Visible Reverse k-Nearest Neighbor Queries

Yunjun Gao[1], Baihua Zheng[1], Gencai Chen[2], Wang-Chien Lee[3], Ken C. K. Lee[3], Qing Li[4]

[1] *Singapore Management University, Singapore* {yjgao, bhzheng}@smu.edu.sg
[2] *Zhejiang University, P. R. China* chengc@zju.edu.cn
[3] *Pennsylvania State University, USA* {wlee, cklee}@cse.psu.edu
[4] *City University of Hong Kong, P. R. China* itqli@cityu.edu.hk

***Abstract*— *Reverse nearest neighbor* (RNN) queries have a broad application base such as decision support, profile-based marketing, resource allocation, data mining, 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, blindages, etc.), and their presence may affect the visibility/distance between two objects. In this paper, we introduce a novel variant of RNN queries, namely *visible reverse nearest neighbor* (VRNN) search, which considers the obstacle influence on the visibility of objects. Given a data set *P*, an obstacle set *O*, and a query point *q*, a VRNN query retrieves the points in *P* that have *q* as their nearest neighbor and are *visible* to *q*. We propose an efficient algorithm for VRNN query processing, assuming that both *P* and *O* are indexed by R-trees. Our methods do not require any pre-processing, and employ *half-plane property* and *visibility check* to prune the search space. In addition, we extend our solution to tackle the *visible reverse k-nearest neighbor* (VR*k*NN) search, which finds the points in *P* that have *q* as one of their *k* nearest neighbors and are *visible* to *q*. Extensive experiments on synthetic and real datasets have been conducted which demonstrate the efficiency and effectiveness of our proposed algorithms.**

I. INTRODUCTION

Reverse nearest neighbor (RNN) search has received considerable attention from the database research community in the past few yeas, due to its importance in a wide spectrum of applications such as decision support [1], profile-based marketing [1], [2], resource allocation [1], [3], data mining [4], 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* (R*k*NN) search, which returns the points in *P* whose *k* nearest neighbors (NNs) include *q*. Formally, $RkNN(q) = \{p \in P \mid q \in kNN(p)\}$, where $RkNN(q)$ and $kNN(p)$ are the set of reverse *k* nearest neighbors of query point *q* and the set of *k* nearest neighbors of point *p*, respectively. Figure 1(a) illustrates an example with four data points, labelled as p_1, p_2, p_3, p_4, in a 2D space. Each point p_i ($1 \le i \le 4$) is associated with a circle centered at p_i and having $dist(p_i, NN(p_i))$[1] as its radius. In other words, 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 nearest neighbor to 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 inside the circle $cir(p_4,NN(p_4))$. It is worth noting the asymmetric NN relationship, i.e., $p \in kNN(q)$ does not necessary imply $q \in kNN(p)$ (i.e., $p \in RkNN(q)$). In Figure 1(a), for instance, we notice that $NN(p_4) = p_3$, but $NN(p_3) = p_2$.

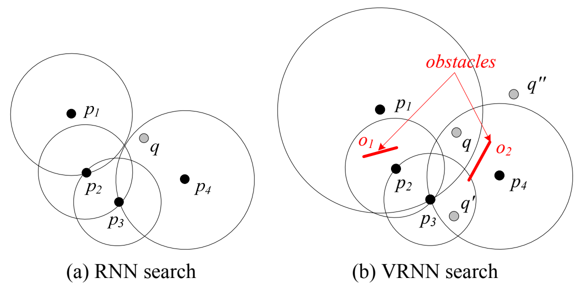


Fig. 1 Example of RNN and VRNN queries

A. Motivation

There are many RNN/R*k*NN query algorithms that have been proposed in the database literature. Basically, they can be classified into three categories: (i) pre-computation based algorithms [1], [3], [5]; (ii) dynamic algorithms [2], [6], [7]; and (iii) algorithms for various RNN/R*k*NN query variants [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. However, none of the existing work on RNN/R*k*NN search has considered physical obstacles (e.g., buildings, blindages, etc.) that exist in the real world. The presence of obstacles may have a significant impact on the visibility/distance between two objects, and hence affects the result of RNN/R*k*NN 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. These include (i) *obstructed nearest neighbor* (ONN) query [18], [19], that is to return the *k* (≥ 1) points in *P* that have the smallest *obstructed distances*[2] to *q*; (ii) *visible k-nearest neighbor* (V*k*NN) search [20], that is to retrieve the *k* nearest points that are *visible* to *q*; and (iii) *clustering spatial data in the presence of obstacles* [21], [22], [23], that is to divide a set of 2D data points into smaller homogeneous groups (i.e., clusters), considering the influence of obstacles. However, to the best of our knowledge, this paper is the first work to consider the obstacles in the context of RNN/R*k*NN search.

[1] Without loss of generality, $dist(p_1, p_2)$ is a function to return the Euclidean distance between two points p_1 and p_2.

[2] The obstructed distance between two points $p_1, p_2 \in P$ is defined as the length of the shortest path that connects p_1 to p_2 without crossing any obstacle from *O*.

B. Contributions

In this paper, we introduce a novel form of RNN queries, namely *visible reverse nearest neighbor* (VRNN) search, which considers the obstacle influence on the visibility of objects. Given a data set P, an obstacle set O, and a query point q, a VRNN query retrieves all the points in P that have q as their NN and are *visible* to q. In other words, there is no other point $p' \in P$ such that p' is visible to p and $dist(p', p) < dist(q, p)$. A natural generalization is the *visible reverse k-nearest neighbor* (VRkNN) retrieval, which finds all the points $p \in P$ that have q as one of their k NNs and are *visible* to q. Take a VRNN query issued at point q as an example (as shown in Figure 1(b)), it returns $\{p_1\}$ as the result set which is different from the result of RNN query.

We focus this paper on VRNN search, not only because the problem has not been studied in the literature but also because it has a large application base. Some of the example applications are listed as follows.

Selection of Promotion Sites. Suppose *Yao Restaurant & Bar* decides to open a new restaurant *YEEHA* in Shanghai, and wants to distribute coupons to its potential customers for business promotions. In order to guarantee the effectiveness of the promotion, it locates all the office buildings and residential buildings that have YEEHA as their top-3 restaurants (in terms of spatial proximity) and identifies customers working or staying in those buildings as its high potential customers. Although RNN search can be applied here to find all the buildings that have YEEHA as one of their 3 nearest restaurants, VRNN considers the visibility of YEEHA (and other restaurants) affected by obstacles such as buildings and malls. VRNN can identify all the buildings that have YEEHA as their 3 visible nearest restaurants. As the coupons are sent to those customers who do not know YEEHA, the visibility plays an important role and it is more likely that those customers who can see YEEHA directly will visit it and try.

Outdoor Advertisement Planning. Suppose *P&G* decides to post advertisements in billboards to promote a new shampoo. In order to encourage customers to try this new product, they decide to distribute samples near billboards as well. Due to the high cost of sample distribution, only those locations that can reach a big pull of potential customers are considered. Ideally, the more people can view the billboards, the more effective the promotion will be. Consequently, VRNN/VRkNN searches can be conducted to compare the optimality of any two locations q_1 and q_2 in terms of the base of potential customers they can reach. Suppose every customer only pays attention to the billboard located closest to him/her, VRNN(q_1)/VRNN(q_2) can be issued. It takes inputs of a set P of office buildings/residential buildings/shopping malls that represents the potential customer base, a set O of obstacles (e.g., buildings) and q_1/q_2 as a query point and returns the customers that will take a look at billboard located at q_1/q_2. The one with more customers is better.

A naive solution to process VRkNN ($k \geq 1$) queries is to find a set of points $p \in P$ (namely dataset S_q) that are visible to a given query point q, perform VkNN search on each of them, and then return those $p \in S_q$ with $q \in$ VkNN(p). However, this approach is very inefficient as it needs to browse the dataset P and obstacle set O multiple times, resulting in high I/O cost and long CPU time, especially when $|\text{VR}k\text{NN}(q)| << |S_q|$[3]. The poor performance of this naive approach will be further demonstrated by our experimental results to be presented in Section VI.

In this paper, we propose an efficient search algorithm for VRNN retrieval, assuming that both the data set and the set of obstacles are indexed by R-trees [24]. Our solution follows a filter-refinement framework, and requires *zero* pre-processing. Specifically, a set of candidate objects (i.e., a superset of the actual query result) is retrieved in the filter step and gets refined in the refinement step, with two steps integrated into a single R-tree traversal. As the size of the candidate objects has a direct impact on the search efficiency, we employ *half-plane properties* (as [7]) and *visibility check* to prune the search space. In addition, the search algorithm is general and can be easily extended to support VRkNN search. In brief, the key contributions of this paper can be summarized as follows:

- We introduce and formalize VRNN query, 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 propose an efficient search algorithm for VRNN (and VRkNN) queries, analyze the cost of VRNN algorithm, and prove its correctness.
- We conduct extensive experiments using both synthetic and real datasets to evaluate the performance of our proposed algorithms in terms of efficiency and effectiveness.

The rest of this paper is organized as follows. Section II formalizes VRNN query and reviews related work. Section III discusses how to decide whether an object is visible to q in the presence of obstacles, and proposes the concept of *visible region* to improve the performance. Section IV proposes an efficient search algorithm for VRNN query processing and conducts analytical analysis to proof its correctness. Section V extends our solution to deal with the VRkNN search. Extensive experimental evaluations and our findings are reported in Section VI. Finally, Section VII concludes the paper with some directions for future work.

II. BACKGROUND

In this section, we present the formal definition of VRkNN query, reveal its properties, and then briefly review some related work, including RNN/RkNN query and visibility queries. Table I summarizes the notations to be used in the rest of this paper.

A. Problem Statement

Given a data set P, an obstacle set O, and a query point q, visible k nearest neighbor and visible reverse k nearest neighbor search are defined in Definition 2 and Definition 3, respectively, with the visibility defined in Definition 1.

Definition 1: **Visibility.** Given a data set P and an obstacle set O, points p and p' ($\in P$) are *visible* to each other iff the

[3]Without loss of generality, $|P|$ represents the cardinality of a set P.

straight line connecting p and p' does not cut through any obstacle o, i.e., $\forall\ o \in O,\ \overline{pp'} \cap o = \varnothing$. □

TABLE I
FREQUENTLY USED SYMBOLS

Notation	Description
p, P	A data point p and the data point set P, with $p \in P$
o, O	An obstacle o and the obstacle set O, with $o \in O$
T_p, T_o	The R-tree on P, and the R-tree on O
q	A query point
e	An entry (point or MBR node) in an R-tree
$RkNN(q)$	Result set of a RkNN query issued at q
$VkNN(q)$	Result set of a VkNN query issued at point q
$VRkNN(q)$	Result set of a VRkNN query issued at point q

Definition 2: **Visible *k* Nearest Neighbor (V*k*NN).** Given a data set P, an obstacle set O, a query point q, and an integer k, the *visible k nearest neighbor* (VkNN) of q retrieves a set of points, denoted by $VkNN(q)$, that satisfy following conditions: (i) $\forall p \in VkNN(q)$ is visible to q; (ii) $|VkNN(q)| = k$; and (iii) $\forall p' \in P - VkNN(q)$ and $\forall p \in VkNN(q)$, if p' is visible to q, $dist(p, q) \le dist(p', q)$. □

Definition 3: **Visible Reverse *k* Nearest Neighbor (VR*k*NN) Query.** Given a data set P, an obstacle set O, a query point q, and a positive integer k, a *visible reverse k-nearest neighbor* (VRkNN) query finds a set of points $VRkNN(q) \subseteq P$, such that $\forall\ p \in VRkNN(q)$, $q \in VkNN(p)$, i.e., $VRkNN(q) = \{p \in P \mid q \in VkNN(p)\}$. □

Property 1: VRkNNs might not be localized to the neighborhood of the query point. □

Property 2: Given a query point q, the cardinality of q's VRkNNs, denoted by $|VRkNN(q)|$, varies which is affected by the position of the query point and the distributions of data points and obstacles. □

Property 3: $p \in VkNN(q)$ does not imply $p \in VRkNN(q)$. □

Some of the important properties of VRkNN query that will be utilized to process VRkNN search are detailed in Property 1, Property 2, and Property 3, respectively. In order to facilitate the understanding, we illustrate those properties using the example depicted in Figure 1(b). First, point p_1 is farthest from the specified query point q compared with other points, but it is still an answer object to the query $VRNN\ (q)$. In contrast, point p_2 that is closer to q than p_1 is not included in $VRNN\ (q)$. Second, for a same k, VRkNN queries issued at different locations obtain different answers with various number of answer points. For example, $|VRNN(q)| = |\{p_1\}| = 1$, $|VRNN(q')| = |\{p_3, p_4\}| = 2$, and $|VRNN(q'')| = |\varnothing| = 0$. Third, the relationship of visible nearest neighbour is asymmetric. For example, $VNN(q) = p_2$, but $VRNN(q) = \{p_1\}$ that does not includes p_2.

B. Related Work

1) Algorithms for RNN/RkNN Search: Since the concept of RNN was first introduced by Korn and Muthukrishnan [1], many algorithms have been proposed which can be clustered into three categories. The first category is based on *pre-computation* [1], [3], [5]. For each point p, it pre-computes the distance between p and 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 answer set, i.e., $RNN(q) = \{p \in P \mid q \in cir(p, NN(p))\}$. To facilitate the examination, all the vicinity circles are indexed with RNN-tree [1] or R*d*NN-tree [3]. Approaches of this category mainly have two shortcomings. First, the index construction cost and update overhead is very expensive. To tackle this issue, bulk insertion in the R*d*NN-tree has been proposed [25]. Second, although these methods can be extended to deal with 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 proposed [26].

The second category does not rely on pre-computation but adopts a filter-refinement framework [2], [6], [7]. In the filter step, the space is pruned according to defined heuristics and a set of candidate objects are retrieved from the dataset. 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 6 equal regions, such that the NNs of q found in each region are the only candidates of RNN query [2]. Thus, in the filter step, 6 *constrained* NN queries [27] 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 search in a 2D space. However, the number of candidates grows exponentially with the increase of the search space dimensionality which implies the search efficiency can only be guaranteed in a low-dimensional space. To process RNN queries in a high-dimensional space, an approximated algorithm is proposed [6]. 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. However, the accuracy and performance of this algorithm is highly dependent on m. The larger m is, the more candidates are identified. Consequently, it is more likely that a complete result set is returned but with a higher processing cost. A small m favours the efficiency but it may incur many *false misses* (points that are RkNNs but missed from the final query result set).

In order to conduct exact RNN search, an efficient algorithm namely TPL is proposed [7]. TPL exploits a *half-plane property* in space to locate RkNN candidates. Applying the *best-first* traversal paradigm [28], 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 point p' or minimum bounding rectangle (MBR) N falling inside $HP_p\ (p, q)$ must have p closer to it than q. As depicted in Figure 2, the bisector $\perp(p_3, q)$ partitions the space into two half-planes. As point p_1 falls inside the half-plane HP_q (p, q), it is closer to q than to p_3. In other words, 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 inside at least k HP_p (p, q) half-planes, it cannot be an 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 Figure 2, points p_1, p_3, and p_4 are identified as the RNN candidates in the filter step, while points 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 inside HP_{p3} $(p_3, q) \cap HP_{p4}$ (p_4, q)) are filtered out. Later, in the refinement step, TPL removes 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 algorithm for VRNN and VRkNN queries employs *half-plane property* and *visibility check* to identify result candidates and prune the search space.

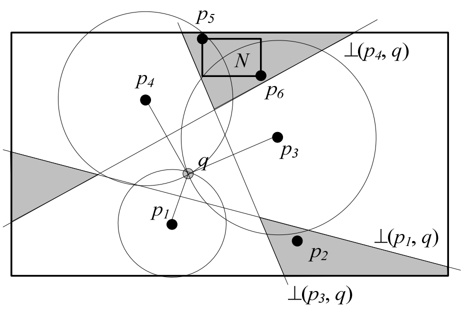


Fig. 2 Example of TPL algorithm

Algorithms belonging to the third category are to process various RNN/RkNN query variants, like bichromatic RNN queries[8], [29], aggregate RNN queries over data stream [9], RkNN query over moving objects with fixed velocities [14], [29], RkNN queries in the context of large graphs and ad-hoc subspaces [10], [11], RkNN query processing in metric spaces [12], [13], continuous RNN/RkNN monitoring [15], [16], [31], and ranked RNN search [17].

2) Visibility Query: 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 [32]. However, there are only a few works on visibility queries in the database community [33], [34], [35]. The main idea is to employ various indexing structures (e.g., LoD-R-tree [33], HDoV-tree [35], etc.) to process visibility queries in visualization systems. These specialized access methods are designed only for the purpose of visualization and contain no distance information. They are not capable of supporting efficient VRkNN query processing. Recently, the visible k-nearest neighbor (VkNN) search has been investigated, where the goal is to retrieve the k nearest objects that are visible to a given query point as mentioned earlier [20].

III. Preliminaries

As VRNN search considers the influence of obstacles in terms of visibility, all the objects that are invisible to q for sure will not be the result. Consequently, an essential issue we have to address is how to determine whether an object is visible to q. A naive 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 Figure 3, a visible region is in irregular shape and we can use vertex to present it. However, it might not be so straightforward to determine whether an object is inside an irregular polygon. Alternative, we propose to use *obstacle lines*, as defined in Definition 4.

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

Suppose the rectangle o as shown in Figure 3 is an obstacle, its corresponding obstacle line is ol_o. As blocked by ol_o, the shadowed area 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 define the *angular bound* and the *distance bound* of an obstacle line in Definition 5 and Definition 6 respectively, to facilitate the visibility checking.

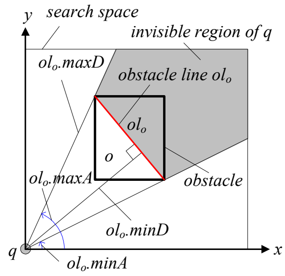


Fig. 3 An example obstacle line, and its angular and distance bounds

Definition 5: **Angular bound of an obstacle line.** 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 by $[ol_o.minA, ol_o.maxA]$ where $ol_o.minA$ and $ol_o.maxA$ are respectively the minimum angle and the maximum angle of ol_o, and $ol_o.minA \le ol_o.maxA$ (see Figure 3). If q is located inside o, the angular bound of ol_o w.r.t. q is set to $[0, 2\pi]$. □

Note that Definition 5 does not hold when ol_o intersects with the positive x-axis in the search space. In this case, we partition ol_o horizontally along the x-axis into ol_{o1} and ol_{o2} such that Definition 5 remains valid for both ol_{o1} and ol_{o2}. Given two obstacles, the intersection of their angular bounds has a direct impact on whether they will affect each other's visibility w.r.t. q, as listed in Property 4.

[4]Although an obstacle o may be an arbitrary convex polygon (e.g., triangle, pentagon, etc.), we assume that o is a rectangle in this paper.

Property 4: Given two obstacles o and o', if their angular bounds are *disjoint*, i.e., $[ol_o.minA, ol_o.maxA] \cap [ol_{o'}.minA, ol_{o'}.maxA] = \emptyset$, then they will not affect each other's visibility w.r.t. q. □

Definition 6: **Distance bound of an obstacle line.** The *distance bound* of o's obstacle line (i.e., ol_o) w.r.t. q is denoted by $[ol_o.minD, ol_o.maxD]$ where $ol_o.minD$ and $ol_o.maxD$ are the minimal distance and maximal distance from q to ol_o, respectively (see Figure 3). □

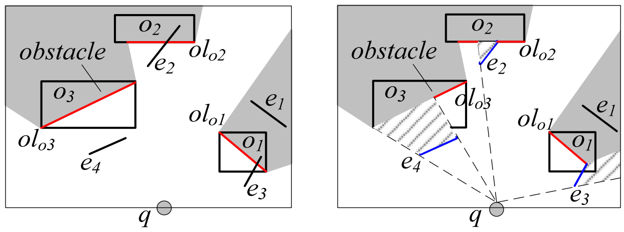
Without any obstacle, the visible region for q (i.e., VR_q) is the entire search space. As obstacles are visited, VR_q gets shrunk. Consequently, an issue we have to solve is how to decide whether a new obstacle might contribute to the formation of VR_q. Although we assume the obstacle is in rectangular shape, we first explain the test based on a line segment (or edges) and then extend the algorithm for rectangles.

Algorithm 1 Edge Visibility Check Algorithm (EVC)

```
algorithm EVC (q, L_q, e, boolean)
flag = invisible
A_min = e.minA; A_max = e.maxA
for each obstacle line l ∈ L_q do
  if l.maxA ≤ A_min then
    continue
  else if l.maxA > A_min and l.minA ≤ A_min then
    e' = edge(e, [A_min, MIN(l.maxA, A_max)])   // get edge
    l' = edge(l, [A_min, MIN(l.maxA, A_max)])
    f = CheckEdges (e', l', q, L_q, boolean)
  else if l.minA ≤ A_max and l.minA > A_min then
    e' = edge(e, [l.minA, MIN(l.maxA, A_max)])
    l' = edge(l, [l.minA, MIN(l.maxA, A_max)])
    f = CheckEdges (e', l', q, L_q, boolean)
  else    // l.minA ≥ A_max
    break
  if flag = invisible then flag = f
return flag

function CheckEdges (l_N, l, q, L_q)
l_N = [s, e]; l = [s', e']
if l.maxD ≤ l_N.minD then
  return IV    // invisible
else if l.minD ≥ l_N.maxD then
  if (boolean = TURE) then L_q = L_q − l + l_N
  return AV    // all-visible
else    // l_N intersects with l
  p = intersection(l_N, l)    // get intersection point
  if dist(q, s) < dist(q, e) then
    if (boolean = TURE) then L_q = L_q − [p, e] + [s, p]
  else
    if (boolean = TURE) then L_q = L_q − [p, s] + [e, p]
  return PV    // part-visible
```

Algorithm 1 lists the pseudo-code of the *Edge Visibility Check* algorithm (EVC), with set L_q keeping all the obstacles found so far that affect the visibility of a given query point q. Based on the angular property of obstacle (i.e., Property 4), a given obstacle o might affect those obstacles with angular bounds overlapping with o's but definitely not the test. Consequently, EVC visits the obstacle lines in L_q according to the ascending order of their minimal angle. An example is illustrated in Figure 4, with $L_q = \{o_1, o_2, o_3\}$, and e_2 being the edge we are going to evaluate. According to the angular bounds of l ($\in L_q$) and e_2, there are three cases: (i) case 1: $l.maxA \le e_2.minA$ (e.g., $l = ol_{o1}$), indicating that e_2 will not affect the visibility of l w.r.t q according to Property 4; (ii) case 2: $[l.minA, l.maxA] \cap [e_2.minA, e_2.maxA] \ne \emptyset$ (e.g., $l = ol_{o2}$), meaning that a detailed examination is necessary as e_2 is very likely to affect the visibility of l w.r.t. of q; and (iii) case 3: $l.minA \ge e_2.maxA$ (e.g., $l = ol_{o3}$), indicating that l and all the rest of obstacles in L_q with larger $minA$ than that of l's will not be affected by e_2 and hence the examination can be terminated.



(a) Obstacle placement (b) New visible region

Fig. 4 Example of EVC algorithm

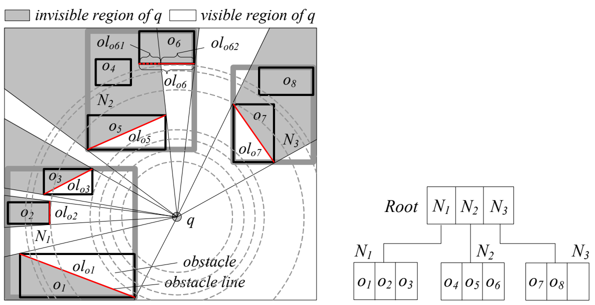
Now the only left task is how to change L_q when a new obstacle line l_N overlaps with some existing obstacle line l in L_q (i.e., case 2), which is handled by Function CheckEdges presented in Algorithm 1. Again, there are three possible cases. First, $l.maxD \le l_N.minD$ and l_N has a zero impact on VR_q. For example, although e_1 overlaps with o_1 in terms of angular bounds, it is invisible to q and thus can be ignored. Second, $l.minD \ge l_N.maxD$ and the entire l_N is visible to q. Hence, l_N is inserted into L_q and the part of l that is blocked by l_N is removed. For example, e_4 is within the angular bound of o_3 and its maximal distance to q is smaller than the minimal distance between o_3 and q. Consequently, e_4 that is visible to q is included into L_q and ol_{o3} is shrunk, as shown in Figure 4(b). Third, l_N and l intersects which means part of l_N is visible to q and the part of l blocked by l_N becomes invisible. L_q needs include the new visible part of l_N and remove the invisible part of l. For instance, the obstacle lines of e_3 and o_1 intersect and that of e_2 and o_2 intersect. We find the intersection points, and update L_q accordingly. After evaluating new edges e_1, e_2, e_3, e_4, the visible region of q is updated to the shaded area shown in Figure 4(b). Please note that the parameter *boolean* in the function is to control if the update operation on L_q is necessary and it is set to *TRUE* only when e refers to a real obstacle.

Algorithm 2 Object Visibility Check Algorithm (OVC)

```
algorithm OVC (e, L_q, q)
if e is an obstacle then
  return EVC (q, L_q, e, TRUE)
else if e is a point then
  return EVC (q, L_q, e, FALSE)
else    // e is a MBR
  for each edge e_i of e do
    f_i = EVC (q, L_q, e_i, FALSE)
  if ∀ f_i = IV then return IV
  else if ∀ f_i = AV then return AV
  else return PV
```

Since we understand how to evaluate the impact of an edge on the visible region of q, we explain how to determine that of a node N (i.e., a rectangle). As a rectangle is consisted of four edges, we evaluate each of them. If four edges are all invisible to q, N is invisible to q and hence N and all its enclosed child nodes can be pruned. If all the edges are visible to q, N is visible to q and its child nodes need further exploration. Otherwise, only edges must be visible/part-visible to q and N might enclose some obstacles that are visible to q and thus its child nodes need further evaluation. Algorithm 2 shows the pseudo-code of *Object Visibility Check* algorithm (OVC). It is important to note that the input e might not be obstacles, as it can be a data point because a result object for VRNN/VR*k*NN search must be visible to the query point. We will explain how VRNN query processing invokes OVC to perform the visibility check in Section IV. A data point p can be regarded as a special case of an edge with $p.minA = p.maxA$ and $p.minD = p.maxD = dist(p, q)$.

Now we are ready to present our *Visible Region Computation* algorithm (VRC). We assume all the obstacles are indexed by an R-tree T_o and VRC traverses T_o in a *best-first* manner with obstacles closer to the query point visited first. A running example is depicted in Figure 5, with T_o for obstacle set $O = \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8\}$ shown in Figure 5(b). We use L_q to store all the obstacle lines that affect the visibility of q, sorted in ascending order of their minimum bounding angles, and a heap H to maintain all the unvisited nodes. Initially, $H = \{N_1, N_2, N_3\}$ and the algorithm always de-heaps the top entry for examination until H becomes empty. First, N_1 is accessed. As it is visible to q, its child nodes are en-heaped for later examination, after which $H = \{o_1, N_2, N_3, o_3, o_2\}$. Next, o_1 is evaluated. As 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 hence its child nodes are en-heaped 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 de-heaped and its child nodes are en-heaped with $H = \{o_7, o_3, o_2, o_4, o_8, o_6\}$. In the sequel, VRC de-heaps obstacles from H and keeps updating L_q until $H = \varnothing$. Finally, $L_q = \{ol_{o7}, ol_{o62}, ol_{o5}, ol_{o3}, ol_{o2}, ol_{o1}\}$.



(a) Obstacle placement (b) The obstacle R-tree

Fig. 5 Example of VRC algorithm

Algorithm 3 presents the pseudo-code 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 maintained 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 entries that are visible to q are en-heaped for later examination (lines 8-10). VRC also explores an early termination condition (lines 4-5), as proved by Lemma 1.

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

Proof: L_q is closed, and suppose there is an entry e with $mindist(e, q) > d_{max}$ 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 (Definition 1). On the other hand, since L_q is closed, $[ol_e.minA, ol_e.maxA] \subseteq \cup_{l \in Lq}[l.minA, l.maxA]$ with ol_e being the obstacle line of e. Without loss of generality, we can assume the extension of line segment $\overline{qp}$ intersects a line $l \in L_q$ at point p' with $dist(p, q) \leq dist(p', q) \leq d_{max}$. As we know $mindist(e, q) \leq dist(p, q)$, consequently $mindist(e, q) \leq d_{max}$ which contradicts our previous assumption. ■

Algorithm 3 Visible Region Computation Algorithm (VRC)

algorithm VRC (T_o, q, L_q)
1: list $L_q = \varnothing$, min-heap $H = \{T_o.root\}$
2: **while** $H \neq \varnothing$ **do**
3: de-heap the top entry (e, key) of H
4: **if** $L_q.isclose = TRUE$ and $mindist(e, q) > d_{max}$ **then**
5: break // terminate
6: **if** e is an obstacle **then**
7: OVC (e, L_q, 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

IV. VRNN Query Processing

In this section, we explain how to process VRNN query. We first present the pruning strategy, detail the search algorithm and then analyse the cost of VRNN algorithm and proof its correctness.

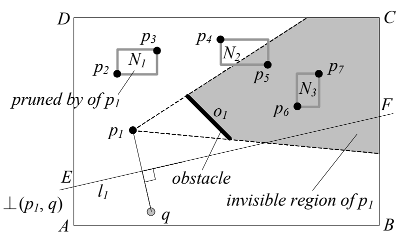


Fig. 6 Illustration of pruning based on half-planes and visibility check

A. Pruning Strategy

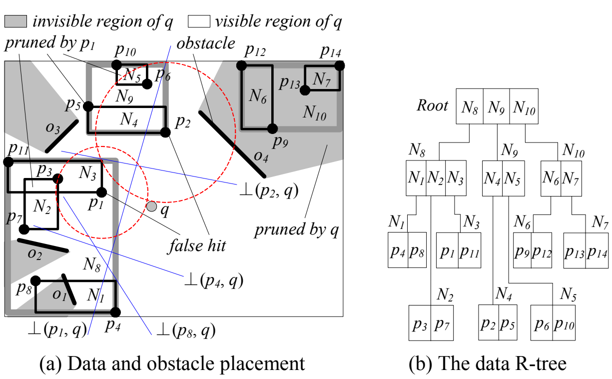
We use *half-plane property* (as [7]) and *visibility check* to prune the search space. Consider the perpendicular bisector between a data point p_1 and a given query point q, i.e., $\perp(p_1, q)$ (i.e., line l_1) as illustrated in Figure 6. The bisector divides the whole data space into two half-planes, 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 inside HP_{p1} (p_1, q) but 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 completely inside $HP_{p1}(p_1, q)$ and are *part-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*.

B. The VRNN Algorithm

We adopt a two-step filter-and-refinement framework to deal with VRNN queries, assuming that both data set P and obstacle set O are indexed by R-trees. In order to improve the performance, these two steps are combined into a single traversal of the trees. In particular, the algorithm accesses nodes/points in ascending order of their distance to the query point q to retrieve a set of potential candidates, maintained by a candidate set S_c. All the points and nodes that cannot be/contain a VRNN of q are pruned by the above mentioned pruning strategy, and inserted (without being visited) into 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 4 presents the pseudo-code 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 Figure 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 Figure 7(b). A *primary heap* H_w is maintained to keep all the unvisited nodes ordered in ascending order of their smallest distance to the query point q.

Algorithm 4 VRNN Search Algorithm (VRNN)

```
algorithm VRNN (Tp, To, q)
  initialize sets Sc = ∅, Sp = ∅, Sn = ∅, Sr = ∅
  VRNN-Filter (Tp, To, q, Sc, Sp, Sn)
  VRNN-Refinement (q, Sc, Sp, Sn, Sr)
  return Sr
```



(a) Data and obstacle placement (b) The data R-tree

Fig. 7 Example of VRNN algorithm

1) The Filter Step: Initially, VRNN visits the root of T_p and 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 de-heaps N_8, accesses its child nodes, and en-heaps 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 de-heaped 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.

Algorithm 5 Filter for VRNN Algorithm (VRNN-Filter)

```
algorithm VRNN-Filter (Tp, To, q, Sc, Sp, Sn)
  point cp = NULL, min-heaps Hw = {To.root} and Ha = ∅
  VRC (To, q, Lq)
  while Hw ≠ ∅ do
  |  de-heap the top entry (e, key) of Hw
  |  if e is a data point then
  |  |  Sc = Sc + {e}; cp = e; VRC (To, cp, Lcp)
  |  |  while Hw ≠ ∅ do
  |  |  |  de-heap the top entry (e′, key) of Hw
  |  |  |  if e′ is a data point and Trim (q, cp, e′) = ∞ then
  |  |  |  |  if OVC (e′, Lcp, cp) = AV then Sp = Sp + {e′}
  |  |  |  |_ else insert (e′, dist(e′, q)) into Ha
  |  |  |  else if e′ is a data point and Trim (q, cp, e′) ≠ ∞ then
  |  |  |  |_ insert (e′, dist(e′, q)) into Ha
  |  |  |  else    // e′ is a MBR (i.e., an intermediate node)
  |  |  |  |  for each entry ei′ ∈ e′ do
  |  |  |  |  |  if OVC (ei′, Lq, q) ≠IV and Trim (q, cp, ei′) = ∞ then
  |  |  |  |  |  |  if OVC (ei′, Lcp, cp) = AV then
  |  |  |  |  |  |  |  if ei′ is a data point then Sp = Sp + {ei′}
  |  |  |  |  |  |  |_ else Sn = Sn + {ei′}
  |  |  |  |  |  |  else if OVC (ei′, Lcp, cp) = PV then
  |  |  |  |  |  |  |_ insert (ei′, mindist(ei′, q)) into Hw
  |  |  |  |  |  |_ else insert (ei′, mindist(ei′, q)) into Ha
  |  |  |  |  |  else if OVC (ei′, Lq, q) ≠ IV and Trim (q, cp, ei′) ≠ ∞
  |  |  |  |  |  |_ insert (ei′, mindist(ei′, q)) into Hw
  |  |  |  |  |  else    // OVC (ei′, Lq, q) = IV
  |  |  |  |  |  |  if ei′ is a data point then Sp = Sp + {ei′}
  |  |  |_ |_ |_ |_ else Sn = Sn + {ei′}
  |  |_ swap (Hw, Ha)    // change the roles between Hw and Ha
  |  else    // e is a MBR (i.e., an intermediate node)
  |  |  for each entry ei ∈ e do
  |  |  |  if OVC (ei, Lq, q) ≠ IV and cp = NULL then
  |  |  |  |_ insert (ei, mindist(ei, q)) into Hw
  |  |  |  else if OVC (ei, Lq, q) ≠ IV and cp ≠ NULL then
  |  |  |  |  if Trim (q, cp, ei) = ∞ then
  |  |  |  |  |  if OVC (ei, Lcp, cp) = AV then
  |  |  |  |  |  |  if ei is a data point then Sp = Sp + {ei}
  |  |  |  |  |  |_ else Sn = Sn + {ei}
  |  |  |  |  |_ else insert (ei, mindist(ei, q)) into Hw
  |  |  |  |_ else insert (ei, mindist(ei, q)) into Hw
  |  |  |  else    // OVC (ei, Lq, q) = IV
  |  |  |  |  if ei is a data point then Sp = Sp + {ei}
  |_ |_ |_ |_ else Sn = Sn + {ei}
```

The next de-heaped entry is N_9. As *cp* (= p_1) is not empty, VRNN uses *Trim* algorithm (as [7]) to check whether N_9 can be pruned. As part of N_9 lies in HP_q (*cp*, q), it has to be accessed and VRNN visits its child nodes. Child node N_5 is discarded as it locates inside HP_{cp} (*cp*, q) and it is *all-visible* to *cp*, meaning that it cannot contain any qualifying candidates. Thus, N_5, which is a MBR, is added to S_n, i.e., $S_n = \{N_{10}, N_5\}$.

The other child entry N_4 is en-heaped into H_w (= $\{N_4, N_1, N_2, p_{11}\}$) because it falls partially into HP_{cp} (cp, q) and is also *all-visible* to cp, indicating that N_4 may contain VRNN candidates. VRNN proceeds to de-heap 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 de-heaped. As it cannot be pruned by current pruning point (p_1), it becomes the second pruning point and maintained by an *auxiliary heap* $H_a = \{p_2\}$. Next, VRNN accesses node N_1 in which points p_4 and p_8 (children of N_1) are inserted into H_w (= $\{N_2, p_4, p_8, p_{11}\}$). Note that although p_8 falls fully inside HP_{cp} (cp, q), it is *invisible* to cp due to the obstruction of obstacle o_2, and hence p_8 cannot be pruned by the current pruning point (i.e., p_1). 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 all-visible to cp. In the sequel, p_4 and p_8 are retrieved and inserted into H_a, after which $H_a = \{p_2, p_4, p_8\}$ ordered based on ascending order of their *mindist* to q. Finally, p_{11} is de-heaped and it is added to $S_p = \{p_5, p_{11}\}$ since it satisfies the pruning condition. 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. 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, but takes H_a as a primary heap. VRNN proceeds in the same loop until $H_w = H_a = \varnothing$, i.e., all the pruning 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 5 shows the pseudo-code of the *Filter for VRNN* algorithm (VRNN-Filter). When an intermediate node is visited, it calls OVC algorithm to check its visibility to the query point q and then processes it. Similarly, when a data point is accessed, it invokes OVC algorithm 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., find the obstacles from T_o that can affect cp's visibility. Note that all pruned entries are stored in their corresponding refinement set but not removed since they are used for verifying candidates in the next refinement step.

2) The Refinement Step: When the filter step finishes, the refinement step starts, with Algorithm 6 depicting the pseudo-code of the *Refinement for VRNN algorithm* (VRNN-Refinement). In the first place, VRNN-Refinement conducts *self-filtering* (lines 2-4), that is, it prunes the candidates that are closer to each other than q. Then, the algorithm enters the next refinement step, where it verifies whether each remaining candidate in S_c is a true result (lines 6-16). First it calls *Round of Refinement* algorithm (Refinement-Round), defined in Algorithm 7, to eliminate false candidates from S_c based on the content 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. Consequently, nodes in *p.toVisit* are visited with each access updating the content of S_p and S_n. Note S_p and S_n are reset to $\varnothing$ after each round of Refinement-Round (line 11) to avoid duplicated checking. The refinement step continues until S_c is empty.

Algorithm 6 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** initialize $p.toVisit = \varnothing$
6: **if** $S_c \neq \varnothing$ **then**
7: **repeat**
8: Refinement-Round (q, S_c, S_p, S_n, S_r)
9: let N be the lowest level node of *p.toVisit* for $p \in S_c$
10: remove N from all *p.toVisit* and access N
11: $S_p = S_n = \varnothing$ // for the next round
12: **if** N is a leaf node **then**
13: $S_p = \{p' \mid p' \in N$ and p' is visible to $p\}$
14: **else**
15: $S_n = \{N' \mid N' \in N$ and N' is visible to $p\}$
16: **else** return // terminate

Algorithm 7 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) = PV **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** $p.toVisit = \varnothing$ **then** $S_c = S_c - \{p\}$; $S_r = S_r + \{p\}$

Now we explain the detail of Refinement-Round algorithm. Specifically, it has three tasks, i.e., pruning false positive, returning result objects, and identifying nodes that might invalidate the remaining points in S_c. First, points p in S_c satisfying following any condition are for sure false positives and can be pruned: (i) $\exists\, p' \in S_p$ such that p' is visible to p and $dist(p', p) < dist(q, p)$ (lines 2-4), or (ii) $\exists\, N \in S_n$ 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. Hence, $minmaxdist(N, p) < dist(q, p)$ means that N contains at least one point that is nearer to p than q. For example, in Figure 7 $p_2 \in S_c$ can be safely discarded as $N_5 \in S_n$ 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: (i) $\forall\, p' \in S_p$, p' is either invisible to p or $dist(p', p) > dist(q, p)$, and (ii) $\forall\, N \in S_n$, it is all-visible/part-visible to p and $mindist(N, p) > dist(q, p)$. In our example, p_4 and p_8 satisfy the above conditions, and thus, they are removed from S_c and reported as the VRNNs of q immediately. The points p in S_c cannot be pruned or reported as real result objects must have some nodes in S_n that contradict above conditions, and we use a set *p.toVisit* to record all the nodes (lines 9-11). Take p_1 as an example. As p_1.*toVisit* $=\{N_2\}$, we access N_2 and find out the enclosed point p_3 is the VNN of p_1 and hence p_1 is invalidated.

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 better pruning. In our example shown in Figure 7, the second refinement round starts with $S_c = \{p_1\}$, $S_p = \{p_3, p_7\}$ (i.e., points enclosed in N_2), $S_n = \varnothing$, and $S_r = \{p_4, p_8\}$. Point p_1 is eliminated as a false positive as p_3 is visible to p_1 and $dist(p_3, p_1) < dist(q, p_1)$, and then the VRNN algorithm terminates.

C. Discussion

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 the data R-tree T_p *at most once*, and the obstacle R-tree T_o at most $(|S_c| + 1)$ times. □

As shown in Algorithm 5, the VRNN-Filter algorithm only traverses T_p *once* to retrieve a set of VRNN candidates. It then uses *half-plane property* and *visibility check* to prune false candidates and calls the VRC algorithm *once* for *each candidate* $p \in S_c$ to find the obstacles affecting its visibility (line 6 in Algorithm 5). Moreover, VRNN-Filter also invokes the VRC algorithm *once* to retrieve the obstacles that can affect the visibility of q (line 2 in Algorithm 5). Consequently, the VRNN algorithm traverses T_o at most $(|S_c| + 1)$ times.

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: 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. A VRNN algorithm calls VRNN-Filter and VRNN-Refinement algorithms with complexities being $O(log|T_p| \times (|S_c| + 1)\ log|T_o|)$ and $O(|S_c|^2 + |S_c|\ (|S_p| + |S_n|))$. 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 non-qualifying points/nodes in the filter step by using our proposed pruning strategy. Hence, no result is 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 may potentially contain VNNs of p, which ensures no false positives. ■

V. VR*k*NN Query Processing

In this section, we discuss how our solution can be adapted to answer more general VR*k*NN queries that find all the points whose V*k*NN set includes q. First, the pruning strategy (described in Section IV-A) can be extended to arbitrary values of k. Assume a VR*k*NN query and a dataset 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 falls completely inside $\bigcap_{i=1}^{k} HP_{\theta i}(\theta_i, q)$ 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. Hence, it can be safely pruned away. On the other hand, if a point/node locates inside $\bigcap_{i=1}^{k} HP_{\theta i}(\theta_i, q)$ and is *part-visible/invisible* to *any subset* of D, it can be/contain a VR*k*NN of q and thus needs further examination.

Next, we explain how to extend the VRNN algorithm for VR*k*NN query processing. Similarly, it follows a filter-refinement framework. Specifically, VR*k*NN first finds a set S_c of VR*k*NN candidates that contains all the actual query results. Then, the algorithm eliminates/validates every candidate in S_c to remove all the false hits. The VRNN-Filter algorithm can be easily adapted to support VR*k*NN query, by integrating the aforementioned pruning strategy. The VRNN-Refinement algorithm can also be extended for VR*k*NN retrieval. During the processing, all the points $p \in S_c$ with *at least k* points *visible* to q within $dist(p, q)$ are pruned as false candidates, while the rest form the final result set. Since the number of points within $dist(p, q)$ and meanwhile visible to p determine whether p is a final result, we associate a counter *cnt* with each $p \in S_c$ during the refinement phase. Every time we find a point $p' \in S_c$ that satisfies the following two conditions: (i) p' is *visible* to p, and (ii) $dist(p', p) < dist(q, p)$, the p's counter *cnt* is increased by one. Eventually, p can be removed as a *false positive* when $cnt \geq k$. The pseudo-codes of the algorithms for VR*k*NN query processing are ignored for space saving.

VI. Experimental Evaluation

In this section, we evaluate the efficiency and effectiveness of our proposed VRNN and VR*k*NN algorithms via experiments on synthetic and real datasets. First, Section VI-A describes the experimental settings, and then Sections VI-B and VI-C present experimental results and our findings for VRNN and VR*k*NN queries, respectively. All the algorithms (i.e., Naive, VRNN, and VR*k*NN) were implemented in C++. Experiments were conducted on a PC with a Pentium IV 3.0 GHz CPU and 2GB RAM, running Microsoft Windows XP Professional Edition.

Here, the Naive algorithm refers to the naive solution introduced in Section I-B. It retrieves all the points that are visible to a given query point q, denoted by S_q; and then performs V*k*NN search on each point $p \in S_q$ in order to determine whether q is included into *VkNN*(p). The set of points p that have $q \in VkNN(p)$ form the final result set.

A. Experimental Setup

We deploy five real datasets[5], which are summarized in Table II. Synthetic datasets are created following the Uniform

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

distribution and Zipf distribution, with the cardinality varying from 0.1 × |LA| to 10 × |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 α = 0.8. All the datasets are mapped to a [0, 10000] × [0, 10000] square. As VRNN and VRkNN queries involve a data set P and an obstacle set O, we deploy five different combinations of the datasets, 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.

TABLE II
DESCRIPTION OF REAL DATASETS USED IN EXPERIMENTS

Dataset	Cardinality	Description
LB	58945	2D point in Long Beach
NA	470759	2D point in North America
LA	131461	2D MBRs of streets in Los Angeles
Cities	5922	2D cities (as point) in Greece
Rivers	21645	2D MBRs of rivers in Greece

TABLE III
PARAMETER RANGES AND DEFAULT VALUES

Parameter	Range	Default
k	1, 2, 4, 8, 16	1, 4
$\|P\|/\|O\|$	0.1, 0.2, 0.5, 1, 2, 5, 10	1
buffer size (% of tree size)	0, 10, 20, 30, 40, 50, 60	0

All data and obstacle sets are indexed by R*-trees [24] with a disk page size of 1K bytes. 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 efficiency and effectiveness of VRNN and VRkNN algorithms under a variety of parameters which are listed in Table III. 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 dataset distribution and the total query cost is evaluated. Both the I/O overhead (by charging 10ms per page fault, as in [7]) and CPU time contribute to the query cost. We assume 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.

B. Results on VRNN Queries

The first set of experiments compares the performance of the Naive algorithm and VRNN algorithm for VRNN queries, with the total query cost (in seconds) of CR datasets depicted in Figure 8. Here, each result is broken into two components, corresponding to the filter step and the refinement step, respectively. The number with percentage on top of each bar is the percentage of I/O time in the total query cost. For example, Naive algorithm incurs extremely high I/O cost, 97% of the total query cost. The number inside the brackets on top of each bar is the cardinality of the candidate set, i.e., $|S_c|$. For instance, Naive algorithm retrieves 68 candidate objects in the filter step while VRNN algorithm only retrieves 2.8 candidates on average. Finally, the number with percentage inside the performance bar indicates the ratio of the cost incurred in the filter step to that of the total query cost. For example, Naive algorithm spends 94% of the cost in the filter step while VRNN algorithm spends 99% of the cost in the filter step.

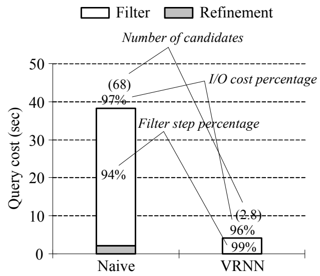


Fig. 8 Naive vs. VRNN (k = 1, CR)

As expected, VRNN outperforms Naive significantly. For 200 queries, VRNN can improves the query cost to up to 11% and reduces the number of candidates to only 4%, compared with that of Naive algorithm. The reason behind is that Naive needs to traverse the data R-tree T_p and the obstacle R-tree T_o multiple times, incurring extremely expensive I/O overhead and distance computation. As demonstrated in Lemma 2 (presented in Section IV-C), VRNN traverses T_p *at most once*, and T_o at most ($|S_c|$ + 1) times, which saves considerable I/O cost.

In addition, we observe that Filter step actually dominates the overall overhead (> 90%), especially for VRNN. This is because: (i) VRNN reuses all the points and nodes pruned from the filter step to perform candidate verification in the refinement step, and hence duplicated accesses to the same points/nodes are avoided; and (ii) 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 verified by visiting a limited number of additional nodes. Since Naive for sure performs worse than VRNN (several orders of magnitude), its performance is omitted in the rest of experimental results.

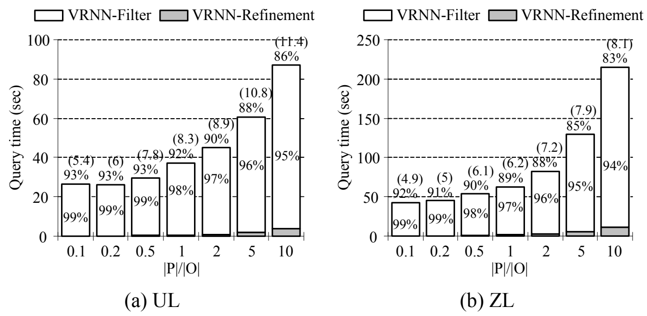
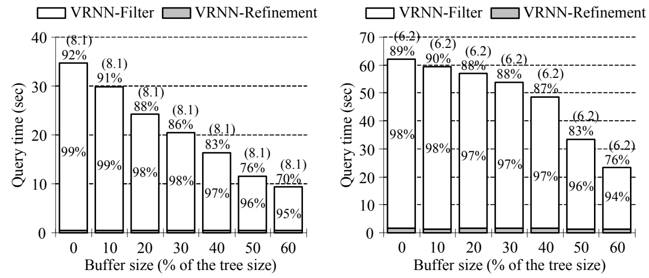


Fig. 9 VRNN cost vs. $|P|/|O|$ (k = 1)

Next, we investigate the effect of the ratio $|P|/|O|$ on the proposed VRNN algorithm using two dataset combinations (i.e., UL and ZL). Figure 9 plots the total query cost of the VRNN algorithm as a function of $|P|/|O|$, fixing $k = 1$. It is observed that the cost of VRNN demonstrates a stepwise behaviour. Specifically, it increases slightly as $|P|/|O|$ changes from 0.1 to 1, but then ascends much faster as $|P|/|O|$ grows further. This is because, 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 check, and more candidate verification. Similar as previous evaluation, VRNN is very efficient in the refinement step, especially when the ratio $|P|/|O|$ is small (e.g., 0.1, 0.2).



(a) UL ($|P|/|O| = 1$) (b) ZL ($|P|/|O| = 1$)

Fig. 10 VRNN cost vs. buffer size ($k = 1$)

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% 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, with its results under UL and ZL dataset combinations shown in Figure 10. The total query cost is reduced as buffer size increases. In particular, as the buffer size enlarges, the VRNN-Filter cost is observed to drop, but the VRNN-Refinement cost remains almost the same. This is because that the filter step of VRNN requires traversing the obstacle R-tree T_o ($|S_c| + 1$) times (by Lemma 2). Consequently, it may access the same nodes (e.g., the root of T_o, i.e., $T_o.root$) multiple times, and hence a buffer space can improve the performance by keeping the nodes locally available.

C. Results on VRkNN Queries

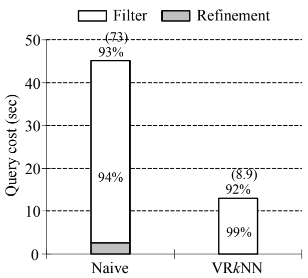
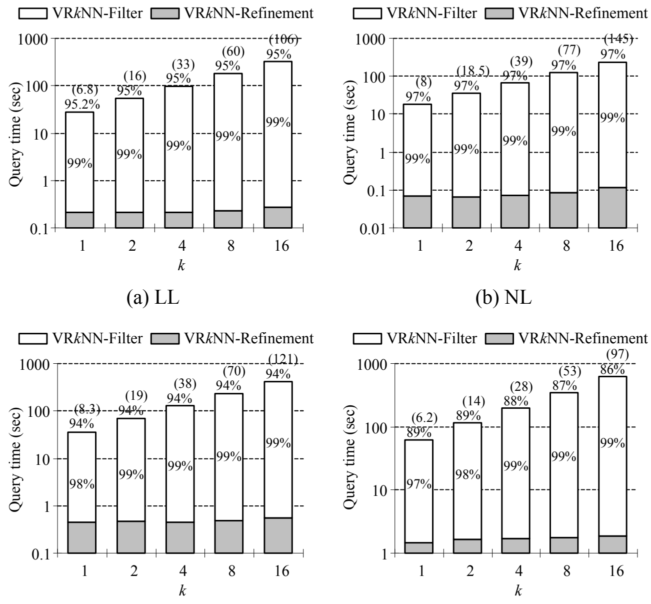


Fig. 11 Naive vs. VRkNN ($k = 4$, CR)

The second set of experiments evaluates the efficiency and effectiveness of VRkNN query processing algorithms. First, we compare the efficiency of alternative algorithms (i.e., Naive and VRkNN) for VRkNN queries, fixing $k = 4$ which is the median value of all the ks we evaluate. Figure 11 presents our experimental results on the CR dataset combination. Similar as performance of VRNN search presented in Figure 8, we also demonstrate the I/O cost percentage, the number of candidates, and the percentage of the filter step. It is observed both Naive and VRkNN search algorithms demonstrate similar performance trends as that under $k = 1$.

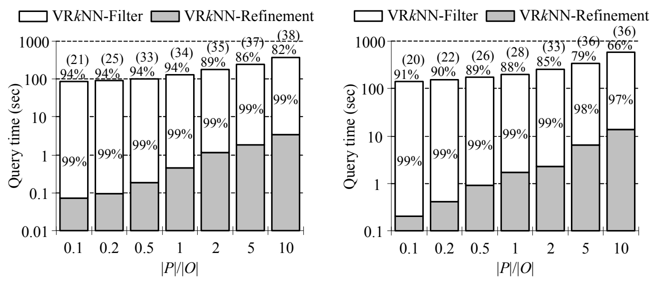
Next, we evaluate the impact of the number k of requested VRNNs on the performance of the VRkNN algorithm, using LL, NL, UL, and ZL dataset combinations. Figure 12 illustrates the total query cost of the VRkNN algorithm as a function of k which varies from 1, to 2, to 4, to 8, and finally to 16. As expected, 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).



(a) LL (b) NL

(c) UL ($|P|/|O| = 1$) (d) ZL ($|P|/|O| = 1$)

Fig. 12 VRkNN cost vs. k (logarithmic scales)

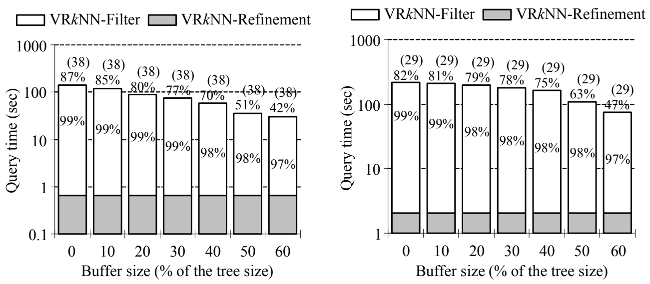


(a) UL (b) ZL

Fig. 13 VRkNN cost vs. $|P|/|O|$ ($k = 4$, logarithmic scales)

In the following experiments, we explore the effects of different parameters, including the ratio $|P|/|O|$ and buffer size, on the performance of the VRkNN algorithm, using UL and

ZL dataset combinations. In Figure 13, we show the efficiency of the algorithm for VRkNN queries by fixing k = 4 and varying $|P|/|O|$ between 0.1 and 10. In Figure 14, we plot the cost of the VRkNN algorithm with respect to the buffer sizes. All the observations made for the VRkNN search are similar to those we make for the VRNN retrieval and thus the detailed explanation is ignored.



(a) UL ($|P|/|O|$ = 1) (b) ZL ($|P|/|O|$ = 1)

Fig. 14 VRkNN cost vs. buffer size (k = 4, logarithmic scales)

VII. CONCLUSIONS

In this paper, we identify and solve a novel type of reverse nearest neighbor queries, namely *visible reverse nearest neighbor* (VRNN) search. Although both the RNN search and the 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 search is useful in many decision support applications involving spatial data and physical obstacles. Consequently, we propose an efficient search algorithm for VRNN query, assuming that both P and 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 proof its correctness. In addition, we generalize our methods to handle *visible reverse k-nearest neighbor* (VRkNN) search. An extensive experimental study with real and synthetic datasets has been conducted which further demonstrates the efficiency and effectiveness of our proposed algorithms for dealing with VRNN and VRkNN queries, under various experimental settings.

In the future, we plan to extend our techniques to other VRNN variations such as constrained VRNN and Top-k VRNN etc. Also, we intend to investigate efficient algorithms for tackling the VRNN retrieval with respect to a line segment which contains continuous query points instead of a fixed query point.

REFERENCES

[1] F. Korn and S. Muthukrishnan, "Influence sets based on reverse nearest neighbor queries," in *SIGMOD*, 2000, pp. 201–212.

[2] I. Stanoi, D. Agrawal, and A. El Abbadi, "Reverse nearest neighbor queries for dynamic databases," in *SIGMOD Workshop DMKD*, 2000, pp. 44–53.

[3] C. Yang and K.-I. Lin, "An index structure for efficient reverse nearest neighbor queries," in *ICDE*, 2001, pp. 485–492.

[4] A. Nanopoulos, Y. Theodoridis, and Y. Manolopoulos, "C2P: Clustering based on closest pairs," in *VLDB*, 2001, pp. 331–340.

[5] A. Maheshwari, J. Vahrenhold, and N. Zeh, "On reverse nearest neighbor queries," in *CCCG*, 2002, pp. 128–132.

[6] A. Singh, H. Ferhatosmanoglu, and A. Tosun, "High dimensional reverse nearest neighbor queries," in *CIKM*, 2003, pp. 91–98.

[7] Y. Tao, D. Papadias, and X. Lian, "Reverse kNN search in arbitrary dimensionality," in *VLDB*, 2004, pp. 744–755.

[8] I. Stanoi, M. Riedewald, D. Agrawal, and A. Abbadi, "Discovery of influence sets in frequently updated databases," in *VLDB*, 2001, pp. 99–108.

[9] F. Korn, S. Muthukrishnan, and D. Srivastava, "Reverse nearest neighbor aggregates over data streams," in *VLDB*, 2002, pp. 814–825.

[10] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao, "Reverse nearest neighbors in large graphs," in *ICDE*, 2005, 186–187.

[11] M. L. Yiu and N. Mamoulis, "Reverse nearest neighbors search in ad-hoc subspaces," in *ICDE*, 2006, p. 76.

[12] E. Achtert, C. Bohm, P. Kroger, P. Kunath, A. Pryakhin, and M. Renz, "Efficient reverse k-nearest neighbor search in arbitrary metric spaces," in *SIGMOD*, 2006, pp. 515–526.

[13] Y. Tao, M. L. Yiu, and N. Mamoulis, "Reverse nearest neighbor search in metric spaces," *TKDE*, vol. 18, no. 9, pp. 1239–1252, 2006.

[14] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis, "Nearest and reverse nearest neighbor queries for moving objects," *VLDB Journal*, vol. 15, no. 3, pp. 229–250, 2006.

[15] T. Xia and D. Zhang, "Continuous reverse nearest neighbor monitoring," in *ICDE*, 2006, p. 77.

[16] J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang, "Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors," in *ICDE*, 2007, pp. 806–815.

[17] C. K. Ken Lee, B. Zheng, and W.-C. Lee, "Ranked reverse nearest neighbor search," *TKDE*, vol. 20, no. 7, pp. 894–910, 2008.

[18] J. Zhang, D. Papadias, K. Mouratidis, and M. Zhu, "Spatial queries in the presence of obstacles," in *EDBT*, 2004, pp. 366–384.

[19] C. Xia, D. Hsu, and A. K. H. Tung, "A fast filter for obstructed nearest neighbor queries," in *BNCOD*, 2004, pp. 203–215.

[20] S. Nutanong, E. Tanin, and R. Zhang, "Visible nearest neighbor queries," in *DASFAA*, 2007, pp. 876–883.

[21] V. Estivill-Castro and I. Lee, "Autoclust+: Automatic clustering of point-data sets in the presence of obstacles," in *TSDM*, 2000, pp. 133–146.

[22] A. K. H. Tung, J. Hou, and J. Han, "Spatial clustering in the presence of obstacles," in *ICDE*, 2001, pp. 359–367.

[23] O. R. Zaiane and C.-H. Lee, "Clustering spatial data in the presence of obstacles: A density-based approach," in *IDEAS*, 2002, pp. 214–223.

[24] 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, pp. 322–331.

[25] K.-I. Lin, M. Nolen, and C. Yang, "Applying bulk insertion techniques for dynamic reverse nearest neighbor problems," in *IDEAS*, 2003, pp. 290–297.

[26] C. Xia, W. Hsu, and M.-L. Lee, "ERkNN: Efficient reverse k-nearest neighbors retrieval with local kNN-distance estimation," in *CIKM*, 2005, pp. 533–540.

[27] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. Abbadi, "Constrained nearest neighbor queries," in *SSTD*, 2001, pp. 257–278.

[28] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *TODS*, vol. 24, no. 2, pp. 265–318, 1999.

[29] T. Xia, D. Zhang, E. Kanoulas, Y. Du, "On computing top-t most influential spatial sites," in *VLDB*, 2005, pp. 946–957.

[30] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis, "Nearest neighbor and reverse nearest neighbor queries for moving objects," in *IDEAS*, 2002, pp. 44–53.

[31] W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan, "Continuous reverse k-nearest-neighbor monitoring," in *MDM*, 2008, pp. 132–139.

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

[33] M. Kofler, M. Gervautz, and M. Gruber, "R-trees for organizing and visualizing 3D GIS databases," *Journal of Visualization and Computer Animation*, vol. 11, no. 3, pp. 129–143, 2000.

[34] 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, pp. 401–410.

[35] L. Shou, Z. Huang, K. L. Tan, "HDoV-tree: The structure, the storage, the speed," in *ICDE*, 2003, pp. 557–568.