Visible Reverse k-Nearest Neighbor Query Processing in Spatial Databases

Yunjun Gao, *Member*, *IEEE*, Baihua Zheng, *Member*, *IEEE*, Gencai Chen, Wang-Chien Lee, *Member*, *IEEE*, Ken C.K. Lee, and Qing Li, *Senior Member*, *IEEE*

Abstract—*Reverse nearest neighbor* (RNN) queries have a broad application base such as decision support, profile-based marketing, resource allocation, etc. Previous work on RNN search does not take obstacles into consideration. In the real world, however, there are many physical obstacles (e.g., buildings) and their presence may affect the visibility between objects. In this paper, we introduce a novel variant of RNN queries, namely, *visible reverse nearest neighbor* (VRNN) search, which considers the impact of obstacles on the *visibility* of objects. Given a data set P, an obstacle set O, and a query point q in a 2D space, a VRNN query retrieves the points in P that have q as their *visible* nearest neighbor. We propose an efficient algorithm for VRNN query processing, assuming that P and O are indexed by R-trees. Our techniques do not require any preprocessing and employ *half-plane property* and *visibility check* to prune the search space. In addition, we extend our solution to several variations of VRNN queries, including: 1) *visible reverse k-nearest neighbor* (VRkNN) search, which finds the points in P that have q as one of their k *visible* nearest neighbors; 2) δ-VRkNN search, which handles VRkNN retrieval with the *maximum visible distance* δ *constraint*; and 3) *constrained* VRkNN (CVRkNN) search, which tackles the VRkNN query with *region constraint*. Extensive experiments on both real and synthetic data sets have been conducted to demonstrate the efficiency and effectiveness of our proposed algorithms under various experimental settings.

Index Terms—Reverse nearest neighbor, visible reverse nearest neighbor, spatial database, query processing, algorithm.

1 INTRODUCTION

REVERSE nearest neighbor (RNN) search has received considerable attention from the database research community in the past few years, due to its importance in a wide spectrum of applications such as decision support [6], profile-based marketing [6], [14], resource allocation [6], [19], etc. Given a set of data points P and a query point q in a multidimensional space, an RNN query finds the points in P that have q as their nearest neighbor (NN). A popular generalization of RNN is the *reverse k-nearest neighbor* (RkNN) search, which returns the points in P whose k NNs include q. Formally, $RkNN(q) = \{p \in P | q \in kNN(p)\}$, where $RkNN(q)$ represents the set of reverse k nearest neighbors to a query point q and $kNN(p)$ denotes the set of k nearest neighbors to a point p. Fig. 1a illustrates an example with four data points, labeled as p_1, p_2, p_3, and p_4, in a 2D space. Each point $p_i (1 \leq i \leq 4)$ is associated with a circle centered at p_i and having $dist(p_i, NN(p_i))$[1] as its radius, i.e., the circle $cir(p_i, NN(p_i))$ covers p_i's NN. For example, the circle $cir(p_3, NN(p_3))$ encloses p_2, the NN of p_3 (i.e., $NN(p_3)$). For a given RNN query issued at point q, its answer set $RNN(q) = \{p_4\}$ as q is only located inside the circle $cir(p_4, NN(p_4))$. It is worth noting the asymmetric NN relationship, that is, $p \in kNN(q)$ does not necessarily imply $q \in kNN(p)$, e.g., in Fig. 1a, we notice that $NN(p_4) = p_3$, but $NN(p_3) = p_2$.

1.1 Motivation

There are many RNN/RkNN query algorithms that have been proposed in the database literature. Basically, they can be classified into three categories: 1) precomputation-based algorithms [6], [19]; 2) dynamic algorithms [13], [14], [16]; and 3) algorithms for various RNN/RkNN query variants [7], [8], [15]. Nevertheless, none of the existing work on RNN/RkNN search has considered physical obstacles (e.g., buildings) that exist in the real world. The presence of obstacles may have a significant impact on the visibility or distance between objects, and hence, affects the result of RNN/RkNN queries. Furthermore, in some applications, users may be only interested in the objects that are visible or reachable to them.

Actually, the existence of physical obstacles has been considered in certain types of spatial queries. They include: 1) *obstructed nearest neighbor* (ONN) query [20], which returns the $k(\geq 1)$ points in P that have the smallest *obstructed distances* (defined as the length of the shortest path that connects any two points without crossing any

- *Y. Gao is with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore, and the College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, PR China. E-mail: yjgao@smu.edu.sg, gaoyj@zju.edu.cn.*
- *B. Zheng is with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore. E-mail: bhzheng@smu.edu.sg.*
- *G. Chen is with the College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, PR China. E-mail: chengc@zju.edu.cn.*
- *W.-C. Lee and K.C.K. Lee are with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802. E-mail: {wlee, cklee}@cse.psu.edu.*
- *Q. Li is with the Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong, PR China. E-mail: itqli@cityu.edu.hk.*

Manuscript received 7 Nov. 2008; revised 10 Mar. 2009; accepted 17 Apr. 2009; published online 28 Apr. 2009.
Recommended for acceptance by M. Garofalakis.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2008-11-0589.
Digital Object Identifier no. 10.1109/TKDE.2009.113.

1. Without loss of generality, function $dist(p_i, p_j)$ returns the euclidean distance between any two points p_i and p_j.

1041-4347/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

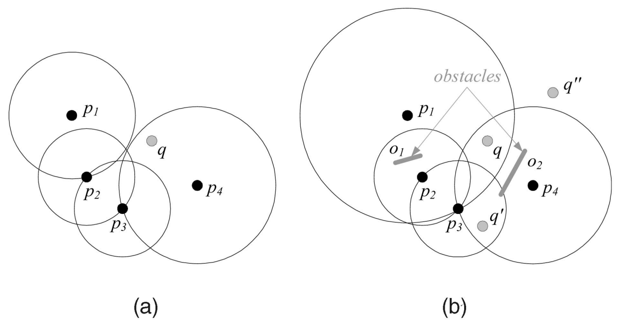


Fig. 1. Example of RNN and VRNN queries. (a) RNN search. (b) VRNN search.

obstacle from an obstacle set) to q; 2) *visible k-nearest neighbor* (VkNN) search [10], which finds the k nearest points that are *visible* to q; and 3) *clustering spatial data in the presence of obstacles* [17], which divides a set of 2D data points into smaller homogeneous groups (i.e., clusters) by taking into account the impact of obstacles. Different from the existing work, this paper considers the obstacles in the context of RNN/RkNN retrieval. To the best of our knowledge, this is the first work to address this problem.

1.2 Contributions

In this paper, we introduce a novel form of RNN queries, namely, *visible reverse nearest neighbor* (VRNN) search, which considers the impact of obstacles on the visibility of objects. Given a data set P, an obstacle set O, and a query point q in a 2D space, a VRNN query retrieves all the points in P that have q as their *visible* NN. Take a VRNN query issued at point q as an example (as depicted in Fig. 1b). It returns $\{p_1\}$ as the result set, which is different from the result of an RNN query issued at q (as shown in Fig. 1a). In addition, we define several variants of VRNN queries, including: 1) *visible reverse k-nearest neighbor* (VRkNN) search, a natural generalization of VRNN retrieval, which finds all the points $p \in P$ that have q as one of their k *visible* NNs; 2) δ-VRkNN search, which answers the VRkNN query with the *maximum visible distance δ constraint*; and 3) *constrained* VRkNN (CVRkNN) search, which processes the VRkNN query with *region constraint*. These potential variants form a suite of interesting and intuitive problems from both the research point of view and application point of view.

We focus this paper on VRNN search, not only because the problem is new to the research community but also because it has a large application base. Some of the example applications are listed as follows:

Outdoor advertisement planning. Suppose *P&G* plans to post advertisements in billboards to promote a new shampoo. In order to encourage customers to try this new product, *P&G* decides to distribute some samples near billboards as well. Due to the high cost of sample distribution, only those billboard locations that can reach a big pool of potential customers are considered. Ideally, the more people can view the billboards, the more effective the promotion will be. We assume that the number of candidate billboard locations is *small* due to limited budget, and each customer only pays attention to the billboard located *closest*, and meanwhile, *visible* to him/her. Hence, VRNN search can be conducted to compare the optimality of any two candidate billboard locations q_1 and q_2 in terms of the potential customer base they can reach. By performing a VRNN query which takes as inputs a set of residential buildings or shopping malls (that represent the potential customer base), a set of obstacles (e.g., buildings), and a query point q_1/q_2, the decision-maker can identify the customers that would watch the billboard located at q_1/q_2. The one with more customers is better.

Selection of promotion sites. Suppose *Yao Restaurant & Bar* plans to open a new restaurant YEEHA in Shanghai and wants to distribute coupons to its potential customers for promotion. Assume those customers who do not know YEEHA previously but have YEEHA as their *visible nearest restaurant* are more likely to visit YEEHA for a trial. Consequently, in order to ensure the effectiveness of the promotion, *Yao Restaurant & Bar* needs to locate all the office buildings and residential buildings that have YEEHA as their visible nearest restaurant, and to identify people working or staying in those buildings as its target consumers. VRNN search can provide a perfect match.[2]

A naive solution to deal with VRkNN ($k \geq 1$) queries is to find a set of points $p \in P$, denoted as S_q, which are visible to a specified query point q, perform VkNN search on each of them, and return these points $p \in S_q$ with $q \in \mathrm{V}k\mathrm{NN}(p)$. However, this method is very inefficient because it needs to traverse the data set P and obstacle set O *multiple times* (i.e., $(|S_q| + 1)$ times[3]), resulting in high I/O overhead and expensive CPU cost, especially when $|VRkNN(q)| \ll |S_q|$.

In this paper, we propose an efficient algorithm for VRNN query processing, assuming that both P and O are indexed by R-trees [2], [4]. Our method follows a *filter-refinement* framework and requires *no* preprocessing. Specifically, a set of candidate objects (i.e., a superset of the final query result) is retrieved in the filter step and gets refined in the subsequent refinement step, with these two steps integrated into a *single* R-tree traversal. Since the size of the candidate set has a direct impact on the search efficiency, we employ *half-plane properties* (as [16]) and *visibility check* to prune the search space. In addition, the search algorithm is general and can be easily extended to support different variants of VRNN queries, such as VRkNN search, δ-VRkNN search, and CVRkNN search.

In brief, the key contributions of this paper can be summarized as follows:

- We introduce and formalize VRNN retrieval, a novel addition to the family of RNN queries, which is very useful in many applications involving spatial data and physical obstacles for decision support.
- We develop an efficient VRNN search algorithm, analyze its cost, and prove its correctness.
- We extend our techniques to several variations of VRNN queries, including VRkNN search, δ-VRkNN search, and CVRkNN search.

2. Note that if we assume that those customers having YEEHA as their closest restaurant (no matter whether YEEHA is visible to them) are more likely to visit YEEHA for a trial, the RNN search based on the obstructed distance would be more suitable.

3. $|P|$ denotes the cardinality of a set P.

TABLE 1
Frequently Used Symbols

Notation	Description
P	A set of data points in a two-dimensional space
O	A set of obstacles in a two-dimensional space
T_p	The R-tree on P
T_o	The R-tree on O
q	A query point
e	An entry (point or MBR node) in an R-tree
VR_q	The visible region of q
L_q	A list that keeps the obstacle lines of the obstacles affecting the visibility of q
CR	A constrained region
$RkNN(q)$	Result set of a RkNN query issued at q
$VkNN(q)$	Result set of a VkNN query issued at q
$VRkNN(q)$	Result set of a VRkNN query issued at q

- We conduct extensive experiments using both real and synthetic data sets to demonstrate the performance of our proposed algorithms in terms of efficiency and effectiveness.

The rest of this paper is organized as follows: Section 2 formalizes VRkNN query and reviews related work. Section 3 discusses how to determine whether an object is visible to q in the presence of obstacles and introduces the concept of *visible region* to improve the search performance. Section 4 proposes an efficient algorithm for processing VRNN queries and conducts an analytical study to prove its correctness. Section 5 extends our solution to tackle several VRNN query variants. Extensive experimental evaluations and our findings are reported in Section 6. Finally, Section 7 concludes the paper with some directions for future work.

2 Background

In this section, we present the formal definition of VRkNN query, reveal its characteristics, and then, survey related work, including RNN/RkNN search algorithms and visibility queries. Table 1 lists the symbols used in this paper.

2.1 Problem Statement

Given a data set P, an obstacle set O, and a query point q in a 2D space, the visibility between two points is defined in Definition 1, based on which we formulate VkNN and VRkNN queries in Definition 2 and Definition 3, respectively.

Definition 1 (Visibility). *Given O in a 2D space, points p and p′ are visible to each other iff the straight line connecting p and p′ does not cut through any obstacle o in O, i.e.,* $\forall o \in O, \overline{pp'} \cap o = \emptyset$.

Definition 2 (Visible k-nearest neighbor query) [10]. *Given P, O, q in a 2D space, and an integer* $k(\geq 1)$, *a* VkNN query *finds a set of points* $VkNN(q) \subseteq P$, *such that: 1)* $\forall p \in VkNN(q)$ *is* visible *to q; 2)* $|VkNN(q)| \leq k^4$; *and 3)* $\forall p' \in P - VkNN(q)$ *and* $\forall p \in VkNN(q)$, *if* p' *is visible to* $q, dist(p,q) \leq dist(p',q)$.

Definition 3 (Visible reverse k-nearest neighbor query). *Given P, O, q in a 2D space, and an integer* $k(\geq 1)$, *a* VRkNN query *retrieves a set of points* $VRkNN(q) \subseteq P$, *such that* $\forall p \in VRkNN(q), q \in VkNN(p)$, *i.e.,* $VRkNN(q) = \{p \in P | q \in VkNN(p)\}$.

4. The cardinality of VkNN(q), i.e., $|\mathrm{V}k\mathrm{NN}(q)|$, may be smaller than k due to the obstruction of obstacles.

Next, some important properties of the VRkNN query that will be utilized to process VRkNN search are presented in Property 1, Property 2, and Property 3, respectively.

Property 1. *The VRkNNs of a query point q might not be localized to the neighborhood of q.*

Property 2. *Given a query point q, the cardinality of q's VRkNNs (i.e.,* $|VRkNN(q)|$*) varies by the position of q and the distributions of data points/obstacles.*

Property 3. $p \in VkNN(q)$ *does not necessarily imply* $p \in VRkNN(q)$ *and vice versa.*

In order to facilitate the understanding, we illustrate these properties using the example depicted in Fig. 1b. First, although point p_1 is the furthest from a specified query point q compared with other points, it is still an answer point to the VRNN query issued at q (i.e., $p_1 \in VRNN(q)$). In contrast, point p_2 that is closer to q than p_1 is not included in $VRNN(q)$. Second, for the same k, VRkNN queries issued at different locations may obtain different results with different number of answer points. As an example, $|VRNN(q)| = |\{p_1\}| = 1, |VRNN(q')| = |\{p_3, p_4\}| = 2$, and $|VRNN(q'')| = |\emptyset| = 0$. Third, the relationship of visible nearest neighbor is asymmetric. For instance, $VNN(q) = \{p_2\}$, *but* $VRNN(q) = \{p_1\}$ that does not contain p_2.

2.2 Related Work

2.2.1 Algorithms for RNN/RkNN Search

Since the concept of RNN was first introduced by Korn and Muthukrishnan [6], many algorithms have been proposed, which can be divided into three categories. The first category is *precomputation*-based [6], [19]. For each point p, it precomputes the distance from p to its nearest neighbor p' (i.e., $NN(p)$) and forms a vicinity circle $cir(p, p')$ that is centered at p and has $dist(p, p')$ as the radius. For a given query point q, it examines q against all the vicinity circles $cir(p, p')$ with $p \in P$, and those having their vicinity circles enclosing q form the final result, i.e., $RNN(q) = \{p \in P | q \in cir(p, NN(p))\}$. To facilitate the examination, all the vicinity circles can be indexed by RNN-tree [6] or RdNN-tree [19]. Approaches of this category mainly have two shortcomings. First, both the index construction cost and the index update overhead are very expensive. To address this problem, bulk insertion in the RdNN-tree has been proposed in [9]. Second, although these methods can be extended to handle the RkNN retrieval (if the corresponding kNN information for each point is available), they are limited to answer RkNN queries for a fixed k. To support various k, an approach for RkNN search with local *kNN-distance* estimation has been developed in [18].

The second category does not rely on precomputation but adopts a filter-refinement framework [13], [14], [16]. In the filter step, the space is pruned according to defined heuristics, and a set of candidate objects is retrieved from the data set. In the refinement step, all the candidates are verified according to kNN search criteria and those *false hits* are removed. For example, based on a given query point q, the original 2D data space can be partitioned around q into six equal regions, such that the NNs of q found in each region are the only candidates of the RNN query [14]. Thus,

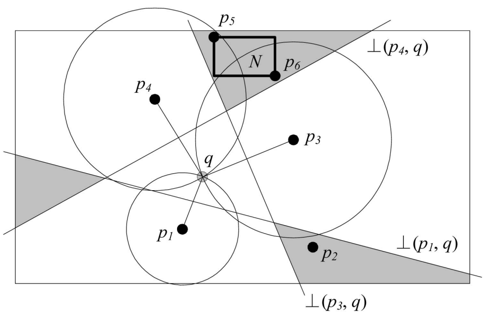


Fig. 2. Example of TPL algorithm.

in the filter step, six *constrained* NN queries are conducted to find the candidates in each region; and then, at the second step, NN queries are applied to eliminate the false hits. The efficiency of this approach is owing to the small number of candidates, e.g., at most 6 for an RNN query in a 2D space. However, the number of candidates grows exponentially with the increase of the search space dimensionality, meaning that the search efficiency can only be guaranteed in a low-dimensional space. To efficiently process RNN queries in a high-dimensional space, an approximated algorithm is proposed in [13]. It retrieves m nearest points to q as candidates with m (a randomly selected number) larger than k, and then, verifies the candidates using range queries. Nevertheless, the accuracy and performance of this algorithm is highly dependent on the value of m. The larger the m value is, the more the identified candidates are. Consequently, it is more likely that a complete result set is returned with a higher processing cost. A small m value favors the efficiency, whereas it may incur *false misses*, i.e., points that are actual reverse k nearest neighbors but missed from the final query result set.

In order to conduct *exact* RNN search, an efficient algorithm, called TPL, is proposed in [16]. TPL exploits a *half-plane property* to locate RkNN candidates. Applying the *best-first* traversal paradigm, TPL traverses the data R-tree to retrieve the NNs of q as RkNN candidates. Every time, an unexplored data point p is retrieved, a *half-plane* is constructed along the perpendicular bisector between p and q, denoted as $\perp(p, q)$. The bisector divides the data space into two half-planes: $HP_q(p, q)$ that contains q and $HP_p(p, q)$ that contains p. Any object, including both points and *minimum bounding rectangle* (MBR), falling completely inside $HP_p(p, q)$ must have p closer to it than q. As shown in Fig. 2, the bisector $\perp(p_3, q)$ partitions the space into two half-planes. As point p_1 falls into the half-plane $HP_q(p_3, q)$, it is closer to q than to p_3. In addition, the number of half-planes $HP_p(p, q)$ that a given point p' falls in represents the number of data points that are closer to p' than q. Hence, if a data point is within at least k $HP_p(p, q)$ half-planes, it cannot be a qualifying RkNN candidate, and thus, can be safely discarded. The filter step terminates when all the nodes of R-tree are either pruned or visited. As illustrated in Fig. 2, points p_1, p_3, and p_4 are identified as the RNN candidates in the filter step, while point p_2 that is inside $HP_{p1}(p_1, q) \cap HP_{p3}(p_3, q)$ and N (enclosing points p_5, p_6) that is within $HP_{p3}(p_3, q) \cap HP_{p4}(p_4, q)$) are filtered out. Later, in the refinement step, TPL eliminates false hits by reusing the pruned points/MBRs. Continuing the running example, points p_3 and p_4 are false hits, as their vicinity circles enclose other points. The final query result set is $\{p_1\}$. Our proposed algorithms for VRNN search and its variations employ *half-plane property* and *visibility check* to identify result candidates and prune the search space.

Algorithms belonging to the third category are to tackle various RNN/RkNN query variants, such as *bichromatic* RNN queries [15], *aggregate* RNN queries over data stream [7], and *ranked* RNN search [8].

2.2.2 Visibility Queries

Visibility computation algorithms that determine object visibility from a given viewpoint or a viewing cell have been well-studied in the area of computer graphics and computational geometry [1]. However, there are only a few works on visibility queries in the database community [5], [11], [12]. The basic idea is to employ various indexing structures (e.g., LoD-R-tree [5], HDoV-tree [12], etc.) to deal with visibility queries in visualization systems. These specialized access methods are designed only for the purpose of visualization test, and hence, contain *zero* distance information. Thus, they are not capable of supporting efficient VRkNN query processing. Recently, VkNN search [10] has been investigated, where the goal is to retrieve the k NNs that are *visible* to a specified query point. Further study along this line includes *continuous* VkNN retrieval [3].

3 Preliminaries

As VRNN search considers the impact of obstacles on objects' visibility, all the objects that are *invisible* to q for sure will not be contained in the result. Consequently, an essential issue we have to address is how to determine whether an object is visible to q. A simple approach is to examine a given object p against all the obstacles w.r.t. q, which is inefficient because the examination of each object p requires a scanning of the obstacles. In this paper, we derive a *visible region* for the query point q, denoted by VR_q, by visiting the obstacle set *once*, and the visibility of an object p w.r.t. q can be determined by checking whether p is located inside VR_q. In this section, we explain the formation of the visible region.

Before we present the detailed formation algorithm, we first discuss the presentation of a visible region. As shown in Fig. 3, a visible region might be in an irregular shape and we can use vertex to represent it. Nevertheless, it might not be so straightforward to determine whether an object is inside an irregular polygon. Alternatively, we propose to use *obstacle lines*, defined in Definition 4, to handle this problem.

Definition 4 (Obstacle line). *The obstacle line of an obstacle o[5] w.r.t. q, denoted by ol_o, is the line segment that obstructs the sight lines from q.*

Suppose the rectangle o depicted in Fig. 3 is an obstacle, and its corresponding obstacle line is ol_o. The shadowed

5. Although an obstacle o may be in an arbitrary shape (e.g., triangle, pentagon, etc.), we assume that o is a rectangle in this paper.

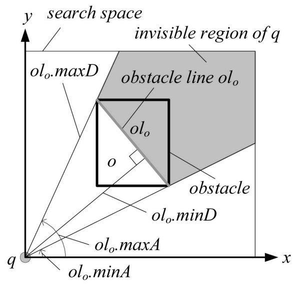


Fig. 3. An example obstacle line and its angular and distance bounds.

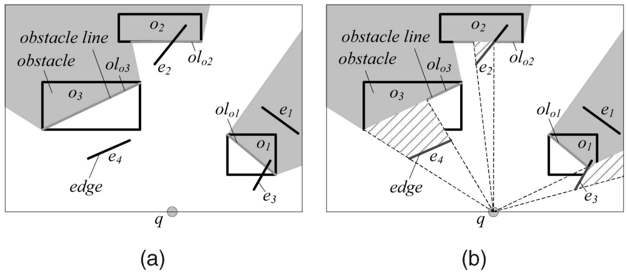


Fig. 4. Example of edge visibility check. (a) Obstacle placement. (b) New visible region.

area, blocked by ol_o, is not visible to q, and the rest (except o) is within the visible region of q (i.e., VR_q). Based on the concept of obstacle line, we can determine the *angular bound* and the *distance bound* of an obstacle line w.r.t. q, which can be utilized to facilitate the visibility checking of objects.

Taking q as an origin in the search space, the *angular bound* of o's obstacle line (i.e., ol_o) w.r.t. q is denoted as $[ol_o.minA, ol_o.maxA]$, in which $ol_o.minA$ and $ol_o.maxA$ are, respectively, the minimum angle and the maximum angle of ol_o, and $ol_o.minA \leq ol_o.maxA$ (see Fig. 3). If q is located inside o, the angular bound of ol_o w.r.t. q is set to [0, 2π]. When ol_o intersects with the positive x-axis in the search space, we partition ol_o horizontally along the x-axis into ol_{o1} and ol_{o2}. In addition, given two obstacles o and o', if their angular bounds are *disjoint*, i.e., $[ol_o.minA, ol_o.maxA] \cap [ol_{o'}.minA, ol_{o'}.maxA] = \emptyset$, they will not affect each other's visibility w.r.t. q. The *distance bound* of o's obstacle line w.r.t. q is denoted as $[ol_o.minD, ol_o.maxD]$, where $ol_o.minD$ and $ol_o.maxD$ are the minimal distance and the maximal distance from q to ol_o, respectively (see Fig. 3).

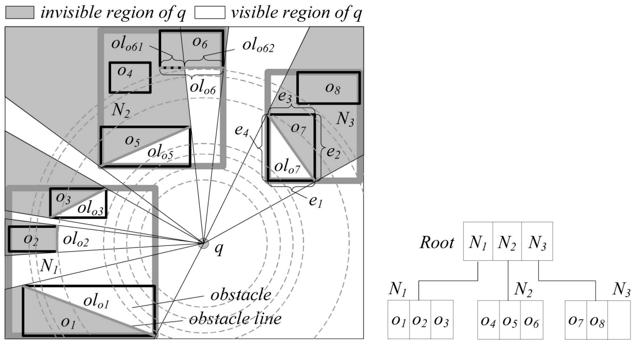
Without any obstacle, the visible region of q (i.e., VR_q) is the entire search space. As obstacles are visited, VR_q gets shrunk. Consequently, an issue we have to solve is how to decide whether a new obstacle might change the size of VR_q. In the following, we first explain the examination-based online segments (or edges), namely, *Edge Visibility Check* (EVC), and then, extend it for obstacles in rectangular shapes.

EVC gradually examines the obstacles and maintains the obstacle lines of all the obstacles found so far which affect the visibility of a given query point q. Given a new obstacle o, o might affect these obstacles with angular bounds overlapping with o's but definitely not the rest. Consequently, EVC evaluates the impact of o on the size of VR_q via comparing o's angular bound against that of obstacle lines in L_q.

Due to the space limitation, the pseudocode of EVC is skipped, while we use an example depicted in Fig. 4 to illustrate the basic idea. Assume that $L_q = \{ol_{o1}, ol_{o2}, ol_{o3}\}$ and e_2 is the edge to be evaluated. According to the angular bound of each obstacle line $l \in L_q$ and that of edge e_2, there are three possible cases: 1) $l.maxA \leq e_2.minA$ (e.g., $l = ol_{o1}$), indicating that e_2 will not affect the visibility of l w.r.t q; 2) $[l.minA, l.maxA] \cap [e_2.minA, e_2.maxA] \neq \emptyset$ (e.g., $l = ol_{o2}$), meaning that a detailed examination is necessary as e_2 is very likely to affect l's visibility w.r.t. q; and 3) $l.minA \geq e_2.maxA$ (e.g., $l = ol_{o3}$), which indicates that l and all the remaining obstacle lines in L_q with $minA$ larger than that of l's will not be affected by e_2, and thus, the evaluation on e_2 can be terminated.

Now the only left task is how to change L_q when a new obstacle line l_n overlaps with some existing obstacle line l in L_q (i.e., **case 2** above). Again, there are three possible cases. First, $l.maxD \leq l_n.minD$ holds, which means that l_n has *zero* impact on q's visible region VR_q. For example, in Fig. 4b, although e_1 overlaps with o_1 in terms of angular bounds, it is *invisible* to q, and hence, can be ignored. Second, $l.minD \geq l_n.maxD$ satisfies, which indicates that the entire l_n is *visible* to q. Thus, l_n is inserted into L_q and the part of l that is blocked by l_n is removed. In Fig. 4b, for instance, e_4 is within the angular bound of o_3 and its maximal distance to q (i.e., $e_4.maxD$) is smaller than the minimal distance between o_3's obstacle line ol_{o3} and q (i.e., $ol_{o3}.minD$). Consequently, e_4 that is visible to q is included into L_q and ol_{o3} is shrunk, as shown in Fig. 4b. Third, l_n and l intersect, meaning that part of l_n is *visible* to q and the other part of l obstructed by l_n becomes *invisible* to q. L_q needs to include the new visible part of l_n and removes the invisible part of l. As an example, in Fig. 4b, edge e_3 and the obstacle line of o_1 (i.e., ol_{o1}) intersect, and edge e_2 and o_2's obstacle line ol_{o2} intersect. Thus, we find the intersection points, and then, update L_q. After evaluating new edges e_1, e_2, e_3, and e_4, the visible region of q (i.e., VR_q) is updated to the shaded area (containing the shaded region highlighted in dashed line), as illustrated in Fig. 4b.

Next, we explain how to extend the algorithm of EVC to determine the impact of a rectangle N on VR_q, namely, *Object Visibility Check* (OVC). The basic idea of OVC is to invoke EVC to evaluate the edges of a rectangle. It is worth noting that OVC only needs to evaluate *at most two* out of *four* edges of a rectangle, because at most two edges may affect the formation of VR_q. Take the obstacle o_7 (i.e., the rectangle that is formed by edges e_1, e_2, e_3, and e_4) in Fig. 5a as an example. Since a specified query point q lies in the southwest of o_7, only the two edges e_1 and e_4 facing toward q need evaluations, whereas the other two edges e_2 and e_3 are ignored. During the processing of OVC, we distinguish the following two possible situations: 1) if two evaluated edges of N are *invisible* to q, OVC returns *IV* to indicate that N is invisible to q, and hence, N and all its enclosed child nodes can be pruned away and otherwise, 2) two evaluated edges of N are *visible* (partially or completely) to q, OVC returns AV or PV to indicate that N is all-visible (i.e., completely visible) or partially visible to q. If N represents an obstacle, the impact of N's edges on VR_q is evaluated by EVC, which updates L_q if necessary. Otherwise, N must be



(a) (b)

Fig. 5. Example of VRC algorithm. (a) Obstacle placement. (b) The obstacle R-tree.

an intermediate node and its child nodes are accessed for further exploration. We omit the pseudocode of OVC due to space limitation.

We are now ready to present our *Visible Region Computation Algorithm* (VRC). We assume that all the obstacles are indexed by an R-tree T_o, and VRC traverses T_o in a *best-first* manner, with unvisited nodes maintained by a min-heap H sorted based on ascending order of their minimal distances to a given query point. Algorithm 1 shows the pseudocode of VRC algorithm. It continuously checks the head entry e of H. The detailed examination varies, dependent on the type of e. If e is an obstacle, it is checked against all the obstacle lines preserved in L_q (lines 6-7). If it is visible to q, e might contribute to the formation of VR_q, and thus, L_q is updated. On the other hand, e must be a node and all its child entries that are *visible* (completely or partially) to q are enheaped for later examination (lines 8-10). VRC also exploits an *early termination condition* (lines 4-5), as proved by Lemma 1.

Algorithm 1. Visible Region Computation Algorithm (VRC)
algorithm VRC (T_o, q, L_q)
/* $T_o.root$: the root node of R-tree T_o; *IV*: *invisible* */
1: insert all entries of $T_o.root$ into min-heap H; list $L_q = \emptyset$
2: **while** $H \neq \emptyset$ **do**
3: de-heap the top entry (*e*, *key*) from H
4: **if** $L_q.isclose =$ TRUE and $mindist(e, q) > \mathrm{MAX}_{l \in Lq}(l.maxD)$ **then**
5: break // terminate
6: **if** *e* is an obstacle **then**
7: OVC (e, L_q, q) // check *e*'s visibility w.r.t. q
8: **else** // *e* is a MBR (i.e., an intermediate node)
9: **for** each entry $e_i \in e$ and OVC (e_i, L_q, q) $\neq$ *IV* **do**
10: insert ($e_i, mindist(e_i, q)$) into H

Lemma 1. *Suppose heap H maintains all the unvisited nodes sorted in ascending order of their minimal distances to the query point q and list L_q keeps the obstacle lines of all the obstacles found so far that affect the visibility of q. If L_q is closed (i.e., $\cup_{l \in Lq}[l.minA, l.maxA] = [0, 2\pi;])$, denoted as $L_q.isclose$ = TRUE, and $mindist(e, q) > \mathrm{MAX}_{l \in Lq}(l.maxD)$, e and all the rest entries in H are invisible to q.*

Proof. Suppose there is an entry e with $mindist(e, q) > \mathrm{MAX}_{l \in Lq}(l.maxD) = d_{max}$ visible to q. As e is visible to q, there must be at least one line segment issued at q and reaching a point of e (denoted as p) without cutting through any other obstacle (by Definition 1). Since L_q is closed, without loss of generality, we can assume that the extension of line segment $\overline{qp}$ intersects an obstacle line $l \in L_q$ at point p' with $dist(p, q) \leq dist(p', q) \leq d_{max}$. As we know that $mindist(e, q) \leq dist(p, q)$ holds. Hence, $mindist(e, q) \leq d_{max} = \mathrm{MAX}_{l \in Lq}(l.maxD)$ satisfies, which contradicts our previous assumption. □

An illustrative example of the VRC algorithm is depicted in Fig. 5, where obstacle set $O = \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8\}$ is indexed by the R-tree T_o shown in Fig. 5b. We use a list L_q to store the obstacle lines of all the obstacles that can affect the visibility of q, sorted according to ascending order of their minimum bounding angles; and a heap H to maintain all the unvisited entries, sorted based on their minimal distances to q. Initially, $H = \{N_1, N_2, N_3\}$ and the algorithm always deheaps the top entry from H for examination until H becomes empty. First, N_1 is accessed. As it is visible to q, its child nodes are enheaped for later examination, after which $H = \{o_1, N_2, N_3, o_3, o_2\}$. Then, o_1 is evaluated. Since it is the first obstacle checked, o_1 for sure affects q's visibility and is added to $L_q(= \{ol_{o1}\})$. Third, N_2 is checked. According to current L_q, N_2 is visible to q, and thus, its child nodes are enheaped, with $H = \{o_5, N_3, o_3, o_2, o_4, o_6\}$. Fourth, o_5 is examined and becomes the second obstacle affecting the visibility of q, i.e., $L_q = \{ol_{o5}, ol_{o1}\}$. Next, N_3 is deheaped and its child nodes are enheaped into $H(= \{o_7, o_3, o_2, o_4, o_8, o_6\})$. In the sequel, VRC deheaps obstacles from H and keeps updating L_q until $H = \emptyset$. Finally, $L_q = \{ol_{o7}, ol_{o62}, ol_{o5}, ol_{o3}, ol_{o2}, ol_{o1}\}$, in which ol_{o62} is the partial obstacle line of obstacle o_6, as illustrated in Fig. 5a.

4 VRNN Query Processing

In this section, we explain how to process VRNN query. We first present the pruning strategy followed by the details of VRNN search algorithm. Then, we analyze the cost of VRNN algorithm and prove its correctness.

4.1 Pruning Strategy

In order to improve the search performance, we utilize *half-plane property* (as [16]) and *visibility check* (discussed in Section 3) to prune the search space. Consider the perpendicular bisector between a data point p_1 and a given query point q, denoted by $\perp(p_1, q)$, i.e., line l_1 in Fig. 6. The bisector divides the whole data space into two half-planes, i.e., $HP_{p1}(p_1, q)$ containing p_1 (i.e., trapezoid *EFCD*) and $HP_q(p_1, q)$ containing q (i.e., trapezoid *ABFE*). All the data points (e.g., p_2, p_3) and nodes (e.g., N_1) that fall completely inside $HP_{p1}(p_1, q)$ and are *visible* to p_1 must have p_1 closer to them than q, and thus, they cannot be/contain a VRNN of q. However, all the data points (e.g., p_6, p_7) and nodes (e.g., N_2, N_3) that fall into $HP_{p1}(p_1, q)$ but are *partially visible/invisible* to p_1 might become or contain a VRNN of q. Therefore, they cannot be discarded and a further examination is necessary. In the following description, we term p_1 as a *pruning point*.

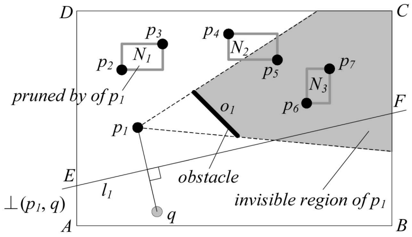


Fig. 6. Illustration of pruning based on half-planes and visibility check.

4.2 The VRNN Algorithm

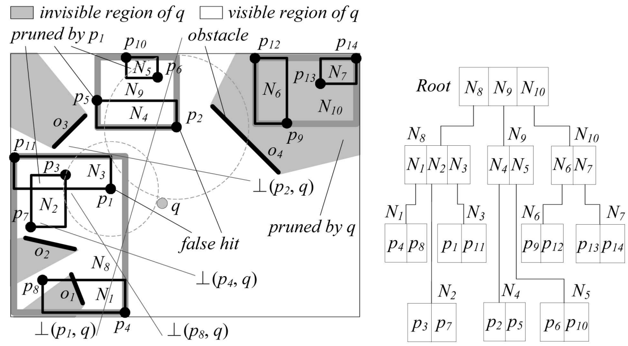
Based on the above pruning strategy, the basic idea of the VRNN algorithm proposed in this paper tries to prune away unqualified data objects/nodes to save the traversal cost. Consequently, it adopts a two-step *filter-and-refinement* framework, assuming that data set P and obstacle set O are indexed by two separate R-trees. In order to enhance the performance, these two steps are well integrated into a *single* traversal of the trees. In particular, the algorithm accesses nodes/points in ascending order of their distances to the query point q to retrieve a set of potential candidates, maintained by a candidate set S_c. All the data points and nodes that cannot be/contain a VRNN of q are discarded by our proposed pruning strategy, preserved in a refinement point *set* S_p and a refinement node set S_n, respectively. At the second step, the entries in both S_p and S_n are used to eliminate false hits.

Algorithm 2. VRNN Search Algorithm (VRNN)
algorithm VRNN (T_p, T_o, q)
/* S_c: candidate set; S_p: refinement point set;
S_n: refinement node set;
S_r: result set of a VRNN query */
initialize sets $S_c = \emptyset, S_p = \emptyset, S_n = \emptyset, S_r = \emptyset$
VRNN-Filter $(T_p, T_o, q, S_c, S_p, S_n)$
VRNN-Refinement (q, S_c, S_p, S_n, S_r)
return S_r

Algorithm 2 presents the pseudocode of the *VRNN Search Algorithm* (VRNN) that takes data R-tree T_p, obstacle R-tree T_o, and a query point q as inputs, and outputs *exactly* all the VRNNs of q. We use an example shown in Fig. 7 to elaborate the VRNN algorithm. Here, $P = \{p_1, p_2, \ldots, p_{13}, p_{14}\}, O = \{o_1, o_2, o_3, o_4\}$, and the corresponding T_p is depicted in Fig. 7b. A *primary heap* H_w is maintained to keep all the unvisited entries ordered in ascending order of their minimal distances to the query point q.

4.2.1 The Filter Step

Initially, VRNN visits the root node of T_p, inserts its child entries N_8 and N_9 that are visible to q into $H_w(= \{N_8, N_9\})$, and adds the entry N_{10} that is invisible to q to $S_n(= \{N_{10}\})$. Then, the algorithm deheaps N_8, accesses its child nodes, and enheaps all the entries that are visible to q, after which $H_w = \{N_3, N_9, N_1, N_2\}$. Next, N_3 is visited and it updates H_w to $\{p_1, N_9, N_1, N_2, p_{11}\}$. The next deheaped entry is p_1. As it is visible to q, p_1 is the first VRNN candidate (i.e., $S_c = \{p_1\}$) and becomes the *current pruning point cp* that is used for pruning in the subsequent execution.



(a) (b)

Fig. 7. Example of VRNN algorithm. (a) Data and obstacle placement. (b) The data R-tree.

The next deheaped entry is N_9. As $cp(= p_1)$ is not *empty*, VRNN uses *Trim algorithm*[6] (as [16]) to check whether N_9 can be pruned. As N_9 overlaps with $HP_q(cp, q)$, its child nodes have to be accessed. Child node N_5 is discarded as it locates inside $HP_{cp}(cp, q)$ and it is *visible* (completely) to *cp*, meaning that it cannot contain any qualified candidate. Thus, N_5, which is an MBR, is added to S_n, i.e., $S_n = \{N_{10}, N_5\}$. The other child entry N_4 is enheaped into $H_w(= \{N_4, N_1, N_2, p_{11}\})$ because it falls partially into $HP_{cp}(cp, q)$ and is *visible* (completely) to *cp*, indicating that N_4 may contain VRNN candidates. VRNN proceeds to deheap N_4 and visits its child entries, i.e., data points p_2 and p_5. As p_2 falls inside $HP_q(cp, q)$ and is visible to *cp*, it is added to $H_w(= \{p_2, N_1, N_2, p_{11}\})$. On the other hand, point p_5 is inserted into $S_p = \{p_5\}$ since it locates inside $HP_{cp}(cp, q)$ and is visible to *cp*. Next, p_2 is deheaped. As it cannot be pruned by current pruning point (p_1), it becomes the second pruning point and maintained by an *auxiliary heap* $H_a = \{p_2\}$.

Subsequently, VRNN accesses node N_1 and inserts its child points p_4 and p_8 into $H_w(= \{N_2, p_4, p_8, p_{11}\})$. Note that although p_8 falls fully into $HP_{cp}(cp, q)$, it is *invisible* to the current pruning point (i.e., p_1) due to the obstruction of obstacle o_2, and hence, p_8 cannot be pruned by *cp*. The next processed entry N_2 is added to $S_n(= \{N_{10}, N_5, N_2\})$ directly, as it locates inside $HP_{cp}(cp, q)$ and is *visible* (completely) to *cp*. In the sequel, p_4 and p_8 are retrieved and inserted into H_a, after which $H_a = \{p_2, p_4, p_8\}$. Finally, p_{11} is deheaped and added to $S_p = \{p_5, p_{11}\}$ since it satisfies the pruning condition. Here, as H_w is empty, *the first loop* stops, with H_a, S_c, S_p, and S_n being $\{p_2, p_4, p_8\}$, $\{p_1\}$, $\{p_5, p_{11}\}$, and $\{N_{10}, N_5, N_2\}$, respectively. The heap contents at each phase during the aforementioned filter process are illustrated in Table 2 where, for simplicity, we omit associated distances to q for node MBRs and data points.

Algorithm 3. Filter for VRNN Algorithm (VRNN-Filter)
algorithm VRNN-Filter $(T_p, T_o, q, S_c, S_p, S_n)$
/* $T_p.root$: the root node of R-tree T_p; *IV*: *invisible*; *AV*: *all-visible*; *PV*: *partially-visible* */

6. If a node MBR can be completely discarded, the Trim algorithm returns ∞; otherwise it returns the minimum distance between a given query point q and the residual MBR. Similarly, it returns the actual distance from a point to q if the point cannot be pruned, or ∞ otherwise. Refer to [16] for details.

TABLE 2
Heap Contents during the First Loop of Filter Step

Action	H_w	H_a	S_c	S_p	S_n
Visit root	$\{N_8, N_9\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{N_{10}\}$
Visit N_8	$\{N_3, N_9, N_1, N_2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{N_{10}\}$
Visit N_3	$\{p_1, N_9, N_1, N_2, p_{11}\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{N_{10}\}$
Process p_1	$\{N_9, N_1, N_2, p_{11}\}$	$\emptyset$	$\{p_1\}$	$\emptyset$	$\{N_{10}\}$
Visit N_9	$\{N_4, N_1, N_2, p_{11}\}$	$\emptyset$	$\{p_1\}$	$\emptyset$	$\{N_{10}, N_5\}$
Visit N_4	$\{p_2, N_1, N_2, p_{11}\}$	$\emptyset$	$\{p_1\}$	$\{p_5\}$	$\{N_{10}, N_5\}$
Process p_2	$\{N_1, N_2, p_{11}\}$	$\{p_2\}$	$\{p_1\}$	$\{p_5\}$	$\{N_{10}, N_5\}$
Visit N_1	$\{N_2, p_4, p_8, p_{11}\}$	$\{p_2\}$	$\{p_1\}$	$\{p_5\}$	$\{N_{10}, N_5\}$
Process N_2	$\{p_4, p_8, p_{11}\}$	$\{p_2\}$	$\{p_1\}$	$\{p_5\}$	$\{N_{10}, N_5, N_2\}$
Process p_4	$\{p_8, p_{11}\}$	$\{p_2, p_4\}$	$\{p_1\}$	$\{p_5\}$	$\{N_{10}, N_5, N_2\}$
Process p_8	$\{p_{11}\}$	$\{p_2, p_4, p_8\}$	$\{p_1\}$	$\{p_5\}$	$\{N_{10}, N_5, N_2\}$
Process p_{11}	$\emptyset$	$\{p_2, p_4, p_8\}$	$\{p_1\}$	$\{p_5, p_{11}\}$	$\{N_{10}, N_5, N_2\}$

insert all entries of $T_p.root$ into min-heap H_w; $cp =$ NULL; $H_a = \emptyset$
VRC (T_o, q, L_q) // compute q's visible region VR_q
while $H_w \neq \emptyset$ **do**
  de-heap the top entry (e, *key*) from H_w
  if e is a data point **then**
    $S_c = S_c \cup \{e\}; cp = e; \mathrm{VRC}(T_o, cp, L_{cp})$
    while $H_w \neq \emptyset$ **do**
      de-heap the top entry (e', key') from H_w
      if e' is a data point **then**
        if Trim $(q, cp, e') = \infty$ and OVC $(e', L_{cp}, cp) = AV$ **then**
          $S_p = S_p \cup \{e'\}$
        else
          insert $(e', dist(e', q))$ into H_a
      else // e' is a MBR (i.e., an intermediate node)
        for each entry $e'_i \in e'$ **do**
          if OVC $(e'_i, L_q, q) \neq IV$ **then**
            if Trim $(q, cp, e'_i) = \infty$ and OVC $(e'_i, L_{cp}, cp) = AV$ **then**
              $S_p = S_p \cup \{e'_i\}$ if e'_i is a data point or $S_n = S_n \cup \{e'_i\}$ if e'_i is a node
            else if Trim $(q, cp, e'_i) = \infty$ and OVC $(e'_i, L_{cp}, cp) = IV$ **then**
              insert $(e'_i, mindist(e'_i, q))$ into H_a
            else
              insert $(e'_i, mindist(e'_i, q))$ into H_w
          else // OVC $(e'_i, L_q, q) = IV$
            $S_p = S_p \cup \{e'_i\}$ if e'_i is a data point or $S_n = S_n \cup \{e'_i\}$ if e'_i is a node
      swap (H_w, H_a) // change the roles between H_w and H_a
  else // e is a MBR (i.e., an intermediate node)
    for each entry $e_i \in e$ **do**
      if OVC $(e_i, L_q, q) \neq IV$ **then**
        if $cp \neq$ NULL and Trim $(q, cp, e_i) = \infty$ and OVC $(e_i, L_{cp}, cp) = AV$ **then**
          $S_p = S_p \cup \{e_i\}$ if e_i is a data point or $S_n = S_n \cup \{e_i\}$ if e_i is a node
        else
          insert $(e_i, mindist(e_i, q))$ into H_w
      else // OVC $(e_i, L_q, q) = IV$
        $S_p = S_p \cup \{e_i\}$ if e_i is a data point or $S_n = S_n \cup \{e_i\}$ if e_i is a node

Next, the roles of H_w and H_a are switched. In other words, in the rest of current iteration, the algorithm uses H_w as an auxiliary heap, while takes H_a as a primary heap. VRNN proceeds in the same loop until $H_w = H_a = \emptyset$, i.e., all the points are either pruned (i.e., inserted into S_p) or become candidates (i.e., inserted into S_c). Finally, we have $S_c = \{p_1, p_2, p_4, p_8\}$, $S_p = \{p_5, p_{11}\}$, and $S_n = \{N_{10}, N_5, N_2\}$.

Algorithm 3 shows the pseudocode of the *Filter for VRNN Algorithm* (VRNN-Filter). When an intermediate node is visited, it utilizes OVC function to check its visibility to the query point q, and then, processes it. Similarly, when a data point is accessed, it uses OVC function to examine its visibility to the current pruning point *cp*, and then, processes it. For each pruning point *cp* discovered, VRNN-Filter applies VRC algorithm to get its visible region, i.e., finding the obstacles from T_o that can affect *cp*'s visibility. Note that all pruned entries are preserved in their corresponding refinement sets but not removed permanently, as they will be used to verify candidates in the next refinement step.

4.2.2 The Refinement Step

When the filter step finishes, the refinement step starts, with the pseudocode of *Refinement for VRNN Algorithm* (VRNN-Refinement) depicted in Algorithm 4. In the first place, VRNN-Refinement conducts *self-filtering* (lines 2-4), that is, it prunes away the candidates that are visible to each other and are closer to each other than to q. Then, the algorithm enters the refinement step, where it verifies whether each remaining candidate in S_c is a true result (lines 7-18). First, it calls *Round of Refinement Algorithm* (Refinement-Round), depicted in Algorithm 5, to eliminate false candidates from S_c based on the contents of S_p and S_n, without any extra node access. The remaining points p in S_c need further refinement, with each associated with *p.toVisit* that records the nodes which might enclose some not-yet visited points that may invalidate p. Hence, nodes in *p.toVisit* are visited, with each access updating the contents of S_p and S_n. Note that S_p and S_n are reset to $\emptyset$ after each round of Refinement-Round (line 12) to avoid duplicated checking. The refinement step continues until $S_c = \emptyset$.

Algorithm 4. Refinement for VRNN Algorithm (VRNN-Refinement)
algorithm VRNN-Refinement (q, S_c, S_p, S_n, S_r)
1: **for** each point $p \in S_c$ **do**
2:   **for** each other point $p' \in S_c$ **do**
3:     **if** OVC $(p', L_p, p) \neq IV$ and $dist(p', p) < dist(q, p)$ **then**
4:       $S_c = S_c - \{p\}$; goto 1
5:   **if** p is not eliminated from S_c **then**
6:     initialize $p.toVisit = \emptyset$
7: **if** $S_c \neq \emptyset$ **then**
8:   **repeat**
9:     Refinement-Round (q, S_c, S_p, S_n, S_r)
10:     let N be the lowest level node of *p.toVisit* for $p \in S_c$
11:     remove N from all *p.toVisit* and access N
12:     $S_p = S_n = \emptyset$ // for the next round
13:     **if** N is a leaf node **then**
14:       $S_p = \{p' | p' \in N \text{ and } p' \text{ is visible to } p\}$

else
$S_n = \{N' | N' \in N$ and N' is visible to $p\}$
else
return // terminate

Now, we explain the details of Refinement-Round algorithm. Specifically, it has three tasks, i.e., pruning false positive, identifying nodes that might invalidate the remaining points in S_c, and returning final result objects. First, points p in S_c satisfying any of following conditions are for sure false positives and can be pruned: 1) $\exists p' \in S_p$ such that p' is visible to p and $dist(p', p) < dist(q, p)$ (lines 2-4), or 2) $\exists N \in S_n$ such that N is all-visible to p and $minmaxdist(N, p) < dist(q, p)$ (lines 5-8). Note that *minmaxdist*(N, p) is the upper bound of the distance between p and its closest point in N. Thus, $minmaxdist(N, p) < dist(q, p)$ meaning that N contains at least one point that is closer to p than to q. For example, in Fig. 7, $p_2 \in S_c$ can be safely discarded because $N_5 \in S_n$ is all-visible to it and $minmaxdist(N_5, p_2) < dist(q, p_2)$. Second, $\forall p \in S_c$ can be reported immediately as an actual VRNN of q when the following two conditions are satisfied: 1) $\forall p' \in S_p, p'$ is either invisible to p or $dist(p', p) > dist(q, p)$ and 2) $\forall N \in S_n$, it is all-visible/partially visible to p and $mindist(N, p) > dist(q, p)$. In our example, p_4 and p_8 satisfy the above conditions, and hence, they are removed from S_c and reported as the VRNNs of q immediately. The point $p \in S_c$ that cannot be pruned or reported as a real result must have some nodes in S_n that contradict above conditions, and we utilize a set *p.toVisit* to record all those nodes (lines 9-11). Take p_1 as an example. As $p_1.toVisit = \{N_2\}$, we access N_2 and find out that the enclosed point p_3 is the VNN of p_1, and thus, p_1 is invalidated.

Algorithm 5. Round of Refinement Algorithm (Refinement-Round)
algorithm Refinement-Round (q, S_c, S_p, S_n, S_r)
1: **for** each point $p \in S_c$ **do**
2: **for** each point $p' \in S_p$ **do**
3: **if** OVC $(p', L_p, p) \neq IV$ and $dist(p', p) < dist(q, p)$ **then**
4: $S_c = S_c - \{p\}$; goto 1
5: **for** each node $N \in S_n$ **do**
6: **if** OVC $(N, L_p, p) = AV$ **then**
7: **if** $minmaxdist(N, p) < dist(q, p)$ **then**
8: $S_c = S_c - \{p\}$; goto 1
9: **for** each node $N \in S_n$ **do**
10: **if** OVC $(N, L_p, p) \neq IV$ and $mindist(N, p) < dist(q, p)$ **then**
11: add N to *p.toVisit*
12: **if** $p.toVisit = \emptyset$ **then**
13: $S_c = S_c - \{p\}; S_r = S_r \cup \{p\}$

If there are multiple nodes in *p.toVisit* for each p remaining in S_c, we can access all of them to invalidate the candidate objects. However, not all the accesses are necessary. Hence, we adopt an incremental approach to access *the lowest level* nodes first in order to achieve a better pruning. In our example shown in Fig. 7, the second refinement round starts with $S_c = \{p_1\}, S_p = \{p_3, p_7\}$ (i.e., points enclosed in N_2), $S_n = \emptyset$, and $S_r = \{p_4, p_8\}$. Point p_1 is eliminated as a false positive since p_3 is visible to p_1 and $dist(p_3, p_1) < dist(q, p_1)$ holds, and then, the VRNN algorithm terminates.

Notice that although VRNN-Refinement and Refinement-Round algorithms are similar to the TPL-Refinement and TPL-Refinement-Round algorithms proposed in [16], they integrate *object visibility check* during the refinement process.

4.3 Discussion

In a 2D space, like the existing SAA [14] and TPL [16] methods for RNN queries, the proposed VRNN algorithm does not require any preprocessing and can return exact result. However, the VRNN algorithm incurs a higher query cost as it considers the obstacle influence on the visibility of objects and it has to traverse not only the data set P but also the obstacle set O. In this section, we present the time complexity of the VRNN algorithm and prove its correctness.

The cost of R-tree traversal dominates the total overhead of the VRNN algorithm. We first derive the upper bound of the number of traversals on the R-trees T_p and T_o, respectively.

Lemma 2. *The VRNN algorithm traverses* T_p *at most once and* T_o *at most* $(|S_c| + 1)$ *times, with* S_c *representing the candidate set.*

Proof. As shown in Algorithm 3, VRNN-Filter algorithm only traverses T_p *once* to obtain a VRNN candidate set S_c. It then uses *half-plane property* and *visibility check* to prune false candidates and invokes the VRC algorithm *once* for *each candidate* $p \in S_c$ to find the obstacles affecting its visibility (line 6 in Algorithm 3). Moreover, VRNN-Filter also calls the VRC algorithm *once* to retrieve the obstacles that can affect the visibility of q (line 2 in Algorithm 3). Consequently, the VRNN algorithm traverses T_o at most $(|S_c| + 1)$ times. □

Let $|T_p|$ and $|T_o|$ be the tree size of T_p and T_o, respectively, and $|S_c|, |S_p|$, and $|S_n|$ be the cardinality of S_c, S_p, and S_n, respectively. We have the following theorems.

Theorem 1. *The time complexity of the VRNN algorithm is* $O(log|T_p| \times (|S_c| + 1)log|T_o| + |S_c|^2 + |S_c|(|S_p| + |S_n|))$.

Proof. The VRNN algorithm follows the filter-refinement framework. In the filter step, it takes $O(log|T_p| \times (|S_c| + 1)log|T_o|)$ for obtaining candidate set S_c; in the refinement step, it incurs $O(|S_c|^2 + |S_c|(|S_p| + |S_n|))$ to eliminate all the false hits. Therefore, the total time complexity of the VRNN algorithm is $O(log|T_p| \times (|S_c| + 1)log|T_o| + |S_c|^2 + |S_c|(|S_p| + |S_n|))$. □

Theorem 2. *The VRNN algorithm retrieves* exactly *the VRNNs of a given query point q, i.e., the algorithm has* no false negatives *and* no false positives.

Proof. First, the VRNN algorithm only prunes away those nonqualifying points or nodes in the filter step by using our proposed pruning strategy. Thus, no answer points are missed (i.e., no false negatives). Second, every candidate $p \in S_c$ is verified in the refinement step by comparing it with each data point retrieved during the filter step and each node that might contain VNNs of p, which ensures no false positives. □

5 Extensions

This section discusses three interesting variants of VRNN queries, namely, VRkNN, δ-VRkNN, and CVRkNN queries.

5.1 The VRkNN Search

A VRkNN query retrieves all the points in a data set whose VkNN sets include q, as formalized in Definition 3. Our solution to VRNN retrieval can be adapted to support VRkNN search. The detailed extensions are described as follows. First, the pruning strategy (presented in Section 4.1) can be extended to an arbitrary value of k. Assume a VRkNN query and a data set P with n ($\geq k$) data points $p_1, p_2, \ldots, p_n$. Let $D = \{\theta_1, \theta_2, \ldots, \theta_k\}$ be a subset of P. If a point/node fully falls into $\cap_{i=1}^{k} HP_{\theta i}(\theta_i, q)$ and is *all-visible* to each point in D, it must have k points (i.e., $\theta_1, \theta_2, \ldots, \theta_k$) closer to it than q. Consequently, it can be safely pruned away. On the other hand, if a point/node locates inside $\cap_{i=1}^{k} HP_{\theta i}(\theta_i, q)$ and is *partially visible/invisible* to *any subset* of D, it can become or contain a VRkNN of q, and thus, needs further examination.

Next, we explain how to extend the proposed algorithms for VRkNN query processing. To solve a VRkNN query, we also follow the filter-refinement framework. In particular, we find a set S_c of VRkNN candidates that contains all the actual answer points, and then, eliminate all the false candidates in S_c. The VRNN-Filter algorithm can be easily modified to support VRkNN retrieval, by integrating the aforementioned pruning strategy. Specifically, the filter step of VRkNN search first finds an initial candidate set S_c which contains the k data points *closest* to a given query point q, and meanwhile, *visible* to q. Then, the algorithm proceeds to retrieve candidates as well as to prune away all the nonqualifying data points and node MBRs that satisfy the aforementioned pruning condition. Data points and node MBRs discarded are kept in the refinement point set S_p and the refinement node set S_n, respectively. The filter phase finishes when all the nodes that may include candidates have been visited.

Algorithm 6. k-Refinement-Round Algorithm (k-Refinement-Round)

algorithm k-Refinement-Round (q, S_c, S_p, S_n, S_r)
for each point $p \in S_c$ **do**
  for each point $p' \in S_p$ **do**
    if OVC $(p', L_p, p) \neq IV$ and $dist(p', p) < dist(q, p)$ **then**
      $p.cnt = p.cnt + 1$
      if $p.cnt = k$ **then**
      $S_c = S_c - \{p\}$; goto 1
  for each node $N \in S_n$ **do**
    if OVC $(N, L_p, p) \neq IV$ and $mindist(N, p) < dist(q, p)$ **then**
      add N to *p.toVisit*
  if $p.toVisit = \emptyset$ **then**
    $S_c = S_c - \{p\}; S_r = S_r \cup \{p\}$

The VRNN-Refinement algorithm can be extended for VRkNN retrieval as well. Similarly, the refinement step of VRkNN search is also executed in rounds, which are shown in Algorithm 6. Different from Refinement-Round, a point $p \in S_c$ can be pruned only if there are *at least* k points *visible* to p within $dist(p, q)$. Hence, we associate a counter *p.cnt* (initially set to 0) with each point p during the processing. Every time, the algorithm finds a point p' that satisfies the following two conditions: 1) p' is *visible* to p and 2) $dist(p', p) < dist(q, p)$, the p's counter *p.cnt* is increased by one. Eventually, p can be removed as a *false hit* when $p.cnt = k$. The refinement phase terminates after all the points in S_c have been eliminated or verified. We omit the pseudocodes of the filter and main refinement algorithms for VRkNN search since they are very similar as VRNN-Filter and VRNN-Refinement presented in Algorithm 3 and Algorithm 4, respectively.

5.2 VRkNN Queries with Constraints

In some real applications, users might enforce some constraints (e.g., distance, spatial region, etc.) on VRkNN queries, and thus, we introduce the VRkNN query with *maximum visible distance* δ constraint (called δ-VRkNN search) and the VRkNN query with *constrained region CR* constraint (called CVRkNN search), respectively. Take the application *outdoor advertisement planning* described in Section 1 as an example. If it is assumed that customers pay *zero* attention to the billboard that is located 50 meters away, δ-VRkNN search with $\delta = 50$ is more suitable, compared with VRkNN search, as it takes the distance constraint into account. On the other hand, if *P&G* only targets for the customers located in certain area (e.g., the customers within a shopping mall), CVRkNN query with *constrained region CR* set to the specified shopping mall is more suitable. In this section, we explain how to extend the CVNN search algorithm to answer δ-VRkNN and/or CVRkNN queries.

Given a data set P, an obstacle set O, a query point q, a distance threshold δ, a constrained region *CR*, and an integer $k(\geq 1)$, 1) a δ-VRkNN query finds a set of points from P, denoted by δ-$VRkNN(q)$, such that $\forall p \in \delta\text{-}VRkNN(q), q \in VkNN(p)$, and $dist(p, q) \leq \delta$, i.e., $\delta\text{-}VRkNN(q) = \{p \in P | q \in VkNN(p) \wedge dist(p, q) \leq \delta\}$ and 2) a CVRkNN query returns a set of points from P, denoted by *CVRkNN(q)*, such that $\forall p \in$ *CVRkNN(q)*, $q \in$ *VkNN(p)*, and $p \cap$ *CR* $\neq \emptyset$ (i.e., p is inside *CR*), formally, $CVRkNN(q) = \{p \in P | q \in VkNN(p) \wedge p \cap CR \neq \emptyset\}$. It is important to note that in addition to the position of q and the distributions of data pints and obstacles, 1) the cardinality of δ-$VRkNN(q)$, i.e., $|\delta\text{-}VRkNN(q)|$, is dependent on the value of δ and 2) the cardinality of *CVRkNN(q)*, i.e., $|CVRkNN(q)|$, is dependent on the size and distribution of *CR*. As an example, a δ-VRNN ($k = 1$) query issued at point q is illustrated in Fig. 8a, where data set $P = \{p_1, p_2, p_3, p_4\}$, obstacle set $O = \{o_1, o_2\}$, and its distance constraint δ is highlighted in the figure. The final result of this query is *empty*, which is different from the result of VRNN search on the same data and obstacle sets (as shown in Fig. 1b) due to the δ constraint.

The proposed algorithms for VRNN search can be easily adjusted to support δ-VRNN and CVRNN queries, by integrating constrained conditions (i.e., distance threshold δ and constrained region *CR*) during the query processing. Moreover, we develop following heuristics to facilitate the search process. First, since the *search region* (SR) of δ-VRNN retrieval is bounded by δ (e.g., the shaded area in Fig. 8a representing the SR of the δ-VRNN query issued at q), 1) any obstacle that does not intersect SR cannot affect the visibility of objects evaluated currently, and can be pruned away safely

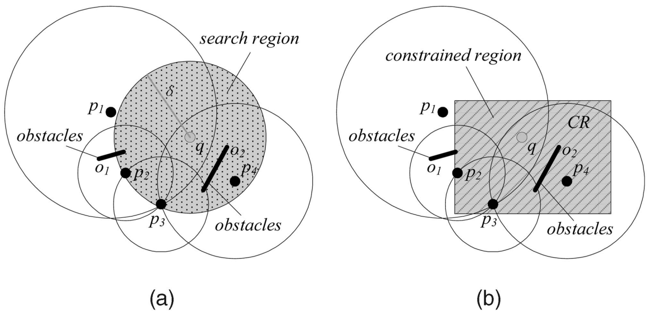


Fig. 8. Variations of VRNN queries with constraints. (a) δ-VRNN search. (b) CVRNN search.

and 2) any point/node that does not locate inside or cross SR can be directly excluded from the further consideration, because it cannot be/contain the final answer object. Second, as the final result of CVRNN search must satisfy the specified region constraint, 1) any obstacle that is outside *CR* can be discarded, since it cannot impact the visibility of objects evaluated currently and 2) any point/node that does not intersect *CR* can be directly excluded from the further examination, because it cannot become/contain the actual answer object. In addition, the algorithm can be extended to support δ-VRkNN and CVRkNN queries, which is similar to the extension for VRkNN search stated above.

6 Experimental Evaluation

In this section, we evaluate the efficiency and effectiveness of our proposed algorithms for VRNN query and its variants through experiments on both real and synthetic data sets. First, Section 6.1 describes the experimental settings, and then, Sections 6.2, 6.3, 6.4, and 6.5 report experimental results and our findings for VRNN, VRkNN, δ-VRkNN, and CVRkNN queries, respectively. All the algorithms were implemented in C++, and all the experiments were conducted on a PC with a Pentium IV 3.0 GHz CPU and 2GB RAM, running Microsoft Windows XP Professional Edition.

6.1 Experimental Setup

We deploy five real data sets,[7] which are summarized in Table 3. Synthetic data sets are created following the 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* data sets is generated uniformly along each dimension, and that of each point in *Zipf* data sets is generated according to zipf distribution with skew coefficient $\alpha = 0.8$. All the data sets are mapped to a [0, 10,000] $\times$ [0, 10,000] square. As VRNN search and its variations involve a data set P and an obstacle set O, we deploy five different data set combinations, namely, ***CR***, ***LL***, ***NL***, ***UL***, and ***ZL***, representing $(P, O) =$ (*Cities*, *Rivers*), (*LB*, *LA*), (*NA*, *LA*), (*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, and the obstacles in O are allowed to overlap each other.

All data and obstacle sets are indexed by R*-trees [2]. The disk page size is fixed to 1K bytes, such that the maximum node capacity equals 50 entries for dimensionality 2 and the number of nodes/pages for *LB*, *NA*, *LA*, *Cities*, and *Rivers* data sets equals 1,178, 9,145, 2,629, 118, and 432, respectively. Note that we choose a small page size to simulate practical scenarios where the cardinalities of the data and obstacle sets are much larger. The experiments investigate the performance of the proposed algorithms under a variety of parameters which are listed in Table 4. In each experiment, we vary only one parameter, while the others are fixed at their default values, and run 200 queries with their average performance reported. The query distribution follows the underlying data set distribution and the overall query cost is measured. Both the I/O overhead (by charging 10 ms per page fault, as in [16]) and CPU time contribute to the query cost. We assume that the server maintains a buffer with LRU as the cache replacement policy. Unless specifically stated, the size of buffer is 0, i.e., the I/O cost is determined by the number of node/page accesses.

7. *LB*, *NA*, and *LA* are available at http://www.maproom.psu.edu/dcw; and *Cities* and *Rivers* are available at http://www.rtreeportal.org.

TABLE 3
Description of Real Data Sets Used in Experiments

Dataset	Cardinality	Description
LB	58,945	2D point in Long Beach
NA	470,759	2D point in North America
LA	131,461	2D MBRs of streets in Los Angeles
Cities	5,922	2D cities (as point) in Greece
Rivers	21,645	2D MBRs of rivers in Greece

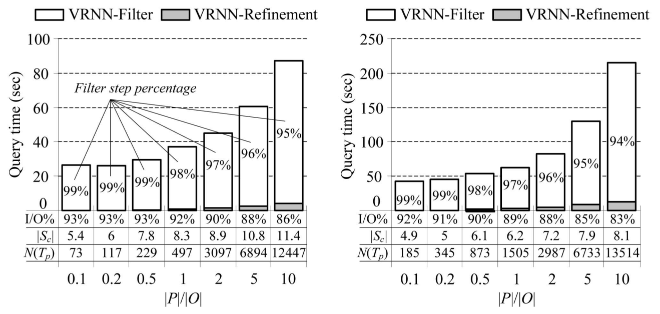
6.2 Results on VRNN Queries

The first set of experiments verifies the performance of the proposed VRNN algorithm for VRNN search. First, we study the effect of the $|P|/|O|$ ratio on the VRNN algorithm using two data set combinations (including *UL* and *ZL*). Fig. 9 plots the total query cost (in seconds) of the VRNN algorithm as a function of $|P|/|O|$, fixing $k = 1$. In Fig. 9, each result is broken into two components, corresponding to the filter step and the refinement step, respectively. The percentage inside the bar indicates the ratio of cost incurred in the filter step to that of the overall query cost. In addition, we show the percentage of I/O time in the entire query cost, denoted by I/O percent; the cardinality of the candidate set, denoted as $|S_c|$; and the number of node accesses on the data R-tree T_p, denoted by $N(T_p)$. For example, as shown in Fig. 9a, when $|P|/|O| = 1$, VRNN accesses 497 out of 2,629 nodes of T_p; its I/O cost contributes to 92 percent of overall query cost; and the candidate set S_c has 8.3 objects on average. The total query cost is around 37 seconds, while the filtering step takes 92 percent of the time.

It is observed that the filter step actually dominates the overall overhead (>90 percent), especially when the $|P|/|O|$ ratio is small (e.g., 0.1, 0.2). This is because: 1) the filter step of VRNN needs to traverse the obstacle R-tree $T_o(|S_c| + 1)$ times

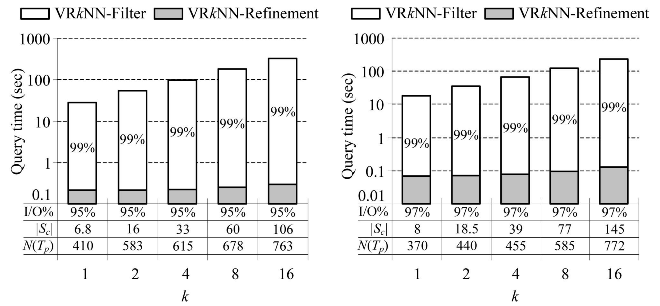
TABLE 4
Parameter Ranges and Default Values

Parameter	Range	Default
k	1, 2, 4, 8, 16	1, 4
$\lvert P\rvert/\lvert O\rvert$	0.1, 0.2, 0.5, 1, 2, 5, 10	1
buffer size (% of the tree size)	0, 10, 20, 30, 40, 50, 60	0
δ (% of the space width)	6, 12, 18, 24, 30	100
CR (% of full space)	10, 20, 30, 40, 50	100



(a) (b)

Fig. 9. VRNN cost versus $|P|/|O|$ ($k = 1, |O| = 131{,}461$). (a) *UL*. (b) *ZL*.



(a) (b)

Fig. 11. VRkNN cost versus k ($|O| = 131{,}461$). (a) *LL*. (b) *NL*.

(according to Lemma 2), incurring expensive I/O cost and a large number of visible region computation operations; 2) VRNN reuses all the points and nodes pruned from the filter step to perform candidate verification in the refinement step, and thus, duplicated accesses to the same points/nodes are avoided; and 3) most candidates in S_c are eliminated as false hits directly by other candidates in S_c or points/nodes maintained in the refinement set S_p or S_n, which does not cause any data access. The remaining candidates can be validated by visiting a limited number of additional nodes. This observation is also confirmed by the rest of experiments. In addition, we observe that the cost of VRNN demonstrates a stepwise behavior. Specifically, it increases slightly as $|P|/|O|$ changes from 0.1 to 1, but then ascends much faster as $|P|/|O|$ grows further. The reason behind is that as the density of data set P grows, the number of the candidates retrieved in the filter step increase as well, which results in more traversals of T_o, more visibility checks, and more candidate verifications.

Finally, we examine the performance of the VRNN algorithm in the presence of an LRU buffer, by fixing k to 1 and varying the buffer size from 0 to 60 percent of the tree size. 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 results under *UL* and *ZL* data set combinations are depicted in Fig. 10. The overall query cost is reduced as buffer size increases. In particular, as the buffer size enlarges, it is observed that the VRNN-Filter cost drops, whereas the VRNN-Refinement cost almost remains the same. This is because the filter step of VRNN requires traversing the obstacle R-tree $T_o(|S_c| + 1)$ times. Consequently, it may access the same nodes (e.g., the root node of T_o) multiple times, and hence, a buffer space can improve the search performance by keeping the nodes locally available.

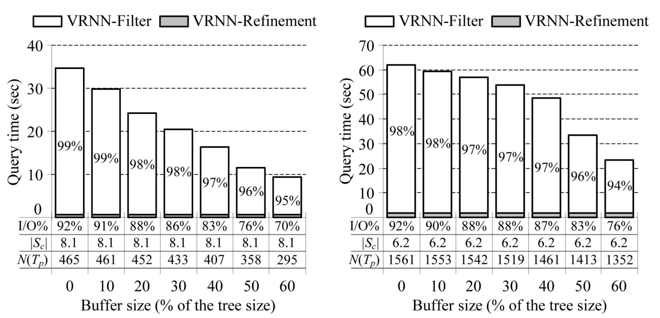
6.3 Results on VRkNN Queries

The second set of experiments evaluates the efficiency and effectiveness of VRkNN query processing algorithm. First, we inspect the impact of k value on the performance of the VRkNN algorithm, using *LL* and *NL* data set combinations. Fig. 11 illustrates the total query cost of the VRkNN algorithm with respect to k which varies from 1 to 16. As expected, the overhead of VRkNN grows with k, due to the significant increase in the cost of VRkNN-Filter. Notice that the number of candidates retrieved during the filter step increases almost linearly with k.

In the following experiments, we investigate the effect of different parameters, including the $|P|/|O|$ ratio and buffer size, on the performance of the VRkNN algorithm, with *UL* and *ZL* data set combinations. In Fig. 12, we show the efficiency of the algorithm for VRkNN queries, by fixing $k = 4$ and varying $|P|/|O|$ between 0.1 and 10. In Fig. 13, we plot the cost of the VRkNN algorithm as a function of the buffer size. As the observations are similar to those made from the VRNN retrieval, we save the detailed explanation due to the space limitation.

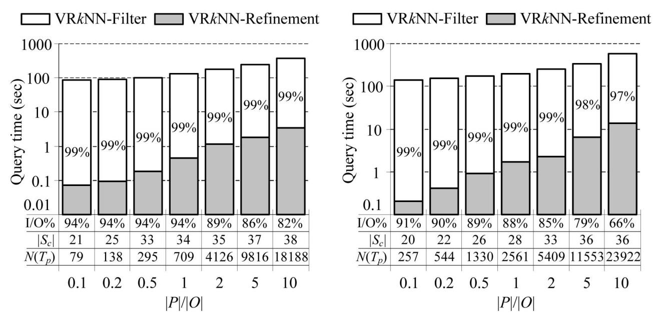
6.4 Results on δ-VRkNN Queries

The third set of experiments explores the influence of the maximal visible distance δ constraint on the efficiency of the



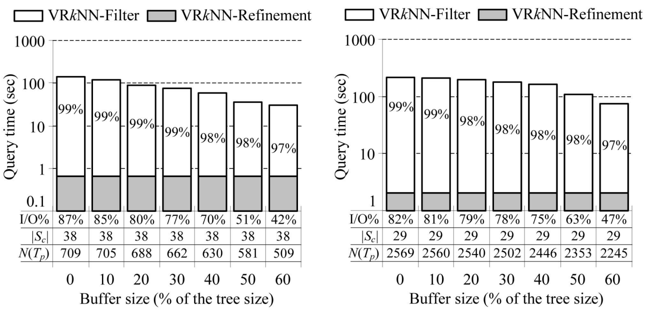
(a) (b)

Fig. 10. VRNN cost versus buffer size ($k = 1, |O| = 131{,}461$). (a) *UL* ($|P|/|O| = 1$). (b) *ZL* ($|P|/|O| = 1$).



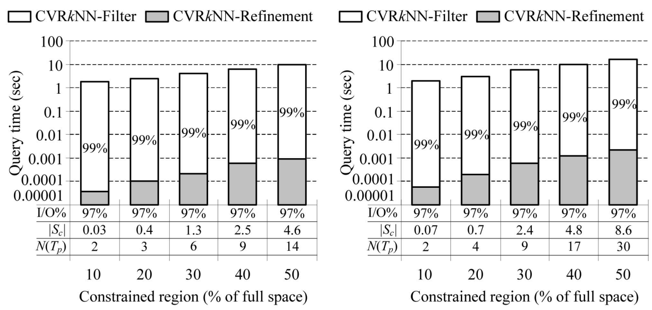
(a) (b)

Fig. 12. VRkNN cost versus $|P|/|O|$ ($k = 4, |O| = 131{,}461$). (a) *UL*. (b) *ZL*.



(a) (b)

Fig. 13. VRkNN cost versus buffer size ($k = 4, |O| = 131{,}461$). (a) *UL* ($|P|/|O| = 1$). (b) *ZL* ($|P|/|O| = 1$).



(a) (b)

Fig. 15. CVRkNN cost versus *CR* ($k = 4, |O| = 131{,}461$). (a) *LL*. (b) *NL*.

δ-VRkNN query processing algorithm. We fix k at 4 and change δ values from 6 to 30 percent of the side length of the search space. Fig. 14 shows the overall query cost of the δ-VRkNN search algorithm with respect to δ for *LL* and *NL* data set combinations. Obviously, δ has a direct impact on the performance of δ-VRkNN retrieval, since it controls the size of the search region. In particular, the cost of the algorithm increases gradually as δ grows. This is because the number of candidates retrieved in the filter step ascends with the growth of δ.

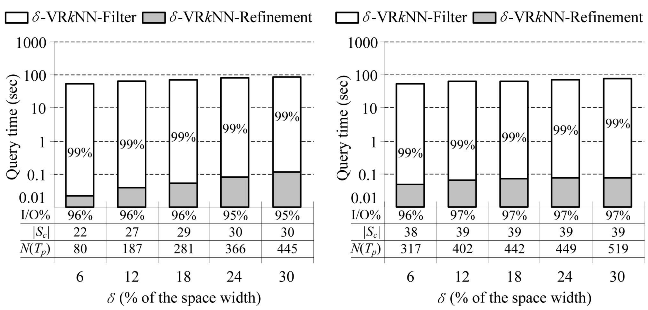
6.5 Results on CVRkNN Queries

The last set of experiments investigates the effect of the constrained region *CR* size on the performance of CVRkNN query processing algorithm. We deploy real data sets, i.e., *LL* and *NL* data set combinations, fix k to 4, vary the size of *CR* from 10 to 50 percent of the whole data space, and present all the experimental results in Fig. 15. As expected, the cost of the algorithm increases with the growth of *CR*. The reason behind is that as constrained region grows, the size of search space enlarges and the number of candidates obtained in the filter step increases, which leads to more traversals of the obstacle R-tree T_o, more visibility checks, and more candidate examinations.

7 Conclusions

In this paper, we identify and solve a novel type of reverse nearest neighbor queries, namely, VRNN search. Although both RNN search and VNN search have been studied, there is no previous work that considers both the visibility and the reversed spatial proximity relationship between objects. On the other hand, VRNN retrieval is useful in many decision support applications involving spatial data and physical obstacles. Consequently, we propose an efficient algorithm for VRNN query processing, assuming that both the data set P and the obstacle set O are indexed by R-trees. We employ half-plane property and visibility check to prune the search space, analyze the cost of the proposed VRNN algorithm, and prove its correctness. In addition, we extend our techniques to tackle three interesting VRNN query variations, including VRkNN, δ-VRkNN, and CVRkNN queries. An extensive experimental evaluation with both real and synthetic data sets has been conducted which demonstrates the performance of our proposed algorithms for handling VRNN search and its variants, under various experimental settings.

This work motivates several directions for future work. First, we plan to study the VRkNN query with respect to multiple query points. Second, we focus only on a 2D space in this paper, and hence, how to extend our solution to a multidimensional space is a challenging topic that is worth further studying. Finally, it would be particularly interesting to develop analytical models for estimating the query cost of VRNN search and its variants. Such models will not only facilitate query optimization, but may also reveal new problem characteristics that could lead to even better algorithms.



(a) (b)

Fig. 14. δ-VRkNN cost versus δ ($k = 4, |O| = 131{,}461$). (a) *LL*. (b) *NL*.

Acknowledgments

This research is partly supported by the Office of Research, Singapore Management University. Wang-Chien Lee was supported in part by the US National Science Foundation under Grant IIS-0534343 and Grant CNS-0626709.

References

[1] T. Asano, S.K. Ghosh, and T.C. Shermer, "Visibility in the Plane," *Handbook of Computation Geometry,* J.-R. Sack and J. Urrutia, eds., pp. 829-876, 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," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 322-331, 1990.

[3] Y. Gao, B. Zheng, W.-C. Lee, and G. Chen, "Continuous Visible Nearest Neighbor Queries," *Proc. Int'l Conf. Extending Database Technology,* pp. 144-155, 2009.

[4] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 47-57, 1984.

[5] M. Kofler, M. Gervautz, and M. Gruber, "R-Trees for Organizing and Visualizing 3D GIS Databases," *J. Visualization and Computer Animation,* vol. 11, no. 3, pp. 129-143, July 2000.

[6] F. Korn and S. Muthukrishnan, "Influence Sets Based on Reverse Nearest Neighbor Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 201-212, 2000.

[7] F. Korn, S. Muthukrishnan, and D. Srivastava, "Reverse Nearest Neighbor Aggregates over Data Streams," *Proc. Int'l Conf. Very Large Data Bases,* pp. 814-825, 2002.

[8] K.C.K. Lee, B. Zheng, and W.-C. Lee, "Ranked Reverse Nearest Neighbor Search," *IEEE Trans. Knowledge and Data Eng.,* vol. 20, no. 7, pp. 894-910, July 2008.

[9] K.-I. Lin, M. Nolen, and C. Yang, "Applying Bulk Insertion Techniques for Dynamic Reverse Nearest Neighbor Problems," *Proc. Int'l Database Eng. and Applications Symp.,* pp. 290-297, 2003.

[10] S. Nutanong, E. Tanin, and R. Zhang, "Visible Nearest Neighbor Queries," *Proc. Int'l Conf. Database Systems for Advanced Applications,* pp. 876-883, 2007.

[11] L. Shou, C. Chionh, Y. Ruan, Z. Huang, and K.L. Tan, "Walking through a Very Large Virtual Environment in Real-Time," *Proc. Int'l Conf. Very Large Data Bases,* pp. 401-410, 2001.

[12] L. Shou, Z. Huang, and K.L. Tan, "HDoV-Tree: The Structure, The Storage, The Speed," *Proc. Int'l Conf. Data Eng.,* pp. 557-568, 2003.

[13] A. Singh, H. Ferhatosmanoglu, and A. Tosun, "High Dimensional Reverse Nearest Neighbor Queries," *Proc. Conf. Information and Knowledge Management,* pp. 91-98, 2003.

[14] I. Stanoi, D. Agrawal, and A. El Abbadi, "Reverse Nearest Neighbor Queries for Dynamic Databases," *Proc. ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery,* pp. 44-53, 2000.

[15] I. Stanoi, M. Riedewald, D. Agrawal, and A. El Abbadi, "Discovery of Influence Sets in Frequently Updated Databases," *Proc. Int'l Conf. Very Large Data Bases,* pp. 99-108, 2001.

[16] Y. Tao, D. Papadias, and X. Lian, "Reverse *k*NN Search in Arbitrary Dimensionality," *Proc. Int'l Conf. Very Large Data Bases,* pp. 744-755, 2004.

[17] A.K.H. Tung, J. Hou, and J. Han, "Spatial Clustering in the Presence of Obstacles," *Proc. Int'l Conf. Data Eng.,* pp. 359-367, 2001.

[18] C. Xia, W. Hsu, and M.-L. Lee, "ER*k*NN: Efficient Reverse *k*-Nearest Neighbors Retrieval with Local *k*NN-Distance Estimation," *Proc. Conf. Information and Knowledge Management,* pp. 533-540, 2005.

[19] C. Yang and K.-I. Lin, "An Index Structure for Efficient Reverse Nearest Neighbor Queries," *Proc. Int'l Conf. Data Eng.,* pp. 485-492, 2001.

[20] J. Zhang, D. Papadias, K. Mouratidis, and M. Zhu, "Spatial Queries in the Presence of Obstacles," *Proc. Int'l Conf. Extending Database Technology,* pp. 366-384, 2004.

Yunjun Gao received the master's degree in computer science from Yunnan University, China, in 2005, and the PhD degree in computer science from Zhejiang University, China, in 2008. He is currently a postdoctoral research fellow in the School of Information Systems, Singapore Management University, Singapore. His research interests include spatial databases, spatiotemporal databases, mobile and pervasive computing, and geographic information systems. He is a member of the ACM, the ACM SIGMOD, and the IEEE.

Baihua Zheng received the bachelor's degree in computer science from Zhejiang University, China, in 1999, and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong, in 2003. She is currently an assistant professor in the School of Information Systems, Singapore Management University, Singapore. Her research interests include mobile and pervasive computing and spatial databases. She is a member of the ACM and the IEEE.

Gencai Chen is a professor in the College of Computer Science, Zhejiang University, China. He was a visiting scholar in the Department of Computer Science, State University of New York at Buffalo, from 1987 to 1988, and the winner of the special allowance, conferred by the State Council of China in 1997. He is currently a vice dean of the College of Computer Science, the director of the Computer Application Engineering Center, and the vice director of the Software Research Institute, Zhejiang University. His research interests include database systems, artificial intelligence, and CSCW.

Wang-Chien Lee received the BS degree from the National Chiao Tung University, Taiwan, the MS degree from the Indiana University, and the PhD degree from the Ohio State University. He is an associate professor in the Department of Computer Science and Engineering, Pennsylvania State University. Prior to joining Pennsylvania State University, he was a principal member of the technical staff at Verizon/GTE Laboratories, Inc. He leads the Pervasive Data Access Research Group at Pennsylvania State University to pursue cross-area research in database systems, pervasive/mobile computing, and networking. He is particularly interested in developing data management techniques for supporting complex queries in a wide spectrum of networking and mobile environments. Meanwhile, he has worked on XML, security, information integration/retrieval, and object-oriented databases. He has published more than 160 technical papers on these topics. He has been active in various IEEE/ACM conferences and has given tutorials for many major conferences. He was the founding program cochair of MDM. He has served as a guest editor for several journal special issues on mobile database-related topics. He has also served as the TPC chairs or the general chairs for a number of conferences. He is a member of the IEEE and the ACM.

Ken C.K. Lee received the BA and MPhil degrees in computing from Hong Kong Polytechnic University, Hong Kong. He is currently working toward the PhD degree at the Department of Computer Science and Engineering, Pennsylvania State University. His research interests include spatial database, mobile and pervasive computing, and location-based services.

Qing Li is a professor in the Department of Computer Science, City University of Hong Kong, where he joined as a faculty member in September 1998. Before that, he has taught at the Hong Kong Polytechnic University, The Hong Kong University of Science and Technology, and The Australian National University, Canberra, Australia. He is a guest professor at the University of Science and Technology of China, Hefei, China, Zhejiang University, Hangzhou, China; a visiting professor at the Institute of Computing Technology (Knowledge Grid), Chinese Academy of Science, Beijing, China; and an adjunct professor of the Hunan University, Changsha, China. His research interests include object modeling, multimedia databases, and Web services. He is a senior member of IEEE, a member of ACM SIGMOD and IEEE Technical Committee on Data Engineering. He is the chairperson of the Hong Kong Web Society and also served/is serving as an executive committee member of the IEEE-Hong Kong Computer Chapter and the ACM Hong Kong Chapter. He serves as a councilor of the Database Society of Chinese Computer Federation, a councilor of the Computer Animation and Digital Entertainment Chapter of Chinese Computer Imaging and Graphics Society, and is a steering committee member of DASFAA, ICWL, and the International WISE Society.