Continuous Visible Nearest Neighbor Queries

Yunjun Gao Baihua Zheng
Singapore Management University
{yjgao, bhzheng}@smu.edu.sg

Wang-Chien Lee
Penn State University
wlee@cse.psu.edu

Gencai Chen
Zhejiang University
chengc@zju.edu.cn

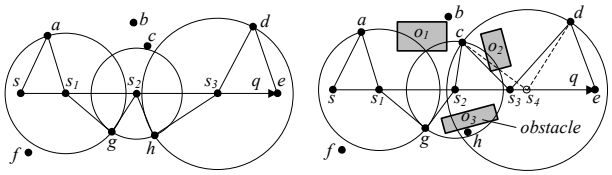
ABSTRACT

In this paper, we identify and solve a new type of spatial queries, called *continuous visible nearest neighbor* (CVNN) search. Given a data set P, an obstacle set O, and a query line segment q, a CVNN query returns a set of $\langle p, R\rangle$ tuples such that $p \in P$ is the nearest neighbor (NN) to every point r along the interval $R \subseteq q$ as well as p is *visible* to r. Note that p may be NULL, meaning that all points in P are *invisible* to all points in R, due to the obstruction of some obstacles in O. In this paper, we formulate the problem and propose efficient algorithms for CVNN query processing, assuming that both P and O are indexed by R-trees. In addition, we extend our techniques to several variations of the CVNN query. Extensive experiments verify the efficiency and effectiveness of our proposed algorithms using both real and synthetic datasets.

1. INTRODUCTION

The *continuous nearest neighbor* (CNN) search is an important operator in spatial databases that has been well-studied [24, 25, 27]. Let P be a set of points in a multi-dimensional space. Given a query line segment q, a CNN query retrieves the nearest neighbor (NN) of every point on segment q over P. The result of CNN retrieval, denoted by $CNN(q)$, contains a set of $\langle p, R\rangle$ tuples, such that $p \in P$ is the NN of each point r along the interval $R \subseteq q$, i.e., $\forall r \in R$, $\forall p' \in P - \{p\}$, $dist(p, r) \leq dist(p', r)$[1]. An example is shown in Figure 1(a), with dataset $P = \{a, b, c, d, f, g, h\}$ and query line segment $q = [s, e]$. $CNN(q) = \{\langle a, [s, s_1]\rangle, \langle g, [s_1, s_2]\rangle, \langle h, [s_2, s_3]\rangle, \langle d, [s_3, e]\rangle\}$, which indicates that point a is the NN for any point along the interval $[s, s_1]$, and point g is the NN for any point along the interval $[s_1, s_2]$, and so on. Points s_1, s_2, s_3 are called *split points*, as the NN object changes at those points.

The CNN query does not take obstacles into consideration. However, many physical obstacles (e.g., buildings, mountains, and blindages, etc.) exist in the real world, and their presence may affect the visibility/distance between two points and hence the result of spatial queries, such as range query, NN search, and spatial join, etc. Moreover, users may be only interested in the objects that are visible or reachable to them. Based on these facts, the *visible nearest neighbor* (VNN) search, which returns the closest object that is *visible* to a given query point, was proposed in [15]. An example of VNN query issued at s_4 is depicted in Figure 1(b). The answer point is d. Although point h is closer to s_4 than d, it is blocked by obstacle o_3, represented by the shaded rectangle[2] in Figure 1(b), and thus is excluded from the result.



(a) CNN search (b) CVNN search

Figure 1: Example of CNN and CVNN queries

So far, the existing work on VNN queries only considers *single query point*. With the ever growing popularity of smart mobile devices and rapid advance of wireless technology, more and more users actually submit queries even when they are moving around. Nowadays, location-based service is expected to locate information for the users not only based on fixed point locations but also based on moving trajectories. Motivated by the lack of efficient algorithm for VNN query processing over moving trajectories, we, in this paper, study *continuous visible nearest neighbor* (CVNN) search, a novel form of VNN retrieval. Given a data set P, an obstacle set O, and a query line segment q, a CVNN query retrieves the VNN for each point on q. In particular, the CVNN search aims at finding a set of $\langle p, R\rangle$ tuples, where $p \in P$ is the VNN for any point r in the interval $R \subseteq q$. It is important to note that p may be *empty*, which means that all points in P are *invisible* to all points on R due to the obstruction of obstacles in O. The query result set, denoted by $RL(q)$, contains a set of $\langle p, R\rangle$ tuples. Continuing the example in Figure 1(b), with $O = \{o_1, o_2, o_3\}$, the result of the CVNN query is $RL(q) = \{\langle a, [s, s_1]\rangle, \langle g, [s_1, s_2]\rangle, \langle c, [s_2, s_3]\rangle, \langle d, [s_3, e]\rangle\}$. It indicates that point a is the VNN for any point along interval $[s, s_1]$; point g is the VNN for any point along interval $[s_1, s_2]$, and so forth. Note that point h is the NN for interval $[s_2, s_3]$ in the traditional CNN retrieval,

[1]Without loss of generality, $dist(p_i, p_j)$ computes the Euclidean distance between any two data points p_i and p_j.

[2]Although an obstacle can be in any shape (e.g., triangle, pentagon, etc.), we assume it is a rectangle in this paper.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

whereas it is not the VNN for $[s_2, s_3]$ in the CVNN search because of obstacle o_3. CVNN search has many variants. Due to space limitation, we only present two variants, namely *continuous visible k nearest neighbor* (CVkNN) query and δ-CVNN retrieval. The former retrieves the k visible nearest neighbors (VNNs) for every point on a specified query line segment q and the latter reports the VNN of each point on q with its distance to q bounded by a *threshold* δ. These potential variants constitute a suite of interesting and practical problems from both the research point of view and the application point of view.

CVNN query is interesting not only because it introduces some new challenges but also because it is really useful in real life applications, ranging from location-based commerce, to interactive online games, and to decision support, to name but a few. Some example applications are listed as follows.

Tourist Recommendation. A CVNN query can identify all the scenes (e.g., temple, stele, pagoda, etc.) for a given tourist traveling route, defined by a starting point s and an ending point e. Different from conventional CNN query, CVNN retrieval considers all the physical obstacles such as mountains and buildings. Consequently, the returned results provide more accurate information in terms of visibility. Note that in this case the purpose of CVNN search differs from that of *opposite query* which finds suitable route(s) from a given set of scenes (e.g., *trip planning query* [11] etc.).

First Person Shooting (FPS) Games. There are many simulative obstacles (e.g., buildings, blindages, etc.) in FPS games (e.g., call of duty). We assume a player or a non-player character (NPC) moves from one shelter to another when heading towards his/her destination. Consequently, CVNN search is invoked for a player or an NPC to find nearest adversary shelters in order to make effective shooting or concealment during the movement.

Outdoor Advertisement. Assume that an advertisement company plans to place outdoor advertisement posters at some popular locations in a downtown area. By conducting a CVNN search which takes as input a set of poster billboards P, a set of obstacles (e.g., buildings) O, and an anticipated trajectory q (representing a specified commercial street), the company can find a set of $\langle loc, R\rangle$ tuples, such that location $loc \in P$ is the VNN of all customers shopping in the corresponding region R (i.e., an interval of q). Thus, it provides a near optimal selection of the poster billboards in order to guarantee the visibility of the posters.

Owing to the big application base, efficient search algorithms for CVNN are required. Several spatial queries have taken the presence of obstacles into consideration, such as (i) *obstructed NN* (ONN) query [33] that is to find top-k (≥ 1) points in a data set P with the smallest *obstructed distances*[3] to a specified query point q, based on an obstacle set O; (ii) *VNN* search that is to find the nearest point *visible* to a given query point [15]; and (iii) *clustering spatial data in the presence of obstacles* that tries to divide a set of data points in a 2D space into smaller homogeneous groups considering the influence of physical obstacles [5, 28, 31]. Nevertheless, to the best of our knowledge, research on exploring the VNN query in a continuous fashion, i.e., CVNN, has not been studied in the literature. We aim at proposing efficient CVNN query processing algorithms in this paper. The main idea is to perform a *single* navigation for the *whole* input line segment, and enable effective *pruning heuristics* on the point set and obstacle set respectively to improve the search performance. Our approaches are based on conventional data-partitioning indexes (e.g., R-trees [2]). The proposed search algorithms are highly flexible which can be easily extended to support different variants of CVNN queries. Finally, extensive experiments with both real and synthetic datasets are conducted to verify the performance of our proposed algorithms in terms of both efficiency and scalability.

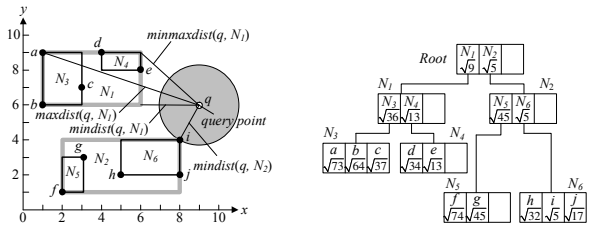
The rest of the paper is organized as follows. Section 2 reviews the related work on NN, CNN, and VNN queries, respectively. Section 3 provides preliminary background for our research, including the definitions and characteristics of CVNN problem. Section 4 proposes efficient CVNN query processing algorithms, with a data set P and an obstacle set O indexed by two separate R-trees and a unified R-tree, respectively. Section 5 extends our solution to deal with CVNN variants. Section 6 presents the performance evaluation of our proposed algorithms and reports our findings. Finally, Section 7 concludes the paper with some directions for future work.

2. RELATED WORK

In this section, we briefly review existing work related to CVNN queries, namely, NN search based on R-trees, CNN search, and VNN search.

2.1 Algorithms for NN Search on R-trees

Among all the index structures available for spatial objects, R-tree and its variants [2, 7, 20] are the most well-received due to their simplicity and efficiency. Figure 2 shows a set of data points $P = \{a, b, \cdots, j\}$ in a 2D space, and the corresponding R-tree with node fanout set to three. Note that in Figure 2(b) the number in each entry refers to the mindist between the query point q and the corresponding *Minimum Bounding Rectangle* (MBR) of the entry. As a leaf entry e refers to a point p of P, its mindist to q is the real distance from p to q. These numbers are not stored in R-tree previously but computed *on-the-fly* during query processing based on a given query point q.



(a) The dataset placement (b) R-tree (fanout = 3)

Figure 2: Example of an R-tree and a NN query

An NN query specifies a query point q, and finds the object in a dataset P that is closest to q. The NN search algorithms on R-trees traverse the R-tree of P in a branch-and-bound manner and use some distance metrics, including (i) mindist(q, N), (ii) maxdist(q, N), and (iii) minmaxdist(q, N), to prune the search space [3, 8, 19]. Here, q is a query point and N is an R-tree node which corresponds to an MBR, together with all the points covered. Figure 2(a) illustrates

[3]The obstructed distance between any two points $p_i, p_j \in P$ is defined as the length of the shortest path that connects p_i to p_j without crossing any obstacle from O.

these pruning metrics between q and nodes N_1 and N_2. Existing algorithms for NN retrieval follow either *depth-first* (DF) or *best-first* (BF) traversal paradigm. DF algorithms [3, 19] start from the root, and visit recursively the node with the smallest mindist from q until the leaf level where a potential NN is found. Subsequently, the algorithm conducts backtracking operation. In particular, during backtracking to the upper levels, DF only visits those entries with minimum distances to the query point smaller than the real distance between the query point and the NN candidate already retrieved. As demonstrated in [18], the DF algorithm is suboptimal, i.e., it accesses more nodes than necessary.

The BF algorithm proposed in [8] achieves the optimal I/O performance by visiting only the nodes necessary for obtaining the NN. In order to implement this, it maintains a priority queue H, with the entries visited so far sorted in ascending order of their mindist. It recursively examines the top entry ϵ of H. If ϵ is a node, its child entries are dequeued for later exploration. If ϵ is an object, it is reported as the answer object. Both DF and BF can be easily extended to retrieve k (> 1) nearest neighbors (kNNs). Compared to DF, BF is incremental as it returns the answer objects in ascending order of their distances to the query point, and hence k does not need to be known in advance, which allows different termination conditions.

In addition, other versions of NN search have been investigated as well, such as *constrained* NN [6], *group* NN [16], *aggregate* NN [17], *all* NN [32], *range* NN [9], and *surface* kNN [4] queries, etc.

2.2 Continuous Nearest Neighbor Search

The CNN search has received considerable attention since it was first introduced by Sistla *et al.* [23] in spatial-temporal databases. The authors presented modeling methods and query languages for the expression of CNN queries, but not the processing approaches. The first algorithm for CNN query processing, based on periodical sampling technique, is proposed in [24]. Due to the natural disadvantage of sampling, its performance highly depends on the number and positions of those sampling points and the accuracy cannot be guaranteed. In order to conduct *exact* search, two CNN query algorithms based on R-trees are proposed in [25, 27]. The first algorithm is based on the concept of *time-parameterized* (TP) queries, which treats a query line segment as the moving trajectory of a query point [25]. Thus, the nearest object to the moving query point is valid only for a limited duration and a new TP query is issued to retrieve the next nearest object once the valid time of the current query expires, i.e., when a split point is reached. Although the TP approach avoids the drawbacks of sampling, it needs issue m TP queries with m being the number of answer objects. In order to improve the performance, the second algorithm [27] retrieves all the answer points for the whole query line segment in one single round.

In addition, there are some existing work related to other variations of CNN search in the literature, such as (i) CNN monitoring in the Euclidean space [12, 29, 30], (ii) CNN monitoring in road network [14], and (iii) CNN monitoring in a distributed environment [13]. As pointed out in Section 1, although conventional CNN query has been well-studied, it does not consider obstacles that exist in many real-life scenarios. Hence, existing solutions for CNN queries cannot be directly applied to handle CVNN search efficiently.

2.3 Visible Nearest Neighbor Search

Although visibility computation algorithms have been well-studied in the area of computer graphics and computational geometry [1], there are only a few works on *visibility queries* in the database community [10, 21, 22]. The existing methods utilize various indexing structures (e.g., LoD-R-tree [10], HDoV-tree [22], etc.) to deal with visibility queries in visualization systems. Since these specialized access methods are designed only for the purpose of visualization without maintaining any distance information, they are not capable of supporting efficient CVNN query processing. Recently, Nutanong *et al.* [15] introduce the VNN search which retrieves the nearest object that is visible to a given query point (e.g., point d is the VNN of s_4 in Figure 1(b)) as mentioned earlier. A VNN query algorithm, based on the fact that a farther object cannot affect the visibility of a nearer object, is proposed in [15]. The basic idea is to conduct NN search and check its visibility condition in an incremental manner. However, the algorithm is only for a fixed query point, but not a line segment which contains multiple query points.

3. PRELIMINARY

In this section, we first present problem definitions for CVNN search and then explain some of its characteristics that can facilitate the development of efficient algorithms for CVNN query.

Given a set of data points $P = \{p_1, p_2, \cdots, p_n\}$, a set of obstacles $O = \{o_1, o_2, \cdots, o_m\}$, and a query line segment $q = [s, e]$ in a 2D space, visibility between two points p and p' is defined in Definition 1, based on which we formulate VNN and CVNN searches in Definition 2 and Definition 3, respectively.

DEFINITION 1. **Visibility**. *Given $p, p' \in P$, p and p' are* visible *to each other iff the straight line connecting p and p'(i.e., line $[p, p']$) does not cut through $\forall o \in O$. A* visible region *of a point p is defined as the union of all the points $p' \in P$ such that p is visible to p'.* □

DEFINITION 2. **Visible Nearest Neighbor (VNN)**. *For $p' \in P$, p' is the visible nearest neighbor of p iff: (i) p' is visible to p; and (ii) $\forall p'' \in P - \{p'\}$, if p'' is visible to p, $dist(p'', p) \geq dist(p', p)$.* □

DEFINITION 3. **Continuous Visible Nearest Neighbor (CVNN)**. *Given a data set P, an obstacle set O, and a query line segment q, a CVNN search finds a result set RL that contains a set of $\langle p_i, R_i \rangle$ ($i \in [1, t]$) tuples such that (i) $\forall i, j \in [1, t](i \neq j)$, $R_i \cap R_j = \emptyset$; (ii) $\cup_{i=1}^{t} R_i = q$; and (iii) $\forall \langle p_i, R_i \rangle \in RL$, if $p_i \neq \emptyset$, p_i is the VNN of any point along R_i.* □

Based on the definition of CVNN search, we understand that CVNN search considers both the *proximity* and *visibility* of the data points to the query line segment. Consequently, we develop Lemma 1 and Lemma 2 to facilitate the proximity checking and visibility checking, respectively, with Lemma 1 based on the dominance relationship defined in Definition 4. Then, by considering both the proximity and the visibility, Lemma 3 presents the condition that a VNN answer point must satisfy.

DEFINITION 4. **Dominance**. *Given a point p, a data set P and an interval R ($= [R.l, R.r]$), the dominance of p over R is defined over two conditions: $\forall p' \in P - \{p\}$, (i) $dist(p', R.l) > dist(p, R.l)$ and (ii) $dist(p', R.r) > dist(p, R.r)$.*

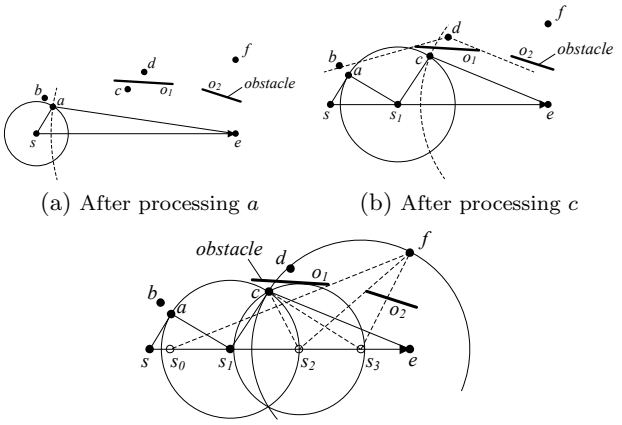
Point p dominates *R iff both conditions are satisfied. A* dominative region *of a point p is defined as the set of all the intervals R that are dominated by p.* □

To illustrate the concept of dominance, Figure 3(a) depicts an example where $P = \{a, b\}$ and $R = [s, e]$. As $dist(b, s) > dist(a, s)$ and $dist(b, e) > dist(a, e)$, point a dominates R. The circle centered at s (e) with $dist(a, s)$ ($dist(a, e)$) as radius is defined as the *vicinity circle* of s (e), denoted by $VC(s)$ ($VC(e)$). Any other point p that can violate a's dominance on R must be within either $VC(s)$ or $VC(e)$, as stated in Lemma 1. For example, point c shown in Figure 3(b) has a shorter distance to e as it is within $VC(e)$ and its appearance partitions the interval R into two intervals R_1 ($[s, s_1]$) and R_2 ($[s_1, e]$), with a dominating R_1 and c dominating R_2, respectively. Point s_1 is defined as the *split point*, i.e., the point along the interval where the VNN object changes.

LEMMA 1. *Assume point p dominates an interval $R = [s, e]$. A new point p' violates p's dominance over R iff p' is within $VC(s)$ or/and $VC(e)$.* □

LEMMA 2. *Given an interval $R = [s, e]$ and a new data point p, p will not be a VNN of any point along R if p is* invisible *to every point in R.* □

LEMMA 3. *Let VR_p be the* visible region *of a data point p and DR_p be the* dominance region *of p. Point p must be the VNN for the interval $R = VR_p \cap DR_p$.* □



(a) After processing a (b) After processing c

(c) After processing f

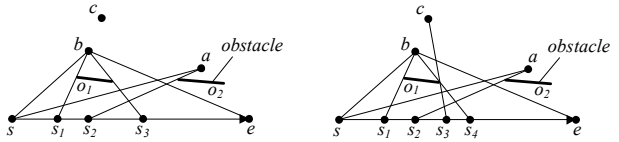
Figure 3: Example of result list updating

Lemma 1, Lemma 2, and Lemma 3 suggest an incremental query processing algorithm that aims at reporting the result set of CVNN with a single dataset traversal. Initially, RL is set to $\langle \emptyset, [s, e] \rangle$, meaning that currently the VNNs of all the points in $[s, e]$ are unknown. Thereafter, we incrementally update RL during query processing. At each step, RL contains the current result with respect to all the data points processed so far. The final result contains each answer object p_i and its dominance region R_i.

A running example is shown in Figure 3 where the set of data points $P = \{a, b, c, d, f\}$ is processed in alphabetic order, with obstacle set $O = \{o_1, o_2\}$ and query line segment $q = [s, e]$. At the beginning, RL is set to $\{\langle \emptyset, [s, e] \rangle\}$. Since a is the first point encountered and its view is not blocked by any obstacle in O, it becomes the current VNN of each point in q, and RL is updated to $\{\langle a, [s, e] \rangle\}$. When the second point b is evaluated, we only need to check whether b falls in the vicinity circles of s or e (i.e., whether b is closer to s or e than its current VNN). The fact that b is outside both circles guarantees that b does not dominate any point along $[s, e]$ and hence b is ignored.

Next, point c is evaluated. As c is inside $VC(e)$ and it is visible to every point in $[s, e]$, a split point s_1 is created. It is the intersection between the query line segment (i.e., $[s, e]$) and the perpendicular bisector of the line segment $[a, c]$, denoted by $\perp_{a,c}$, meaning that points to the left of s_1 are closer to a while points to the right of s_1 are closer to c. Consequently, RL is updated to $\{\langle a, [s, s_1] \rangle, \langle c, [s_1, e] \rangle\}$. Figure 3(b) shows the case after the processing of point c. Then, point d is pruned because it is not visible to any point along q, even though d violates c's dominance on $[s_1, e]$ (see Figure 3(b)). Then, point f is evaluated. It does not contribute to the answer set of CVNN issued at $[s, e]$ as its visible region (i.e., $[s_0, s_2]$) and dominance region (i.e., $[s_3, e]$) do not overlap. After the processing of f (as shown in Figure 3(c)), the algorithm outputs the final result (i.e., $RL = \{\langle a, [s, s_1 \rangle, \langle c, [s_1, e] \rangle\}$) to complete the search.

In addition, we observe two important properties, namely, *VNN discontinuity* and *invisible interval*, that are unique to the CVNN search.



(a) Dominative discontinuity (b) Invisible interval

Figure 4: Illustration of problem properties

PROPERTY 1. **VNN Discontinuity**. *A result point p may be VNN to multiple intervals that are not adjacent.* □

For instance, Figure 4(a) illustrates a situation where data points a and b have been processed, and the corresponding RL=$\{\langle b, [s, s_1] \rangle, \langle a, [s_1, s_2] \rangle, \langle \emptyset, [s_2, s_3] \rangle, \langle b, [s_3, e] \rangle\}$. Point b is the VNN for all the points along intervals $[s, s_1]$ and $[s_3, e]$ that are not adjacent. This property implies that a binary search heuristic that is used in conventional CNN search to retrieve the dominance region for a given point cannot be applied to CVNN search.

PROPERTY 2. **Invisible Interval**. *RL of CVNN query may have k (≥ 1) invisible intervals $\langle \emptyset, R \rangle$, where no point in a given data set P is visible to any point in R.* □

Continue the running example. Suppose a new point c is processed and it updates RL to $\{\langle b, [s, s_1] \rangle, \langle a, [s_1, s_2] \rangle, \langle \emptyset, [s_2, s_3] \rangle, \langle c, [s_3, s_4] \rangle, \langle b, [s_4, e] \rangle\}$, as illustrated in Figure 4(b). In this case, $[s_2, s_3]$ is an invisible interval.

4. CVNN QUERY PROCESSING

In this section, we present efficient algorithms for CVNN queries, assuming that the data set P and the obstacle set O are indexed by two separate R-trees. We first define several pruning heuristics in Section 4.1, then elaborate the proposed CVNN query processing algorithm in Section 4.2, and finally analyze the correctness of our solution in Section 4.3. We further extend our techniques to handle the case where P and O are indexed by a unified R-tree, as discussed in Section 4.4.

4.1 Pruning Heuristics

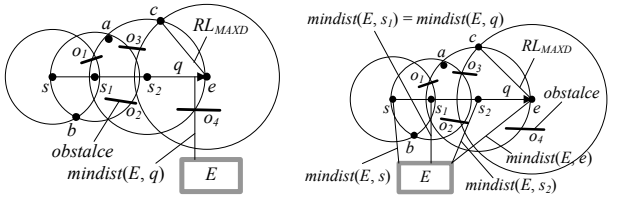
Like the conventional NN search methods discussed in Section 2, CVNN search algorithms employ *branch-and-bound*

techniques to prune the search space. A series of heuristics are developed for effective space pruning. According to the type of objects they prune, those heuristics are divided into two categories, with one for data set P and the other for obstacle set O.

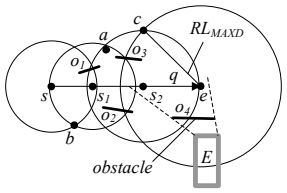
4.1.1 Pruning on Data Set

HEURISTIC 1. *Suppose the current result list* $RL = \cup_{1\leq i\leq t} \langle p_i, R_i\rangle$, *with* $R_i = [R_i.l, R_i.r]$. *Given an intermediate node entry* E *and a query line segment* q, *the subtree of* E *may contain some answer points only if* $\mathsf{mindist}(E,q) < \mathsf{RL_{MAXD}}$, *where* $\mathsf{mindist}(E,q)$ *denotes the minimum distance from the MBR of* E *to* q, *and* $\mathsf{RL_{MAXD}} = MAX_{1\leq i\leq t}\ (dist(p_i, R_i.l), dist(p_i, R_i.r))$. □

Figure 5(a) shows a data set $P = \{a, b, c\}$, an obstacle set $O = \{o_1, o_2, o_3, o_4\}$, a query line segment $q = [s, e]$, and current $RL = \{\langle b, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle c, [s_2, e]\rangle\}$. Rectangle E represents the MBR of an intermediate node (i.e., a non-leaf node). Since $\mathsf{mindist}(E,q) > \mathsf{RL_{MAXD}} = dist(c, e)$, E does not contain any point that dominates any interval of q and thus the search space covered by E can be safely pruned.



(a) With $\mathsf{RL_{MAXD}}$ (b) With dominance



(c) With visibility

Figure 5: Pruning skills

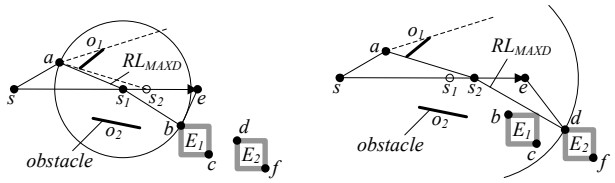
Heuristic 1 can serve as the initial pruning criteria as its computational overhead is very small. However, an entry E with $\mathsf{mindist}(E,q) < \mathsf{RL_{MAXD}}$ does not necessarily contain any answer object, which means that the pruning condition can be improved further. To verify this, consider Figure 5(b), which is similar to Figure 5(a) except that the $\mathsf{RL_{MAXD}}$ is larger. Notice that although E satisfies Heuristic 1 as $\mathsf{mindist}(E,q)$ $(=\mathsf{mindist}(E,s_1)) < \mathsf{RL_{MAXD}}$, E does not contain any qualified data point that dominates an interval of q. Consequently, Heuristic 2 is devised to prune away such entries.

HEURISTIC 2. *Given an intermediate node entry* E *and a query line segment* q, *the subtree of* E *may contain answer points only if there exists at least one interval* R *in* RL *such that* R *is dominated by* E (*i.e., at least one point in* E *dominates* R, *which can be decided by Lemma 1*). □

Heuristic 2 gives a stronger pruning criterion, but it incurs higher CPU cost compared with Heuristic 1, as it requires the calculation of the minimal distance from E to each interval in the current RL. Therefore, it is applied only for the entries that meet Heuristic 1. However, the access of some entries satisfying both Heuristic 1 and Heuristic 2 is not always necessary. Consider Figure 5(c) where the MBR of entry E is invisible to $[s_2, e]$ due to the obstruction of obstacle o_4. Hence, E can be discarded even though it satisfies Heuristic 1 and Heuristic 2. Heuristic 3 enables this pruning.

HEURISTIC 3. *Given an intermediate node entry* E *and a query line segment* q, *the subtree of* E *must be accessed iff there exists an interval* R *in* RL *that is dominated by* E *as well as* E *is visible to* R. □

By taking the visibility into consideration, Heuristic 3 further eliminates non-qualifying entries, but it incurs higher CPU overhead. Therefore, it is utilized only for the entries that satisfy both Heuristic 1 and Heuristic 2.



(a) Processing E_1 first (b) Processing E_2 first

Figure 6: Sequence of entry accesses

In addition, the entry access order also plays an important role. Consider Figure 6 in which point a has been processed, but not entries E_1 and E_2. Here $RL = \{\langle a, [s, s_2]\rangle, \langle \emptyset, [s_2, e]\rangle\}$, and hence $\mathsf{RL_{MAXD}} = \infty$. Both E_1 and E_2 satisfy Heuristics 1, 2, and 3 and thus the access to E_1 and E_2 is triggered. Suppose that E_1 is visited first, data points b and c are processed, and RL is updated as shown in Figure 6(a). Thereafter, E_2 can be pruned away from further exploration according to Heuristic 1. On the other hand, if E_2 is accessed first, E_1 has to be visited as well (see Figure 6(b)). To minimize the number of node accesses, we propose the following visiting order heuristic, which is based on the intuition that entries closer to the query line segment are more likely to contain qualified data points.

HEURISTIC 4. *Entries* E (*satisfying Heuristics 1, 2, and 3*) *are accessed in a* best-first *fashion according to ascending order of their* mindist *to the query line segment* q. □

4.1.2 Pruning on Obstacle Set

A line segment $q = [s, e]$ in a 2D space can partition the data space into two half-planes, with $HP_q^{\perp}$ above q and the other one $HP_q^{\top}$ below q. Observe that if a data point p lies in plane $HP_q^{\top}$ $(HP_q^{\perp})$, i.e., $p \in HP_q^{\top}$ $(p \in HP_q^{\perp})$, the obstacles that can affect p's visibility w.r.t. q must intersect the half-plane $HP_q^{\top}$ $(HP_q^{\perp})$. For instance, in Figure 7 the obstacles affecting the visibility of point a include o_1 and o_3; and the obstacles affecting c's visibility contain o_2 and o_3. Based on this observation, we propose the obstacle distribution heuristic below. Notice that Heuristics 6 and Heuristics 7 provide a much stronger pruning than Heuristics 5, but Heuristics 5 incurs a lower computation cost and can be implemented as first-level pruning.

HEURISTIC 5. *Given a data point* p *and a query line segment* q, *an obstacle* o *that may affect the visibility of* p *w.r.t.* q *must have at least one point* $p' \in o$ *that locates in the same half-plane partitioned by* q *as* p *does.* □

HEURISTIC 6. *Given a data point* p *and a query line segment* q, *the obstacles for sure affect the visibility of* p *w.r.t.* q *if and only if they intersect or fall completely into the triangle formed by* p *and* q. □

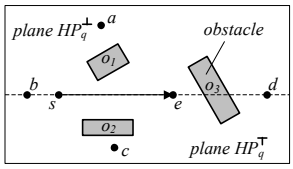
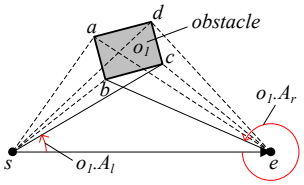
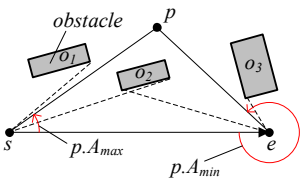


Figure 7: Pruning with obstacle distribution



(a) The $o_1.A_l$ and $o_1.A_r$



(b) The $p.A_{max}$ and $p.A_{min}$

Figure 8: Pruning with angular domain

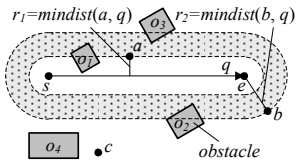


Figure 9: Incremental visit of obstacles

HEURISTIC 7. *Given a data point p and a query line segment q, any obstacle o that may affect the visibility of p w.r.t. q only if* $\mathsf{mindist}(o, q) < \mathsf{mindist}(p, q)$. □

As Heuristic 5 and Heuristic 7 are straightforward, we only explain Heuristic 6 in the following. Given a point p and a query line segment $q = [s, e]$, Heuristic 6 indicates that any obstacle o with $o \cap \triangle pse = \emptyset$ can be pruned away because it has zero impact on p's visibility w.r.t. q. Consequently, we can reduce the number of obstacles that need evaluations significantly by applying Heuristic 6. Now, we explain how to evaluate whether an obstacle shares some common area with $\triangle pse$. Our approach is as follows. For a new obstacle o, we compute in counter-clockwise direction its minimum (maximum) angle, denoted by $o.A_l$ ($o.A_r$), between a specified query line segment q and the line segments connecting the starting (ending) point s (e) of q and the vertexes of o. For example, $o_1.A_l = \angle cse$ and $o_1.A_r = \angle seb$, as shown in Figure 8(a). When processing a candidate data point p, we first calculate in counter-clockwise direction its maximal (minimal) angle, denoted by $p.A_{max}$ ($p.A_{min}$), between the query line segment q and the line segment connecting p and the starting (ending) point s (e) of q. Thereafter, any obstacle o that satisfies $o.A_l > p.A_{max}$ or $o.A_r < p.A_{min}$ does not need to be processed since it cannot intersect or locate inside $\triangle pse$. Consider, for instance, Figure 8(b) where $p.A_{max} = \angle pse$ and $p.A_{min} = \angle sep$; and hence, obstacle o_2 can affect p's visibility w.r.t. q, but not obstacles o_1, o_3.

4.2 CVNN Search Algorithm

As pointed out in Heuristic 4, we can traverse the R-tree built on a data set P (denoted by T_p) following a best-first paradigm to answer CVNN query. For each new data point p accessed, we need check whether p will update the current result list RL which involves two main issues: (i) how to get the visible region of p (denoted by VR_p) on a given query line segment $q = [s, e]$ in the presence of obstacles; and (ii) how to evaluate p's impact on a result list and how to do the update. In what follows, we explain the detailed solutions to above two issues and then present the details of CVNN search algorithm.

4.2.1 Compute Visible Region

In order to derive the visible region of p, we need find out all the obstacles that may affect the visibility of p on the query segment q. The algorithm GetObs, as presented in Algorithm 1, provides a solution. It accesses the obstacles according to ascending order of their distances to q, and stops the traversal once the accessed obstacle has its distance to q larger than $\mathsf{mindist}(p, q)$. The result obstacles are stored in a linked list L_o. We would like to highlight that as objects in P are accessed according to the ascending order of their distances to q, GetObs, for an object $p \in P$, does not need to start from scratch. Suppose p_2 is examined right after p_1. As $\mathsf{mindist}(p_1, q) \leq \mathsf{mindist}(p_2, q)$, all the obstacles that might affect p_1's visibility, denoted as GetObs(p_1), also have the possibility to affect p_2's visibility. As GetObs(p_1) is locally available, GetObs corresponding to p_2 only need to retrieve obstacles with distances to q bounded by $\mathsf{mindist}(p_1, q)$ and $\mathsf{mindist}(p_2, q)$. In other words, GetObs corresponding to a data point p_{i+1} that is examined right after data point p_i only needs to find out all the obstacles with their distances to q falling inside the range $[\mathsf{mindist}(p_i, q), \mathsf{mindist}(p_{i+1}, q)]$. Consequently, GetObs is an incremental process while the GetObs, for all the data points in P, can be finished via *one* traversal of O. For example, Figure 9 illustrates the incremental process of GetObs algorithm. It is first employed to obtain the obstacle o_1 that may influence the visibility of point a, maintained in L_o (i.e., $L_o = \{o_1\}$); and then for data point b, all the obstacles in current L_o for sure might affect its visibility. GetObs is called again to find all the obstacles other than those in L_o (i.e., o_2 and o_3), after which L_o is updated to $\{o_1, o_2, o_3\}$. If there is a new data point accessed after b, all the obstacles in L_o can be reused and the search on O can be continued to find the rest.

Algorithm 1 Get Obstacle (GetObs)

Input: an obstacle R-tree T_o, a priority queue maintaining entries H_o, a query line segment q, search distance r, a linked list storing obstacles L_o;
Procedure:
1: **while** $e :=$ deheap(H_o) $\neq \emptyset$ **do**
2: **if** $\mathsf{mindist}(e, q) > r$ **then**
3: return L_o;
4: **else if** e is an obstacle **then**
5: insert e into L_o;
6: **else**
7: $\forall e_i \in e$, insert e_i into H_o;

Once all the obstacles that might affect the visibility of p are retrieved via GetObs and maintained by L_o, we identify the invisible region IR_o along q that is blocked by scanning each obstacle $o \in L_o$ and the visible region of p, denoted as VR_p, can be derived based on $VR_p = q - \cap_{o \in O} IR_o$. Algorithm 2 depicts the pseudo-code of the Visible Region Computation Algorithm (VRC), which takes as input p, q, and L_o, and outputs VR_p over q. Here, the function Obstruction(q, p, o) is to return the regions inside q that are invisible to p due to the block of o.

EXAMPLE 1. *We illustrate Algorithm 2 with the example shown in Figure 10, where the obstacles affecting the visibility of p have been obtained and stored in* $L_o = \{o_1, o_2, o_3\}$. *First, VRC examines* o_1 *and gets its invisible region* $IR_{o_1} = [s_1, s_3]$. *Consequently,* VR_p *is updated to* $q - IR_{o_1} = \{[s, s_1], [s_3, e]\}$. *Next, VRC examines* o_2 *and gets its invisible region* $IR_{o_1} = [s_2, s_4]$ *which updates* VR_p *to* $\{[s, s_1], [s_4, e]\}$. *Finally,* o_3 *is evaluated. As its invisible region* IR_{o_3} *is* $[s_5, s_6]$,

VR_p is updated to $\{[s, s_1], [s_4, s_5], [s_6, e]\}$. □

Algorithm 2 Visible Region Computation (VRC)

Input: a data point p, a query line segment $q = [s, e]$, and a linked list maintaining obstacles L_o;
Output: p's visible region VR_p;
Procedure:
1: **for** each obstacle $o \in L_o$ **do**
2: **if** $o \cap HP_q(p) \neq \emptyset$ and $o \cap \triangle pse \neq \emptyset$ **then**
3: $IR :=$ Obstruction(q, p, o)
4: **for** each region $[l, r] \in IR$ **do**
5: $VR_p := q - [l, r]$
6: return VR_p

4.2.2 Update Result List

A Result List Update Algorithm (RLU) is proposed to incrementally update the result list for a CVNN search upon the evaluation of a new data point. It takes as input the current result list RL, a new point p, and p's visible region VR_p to evaluate the impact of p on RL. Specifically, for every region R in RL, RLU distinguishes two cases: (i) if $R \cap VR_p \neq \emptyset$ (i.e., p is visible to R), the algorithm first computes the intersection R_{int} $(= R \cap VR_p)$ and difference R_{dif} $(= R - VR_p)$ between R and VR_p; and then, it performs the following operations. If VNN of R (denoted by R.VNN) is empty, $\langle p, R_{int}\rangle$ and $\langle \emptyset, R_{dif}\rangle$ (if $R_{dif} \neq \emptyset$) are inserted into a temporary result list TRL. Otherwise (i.e., R.VNN is not empty), RLU inserts $\langle R.\text{VNN}, R_{dif}\rangle$ into TRL if $R_{dif} \neq \emptyset$; and then, the algorithm invokes a RS-CVNN algorithm to determine whether R.VNN can be fully/partially replaced by p over region R_{int}. We defer the discussion of RS-CVNN later. (ii) $R \cap VR_p = \emptyset$ (i.e., p is invisible to R) which means p has zero impact on region R and thus RLU inserts $\langle R.\text{VNN}, R\rangle$ into TRL. At the end of the algorithm, TRL keeps the new result list. The pseudo-code of RLU is depicted in Algorithm 3. It is important to note that every time when a new tuple $\langle p', R'\rangle$ is inserted into TRL, it might be merged with the existing region R'' in TRL if R' and R'' are continuous and they share the same VNN.

Algorithm 3 Result List Update (RLU)

Input: a result list RL, a data point p, p's visible region VR_p, and the query segment $q = [s, e]$;
Output: the updated result list;
Procedure:
1: TRL := $\{\langle \emptyset, [s, e]\rangle\}$;
2: **for** each region $R \in RL$ **do**
3: **if** $R \cap VR_P \neq \emptyset$ **then**
4: $R_{int} := R \cap VR_p$; $R_{dif} := R - VR_p$;
5: **if** R.VNN $= \emptyset$ **then**
6: insert $\langle p, R_{int}\rangle$ into TRL, insert $\langle \emptyset, R_{dif}\rangle$ into TRL if $R_{def} \neq \emptyset$, and merge them with the existing regions in TRL if necessary;
7: **else**
8: insert $\langle R.\text{VNN}, R_{dif}\rangle$ into TRL if $R_{dif} \neq \emptyset$, and merge it with the existing regions in TRL if necessary;
9: RS-CVNN(TRL, R_{int}, p); // see Algorithm 4
10: **else**
11: insert $\langle R.\text{VNN}, R\rangle$ into TRL and merge it with the existing regions in TRL if necessary;
12: return TRL;

The RS-CVNN algorithm is used to check the validity of R.VNN upon p, and fully/partially replace R.VNN with p if necessary. The pseudo-code is described in Algorithm 4. Notice that the region R $(\subseteq VR_p)$ is for sure visible to p and hence we only need check the dominance relationship, according to Lemma 1. RS-CVNN distinguishes four cases: (i) If p does not dominate R, the original tuple $\langle R.\text{VNN}, R\rangle$ remains valid, and is added to TRL (lines 1-2). (ii) If p dominates entire R, the algorithm replaces R.VNN with p and inserts $\langle p, R\rangle$ into TRL (lines 3-4). (iii) If p is only within the vicinity circle of $R.l$, the algorithm calculates the intersection s_1 between the query line segment q and the perpendicular bisector of segment $[v, p]$ (i.e., $\perp_{v,p}$) with $v = R.\text{VNN}$, and adds $\langle p, [R.l, s_1]\rangle$ and $\langle v, [s_1, R.r]\rangle$ to TRL (lines 5-7). (iv) Similar to case (iii), if p is only within the vicinity circle of $R.r$, the algorithm derives the intersection s_2 between the query line segment q and $\perp_{v,p}$, and inserts $\langle v, [R.l, s_2]\rangle$ and $\langle p, [s_2, R.r]\rangle$ into TRL (lines 8-10).

Algorithm 4 Region Split for CVNN (RS-CVNN)

Input: TRL, a region R, a data point p;
Procedure:
1: **if** $p \notin VC(R.l)$ and $p \notin VC(R.r)$ **then**
2: insert $\langle v, R\rangle$ into TRL; // R.VNN does not change
3: **else if** $p \in VC(R.l)$ and $p \in VC(R.r)$ **then**
4: insert $\langle p, R\rangle$ into TRL; // replace R.VNN with p
5: **else if** $p \in VC(R.l)$ **then**
6: $s_1 := R \cap \perp_{v,p}$;
7: insert both $\langle p, [R.l, s_1]\rangle$ and $\langle v, [s_1, R.r]\rangle$ into TRL;
8: **else**
9: $s_2 := R \cap \perp_{v,p}$;
10: insert both $\langle v, [R.l, s_2]\rangle$ and $\langle p, [s_2, R.r]\rangle$ into TRL;

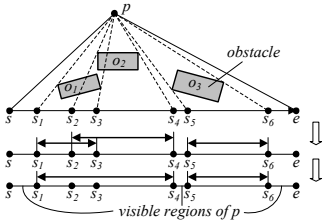


Figure 10: Example of VRC algorithm

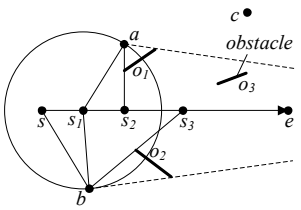


Figure 11: Example of RLU algorithm

EXAMPLE 2. *We illustrate the RLU algorithm using the data in Figure 11, where* $P = \{a, b, c\}$ *and* $O = \{o_1, o_2, o_3\}$. *As seen from Figure 11,* $VR_a = \{[s, s_2]\}$ *and* $VR_b = \{[s, s_3]\}$. *Suppose that point a has been processed and current* $RL = \{\langle a, [s, s_2]\rangle, \langle \emptyset, [s_2, e]\rangle\}$. *Now we call RLU to evaluate the impact of a new point b on the RL. RLU checks each region in RL. First,* $[s, s_2]$ *is evaluated. As it overlaps with* VR_b, *we derive* R_{int} $(= [s, s_2] \cap [s, s_3])$ *and* R_{dif} $(= [s, s_2] - [s, s_2] = \emptyset)$, *and calls RS-CVNN to examine whether a, the current VNN of* R_{ins}, *can be fully/partially replaced by b. As b is within the vicinity circle of s, RS-CVNN computes the intersection* s_1 *between* $[s, s_2]$ *and* $\perp_{a,b}$, *i.e., the perpendicular bisector of segment* $[a, b]$, *and adds* $\langle b, [s, s_1]\rangle$ *and* $\langle a, [s_1, s_2]\rangle$ *to TRL. Next, RLU checks the second region in RL (i.e.,* $[s_2, e]$*) and finds out it also overlaps with* VR_b. *Consequently, both* R_{int} $(= [s_2, e] \cap [s, s_3])$ *and* R_{dif} $(= [s_2, e] - [s_2, s_3])$ *are calculated. As the current VNN of* $[s_2, e]$ *is* $\emptyset$, *RLU adds directly* $\langle b, [s_2, s_3]\rangle$ *and* $\langle \emptyset, [s_3, e]\rangle$ *to TRL. Finally, RLU returns* $TRL = \{\langle b, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle b, [s_2, s_3]\rangle, \langle \emptyset, [s_3, e]\rangle\}$. □

4.2.3 The Complete CVNN Query Algorithm

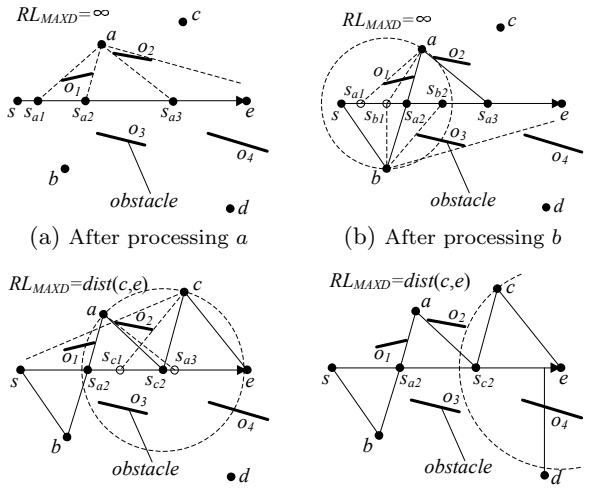
Having explained the *visible region computation* algorithm and the *result list update* algorithm, we are ready to present the complete CVNN query processing algorithm, namely CVNN, which is described in Algorithm 5. It takes as input the R-trees T_p and T_o, on dataset P and obstacle set O, respectively, and a query line segment q. Although the proposed CVNN algorithm can be applied with both the *depth-first* and *best-first* paradigms as discussed in Section 2, for simplicity, we only elaborate the complete CVNN algorithm following the best-first fashion.

In order to enable the best-first traversal, the algorithm maintains two heaps H_p and H_o storing the data and obstacle entries visited so far respectively, sorted by ascending order of their minimal distance (i.e., mindist) to q. CVNN starts from the root of T_p, and inserts all its child entries into H_p (line 2). Then, at each step, CVNN visits the head entry e in H_p that has the smallest mindist to q and performs the following tasks (lines 3-14). (i) CVNN checks whether $\mathsf{mindist}(e,q) \geq \mathsf{RL_{MAXD}}$. If yes, the algorithm terminates because the remaining entries in H_p can not contain any answer point according to Heuristic 1 (described in Section 4.1). (ii) If e is a data point, CVNN calls the GetObs algorithm to obtain all the obstacles that may influence the visibility of e, calls the VRC algorithm to derive the visible region of e w.r.t. q, and updates the current result list RL based on the RLU algorithm. (iii) If e is an intermediate node (i.e., a non-leaf entry), CVNN visits its subtree only if it may contain any qualifying data point. Note that here CVNN applies Heuristic 2 and Heuristic 3 (discussed in Section 4.1). The advantage of the algorithm over exhaustive scan is that the access to some unnecessary nodes, i.e., those for sure not containing any qualified objects, is eliminated. Finally, we illustrate the CVNN algorithm using a running example.

Algorithm 5 CVNN Search (CVNN)

Input: a data R-tree T_p, an obstacle R-tree T_o, and a query line segment q;
Procedure:
1: $RL := \{\langle \emptyset, [s,e]\rangle\}$; $\mathsf{RL_{MAXD}} = \infty$; $L_o := \emptyset$;
2: $H_p :=$ root(T_p); $H_o :=$ root(T_o);
3: **while** $H_p \neq \emptyset$ **do**
4: $e :=$ deheap(H_p);
5: **if** $\mathsf{mindist}(e,q) \geq \mathsf{RL_{MAXD}}$ **then**
6: break;
7: **else if** e is a data point **then**
8: GetObs(T_o, H_o, q, $\mathsf{mindist}(e,q)$, L_o);
9: $VR_e :=$ VRC(e,q,L_o);
10: $RL :=$RLU(RL, e, VR_e);
11: **else**
12: **for** each child entry $e_i \in e$ **do**
13: **if** e_i dominates some region in RL and it is visible to q **then**
14: insert e_i into H_p;
15: return RL;

Example 3. *Consider the example depicted in Figure 12 with $P = \{a,b,c,d\}$, $O = \{o_1,o_2,o_3,o_4\}$, and $q = [s,e]$. Initially, the result list RL is set to $\{\langle\emptyset,[s,e]\rangle\}$. When the first data point a (that is the closest to q without considering obstacles) is visited, CVNN invokes GetObs to obtain all the obstacles that may affect the visibility of a (i.e., o_1 and o_2). Then, it uses VRC to get $VR_a = \{[s,s_{a_1}], [s_{a_2},s_{a_3}]\}$, i.e., the a's visible regions w.r.t. q. Next, RLU is called to update the current RL, after which $RL = \{\langle a,[s,s_{a_1}]\rangle, \langle\emptyset,[s_{a_1},s_{a_2}]\rangle, \langle a,[s_{a_2},s_{a_3}]\rangle, \langle\emptyset,[s_{a_3},e]\rangle\}$ as shown in Figure 12(a). The second point examined is b. As b dominates $[s,s_{a_1}]$ and $[s_{a_1},s_{a_2}]$, the corresponding VNNs are replaced by b, as shown in Figure 12(b) with $RL = \{\langle b,[s,s_{a_2}]\rangle, \langle a,[s_{a_2},s_{a_3}]\rangle, \langle\emptyset,[s_{a_3},e]\rangle\}$. Subsequently, it evaluates the third point c and updates RL to $\{\langle b,[s,s_{a_2}]\rangle, \langle a,[s_{a_2},s_{c_2}]\rangle, \langle c,[s_{c_2},e]\rangle\}$, which is illustrated in Figure 12(c). Finally, when the last point d is encountered, d is pruned directly because $\mathsf{mindist}(d,q) > \mathsf{RL_{MAXD}}$ $(= dist(c,e))$. The algorithm terminates with the final result $RL = \{\langle b,[s,s_{a_2}]\rangle, \langle a,[s_{a_2},s_{c_2}]\rangle, \langle c,[s_{c_2},e]\rangle\}$, as shown in Figure 12(d).* □



(a) After processing a (b) After processing b
(c) After processing c (d) After processing d

Figure 12: Illustration of CVNN search processing

4.3 Analysis

In what follows, we reveal some characteristics about our proposed CVNN query processing algorithm, analyse the cost of the CVNN algorithm and prove its correctness.

Lemma 4. *Every data point in a data set P will be examined during the CVNN search, unless one of its ancestor nodes has been pruned.* □

Lemma 5. *It is guaranteed that any obstacle in an obstacle set O added to L_o during the CVNN search may impact the visibility of the current data point processed.* □

Lemma 6. *The CVNN algorithm traverses the data R-tree T_p and the obstacle R-tree T_o* at most once. □

Theorem 1. *The time complexity of the CVNN algorithm is $O(N \log |T_p| \times (\log |T_o| + |L_o| + |RL|))$.* □

Proof. Let $|L_o|$ be the maximal number of obstacles in a linked list L_o, $|RL|$ be the maximal number of entries in a result list RL, $|T_p|$ and $|T_o|$ be the tree size of T_p and T_o respectively, and N be the number of data points accessed. A CVNN algorithm invokes GetObs, VRC, and RLU algorithms with complexities being $O(\log |T_o|)$, $O(|L_o|)$, and $O(|RL|)$, respectively. Thus, the time complexity of the CVNN algorithm is $O(N \log |T_p| \times (\log |T_o| + |L_o| + |RL|))$. ■

Theorem 2. *The CVNN algorithm retrieves* exactly *the VNN of every point on a given query line segment, i.e., the algorithm has no* false misses *and no* false hits. □

Proof. Theorem 2 holds, as Lemma 4 makes sure that the CVNN algorithm processes only those qualifying data points in P that may be included in the final result list RL, and all unqualified data points in P are pruned away safely according to the Heuristics 1 through 4. ■

4.4 Discussion

Our previously presented CVNN search algorithm assumes dataset O and obstacle set P are indexed by two different R-trees. In the following, we explain how to extend it to conduct the search based on a *single* R-tree that indexes both data points and obstacles. The detailed extensions are listed as follows: (i) It requires only one heap H to store the candidate entries (containing data points, obstacles, and nodes), ordered based on ascending order of their minimum distances to the query line segment q. (ii) When processing the top entry E removed from H, it distinguishes three cases. (1) E is an obstacle. It adds E to a linked list L_o, which keeps all the obstacles that may affect the visibility of the data points processed so far w.r.t. q. (2) E is a data point p. It computes the visible region of E w.r.t. q, and updates the current result list RL if necessary. It is guaranteed that all the obstacles that might affect the dominance of p must have been visited before p. According to the Heuristic 7 (proposed in Section 4.1), any obstacle o that may impact the visibility of E w.r.t. q must satisfy the condition: $\mathsf{mindist}(o, q) < \mathsf{mindist}(E, q)$. Consequently, when E is visited, the algorithm must have obtained all the obstacles with smaller $\mathsf{mindist}$ to q. Note that here we do not have to call the GetObs algorithm to get all the obstacles that may affect E's visibility, because both data points and obstacles are indexed by one unified R-tree. (3) E is an intermediate node, meaning that it may contain data points and/or obstacles. It expands the subtree of E only if E may contain any qualifying data point. Note that all the proposed heuristics (described in Section 4.1) can still be applied for pruning unnecessary node accesses significantly.

5. CVNN QUERY VARIANTS

The CVNN search has several interesting variations. Due to the space limitation, we only discuss two variants in this section, namely CVkNN and δ-CVNN queries.

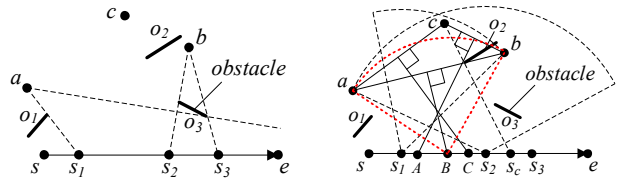
5.1 The CVkNN Search

Given a data set P, an obstacle set O, and a query line segment $q = [s, e]$, CVkNN query is to retrieve k VNNs for each point on q. A tuple $\langle S, R\rangle$ in the result list RL represents that all the points along interval R share the same k VNNs, denoted by S. Different from conventional kNN search, the answer set S might not exist (i.e., $S = \emptyset$) or it might not hold k objects (i.e., $|S| < k$) because of the presence of obstacles. The proposed algorithms for CVNN queries can be easily extended to support CVkNN queries as well. The detailed extensions are listed as follows. The dominance conditions defined in Definition 4 needs updating. The distance between the answer point p and the interval R, i.e., $dist(p, R.l)$ and $dist(p, R.r)$, need to be replaced by $\mathsf{maximumdist}(S, R.l)$ and $\mathsf{maximumdist}(S, R.r)$ respectively, where $\mathsf{maximumdist}$ is defined as follows.

$$\mathsf{maximumdist}(S, r) = \begin{cases} MAX_{\forall s_i \in S} dist(s_i, r) & \text{if } |S| = k \\ \infty & \text{otherwise} \end{cases}$$

The Heuristics defined in Section 4 are directly applicable except that, in Heuristic 1, $\mathsf{RL_{MAXD}} = MAX_{i\in[1,|R|]}(\mathsf{maximumdist}(S_i, R_i.l), \mathsf{maximumdist}(S_i, R_i.r))$, in which $|R|$ denotes the number of regions in R, and S_i contains all the VNNs for R_i. The pruning process is the same as the CVNN retrieval. The handling of data points is also similar. Specifically, each data point p is processed in the following steps. The first step obtains all obstacles affecting the visibility of p. The second step computes the visible regions of p (i.e., VR_p). The third step is to update current result list RL if necessary, which is more complex than CVNN (i.e., $k = 1$).

We use following example to illustrate the result update process for a CV2NN ($k = 2$) search, with a data set $P = \{a, b, c\}$, an obstacle set $O = \{o_1, o_2, o_3\}$, and $q = [s, e]$ (as shown in Figure 13). Suppose points a and b have been processed and currently $RL = \{\langle\{b\}, [s, s_1]\rangle, \langle\{a, b\}, [s_1, s_2]\rangle, \langle\{a\}, [s_2, s_3]\rangle, \langle\{a, b\}, [s_3, e]\rangle\}$, as shown in Figure 13(a). Notice that the number of the current VNN(s) for intervals $[s, s_1]$ and $[s_2, s_3]$ is only one due to the obstruction of obstacles. Now the evaluation of a new data point c starts, assuming that we have got $VR_c = \{[s, s_c]\}$ w.r.t. q.



(a) Before updating RL (b) After updating RL

Figure 13: Illustration of updating RL ($k = 2$)

In order to simplify the discussion, we just focus on the evaluation of c based on a given interval R, but the same process can be applied to other intervals in RL. First, according to the visibility, c partitions the interval R into two regions R_{int} and R_{dif} with $R_{int} = R \cap VR_c$ and $R_{dif} = R - VR_c$. Point c might change the result corresponding to R_{int} but definitely not R_{dif}. Consequently, the evaluation can be safely terminated if $R_{int} = \emptyset$, which means the interval R is invisible to c. Now suppose $R_{int} = [s, e]$, and its corresponding answer set is S. We need check whether (i) $\mathsf{maximumdist}(S, s) > dist(c, s)$ and/or (ii) $\mathsf{maximumdist}(S, e) > dist(c, e)$. Similar as RS-CVNN algorithm presented in Algorithm 4, the new point c partitions the interval R accordingly based on conditions (i) and (ii). If neither condition is satisfied, c is discarded as it is not an answer object to any point along R. Otherwise, c must satisfy at least one condition and the interval R needs to be split. Notice that the split point is identified by sweeping algorithm.

Take interval $[s_1, s_2]$ as an example. Point c is fully visible to $[s_1, s_2]$, and $dist(c, s_1) < \mathsf{maximumdist}(S, s_1)$ and $dist(c, s_2) < \mathsf{maximumdist}(S, s_2)$. Consequently, sweeping algorithm is called to find the split points along interval $[s_1, s_2]$. As $dist(c, s_1) < dist(b, s_1)$, c will replace b as the new 2-VNN for s_1. The intersection between q and $\perp_{b,c}$ (i.e., A in Figure 13(b)) is computed (i.e., the first split point), and $\langle\{a, c\}, [s_1, A]\rangle$ is inserted into TRL. The next split point is derived based on the intersection between q and $\perp_{a,c}$ (i.e., C in Figure 13(b)). Based on bisectors $\perp_{b,c}$ and $\perp_{a,c}$, the entire interval $[A, C]$ is closer to a and b, compared with c. Consequently, $\langle\{a, b\}, [A, C]\rangle$ is inserted into TRL. Finally, the interval $[C, s_2]$ is closer to b and c, compared with a, and hence $\langle\{b, c\}, [C, s_2]\rangle$ is inserted into TPL to finish the update of c on $\langle\{a, b\}, [s_1, s_2]\rangle$.

5.2 The δ-CVNN Search

In practice, users may also enforce some constraints (e.g., distance, region, etc.) on CVNN queries. In view of this, we introduce δ-CVNN search, a CVNN query with *maximum*

visible distance δ constraint. Given a data set P, an obstacle set O, a query line segment q, and distance threshold δ, a δ-CVNN search is to return the VNN of every point on q with its distance to q bounded by δ. The proposed algorithms for CVNN queries can directly deal with δ-CVNN search by integrating constrained condition (i.e., δ) during query processing.

In addition to all the Heuristics defined in Section 4 that are still applicable, we propose following heuristics to fully utilize the distance constraint: (i) As we evaluate the nodes based on best-first paradigm, the algorithm can be safely terminated once an entry E with mindist to q larger than δ is encountered; (ii) The searching space of δ-CVNN query is limited by δ. For example, the shadowed area in Figure 14 represents the search space of a δ-CVNN query with $q = [s, e]$ and δ being the threshold. Hence, any entry (containing data point, obstacle, and node) that is outside the search range should be discarded; (iii) For a candidate data point p, it has a influence region on q, i.e., the region R such that the distance between p and any point $r \in R$ is bounded by δ, e.g., interval $[s_1, s_2]$ is the influence region of a data point p as shown in Figure 14. Any obstacle falling out of $\triangle s_1 s_2 p$ does not affect p's visibility.

We use the example data in Figure 12 to conduct the δ-CVNN search. Figure 15 shows the final results (i.e., $\{\langle b, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle c, [s_2, s_3]\rangle, \langle \emptyset, [s_3, e]\rangle\}$), which is different from the answer of CVNN on the same data due to the influence of δ.

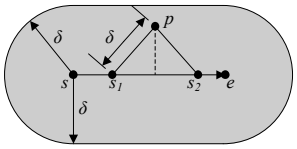


Figure 14: Search area and influence region

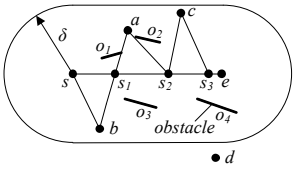


Figure 15: Example of a δ-CVNN query

6. PERFORMANCE EVALUATION

This section evaluates the performance of the proposed algorithms. The algorithms were implemented in C++ and the experiments were conducted on a Pentium IV 3.0 GHz PC with 2GB RAM. We first describe the experimental settings, and then present the experimental results and our findings.

6.1 Experimental Settings

Our experiments are based on both synthetic and real datasets, with the search space fixed at $[0, 10000] \times [0, 10000]$ square shaped range. Four real datasets are deployed, namely **CA**, **Cities**, **LA** and **Rivers** [4]. CA and Cities contain 2D points, representing $62,556$ locations in California and $5,922$ cities and villages in Greece, respectively; LA and Rivers include 2D rectangles, representing $131,461$ MBRs of streets in Los Angeles and $24,650$ MBRs of rivers in Greece, respectively. All the datasets are normalized in order to fit the search range. Synthetic datasets are generated based on uniform distribution and zipf distribution, with the cardinality varying from $0.1 \times |LA|$ to $10 \times |LA|$. The coordinate of each point in Uniform datasets is generated uniformly along each dimension, and that of each point in Zipf datasets is generated according to zipf distribution with skew coefficient $\alpha = 0.8$. We assume a point's coordinates on both dimensions are mutually independent. As CVNN search involves a data set P and an obstacle set O, we deploy four different combinations of the datasets, namely **CL**, **CR**, **UL**, and **ZL**, representing (P, O) = (CA, LA), (Cities, Rivers), (Uniform, LA), and (Zipf, LA), respectively. Note that the data points in P are allowed to lie on the boundaries of the obstacles but not in their interior.

[4]CA, Cities, LA, and Rivers datasets are available in the R-tree Portal (http://www.rtreeportal.org).

Table 1: Parameter ranges and default values

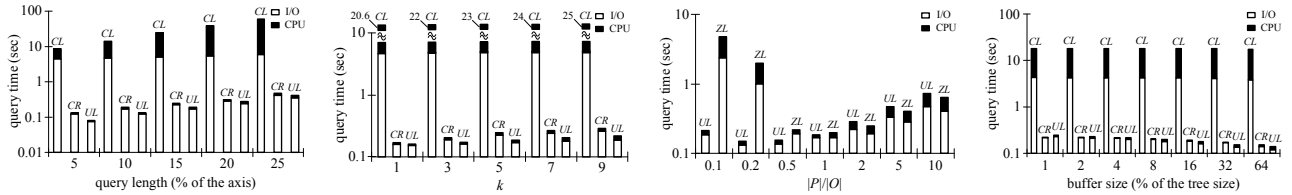
Parameter	Range
$\|P\|/\|O\|$	0.1, 0.2, 0.5, **1**, 2, 5, 10
k	1, 3, **5**, 7, 9
query length q_l (% of the axis)	5, 10, **15**, 20, 25
buffer size (% of tree size)	1, 2, 4, 8, **10** 16, 32, 64
δ (% of the axis)	5, 10, 15, 20, 25 ∞

All data and obstacle sets are indexed by R*-trees [2], with the page size fixed at 4KB. We also employ a memory cache of 10% of index size with LRU as the replacement scheme to buffer loaded pages. The query time, that is the summation of the I/O time and CPU time where the I/O time is computed by charging 10ms for each page fault as [26], is the major performance metric used in our study. We evaluate the efficiency and effectiveness of our proposed algorithms in terms of average query time under various parameters which are summarized in Table 1. The numbers in bold represent the default settings for each parameter. In each set of experiments, only one parameter is changed in order to evaluate its impact on the performance, while all the other parameters are fixed at default values. Each experiment runs 200 trials of CVNN queries and the average performance is reported. Each CVNN query is generated by (i) selecting uniformly a point in the data space as the starting point of the query line segment, and (ii) selecting an orientation (angle with the x-axis) from the range $[0, 2\pi)$ randomly, with its length controlled by the parameter q_l.

6.2 Performance Study

Effect of query length q_l. Figure 16(a) shows the efficiency of the CVkNN algorithm as a function of the query length q_l ($k = 5$). The letters on top of each column specify dataset combination name. e.g., CR denotes P = Cities and O = Rivers. Obviously, the query cost of CVkNN, especially the CPU time, increases with q_l. This is because, as the query length becomes longer, both the number of candidate data points processed and the number of the splitting regions in the specified query line segment increase, which results in more distance computation, more visibility verification, and more result list updating.

Effect of k. Figure 16(b) illustrates the performance of the CVkNN algorithm under different k values, fixing $q_l = 15\%$. The sign $\approx$ on top of some bars means the cost is too high, compared with others. Instead of depicting the bar in the Figure 16(b), we provide the real performance, shown as the numbers above some bars. As expected, both I/O cost and CPU time grow with k, because a higher value of k implies a larger search range (for data points and obstacles) and more distance computations. Moreover, as k increases, the number of objects in the final result list increases, which contributes to the more frequent update operations and hence more expensive maintenance cost of the result list.

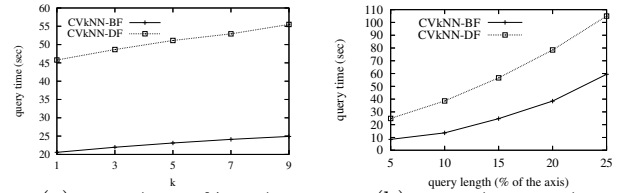


(a) vs. q_l ($|P|=|O|$ for UL) (b) vs. k ($|P|=|O|$ for UL) (c) vs. $|P|/|O|$ (d) vs. buffer ($|P|=|O|$ for UL)

Figure 16: CVkNN performance under different parameters

Effect of $|P|/|O|$. Figure 16(c) depicts the cost of the CVkNN algorithm as a function of the ratio $|P|/|O|$, fixing $k = 5$ and $q_l = 15\%$. A crucial observation is that the cost of the CVkNN algorithm first drops and then increases as $|P|/|O|$ changes its value from 0.1 to 10. In particular, the query time of CVkNN decreases when $|P|/|O|$ increases from 0.1 to 1. This is because, as the density of object set P grows, the search space of CVkNN becomes smaller and the number of obstacles that participate in the visibility verification of data points is decreased. Consequently, the performance increases. However, as $|P|/|O|$ changes from 1 to 10, the cost of CVkNN gradually increases. This is because the interval dominated by each object becomes shorter, and the result list contains more answer objects. In other words, more candidate data points need evaluation, which in turn increases the number of split points and result list update cost. Notice that when P and O share similar cardinalities (e.g., $|P|/|O| = 0.5$ or 1 in Figure 16(c)), the performance of CVkNN is the best in the whole cost varying process.

Effect of query buffer size. As mentioned in Section 6.1, all previous experiments are performed with an LRU buffer that is set to 10% of the tree size. In this set of experiments, we examine the performance of CVkNN with various LRU buffer sizes, while fixing k and q_l to their default values (i.e., 5% and 15% respectively). To obtain stable statistics, we measure the average cost of the last 100 queries, after the first 100 queries have been performed for warming up the buffer. The experimental result is depicted in Figure 16(d). It is observed that as the buffer size increases, the I/O cost drops but the CPU cost remains almost the same.

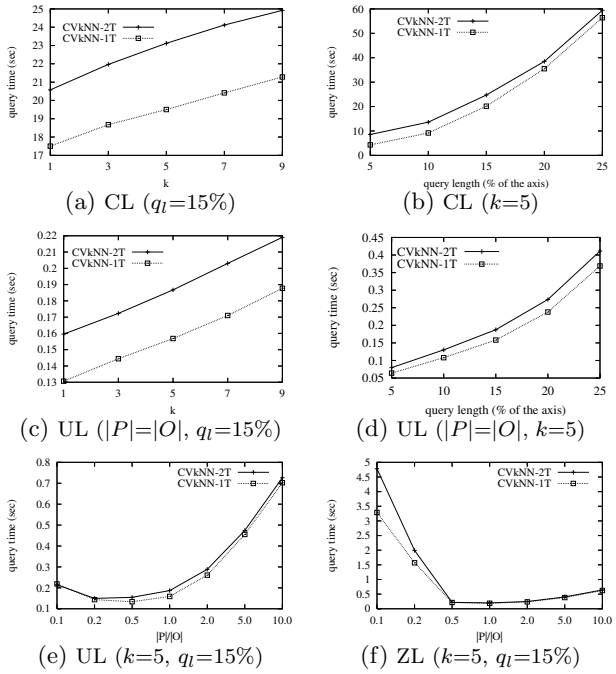


(a) vs. k (q_l=15%, CL) (b) vs. q_l (k=5, CL)

Figure 17: CVkNN under Best-First and Depth-First

Best-First based CVkNN search versus Depth-First based CVkNN search. As mentioned earlier, our proposed CVkNN query processing approaches can be applied with both the best-first (BF) and depth-first (DF) traversal paradigms. In view of this, we implement both paradigms, denoted as CVkNN-BF and CVkNN-DF. Figure 17 plots their performance as a function of k and q_l, respectively. Clearly, CVkNN-BF outperforms CVkNN-DF significantly for all cases, which is consistent with previous results on conventional NN queries [8].

CVNN search on two R-trees versus CVNN search on one R-tree. Figure 18 presents the query efficiency when the data point and obstacle sets are indexed by two separate R-trees (2T) and when they are indexed by one single R-tree (1T), under a variety of factors (containing k, q_l, and $|P|/|O|$) which affect the performance of the algorithms. It shows 1T is more efficient than 2T in all cases. This is because when both data points and obstacles are indexed by one R-tree; only one traversal of the unified R-tree is required. Data point and obstacles that are close to each other could be found in the same leaf node of the R-tree. For the 2T case, two R-tree should be traversed and at least two leaf nodes should be accessed to retrieve the data point and the obstacles nearby. Based on this result, when we perform expensive spatial data mining tasks, we recommend to build a single R-tree for both data points and obstacles to obtain better performance.



(a) CL (q_l=15%) (b) CL (k=5)
(c) UL ($|P|=|O|$, q_l=15%) (d) UL ($|P|=|O|$, k=5)
(e) UL (k=5, q_l=15%) (f) ZL (k=5, q_l=15%)

Figure 18: CVkNN on two R-trees vs. CVkNN on one R-tree

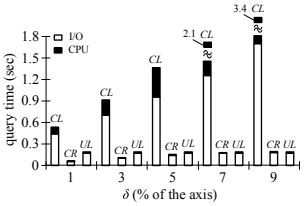


Figure 19: δ-CVNN cost vs. δ (k=5, q_l=15%, $|P|=|O|$ for UL)

Effect of δ. Finally, Figure 19 shows the performance of the algorithm for δ-CVNN queries, with different maximum visible distance δ. Clearly, δ has a direct impact on the performance, as it controls the size of the search space.

7. CONCLUSION

This paper proposes a novel type of spatial queries called *continuous visible nearest neighbor* (CVNN) search, which is not only interesting from a research point of view but also useful in many practical applications involving spatial data and obstacles such as location-based commerce, interactive online games, and decision support. We carry out a systematic study of CVNN. First, we provide a formal definition of the problem and reveal its unique characteristics. Then, we present a suite of effective pruning heuristics and develop an efficient algorithm to tackle the problem. Next, we extend our methods to handle several CVNN variants, including CVkNN and δ-CVNN queries. Finally extensive experiments are conducted to verify the performance of our proposed algorithms.

As for our future work, we plan to extend our techniques to other CVNN query variations (e.g., *trajectory* VNN, *constrained* CVNN, *ranked* CVNN, etc.). In addition, we are interested in efficient algorithms for processing complex spatial queries (e.g., reverse nearest neighbor search, etc.) in the presence of obstacles. We would like to consider other distance metrics, like shortest path or travel time in road network.

8. ACKNOWLEDGMENT

In this research, Wang-Chien Lee is supported in part by the National Science Foundation under Grant No. IIS-0328881, IIS-0534343, and CNS-0626709.

9. REFERENCES

[1] T. Asano, S. K. Ghosh, and T. C. Shermer. Visibility in the plane. In *Handbook of Computation Geometry*. Elsevier, 2000.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[3] K. L. Cheung and A. W.-C. Fu. Enhanced nearest neighbour search on the R-tree. *SIGMOD Record*, 27(3):16–21, 1998.

[4] K. Deng, X. Zhou, H. Shen, K. Xu, and X. Lin. Surface k-NN query processing. In *ICDE*, 2006.

[5] V. Estivill-Castro and I. Lee. Autoclust+: Automatic clustering of point-data sets in the presence of obstacles. In *TSDM*, 2000.

[6] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. Abbadi. Constrained nearest neighbor queries. In *SSTD*, 2001.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[8] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[9] H. Hu and D. L. Lee. Range nearest-neighbor query. *TKDE*, 18(1):78–91, 2006.

[10] M. Kofler, M. Gervautz, and M. Gruber. R-trees for organizing and visualizing 3D GIS databases. *Journal of Visualization and Computer Animation*, 11(3):129–143, 2000.

[11] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *SSTD*, 2005.

[12] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.

[13] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *TKDE*, 17(11):1451–1464, 2005.

[14] K. Mouratidis, M. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, 2006.

[15] S. Nutanong, E. Tanin, and R. Zhang. Visible nearest neighbor queries. In *DASFAA*, 2007.

[16] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *ICDE*, 2004.

[17] D. Papadias, Y. Tao, K. Mouratidis, and K. Hui. Aggregate nearest neighbor queries in spatial databases. *TODS*, 30(2):529–576, 2005.

[18] A. Papadopoulos and Y. Manolopoulos. Performance of nearest neighbor queries in R-trees. In *ICDT*, 1997.

[19] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.

[20] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, 1987.

[21] L. Shou, C. Chionh, Y. Ruan, Z. Huang, and K. L. Tan. Walking through a very large virtual environment in real-time. In *VLDB*, 2001.

[22] L. Shou, Z. Huang, and K. L. Tan. HDoV-tree: The structure, the storage, the speed. In *ICDE*, 2003.

[23] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *ICDE*, 1997.

[24] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, 2001.

[25] Y. Tao and D. Papadias. Time parameterized queries in spatio-temporal databases. In *SIGMOD*, pages 334–345.

[26] Y. Tao, D. Papadias, and X. Lian. Reverse kNN search in arbitrary dimensionality. In *VLDB*, 2004.

[27] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298.

[28] A. K. H. Tung, J. Hou, and J. Han. Spatial clustering in the presence of obstacles. In *ICDE*, 2001.

[29] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, 2005.

[30] X. Yu, K. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.

[31] O. R. Zaiane and C.-H. Lee. Clustering spatial data in the presence of obstacles: A density-based approach. In *IDEAS*, 2002.

[32] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-nearest-neighbors queries in spatial databases. In *SSDBM*, 2004.

[33] J. Zhang, D. Papadias, K. Mouratidis, and M. Zhu. Spatial queries in the presence of obstacles. In *EDBT*, 2004.