects of interest are distributed sparsely (e.g., shopping centers). In these cases, UBA can filter out several adjacent nodes from the computation of NNs. Based on Lemmas 3 and 4, UNICONS can also avoid the execution of NN queries from intersection points on the query path. The use of condensing points is very helpful for answering CNN queries. This is due to the fact that the CNN search algorithms

of UNICONS require the execution of consecutive NN queries at intersection points only regardless of the density of objects and the value of k.



(a) Shopping centers (178) (b) Campgrounds (423)

(c) Parks (1154) (d) Schools (2979)

(e) Lakes or ponds (5177) (f) Composite data (9911)

Figure 20: Execution time for CNN queries

Figure 20 illustrates the query execution times for UNICONS and UBA when the length of query paths is fixed to 0.05 and the value of k varies from 1 to 64. Naturally, the execution time of UBA increases greatly with the value of k. The increase in the value of k generates more split points and leads to the execution of a large number of (k+1) NN queries on the query path. However, UNICONS issues NN queries at intersection points on the query path regardless of the density of objects. The experiments for traveling paths of length between 0.01 and 0.1 show similar trends. The experimental results confirm that UNICONS is superior to UBA and it is optimized for CNN queries.

6 Conclusions

In this paper, we developed new continuous search algorithms which answer NN queries at any point of a given path. We also verified that our continuous search algorithms require a small number of static queries in producing the continuous search result. Experimental results with TIGER/Line data demonstrated that UNICONS outperforms its competitors for various numbers of NNs and data sets.

7 Acknowledgments

This research was supported in part by the Agency for Defense Development, Korea, through the Image Information Research Center at Korea Advanced Institute of Science and Technology, and in part by the Ministry of Information and Communications, Korea, under the Information Technology Research Center (ITRC) Support Program.

References

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. of the ACM SIGMOD*, 1990.

[2] S. Borzsonyi, D. Kossmann, and K. Stocker: The Skyline Operator. In *Proc. of ICDE*, 2001.

[3] E. Dijkstra: A Note on Two Problems in Connection with Graphs. *Numeriche Mathematik, Volume(1)*, 1959.

[4] J. Feng and T. Watanabe: A Fast Method for Continuous Nearest Target Objects Query on Road Network. In *Proc. of Virtual Systems and Multi-Media*, 2002.

[5] J. Feng and T. Watanabe: Search of Continuous Nearest Target Objects along Route on Large Hierarchical Road Network. In *Proc. of the Data Engineering Workshop*, 2003.

[6] C. Jensen, J. Kolarvr, T. Pedersen, and I. Timko: Nearest neighbor queries in road networks. In *Proc. of ACM GIS*, 2003.

[7] M. Kolahdouzan and C. Shahabi: Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases. In *Proc. of VLDB*, 2004.

[8] M. Kolahdouzan and C. Shahabi: Continuous K-Nearest Neighbor Queries in Spatial Network Databases. In *Proc. of STDBM*, 2004.

[9] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao: Query Processing in Spatial Network Databases. In *Proc. of VLDB*, 2003.

[10] C. Shahabi, M. Kolahdouzan, and M. Sharifzadeh: A road network embedding technique for k-nearest neighbor search in moving object databases. In *Proc. of ACM GIS*, 2002.

[11] S. Shekhar and J. Yoo: Processing in-route nearest neighbor queries: a comparison of alternative approaches. In *Proc. of ACM GIS*, 2003.

[12] US Bureau of the Census: Technical Documentation. TIGER/Line Files. 1995.