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]

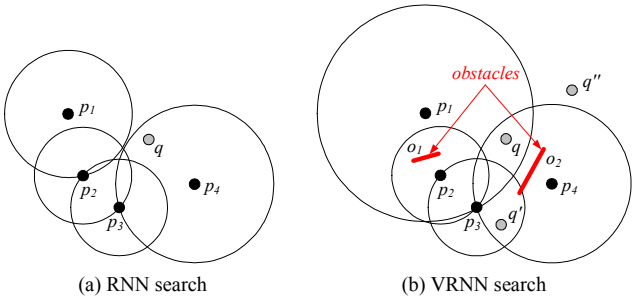
[1]*Singapore Management University, Singapore* {yjgao, bhzheng}@smu.edu.sg
[2]*Zhejiang University, P. R. China* chengc@zju.edu.cn
[3]*Pennsylvania State University, USA* {wlee, cklee}@cse.psu.edu
[4]*City University of Hong Kong, P. R. China* itqli@cityu.edu.hk

***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 method does not require any pre-processing, and employs *half-plane property* and *visibility check* to prune the search space.**

I. Introduction

Given a set of data points P and a query point q in a multi-dimensional space, a *reverse nearest neighbor* (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 necessarily 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 $NN(p_3) = p_2$.

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], [2]; (ii) dynamic algorithms [3], [4]; and (iii) algorithms for various RNN/R*k*NN query variants [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. 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.



(a) RNN search (b) VRNN search

Fig. 1 Example of RNN and VRNN queries

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* (VR*k*NN) 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 propose an efficient algorithm for VRNN query processing, assuming that both the data set and the set of obstacles are indexed by R-trees [15]. 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 subsequent 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 [4]) and *visibility check* to prune the search space.

II. Problem Statement

[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.

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.201

Given a data set P, an obstacle set O, and a query point q, visible k nearest neighbors and visible reverse k nearest neighbor search are formalized 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 straight line connecting p and p' does not cut through any obstacle o in O, i.e., $\forall\, o \in O,\ \overline{pp'} \cap o = \varnothing$. □

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 $o \in O$
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 point 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 Neighbors [16].** Given a data set P, an obstacle set O, a query point q, and an integer k, the *visible k nearest neighbors* of q retrieves a set of points, denoted by $VkNN(q)$, which satisfies following conditions: (i) $\forall p \in VkNN(q)$ is *visible* to q; (ii) $|VkNN(q)| \le k$; 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., $VRkNN(q) = \{p \in P \mid q \in VkNN(p)\}$. □

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. Thus, 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 if p is located inside VR_q.

Before we present the detailed formation algorithm, we first discuss the presentation of a visible region. As shown in Figure 2, a visible region is in 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*, as defined in Definition 4, to address this problem.

Definition 4: **Obstacle line.** The *obstacle line* of an obstacle o[2] w.r.t. q, denoted by ol_o, is the line segment that obstructs the sight lines from q. □

[2]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.

Suppose the rectangle o as shown in Figure 2 is an obstacle, its corresponding obstacle line is ol_o. Since 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.

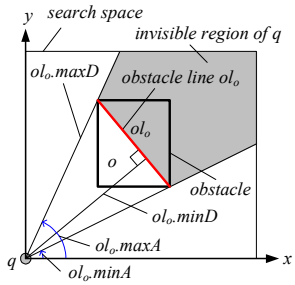


Fig. 2 An example obstacle line, and its angular and distance bounds

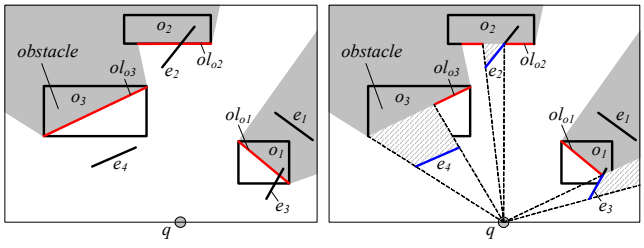
Definition 5: **Angular bound of an obstacle line.** Taking q as an origin in the search space, the *angular bound* of o's obstacle line (i.e., ol_o) w.r.t. q is denoted by $[ol_o.minA, ol_o.maxA]$ where $ol_o.minA$ and $ol_o.maxA$ are respectively the minimum angle and the maximum angle of ol_o, and $ol_o.minA \le ol_o.maxA$ (see Figure 2). If q is located inside o, the angular bound of ol_o w.r.t. q is set to $[0, 2\pi]$. □

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 2). □

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), namely *Edge Visibility Check algorithm* (EVC), and then extend the algorithm for rectangles.

Based on angular bound of obstacle, a specified 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 set L_q (that keeps all the obstacles found so far which affect the visibility of a given query point q), according to the ascending order of their minimal angles. An example is illustrated in Figure 3, with $L_q = \{ol_{o1}, ol_{o2}, ol_{o3}\}$, and e_2 being the edge we are going to evaluate. According to the angular bound of any obstacle line l ($\in L_q$) and e_2, there are three cases: (i) $l.maxA \le e_2.minA$ (e.g., $l = ol_{o1}$), indicating that e_2 will not affect the visibility of l w.r.t q; (ii) $[l.minA, l.maxA] \cap [e_2.minA, e_2.maxA] \ne \varnothing$ (e.g., $l = ol_{o2}$), meaning that a detailed examination is necessary as e_2 is very likely to affect the l's visibility w.r.t. q; and (iii) $l.minA \ge e_2.maxA$ (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. 3 Example of EVC algorithm

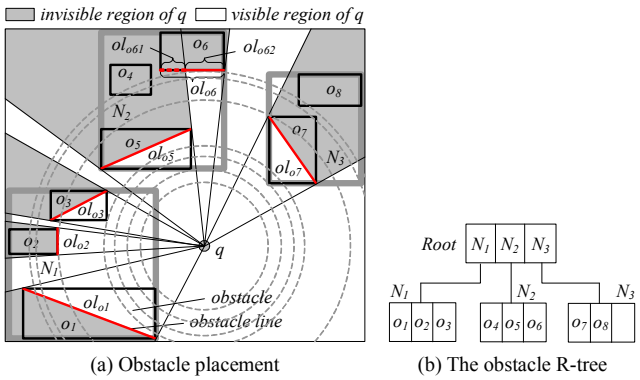
Now the only left task is how to change L_q when a new obstacle line l_N overlaps with some existing obstacle line l in L_q (i.e., case (ii)). Again, there are three possible cases. First, $l.maxD \le l_N.minD$ and l_N has a *zero* impact on VR_q. For example, although e_1 overlaps with o_1 in terms 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 3(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, and e_4, the visible region of q is updated to the shaded area (that contains the shadowed region highlighted in dashed line), as shown in Figure 3(b).

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, 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.

We are now 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 unvisited nodes maintained by a heap H sorted based on ascending order of their minimal distances to the query point. It continuously checks the head entry e of H. The detailed examination varies, dependent on the type of e. If e is an actual obstacle, it is checked against all the obstacle lines maintained in L_q. If it is visible to q, e might contribute to the formation of VR_q and hence L_q is updated. On the other hand, e must be a node and all its child entries that are visible to q are en-heaped for later examination. VRC also exploits an early termination condition, as demonstrated by the following Lemma 1.

Lemma 1: Suppose heap H maintains all the unvisited nodes sorted according to ascending order of their minimal distances to the specified 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 $mindist(e, q) > d_{max} = \text{MAX}_{l \in Lq}(l.maxD)$ holds, e and all the remaining 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). Since L_q is closed, on the other hand, $[ol_e.minA, ol_e.maxA] \subseteq \cup_{l \in Lq}[l.minA, l.maxA]$ with ol_e being the obstacle line of e. Without loss of generality, we can assume that the extension of line segment $\overline{qp}$ intersects a line $l \in L_q$ at point p' with $dist(p, q) \le dist(p', q) \le d_{max}$. As we know $mindist(e, q) \le dist(p, q)$, consequently $mindist(e, q) \le d_{max}$, which contradicts our previous assumption. ■



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

Fig. 4 Example of VRC algorithm

A running example is depicted in Figure 4, 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 4(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, sorted according to their minimal distances to q. 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}}, where ol_{o62} is the partial obstacle line of obstacle o_6 (as shown in Figure 4(a)).

IV. VRNN Query Processing

In this section, we explain how to process VRNN query. In order to improve the search performance, we employ *half-plane property* (as [4]) and *visibility check* 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) as illustrated in Figure 5. 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*.

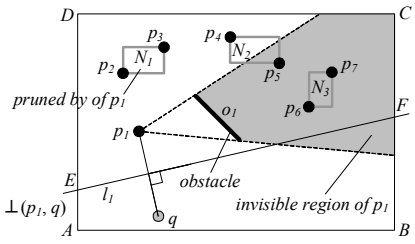


Fig. 5 Illustration of pruning based on half-planes and visibility check

Algorithm 1 VRNN Search Algorithm (VRNN)

algorithm VRNN (T_p, T_o, q)
1: initialize sets $S_c = \emptyset$, $S_p = \emptyset$, $S_n = \emptyset$, $S_r = \emptyset$
2: VRNN-Filter (T_p, T_o, q, S_c, S_p, S_n)
3: VRNN-Refinement (q, S_c, S_p, S_n, S_r)
4: return S_r

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 enhance the efficiency of VRNN query processing, these two steps are combined into a single traversal of the trees. In particular, the algorithm accesses nodes/points in ascending order of their distances to the specified 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 1 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 visible reverse nearest neighbors of q. The details of filtering step and refinement step are omitted due to space limitation.

V. Conclusions

In this paper, we identify and solve a novel type of reverse nearest neighbor queries, namely *visible reverse nearest neighbor* (VRNN) search. Although the RNN search and its variants have been well-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 VRNN query processing algorithm, assuming that both P and O are indexed by R-trees.

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

Acknowledgement

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

References

[1] F. Korn and S. Muthukrishnan, "Influence sets based on reverse nearest neighbor queries," in *SIGMOD*, 2000, pp. 201–212.
[2] C. Yang and K.-I. Lin, "An index structure for efficient reverse nearest neighbor queries," in *ICDE*, 2001, pp. 485–492.
[3] A. Singh, H. Ferhatosmanoglu, and A. Tosun, "High dimensional reverse nearest neighbor queries," in *CIKM*, 2003, pp. 91–98.
[4] Y. Tao, D. Papadias, and X. Lian, "Reverse *k*NN search in arbitrary dimensionality," in *VLDB*, 2004, pp. 744–755.
[5] I. Stanoi, M. Riedewald, D. Agrawal, and A. Abbadi, "Discovery of influence sets in frequently updated databases," in *VLDB*, 2001, pp. 99–108.
[6] F. Korn, S. Muthukrishnan, and D. Srivastava, "Reverse nearest neighbor aggregates over data streams," in *VLDB*, 2002, pp. 814–825.
[7] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao, "Reverse nearest neighbors in large graphs," in *ICDE*, 2005, 186–187.
[8] M. L. Yiu and N. Mamoulis, "Reverse nearest neighbors search in ad-hoc subspaces," in *ICDE*, 2006, p. 76.
[9] 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.
[10] Y. Tao, M. L. Yiu, and N. Mamoulis, "Reverse nearest neighbor search in metric spaces," *TKDE*, vol. 18, no. 9, pp. 1239–1252, 2006.
[11] 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.
[12] T. Xia and D. Zhang, "Continuous reverse nearest neighbor monitoring," in *ICDE*, 2006, p. 77.
[13] 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.
[14] Ken C. K. Lee, B. Zheng, and W.-C. Lee, "Ranked reverse nearest neighbor search," *TKDE*, vol. 20, no. 7, pp. 894–910, 2008
[15] 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.
[16] S. Nutanong, E. Tanin, and R. Zhang, "Visible nearest neighbor queries," in *DASFAA*, 2007, pp. 876–883.