Incremental Evaluation of Visible Nearest Neighbor Queries

Sarana Nutanong[†‡], Egemen Tanin[†‡], Rui Zhang[†]
[†]Department of Computer Science and Software Engineering
University of Melbourne, Victoria, Australia
{sarana,egemen,rui}@csse.unimelb.edu.au
[‡]NICTA Victoria Laboratory, Australia

***Abstract*— In many applications involving spatial objects, we are only interested in objects that are directly visible from query points. In this article, we formulate the visible k nearest neighbor (VkNN) query and present incremental algorithms as a solution, with two variants differing in how to prune objects during the search process. One variant applies visibility pruning to only objects, whereas the other variant applies visibility pruning to index nodes as well. Our experimental results show that the latter outperforms the former. We further propose the *aggregate VkNN query*, which finds the visible k nearest objects to a set of query points based on an aggregate distance function. We also propose two approaches to processing the aggregate VkNN query. One accesses the database via multiple VkNN queries, whereas the other issues an aggregate k nearest neighbor query to retrieve objects from the database and then re-rank the results based on the aggregate visible distance metric. With extensive experiments, we show that the latter approach consistently outperforms the former one.**

***Index Terms*— Geographical information systems, Spatial databases, Query processing.**

I. Introduction

VISIBILITY is an extensively studied topic in computational geometry and computer graphics. Many algorithms have been developed to efficiently compute the region visible to a given query point [2], [3], [13], [25], [29]. Many problems in spatial databases also involve visibility. For example, a tourist can be interested in locations where views of scenes such as sea or mountains are available. In an interactive online game, a player commonly needs to know enemy locations that can be seen from his/her position. In such problems, only objects directly visible from a user's location are relevant. In this article, we investigate the *visible k nearest neighbor (VkNN) query* [18], which incorporates the requirement of visibility into the k nearest neighbor (kNN) query.

A VkNN query retrieves k objects with the smallest visible distances to a query point $\boldsymbol{q}$. In Figure 1, the V3NN of $\boldsymbol{q}$ are B, A and D (in order of visible distance). Object C is excluded because it is blocked by B. Object A is considered nearer to $\boldsymbol{q}$ than D because the visible part of A is nearer to $\boldsymbol{q}$ than that of D.

Processing the VkNN query requires determining the visibility of objects. One straightforward method consists of the following steps: (i) calculating the visibility region of a query point, (ii) using the query's visibility region to "clip" data objects to obtain the visible parts of each object, and (iii) executing a kNN query on the clipped data objects. The drawback of this approach is that the visibility region computation requires accessing all objects in the database.

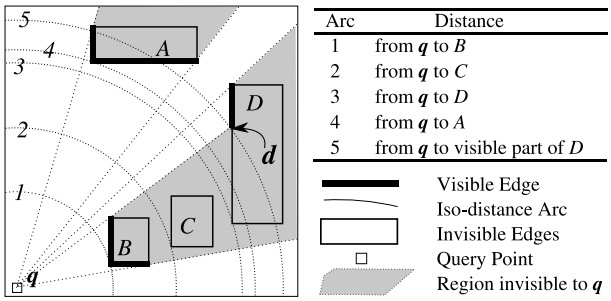


Fig. 1. The VkNN query

We propose more efficient VkNN algorithms, based on the observation that finding the k *visible NNs (VNNs)* requires only a subset of the complete visibility region. Specifically, to determine the visible distance between the query point $\boldsymbol{q}$ and an object X, it is sufficient to consider only the objects nearer to $\boldsymbol{q}$ than X. The above observation allows us to adapt an *incremental nearest neighbor* algorithm [14] to simultaneously obtain the relevant obstacles and VNNs. This adapted incremental VkNN algorithm makes use of a new distance function, MinViDist, to rank the VNNs and order the tree branches during the search. The MinViDist between X and $\boldsymbol{q}$ is defined as the distance from $\boldsymbol{q}$ to the nearest visible point on X. For example (Figure 1), the MinViDist between $\boldsymbol{q}$ and D is the distance between $\boldsymbol{q}$ and $\boldsymbol{d}$, which is the nearest visible point on D. A problem scenario that may benefit from the VkNN query is as follows.

Scenario 1 (Placement of Security Cameras): Suppose that a security company wants to attach k security cameras to k different buildings to monitor a site $\boldsymbol{q}$. Clearly, it would require the monitored site $\boldsymbol{q}$ to be visible to all of these k buildings. Furthermore, the security company may also want the distances from these security cameras to $\boldsymbol{q}$ to be minimized.

In this scenario, the user (the security company) can use the VkNN query to find these k visible nearest buildings. Our incremental VkNN algorithm also allows postconditions to be applied to query results. For example, when a security camera cannot be attached to some of the k nearest buildings, the user can incrementally retrieve more results until the user obtains k buildings that can accommodate security cameras.

Furthermore, we propose a multi-query-point generalization to the VkNN query, called the *aggregate VkNN (AVkNN) query*. An AVkNN query finds k objects with the smallest aggregate visible distances to a given set of query points, rather than a single query point. A problem scenario for the AVkNN query is as follows.

Scenario 2 (Placement of Network Antennas): Suppose that a telecommunication company is searching for a building to install an antenna (or multiple antennas) to provide network access to m different sites. This building must have a line of sight to each of these m sites. Furthermore, since the signal strength has a negative correlation with the distance from an antenna, the company also wants to minimize the worst-case distance to the sites.

In this scenario, the user (the telecommunication company) can use our AVkNN algorithms to find the nearest building visible to the m sites (if exists). In addition, similar to the VkNN algorithms, our AVkNN algorithms are incremental so postconditions can be applied to the problem. The user can incrementally retrieve possible solutions until the first one that satisfies the postconditions is found.

Our investigation of the AVkNN query focuses on three aggregate functions, SUM, MAX and MIN. By exploiting the concept of *aggregate search region* (AGGSR), we are able to apply an incremental retrieval strategy to the AVkNN query. We propose two incremental approaches (sets of algorithms) for the AVkNN query. The first one uses a brute-force strategy, which issues a VkNN query at each query point, although an effective pruning technique based on visible distance is applied to improve the performance. We call this approach *multiple retrieval front (MRF)*. The second approach issues just one aggregate query to retrieve objects from the database and then re-rank the results based on the aggregate visible distance metric. We call this approach *single retrieval front (SRF)*. Our experimental results show that SRF consistently outperforms MRF.

The contributions of this article are summarized as follows:

- formalization and investigation of the VkNN query and the MINVIDIST distance metric;
- two incremental algorithms, POSTPRUNING and PREPRUNING, for processing VkNN queries without pre-computing visibility regions, and an optimality proof of PREPRUNING in terms of the I/O cost;
- a multi-query-point generalization of the VkNN query (i.e., the AVkNN query) with two sets of associated algorithms;
- experimental studies on the VkNN and AVkNN algorithms.

This article is an extended version of our previous paper [18]. In our previous paper, we have proposed the VkNN query and two approaches to processing it, POSTPRUNING and PREPRUNING. In this article, we first provide a new PREPRUNING algorithm which is optimal in terms of the I/O cost. Second, we generalize the VkNN query to a multi-query-point version, the AVkNN query, and propose two approaches for the AVkNN query. Third, we perform a thorough experimental study on the algorithms for both types of queries.

The rest of the article is organized as follows. Section II discusses related work on spatial data structures and queries. Section III provides preliminaries on the MINVIDIST metric, the aggregate k nearest neighbor query and search regions. Section IV presents two algorithms for processing VkNN queries. Section V formulates the AVkNN query and presents two approaches to processing the AVkNN query. Results of our experimental study are reported in Section VI. Finally, Sections VII and VIII give the conclusions and future research directions respectively.

II. RELATED WORK

A. Algorithms to Construct Visibility Regions

Construction of a visibility region (also known as a *visibility polygon*) inside a polygon with obstacles has been investigated in the context of computational geometry. Asano et al. [3] propose a method which requires $\mathcal{O}(n^2)$ time and space for preprocessing and $\mathcal{O}(n)$ to compute the visibility polygon for each view point (n denotes the total number of edges of obstacles). Asano et al. [2] propose an algorithm that runs in $\mathcal{O}(n \log n)$ time and the same result is also independently obtained by Suri and O'Rourke [25]. Heffernan and Mitchell [13] propose an algorithm with the time complexity of $\mathcal{O}(n + h \log h)$ (where h is the number of obstacles). Zarei and Ghodsi [29] propose an algorithm that requires $\mathcal{O}(n^3 \log n)$ time and $\mathcal{O}(n^3)$ space for preprocessing. The algorithm runs in $\mathcal{O}((1 + h') \log(n + |V(\boldsymbol{q})|))$ time, where $|V(\boldsymbol{q})|$ is the size of the visibility polygon $V(\boldsymbol{q})$, and h' is bounded by $\text{MIN}(h, |V(\boldsymbol{q})|)$.

These algorithms efficiently solve the problem of visibility polygon construction, but must rely on preprocessing and/or accessing all obstacles. As a result, they are not suitable for many applications in the domain of spatial databases due to the following reasons: (i) any update will invalidate the preprocessed data; (ii) accessing all objects for each query is expensive.

B. Distance Metrics

We use the R*-tree [4], which is a variant of the popular spatial indexing structure R-tree [12] in our experiments. Our algorithms can also be applied to other hierarchical structures such as the quadtree [24]. An R-tree consists of a hierarchy of *minimum bounding rectangles (MBRs)*, where each corresponds to a tree node and bounds all the MBRs in its sub-tree. Data objects are stored in leaf nodes and they are partitioned based on a heuristic that aims to minimize the I/O cost.

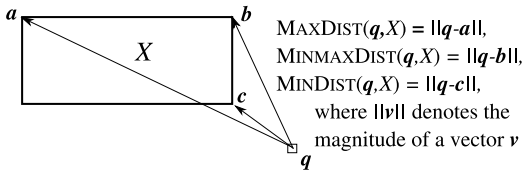


Fig. 2. MINDIST, MAXDIST, and MINMAXDIST metrics

KNN search algorithms using R-trees usually depend on some distance estimators to decide in which order to access the tree nodes and data objects. Figure 2 illustrates commonly used distance estimators [24], such as MAXDIST, MINMAXDIST and MINDIST. The MINDIST between the query point $\boldsymbol{q}$ and an MBR X is the smallest Euclidean distance between $\boldsymbol{q}$ and X. The MAXDIST between $\boldsymbol{q}$ and X is the largest Euclidean distance between $\boldsymbol{q}$ and X. The MINMAXDIST [22] or MAXNEARESTDIST [23] is the greatest possible distance between the nearest object in X and $\boldsymbol{q}$. The MINDIST function is optimistic in the sense that the MINDIST of an MBR is guaranteed to be smaller than or equal to the distance of the nearest object in the MBR. Both MAXDIST and MINMAXDIST are pessimistic [24] because the MAXDIST and MINMAXDIST of an MBR are guaranteed to be greater than or equal to the distance of the nearest object in that MBR.

C. Nearest Neighbor Query Processing

The k nearest neighbor (kNN) query finds k objects nearest to a given query point. A formal definition of the query can be given as follows.

Definition 1 (k Nearest Neighbor (kNN) Query): Given a set $\mathcal{S}$ of objects and a query point $\boldsymbol{q}$, the kNN of $\boldsymbol{q}$ is a set $\mathcal{A}$ of objects such that: (i) $\mathcal{A}$ contains k objects from $\mathcal{S}$; (ii) for any object $X \in \mathcal{A}$ and object $Y \in (\mathcal{S} - \mathcal{A})$, $\text{MINDIST}(\boldsymbol{q}, X) \leq \text{MINDIST}(\boldsymbol{q}, Y)$.

Two well known algorithms for processing kNN queries are *depth-first (DF)* kNN [22] and *best-first (BF)* kNN [14]. They differ in the order of tree traversal. DF-kNN visits tree nodes in a depth-first manner and meanwhile maintains the k nearest objects discovered so far as candidates. The k^{th} nearest object's distance to $\boldsymbol{q}$ is used as a pruning distance to discard subsequent tree nodes and objects. When every node is either visited or discarded, the k objects remaining in the candidate set are the resultant k nearest neighbors (NNs).

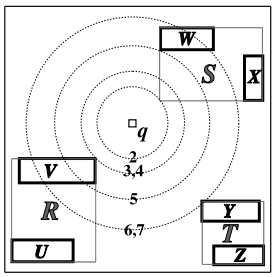
BF-kNN visits tree nodes and data objects in the order of their distances to the query point. Farther nodes are never pruned but scheduled to be visited later on, and they may not be visited at all if the k NNs are discovered first. The main benefit of BF-kNN is threefold: (i) the value of k need not be specified in advance; (ii) the results are ranked according to their distances by default; (iii) the number of visited nodes is minimal (that is, the algorithm is I/O optimal.) Since our VkNN algorithms are based on BF-kNN, we further elaborate the discussion as follows.

Algorithm 1 BF-kNN($Tree$, $\boldsymbol{q}$, k)

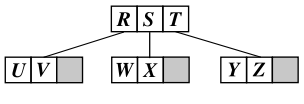
```
1: Create PQ with Tree.ROOT as the first entry
2: Create an empty set A of answers
3: repeat
4:    E ← PQ.POPHEAD()
5:    if E contains an object then
6:       Insert E.OBJ into A
7:    else if E contains a node then
8:       Children ← Tree.GETCHILDNODES(E.NODE)
9:       for all C in Children do
10:         D ← Calculate MINDIST(q, C)
11:         Create NewEntry from C and D
12:         Insert NewEntry into PQ
13:      end for
14:   end if
15: until k objects in A or PQ is empty
16: return A
```

Algorithm 1 gives the detailed steps of BF-kNN. We start with a priority queue PQ with the root node as the first entry and an empty set $\mathcal{A}$ that will contain the resultant k NNs (Lines 1 and 2). If the entry retrieved from PQ is an object, the object is the next NN (Lines 5 and 6); otherwise (the entry contains a tree node) (Line 7), we retrieve the child nodes stored in the node (Line 8). For each of its child nodes, a new entry is created, the MINDIST is calculated and the entry is then inserted into PQ (Lines 9 to 13). The repeat-until loop stops when the k NNs has been discovered or PQ is exhausted (Line 15). Finally, the set $\mathcal{A}$ of resultant k NNs is returned (Line 16).

An example run of the algorithm is given in Figure 3. The upper part of Figure 3(b) shows the R-tree for the dataset in Figure 3(a). The lower part of Figure 3(b) lists the execution steps of BF-kNN. The priority queue PQ keeps all the nodes and data objects to be visited in the order of their distances to $\boldsymbol{q}$. In Step 1, R, S and T are inserted into PQ. In Step 2, S is retrieved from PQ and its two child entries X and W are inserted into PQ. In Step 3, R is retrieved from PQ, and nodes U and V are inserted into PQ. In Step 4, V is retrieved from PQ and it is the first NN. If another NN is needed, the process continues until another data object is discovered. In this manner, an arbitrary number of NNs can be incrementally obtained.



(a) R-tree of objects with MBRs



Step	Priority Queue *PQ*	NNs
1	<***S***,***R***,***T***>	{}
2	<R,***W***,T,***X***>	{}
3	<***V***,W,T,X,***U***>	{}
4	<W,T,X,U>	{V}
5	<T,X,U>	{V,W}
6	<***Y***,X,U,***Z***>	{V,W}
7	<X,U,Z>	{V,W,Y}

(b) R-tree and execution steps of a BF-kNN search

Fig. 3. An example run of BF-kNN

D. Related Spatial Problems

Ferhatosmanoglu et al. [9] propose the *constrained kNN query* which finds the NNs inside a polygon defined as linear constraints (or a disjunction of linear constraints). Although the visibility region can be represented as a disjunction of constraints, it is inefficient to use the constrained kNN algorithm to solve the VkNN problem. This is because the visibility region depends on the location of the query point, i.e., each query point has its unique visibility region. Solving VkNN using the constrained NN query requires an additional step of visibility region computation.

The *nearest surrounder (NS) query* is proposed by Lee et al. [16]. An NS query finds the nearest object for all orientations around the query point $\boldsymbol{q}$. Consequently, only objects visible to $\boldsymbol{q}$ can be an NS. The main difference between the NS query and the VkNN query is that an NS query finds all "visible" objects around the query point whereas the number of visible objects for VkNN is user-determined. Two NS algorithms were proposed: the angle-based *sweep* algorithm and the distance-based *ripple* algorithm. Since both of our VkNN algorithms are distance-based, we further discuss the ripple algorithm as follows.

The ripple algorithm retrieves NS candidates in the order of MINDIST using a priority queue. The algorithm keeps track of the NS set and the associated orientation of each NS candidate discovered so far. Upon retrieval of each object, the NS set is accordingly updated. The algorithm halts when the priority queue is exhausted or it satisfies the following *NS termination check (NS-TC)* conditions: (i) each orientation has an associated NS; (ii) all objects in the priority queue are outside the smallest circle that encloses all NS answers (centered at $\boldsymbol{q}$).

Papadias et al. [20] propose a generalization to the kNN query, called the *aggregate kNN (AkNN) query*. An AkNN query finds k objects with the smallest aggregate distances to a set $\mathcal{Q}$ of query points. Papadias et al. investigate three types of aggregate functions: SUM, MAX and MIN; and propose two approaches for processing AkNN queries, *multiple-* and *single query*. They have shown that the single query algorithm is more efficient than the multiple-query one in terms of I/O cost and response time. This study however does not address AVkNN queries.

The *closest-pair query* in spatial databases [6] involves finding two objects from two different datasets where the distance between them is minimized. The similarity between the closest-pair and aggregate NN problems is that they both involves comparing distances of objects from different reference points (objects). However, the two problems differs in the following ways: (i) the number of aggregate query points is much smaller than the cardinality of the dataset, while the two datasets in a closest-pair query may have similar sizes; (ii) the aggregate query points are usually localized, while the two closest-pair datasets may span the same dataspace.

The *visibility graph* [11] involves problems related to the obstructed distance between two points in a 2D space with obstacles. Specifically, the obstructed distance between two points is the length of the path between the two points that (i) does not pass the interior of any obstacle, and (ii) minimizes the travelling distance. A visibility graph can be constructed by connecting obstacles' corners that are visible to each other. The visibility graph in turn allows the problem of obstructed distance calculations to be solved in a spatial-network manner [21].

Zhang et al. [30] propose a database-oriented solution to spatial problems with obstacles. Their solution does not require a complete visibility graph to be constructed beforehand but creates a *local visibility graph* on the fly. Among a wide range of spatial queries in presence of obstacles, the *obstructed NN (ONN) query* is proposed. The ONN query retrieves k objects with the smallest *obstructed distances* in a setting of polygonal obstacles and point data objects.

Although both VkNN and ONN are NN variants that involve obstacles, they require different techniques. For VkNN, any object blocked by obstacles has the distance of infinity, while ONN instead uses the distance of the shortest detour. Since blocked objects could be returned as ONN results, the visibility-culling strategy used in VkNN algorithms is inapplicable to ONN. The emphasis of the ONN algorithm is the use of a local visibility graph to calculate obstructed distances via Dijkstra's algorithm [7]. For VkNN, the MINVIDIST between $\boldsymbol{q}$ and an object X is the Euclidean distance between $\boldsymbol{q}$ and the nearest visible point on X. One may generalize MINVIDIST as a single-hop variant of the obstructed distance measure. Specifically, any object unreachable by a single hop from $\boldsymbol{q}$ has the MINVIDIST of infinity and is ignored. This property of MINVIDIST eliminates the need for Dijkstra's algorithm. As a result, a visibility graph is not needed for MINVIDIST calculations in VkNN.

Tung et al. [27] propose an obstacle-aware clustering technique. The technique can be used to construct a spatial data structure that is more suitable for spatial queries that use the obstructed distance as the proximity measure [30] than the R-tree [12]. However, the technique requires the visibility graph to be constructed beforehand. As pointed out by Zhang et al. [30], this requirement incurs additional effort to maintain the visibility graph when updating the set of obstacles.

Recently, Gao et al. [10] propose the *visible reverse kNN (VRkNN) query* in a setting of point data objects and rectangular obstacles. A VRkNN query finds all objects with the query point $\boldsymbol{q}$ as a member of the VkNN [18] set. They also propose a VRkNN algorithm which applies the visibility culling concept to a well known RkNN algorithm, the TPL algorithm [26]. Similar to our VkNN algorithms, the VR-kNN algorithm retrieve obstacles using a best-first search to construct the region visible to $\boldsymbol{q}$.

III. PRELIMINARIES

A. The MINVIDIST Metric

In order to formally define MINVIDIST, we first need to define two functions: the *visibility clipping* function and the *shadow* function, whose definitions are given as follow.

The visibility clipping function CLIP is based on the polygon clipping algorithm proposed by Vatti [28]. In Vatti's algorithm, clipping two polygons is done by partitioning the space according to the y-coordinates of the two polygons' vertices. These partitions are then processed in an orderly fashion. For each partition, a partial resultant contour is obtained by scanning for possible intersections between the two polygons. After all partitions are processed, the complete resultant polygon is obtained without post-processing, e.g., sorting the edges.

In this paper, we define CLIP as a function that returns the visible part of an object X with respect to a query point $\boldsymbol{q}$ and a given set $\mathcal{S}$ of objects (functioning as obstacles). That is,

$$\text{CLIP}(\boldsymbol{q}, X, \mathcal{S}) = X - \bigcup_{Y \in \mathcal{S}} \text{SHADOW}(\boldsymbol{q}, Y).$$

The shadow of an object Y is the region obscured by Y from the perspective of a given query point $\boldsymbol{q}$. That is,

$$\text{SHADOW}(\boldsymbol{q}, Y) = \bigcup_{\boldsymbol{y} \in \text{INTERIOR}(Y)} \{\boldsymbol{s} : \boldsymbol{y} \in \overline{\boldsymbol{qs}}\},$$

where INTERIOR(Y) denotes the set of points in Y that are not on the edges. Using only the interior of Y instead of the complete object Y means that Y cannot block itself.

MINVIDIST is the distance between the query point and the nearest visible point of an object, formally defined as follows.

Definition 2 (Minimum Visible Distance — MINVIDIST): Given a set $\mathcal{S}$ of objects (functioning as obstacles), the MINVIDIST between $\boldsymbol{q}$ and X given as

$$\text{MINVIDIST}(\boldsymbol{q}, X, \mathcal{S}) = \begin{cases} \text{MINDIST}(\boldsymbol{q}, X'), & \text{if } X' \neq \emptyset \\ \infty, & \text{otherwise,} \end{cases}$$

where X' is equal to CLIP$(\boldsymbol{q}, X, \mathcal{S})$.

Our incremental processing technique allows us to use only a small subset of $\mathcal{S}$ to calculate the MINVIDIST of an object. Detailed discussion on MINVIDIST calculations in the context of incremental query processing will be given in Section IV.

According to Definition 2, MINVIDIST calculations in 3D can be achieved by replacing the polygon clipping algorithm [28] with a 3D volume clipping algorithm [8]. Discussion on the effect of MINVIDIST calculations in 3D on the proposed VkNN algorithms is given in Section IV-C.

B. Aggregate Nearest Neighbor Query

An aggregate kNN (AkNN) query finds k objects with the smallest aggregate distances to a set $\mathcal{Q}$ of query points. A formal definition of the AkNN query can be given as follows.

Definition 3 (Aggregate kNN Query): Given a set $\mathcal{Q}$ of query points and a set $\mathcal{S}$ of objects, the aggregate kNN of $\mathcal{Q}$ is a set $\mathcal{A}$ of objects such that: (i) $\mathcal{A}$ contains k from $\mathcal{S}$; (ii) for any given X that is in $\mathcal{A}$ and Y in $(\mathcal{S} - \mathcal{A})$, the aggregate MINDIST between $\mathcal{Q}$ and X, AGGMINDIST$(\mathcal{Q}, X)$, is less than or equal to AGGMINDIST$(\mathcal{Q}, Y)$.

The AGGMINDIST function is defined as follows.

Definition 4 (Aggregate Minimum Distance — AGGMINDIST): Given a set $\mathcal{Q}$ of query points and a selection on the aggregate

function, AGGMINDIST$(\mathcal{Q}, X)$ returns either the minimum (MINMINDIST$(\mathcal{Q}, X)$), maximum (MAXMINDIST$(\mathcal{Q}, X)$) or sum (SUMMINDIST$(\mathcal{Q}, X)$) of MINDIST$(\boldsymbol{q}, X)$ for all $\boldsymbol{q}$ in $\mathcal{Q}$.

An example AkNN query is given in Figure 4. According to the sum-aggregate distance (SUMMINDIST) function, the aggregate 3 NNs of $\boldsymbol{q_1}$ and $\boldsymbol{q_2}$ are X, Y and Z, in the order of SUMMINDIST.

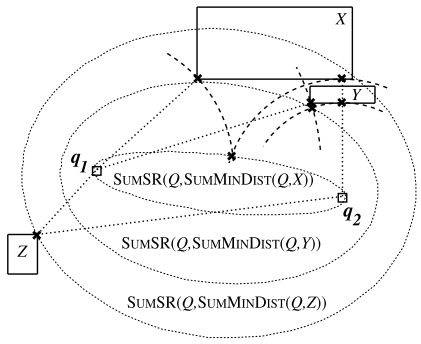


Fig. 4. Aggregate query example with the query set $\mathcal{Q} = \{\boldsymbol{q_1}, \boldsymbol{q_2}\}$ and data objects X, Y and Z. The ellipses show the boundaries of the search regions SUMSR$(\mathcal{Q}$, SUMMINDIST$(\mathcal{Q}, X))$, SUMSR$(\mathcal{Q}$, SUMMINDIST$(\mathcal{Q}, Y))$ and SUMSR$(\mathcal{Q}$, SUMMINDIST$(\mathcal{Q}, Z))$.

We can adapt the BF-kNN algorithm (Algorithm 1) to obtain an algorithm to process AkNN queries by changing the distance function (Line 10 of Algorithm 1) from MINDIST to AGGMINDIST. The BF-search principle in the BF-kNN algorithm is still applicable to AkNN queries. It is because AGGMINDIST is optimistic for all aggregate functions (i.e., SUM, MAX and MIN).

C. Search Region

For each nearest neighbor retrieved from the priority queue, there is a corresponding search region (SR) which delimits the current coverage of the search. According to the example given in Figure 3(a), the region enclosed by Circle 4, $\{\boldsymbol{p} : \|\boldsymbol{q} - \boldsymbol{p}\| \leq$ MINDIST$(\boldsymbol{q}, V)\}$, corresponds to V. We define an SR as a function of $\boldsymbol{q}$ and a coverage c as SR$(\boldsymbol{q}, c) = \{\boldsymbol{p} : \|\boldsymbol{q} - \boldsymbol{p}\| \leq c\}$.

Similarly, for an AkNN query, an aggregate SR (AGGSR) can be formally defined as follows.

Definition 5 (Aggregate Search Region): Given a set $\mathcal{Q}$ of query points, the search region AGGSR$(\mathcal{Q}, c)$ is a set of points $\boldsymbol{p}$ such that AGGMINDIST$(\mathcal{Q}, \boldsymbol{p})$ [1] is less than or equal to c, i.e.,

$$\text{AGGSR}(\mathcal{Q}, c) = \{\boldsymbol{p} : \text{AGGMINDIST}(\mathcal{Q}, \boldsymbol{p}) \leq c\}.$$

Since we consider three aggregate functions: SUM, MAX and MIN; there are three types of AGGSRs: SUMSR, MAXSR and MINSR respectively. For example, Figure 4 shows three SUMSRs of the three objects X, Y and Z. The region SUMSR$(\mathcal{Q}$, SUMMINDIST$(\mathcal{Q}, X))$ is a set of points $\boldsymbol{p}$ where SUMMINDIST$(\mathcal{Q}, \boldsymbol{p})$ is less than or equal to SUMMINDIST$(\mathcal{Q}, X)$. Any object that is overlapped with SUMSR$(\mathcal{Q}$, SUMMINDIST$(\mathcal{Q}, X))$ has a AGGMINDIST smaller than or equal to X. The reverse however does not hold. In other words, SUMSR$(\mathcal{Q}$, SUMMINDIST$(\mathcal{Q}, X))$ may not overlap with all objects that have aggregate distances smaller than SUMMINDIST$(\mathcal{Q}, X)$. For example, SUMMINDIST$(\mathcal{Q}, X)$ is smaller than SUMMINDIST$(\mathcal{Q}, Y)$, but X does not overlap with SUMSR$(\mathcal{Q}$, SUMMINDIST$(\mathcal{Q}, Y))$.

[1] To avoid an excessive number of distance functions, AGGMINDIST$(\mathcal{Q}, \boldsymbol{p})$ also denotes the aggregate value of $\{\|\boldsymbol{q} - \boldsymbol{p}\| : \boldsymbol{q} \in \mathcal{Q}\}$.

Lemma 1: The SUMSR of a SUM-AkNN query is convex.

Proof: According to Definition 5, the SUMSR of a set $\mathcal{Q}$ of query points and the coverage c can be expressed as follows.

$$\text{SUMSR}(\mathcal{Q}, c) = \{\boldsymbol{p} : \sum_{\boldsymbol{q} \in \mathcal{Q}} \|\boldsymbol{q} - \boldsymbol{p}\| \leq c\}.$$

To prove that such a region is convex, we show that all points on the line segment $\overline{\boldsymbol{ab}}$ has to be in the region for any two points $\boldsymbol{a}$ and $\boldsymbol{b}$ in the region, i.e.,

$$\left(\sum_{\boldsymbol{q} \in \mathcal{Q}} \|\boldsymbol{q} - \boldsymbol{a}\| \leq c\right) \bigwedge \left(\sum_{\boldsymbol{q} \in \mathcal{Q}} \|\boldsymbol{q} - \boldsymbol{b}\| \leq c\right).$$

Let $\boldsymbol{x}$ be any point on $\overline{\boldsymbol{ab}}$. That is, $\boldsymbol{x}$ is $\lambda\boldsymbol{a} + \mu\boldsymbol{b}$, where λ and μ are nonnegative real numbers and $\lambda + \mu$ is 1. The sum of distances between $\boldsymbol{x}$ and all query points in $\mathcal{Q}$ is $\sum_{\boldsymbol{q} \in \mathcal{Q}} \|\lambda\boldsymbol{a} + \mu\boldsymbol{b} - \boldsymbol{q}\|$, which is also smaller than or equal to c because of the following relations.

$$\begin{aligned}\sum_{\boldsymbol{q} \in \mathcal{Q}} \|\lambda\boldsymbol{a} + \mu\boldsymbol{b} - \boldsymbol{q}\| &= \sum_{\boldsymbol{q} \in \mathcal{Q}} \|\lambda(\boldsymbol{a} - \boldsymbol{q}) + \mu(\boldsymbol{b} - \boldsymbol{q})\| \\ &\leq \sum_{\boldsymbol{q} \in \mathcal{Q}} \left(\|\lambda(\boldsymbol{a} - \boldsymbol{q})\| + \|\mu(\boldsymbol{b} - \boldsymbol{q})\|\right) \leq \lambda c + \mu c = c\end{aligned}$$

Therefore, any point $\boldsymbol{x}$ on $\overline{\boldsymbol{ab}}$ is also in the SUMSR. ■

Applying the same principle to the MAX function, we will also obtain the same result. By exploiting the convexity of SUMSRs and MAXSRs, we can determine whether we have obtained enough obstacles to calculate the aggregate MINVIDIST (AGGMINVIDIST) of an object. Consequently, we will see that both data retrieval and visibility region construction can be done in an incremental manner. For the MIN aggregate function, MINSRs do not share the same property of convexity. This will be further discussed in Section V-B.

IV. VISIBLE NEAREST NEIGHBOR QUERY

A *Visible k Nearest Neighbor (VkNN) query* finds k nearest objects visible to a query point. We consider the VkNN problem in a setting where (i) data objects are represented as polygons, and (ii) each data objects is also an obstacle. A formal definition of the query is given as follows.

Definition 6 (Visible k Nearest Neighbor (VkNN) Query): Given a set $\mathcal{S}$ of objects (represented by polygons), the visible kNN of $\boldsymbol{q}$ is a set $\mathcal{A}$ of objects such that: (i) $\mathcal{A}$ contains k visible objects from $\mathcal{S}$ (given that the number of visible objects is greater than or equal to k); (ii) for any given X that is in $\mathcal{A}$ and Y that is not in $\mathcal{A}$, $Y \in \mathcal{S} - \mathcal{A}$, MINVIDIST$(\boldsymbol{q}, X, \mathcal{S})$ is less than or equal to MINVIDIST$(\boldsymbol{q}, Y, \mathcal{S})$.

Using MINVIDIST (Definition 2) to rank VkNN results means that invisible objects, which has the distances of infinity, are ignored. Calculating the MINVIDIST between an object X and a query point $\boldsymbol{q}$ does not require the complete $\mathcal{S}$. Lemma 2 can be used to determine a subset $\mathcal{B}$ of $\mathcal{S}$ such that MINVIDIST$(\boldsymbol{q}, X, \mathcal{S})$ yields the same result as MINVIDIST$(\boldsymbol{q}, X, \mathcal{B})$.

Lemma 2: If MINVIDIST$(\boldsymbol{q}, Z, \mathcal{S})$ is greater than MINVIDIST$(\boldsymbol{q}, X, \mathcal{S})$ then MINVIDIST$(\boldsymbol{q}, X, \mathcal{S})$ is equal to MINVIDIST$(\boldsymbol{q}, X, \mathcal{S} - \{Z\})$.

Proof: Let $\boldsymbol{v}$ be a point such that $\|\boldsymbol{q} - \boldsymbol{v}\|$ is equal to MINVIDIST$(\boldsymbol{q}, X, \mathcal{S})$. The line segment $\overline{\boldsymbol{qv}}$ can be one of the two cases: (i) $\boldsymbol{v}$ is the nearest point on X to $\boldsymbol{q}$ (the MINVIDISTs

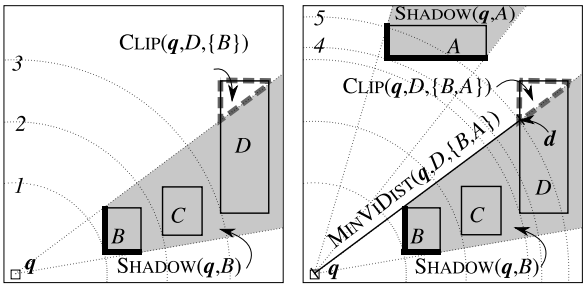
of B and A in Figure 1 for examples), which means that the MINVIDIST of the object does not depend on any other objects; (ii) $\overline{\boldsymbol{qv}}$ is determined by a corner or an edge of an object. Since such object needs to at least have a corner on $\overline{\boldsymbol{qv}}$, the object has to be nearer to $\boldsymbol{q}$ than X. (For the example in Figure 1, the MINVIDIST of D is determined by the top-left corner of B.) ■

Lemma 2 implies that only objects with the MINVIDIST greater than X can be safely ignored (as obstacles) when calculating the MINVIDIST between X and $\boldsymbol{q}$. Thus, a subset $\mathcal{B}$ of $\mathcal{S}$ that makes $\text{MINVIDIST}(\boldsymbol{q}, X, \mathcal{B})$ equivalent to $\text{MINVIDIST}(\boldsymbol{q}, X, \mathcal{S})$ can be given as follows.

$$\mathcal{B} = \{Y : Y \in \mathcal{S}, \text{MINVIDIST}(\boldsymbol{q}, Y, \mathcal{S}) < \text{MINVIDIST}(\boldsymbol{q}, X, \mathcal{S})\}.$$

This lemma allows us to incrementally retrieve VNNs and construct the visibility region at the same time. Consequently, the required amount of visibility knowledge is optimized. An optimistic estimator is used to rule out objects with MINVIDISTs greater than that of the object being considered. For example, if the MINDIST of X is greater than c the MINVIDIST of X has to be greater than c as well. Let us consider Figure 5, where objects in the figure are considered according to the order of MINDIST. In Step 1, we know that MINVIDIST of B is equal to $\text{MINDIST}(\boldsymbol{q}, B)$, because no other object has a MINDIST smaller than B. In Step 2, C is obscured by B so C is not a VNN of $\boldsymbol{q}$. In Step 3, D is found to be partially blocked by B. As B is the only know obstacle, $\text{MINVIDIST}(\boldsymbol{q}, D, \{B\})$ becomes the tentative MINVIDIST of D. Since $\text{CLIP}(\boldsymbol{q}, D, \{B\})$ is farther than A (which is the next object in line), D may not be the next VNN and we have to consider A first. In Step 4, The MINVIDIST of A is calculated. The visible part of A is nearer than $\text{CLIP}(\boldsymbol{q}, D, \{B\})$, so A becomes the second VNN. In Step 5, the MINVIDIST of D is recalculated (with A taken in to consideration this time). The MINVIDIST of D is unaltered and D becomes the third VNN.

Figure 5(b) also shows how the visibility clipping (CLIP) function is used to calculate the MINVIDIST. The MINVIDIST between D and $\boldsymbol{q}$ is is equivalent to the MINDIST between $\text{CLIP}(\boldsymbol{q}, D, \{B, A\})$ and $\boldsymbol{q}$.



(a) Steps 1, 2 and 3 (b) Steps 4 and 5

Fig. 5. MINVIDIST calculations using the visibility clipping function

We now describe two incremental algorithms to process VkNN queries. In our presentation, we assume that all objects are indexed in an R-tree [12], although our algorithms are applicable to many hierarchical spatial indices such as the k-d-tree [5] or the quadtree [24]. We propose two variations, POSTPRUNING (Algorithm 2) and PREPRUNING (Algorithm 3), which differ in the distance estimator used to order entries in the priority queue but produce the same results.

A. The POSTPRUNING Algorithm

The POSTPRUNING algorithm (Algorithm 2) is based on the BF-kNN algorithm (Algorithm 1). In Line 6, the distance of the object entry is set to MINVIDIST. If the newly assigned MINVIDIST is still smaller than the distance of the head of the priority queue[2], the object is added to $\mathcal{A}$ as the next VNN (Lines 7 and 8). Otherwise, the entry is inserted back into the priority queue for reassessment if the distance is not infinity (Lines 9 and 10). In terms of node processing (Lines 12 to 19), MINDIST is used as the estimator for each child node which is the same as the BF-kNN algorithm. The MINDIST metric can be used as a VkNN estimator because MINDIST is also optimistic for VkNN, i.e., the MINDIST of a node is always less than or equal to the object with the smallest MINVIDIST in the node.

Algorithm 2 POSTPRUNING($Tree$, $\boldsymbol{q}$, k)

```
1: Create PQ with Tree.ROOT as the first entry
2: Create an empty set A of answers
3: while PQ is not empty and |A| is less than k do
4:    E ← PQ.POPHEAD()
5:    if E contains an object then
6:       E.DST ← Calculate MINVIDIST(q, E, A)
7:       if E.DST ≤ PQ.HEAD().DST then
8:          Insert E.OBJ into A
9:       else if E.DST is not infinity then
10:         Insert E back into PQ
11:      end if
12:   else if E contains a node then
13:      Children ← Tree.GETCHILDNODES(E.NODE)
14:      for all C in Children do
15:         D ← Calculate MINDIST(q, C)
16:         Create NewEntry from C and D
17:         Insert NewEntry into PQ
18:      end for
19:   end if
20: end while
21: return A
```

Modifying the NS (nearest surrounder) ripple algorithm: POSTPRUNING-NS-TC. In the original definition of the NS ripple algorithm [16], data objects are retrieved from the priority queue according to the MINDIST metric. The NS ripple algorithm can be modified to incrementally retrieve VNNs and to stop after obtaining the k VNNs. This modification is done by applying the MINVIDIST metric and reinserting objects that may not be the next VNN into the priority queue. This modification will result in an algorithm similar to POSTPRUNING (Algorithm 2) with the termination check NS-TC (Section II). We hence call this modification POSTPRUNING-NS-TC.

B. The PREPRUNING Algorithm

The PREPRUNING algorithm (Algorithm 3) is an optimization of POSTPRUNING (Algorithm 2) in terms of the I/O cost. Unlike POSTPRUNING, PREPRUNING applies MINVIDIST to objects as well as index nodes. Index nodes are hence "pre-pruned" according to their visibilities before being visited. At each iteration, we first retrieve the head of PQ (Line 4) and calculate its MINVIDIST (Line 5). We then check whether the updated distance is larger than the distance of the new head of

[2]For brevity, we omit the handling of a marginal case where PQ is empty. This omission is also applied to the rest of algorithms.

PQ (Line 6). If that is the case, we check whether the entry is visible, i.e., the distance is not infinity (Line 7). If the entry is visible, it is reinserted into PQ (Line 8). The entry is discarded if it is found to be invisible. If the updated distance is otherwise smaller than the new head of PQ, we check if the entry is an object (Line 10). If yes, the object is inserted into $\mathcal{A}$ as the next VNN (Line 11); otherwise (an index node), for each child node of the index node, a new entry is created and inserted into PQ (Lines 12 to 19).

Algorithm 3 PREPRUNING($Tree$, $\boldsymbol{q}$, k)

```
1: Create PQ with Tree.ROOT as the first entry
2: Create an empty set A of answers
3: while PQ is not empty and |A| is less than k do
4:    E ← PQ.POPHEAD()
5:    E.DST ← Calculate MINVIDIST(q, E, A)
6:    if E.DST > PQ.HEAD().DST then
7:       if E.DST is not infinity then
8:          Insert E back into PQ
9:       end if
10:   else if E contains an object then
11:      Insert E.OBJ into A
12:   else if E contains a node then
13:      Children ← Tree.GETCHILDNODES(E.NODE)
14:      for all C in Children do
15:         D ← Calculate MINDIST(q, C)
16:         Create NewEntry from C and D
17:         Insert NewEntry into PQ
18:      end for
19:   end if
20: end while
21: return A
```

Note that another possible PREPRUNING variant is to use MINVIDIST$(\boldsymbol{q}, C, \mathcal{A})$ as the distance of a child node C in Line 15. However, the MINVIDIST of C calculated based on $\mathcal{A}$ could be inaccurate, since $\mathcal{A}$ may not contain all objects with MINVIDISTs less than that of C. We thus cannot avoid recalculating the MINVIDIST for every entry retrieved from PQ (Line 5). Since MINVIDIST is significantly more expensive than MINDIST, this modification introduces a higher computational overhead. We will not further consider this PREPRUNING variant in this article.

C. Comparison Between POSTPRUNING and PREPRUNING

We analyze the VkNN query cost in two major components, the I/O and CPU costs. The I/O cost concerns the number of pages retrieved from the disk. The CPU cost is dominated by the MINVIDIST computation.

Generally, we expect POSTPRUNING to be more expensive than PREPRUNING in terms of both I/O and CPU costs for large values of k due to the following reasons. POSTPRUNING does not prune invisible nodes so it has a higher I/O cost than PREPRUNING. In terms of the CPU cost, although the MINVIDIST function (which is much more expensive than MINDIST) is only applied to objects (not to R-tree nodes) for POSTPRUNING, the algorithm ends up with more entries to compute the MINVIDIST. This is because the lack of pruning eventually creates more objects to consider. Furthermore, MINVIDIST also provides a better search ordering than MINDIST on visible nodes. An example comparing the difference that POSTPRUNING and PREPRUNING have in terms of search orders are given in Figures 6 and 7.

Assume that F is recently discovered as the first VNN (after Step 2 in Figure 7(a)). According to Algorithm 2 where MINDIST is used for search ordering, B is searched before A because MINDIST$(\boldsymbol{q}, B)$ is smaller than MINDIST$(\boldsymbol{q}, A)$. In Step 3, I, H and G are inserted into the priority queue PQ. In Step 4, the nearest entry in PQ is I and it is retrieved from the priority queue. Then I is discarded because it is invisible. Node A is now the nearest. Objects C, D and E from A are inserted into PQ in Step 5. Next, D is discovered as the second VNN in Step 6.

Let us now consider the search order (Figure 7(b)) produced by PREPRUNING (Algorithm 3), where MINVIDIST is also applied to nodes. In Step 2, F is discovered as the first VNN. In Step 3, we examine B and find out that MINVIDIST$(\boldsymbol{q}, B, \{F\})$ is greater than MINDIST$(\boldsymbol{q}, A)$ so B is inserted back into PQ and Node A becomes the nearest entry. Objects C, D and E, are inserted into PQ (Step 4), then Object D which currently has the smallest MINVIDIST is discovered as the second VNN (Step 5). PREPRUNING visits fewer nodes than POSTPRUNING, because PREPRUNING is in fact I/O optimal (Theorem 1).

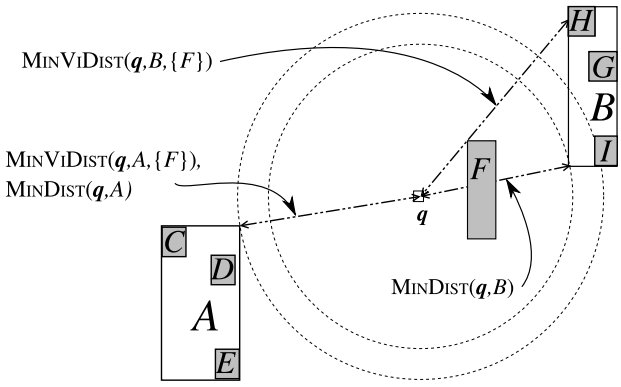


Fig. 6. An R-tree of $\{C, D, E, F, G, H, I\}$; Objects C, D and E are in Node A; H, G and I are in B; F is by itself.

Theorem 1: The I/O cost of the PREPRUNING algorithm is optimal.

Proof: According to Lemma 2, the MINVIDIST assigned to the head entry based on the obstacles retrieved so far (Line 5 of Algorithm 3) is the correct MINVIDIST. This implies that the algorithm *strictly* visits the node with the smallest MINVIDIST before any other nodes. Since the next VNN cannot be retrieved without exploring the node with the current smallest MINVIDIST, the algorithm visits the minimum number of nodes and hence it is I/O optimal. ■

This however does not mean that PREPRUNING always performs better than POSTPRUNING. The I/O cost reduction comes with an additional processing cost, i.e., the computation of

POSTPRUNING

Step	PQ	$\mathcal{A}$
1	$\langle F, B, A\rangle$	$\{\}$
2	$\langle B, A\rangle$	$\{F\}$
3	$\langle I, A, G, H\rangle$	$\{F\}$
4	$\langle A, G, H\rangle$	$\{F\}$
5	$\langle D, G, C, H, E\rangle$	$\{F\}$
6	$\langle G, C, H, E\rangle$	$\{F, D\}$

(a)

PREPRUNING

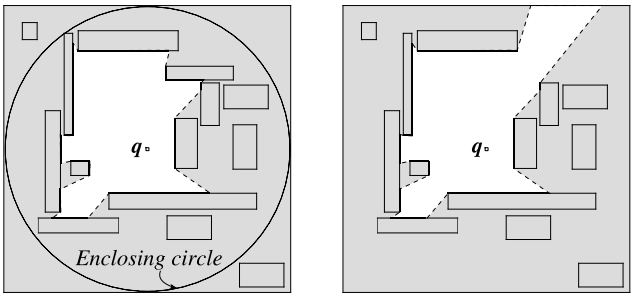
Step	PQ	$\mathcal{A}$
1	$\langle F, B, A\rangle$	$\{\}$
2	$\langle B, A\rangle$	$\{F\}$
3	$\langle A, B\rangle$	$\{F\}$
4	$\langle D, B, C, E\rangle$	$\{F\}$
5	$\langle B, C, E\rangle$	$\{F, D\}$

(b)

Fig. 7. Search orders of the POSTPRUNING and PREPRUNING algorithms for the example in Figure 6

MINVIDIST for every node visited. The MINVIDIST function is more expensive than MINDIST due to the polygon clipping operations. We will further investigate their practical performance especially for different values of k in our experimental study (Section VI-A).

The NS adaptation, POSTPRUNING-NS-TC, has a similar behavior to POSTPRUNING when k is smaller than the number of VNNs. When using the two variants to rank all VNNs in the dataset, POSTPRUNING always visits all R-tree nodes due to the absence of termination check. POSTPRUNING-NS-TC, on the other hand, terminates when (i) the query point is completely surrounded by VNNs, and (ii) the next entry in the priority queue is outside the minimum circle centered at $\boldsymbol{q}$ that encloses all current VNNs candidates (termed as the *enclosing circle*). Figure 8 shows the visibility region (as the white area) in two cases. Figure 8(a) shows a case where the query point is surrounded by VNNs. In this cases, POSTPRUNING-NS-TC terminates when the next entry in the priority queue is outside the enclosing circle. Figure 8(b) shows a case where there exists an angular gap of VNNs. In this case, the enclosing circle is inapplicable and like POSTPRUNING, POSTPRUNING-NS-TC visits all nodes in the R-tree. In both cases, PREPRUNING visits only nodes overlapped with the visibility region. Therefore, PREPRUNING incurs a lower I/O cost than the two POSTPRUNING variants.



(a) Fully surrounded query point (b) Visibility region with a VNN Gap

Fig. 8. Visibility region in two different cases

A setting that could be favorable to POSTPRUNING-NS-TC is when the query point is fully surrounded by VNNs and all objects in the enclosing circle are visible. This could happen when (i) the number of visible of objects is low enough, or (ii) the query point is situated in the middle of a circle formation of objects. In such cases, POSTPRUNING-NS-TC could have a smaller response time than PREPRUNING, since no benefits can be gained from pruning index nodes beforehand.

In a 3D application, the cost of MINVIDIST calculations is higher than the 2D one. This may affect the preference between the POSTPRUNING-NS-TC and PREPRUNING algorithms. In a setting of centralized processing, the cost of MINVIDIST calculations could outweigh the I/O cost. As a result, POSTPRUNING-NS-TC could be the preferred option. In contrast, in a distributed setting, PREPRUNING could perform better than POSTPRUNING-NS-TC, since the I/O cost is determined by the network latency and bandwidth. Experimental studies on VkNN in 3D will be investigated as future work.

V. AGGREGATE VISIBLE NEAREST NEIGHBOR QUERY

In Section I, we have motivated the aggregate visible k nearest neighbor (AVkNN) query, which is a multi-query-point generalization to the VkNN query. A formal definition of the AVkNN query is given as follows.

Definition 7 (Aggregate VkNN (AVkNN) Query): Given a set $\mathcal{S}$ of objects (represented by polygons) and a set $\mathcal{Q}$ of query points, the aggregate visible k NNs of $\mathcal{Q}$ is a set $\mathcal{A}$ of objects such that: (i) $\mathcal{A}$ contains k objects from $\mathcal{S}$ that are visible to $\mathcal{Q}$; (ii) for any given X in $\mathcal{A}$ and Y in $(\mathcal{S}-\mathcal{A})$, $\text{AGGMINVIDIST}(\mathcal{Q}, X, \mathcal{S})$ is less than or equal to $\text{AGGMINVIDIST}(\mathcal{Q}, Y, \mathcal{S})$.

The AGGMINVIDIST function is defined as follows.

Definition 8 (Aggregate MINVIDIST — AGGMINVIDIST*):* Given a set $\mathcal{Q}$ of query points, the distance function $\text{AGGMINVIDIST}(\mathcal{Q}, X, \mathcal{S})$ is the aggregate distance of $\text{MINVIDIST}(\boldsymbol{q}, X, \mathcal{S})$ for all $\boldsymbol{q}$ in $\mathcal{Q}$.

We focus on three aggregate functions, SUM, MAX and MIN, which correspond to three distance functions: SUMMINVIDIST, MAXMINVIDIST and MINMINVIDIST, respectively. Figure 9 shows the visibility regions generated from the set $\mathcal{Q}$ of query points $\{\boldsymbol{q_1}, \boldsymbol{q_2}\}$ and the dataset $\mathcal{S}$, $\{U, V, W, X, Y, Z\}$. The SUMMINVIDIST between $\mathcal{Q}$ and X can be given as $(\text{MINVIDIST}(\boldsymbol{q_1}, X, \mathcal{S}) + \text{MINVIDIST}(\boldsymbol{q_2}, X, \mathcal{S}))$, which is in turn equal to $(\|\boldsymbol{q_1} - \boldsymbol{x_1}\| + \|\boldsymbol{q_2} - \boldsymbol{x_2}\|)$. Similarly, MAXMINVIDIST and MINMINVIDIST of the same object and query points are equal to $\text{MAX}\{\|\boldsymbol{q_1} - \boldsymbol{x_1}\|, \|\boldsymbol{q_2} - \boldsymbol{x_2}\|\} = \|\boldsymbol{q_1} - \boldsymbol{x_1}\|$ and $\text{MIN}\{\|\boldsymbol{q_1} - \boldsymbol{x_1}\|, \|\boldsymbol{q_2} - \boldsymbol{x_2}\|\} = \|\boldsymbol{q_2} - \boldsymbol{x_2}\|$ respectively.

In the same way as the MINVIDIST metric is defined (Definition 2), an object X is *invisible* to $\mathcal{Q}$ *iff* the distance $\text{AGGMINVIDIST}(\mathcal{Q}, X, \mathcal{S})$ is infinity. This implies the following properties.

(i) For SUMMINVIDIST and MAXMINVIDIST, X is invisible to $\mathcal{Q}$ *iff* there exists a query point $\boldsymbol{q}$ in $\mathcal{Q}$ such that $\text{MINVIDIST}(\boldsymbol{q}, X, \mathcal{S})$ is infinity. Figure 9 gives an example where both sum and maximum of $\text{MINVIDIST}(\boldsymbol{q_1}, U, \mathcal{S})$ and $\text{MINVIDIST}(\boldsymbol{q_2}, U, \mathcal{S})$ are infinity because $\text{MINVIDIST}(\boldsymbol{q_1}, U, \mathcal{S})$ is infinity.

(ii) For MINMINVIDIST, X is invisible to $\mathcal{Q}$ *iff* $\text{MINVIDIST}(\boldsymbol{q}, X, \mathcal{S})$ is infinity for all query points $\boldsymbol{q}$ in $\mathcal{Q}$. Figure 9 gives an example where the minimum of $\text{MINVIDIST}(\boldsymbol{q_1}, U, \mathcal{S})$ and $\text{MINVIDIST}(\boldsymbol{q_2}, U, \mathcal{S})$ is non-infinity because $\text{MINVIDIST}(\boldsymbol{q_2}, U, \mathcal{S})$ is non-infinity.

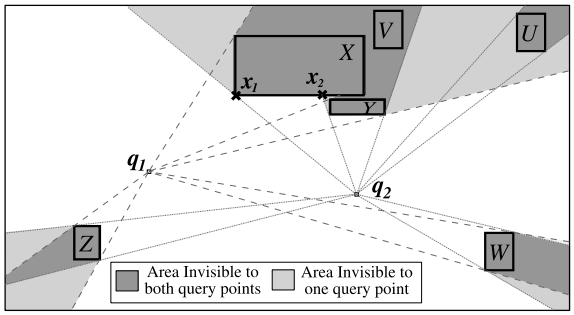
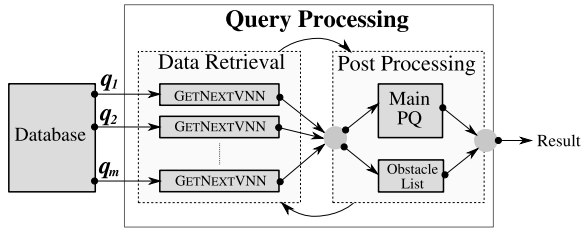


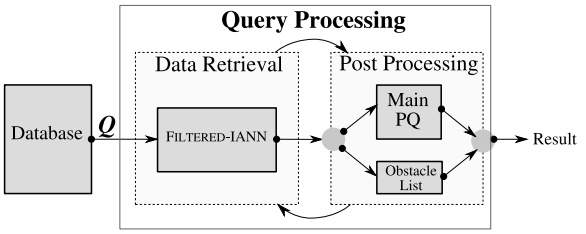
Fig. 9. Visibility regions generated from two query points $\boldsymbol{q_1}$ and $\boldsymbol{q_2}$ with the data set $\mathcal{S}$ of $\{U, V, W, X, Y, Z\}$

The problem of AVkNN cannot be solved using conventional aggregate kNN (AkNN) query algorithms, since each query point has a different set of visible objects and each visible object may have a different visible part for each query point, as illustrated in Figure 9. Therefore, we propose two incremental approaches to processing AVkNN queries: *multiple retrieval front (MRF)* and *single retrieval front (SRF)*, for the three aggregate functions.

A *retrieval front* is a sub-query used to access the database. Figure 10 shows how the two approaches differ in the way they access the database. MRF executes multiple instances of the GETNEXTVNN algorithm (Algorithm 4), which is an algorithm to incrementally retrieve VNNs based on the PREPRUNING algorithm (Algorithm 3), at each query point. The results from different query points are combined in a priority queue. SRF, in contrast, accesses the database via a single FILTERED-IANN query (Algorithm 7). Both approaches have a *post-processing* component. For MRF, the post-processing component is used to reorder objects retrieved from the m query points according to their AGGMINVIDIST to $\mathcal{Q}$. For SRF, the post-processing component is used to reorder objects retrieved from FILTERED-IANN according to AGGMINVIDIST. For both approaches, we maintain all retrieved objects as obstacles to calculate the AGGMINVIDIST of the objects in the priority queue ($MainPQ$). The priority queue $MainPQ$ uses the AGGMINDIST metric as an optimistic estimator and AGGMINVIDIST as the actual ranking distance metric. Therefore, objects retrieved from the head of $MainPQ$ are in the increasing order of AGGMINVIDIST. As a result, both approaches can be used to incrementally retrieve *aggregate VNNs (AVNNs)* from the database.



(a) Multiple Retrieval Front (MRF)



(b) Single Retrieval Front (SRF)

Fig. 10. Structural comparison between MRF and SRF

A. Multiple Retrieval Front (MRF)

In the MRF approach, the query processing is divided into two components: data retrieval and post-processing as shown in Figure 10(a). The data retrieval component consists of m retrieval fronts, where m is the number of query points. Each retrieval front is an instance of GETNEXTVNN (Algorithm 4), which is an incremental VNN retrieval performed at each query point. The post-processing component consists of a priority queue $MainPQ$ and a list $\mathcal{L}$ of obstacles. We use $MainPQ$ to rank objects according to their AGGMINVIDIST to $\mathcal{Q}$, where the AGGMINVIDIST of each object is calculated based on $\mathcal{L}$.

In what follows, we present two MRF algorithms: Algorithm 5 and Algorithm 6. Algorithm 5 can be used to process AVkNN queries for the SUM, MAX and MIN aggregate functions. An optimization can be applied for the MIN aggregate function, which results in Algorithm 6.

Algorithm 4 GETNEXTVNN($Tree$, $\boldsymbol{q}$, PQ, $\mathcal{B}$)

```
1: while PQ is not empty do
2:   E ← PQ.POPHEAD()
3:   E.DST ← MINVIDIST(q, E, B)
4:   if E.DST > PQ.HEAD().DST then
5:     if E.DST is not infinity then
6:       Insert E back into PQ
7:     end if
8:   else if E contains an object then
9:     return (E.OBJ, E.DST)
10:  else if E contains a node then
11:    Children ← Tree.GETCHILDNODES(E.NODE)
12:    for all C in Children do
13:      D ← Calculate MINDIST(q, C)
14:      Create NewEntry from C and D
15:      Insert NewEntry into PQ
16:    end for
17:  end if
18: end while
19: return (null, infinity)
```

We first explain Algorithm 5. The initialization steps (Lines 1 to 8) of the algorithm involves: (i) creating a priority queue $MainPQ$, the list $\mathcal{L}$ of all discovered obstacles and the set $\mathcal{A}$ of results; (ii) retrieving the first VNN for each query point; (iii) initializing the minimum coverage ($MinCov$) to zero.

The main part of query processing takes place in the repeat-until loop (Lines 9 to 30). For each iteration, we check whether the head object of $MainPQ$ is contained by all SRs (Line 10). Consequently, for every $\boldsymbol{q_i}$ in $\mathcal{Q}$, we ensure that any object that may block any part of the head object is discovered. As a byproduct, this condition also ensures that any object that has the AGGMINVIDIST smaller than the head object's AGGMINVIDIST is discovered. As a result, each iteration of the repeat-until loop can be one of the two cases:

(i) **The AGGMINVIDIST of the head object can be calculated (Lines 11 to 17).** For this case, we retrieve the head object from $MainPQ$ and calculate the AGGMINVIDIST of the head object (Lines 11 and 12). Then we check whether the newly calculated distance is smaller than the distance/estimate of the next head object (Lines 13). If yes, the head object is the next AVNN (Line 14). Otherwise, the object is reinserted into $MainPQ$, or discarded if its AGGMINVIDIST is infinity (Lines 15 to 17).

(ii) **More objects need to be retrieved (Lines 19 to 28).** For this case, we select the query with the minimum coverage $MinCov$ (Lines 19 and 20)[3], and insert its corresponding object X_i into $MainPQ$ if it is not a duplicate of a previously retrieved object (Lines 21 to 26). Object X_i is replaced and the coverage of the corresponding query is updated (Line 27). The new X_i is inserted into $\mathcal{B}_i$ (Line 28)[4].

The loop repeats until k AVNNs are found or all VNNs from each of $\boldsymbol{q_i}$ in $\mathcal{Q}$ have been considered (Line 30). Finally, $\mathcal{A}$ is returned as the result (Line 31).

An example run of Algorithm 5 with the aggregate function of SUM is shown in Figure 11. The set $\mathcal{S}$ of objects is $\{X, Y, Z, W\}$, and the set $\mathcal{Q}$ of query points is $\{\boldsymbol{q_1}, \boldsymbol{q_2}\}$. In the initialization

[3]For brevity, we omit the handling of a marginal case where all $Cov_1, Cov_2, ..., Cov_m$ are infinity and $X_1, X_2, ..., X_m$ are $null$. This omission is applied to all MRF algorithms.

[4]We again here omit the handling of a marginal case where X_i is $null$. This omission is also applied to all MRF algorithms

Algorithm 5 MRF-AVkNN($Tree$, $\mathcal{Q}$, k)

```
1: Create MainPQ, an obstacle list L and an answer set A
2: for all q_i in Q = {q_1, q_2, ..., q_m} do
3:    Create a list B_i of obstacles
4:    Create PQ_i with Tree.ROOT as the first entry
5:    (X_i, Cov_i) ← GETNEXTVNN(Tree, q_i, PQ_i, B_i)
6:    Insert X_i into B_i
7: end for
8: MinCov ← 0
9: repeat
10:   if MainPQ is not empty and ∀q_i ∈ Q,
      MainPQ.HEAD().OBJ ⊆ SR(q_i, MinCov) then
11:      E ← MainPQ.POPHEAD()
12:      E.DST ← Calculate AGGMINVIDIST(Q, E.OBJ, L)
13:      if E.DST ≤ MainPQ.HEAD().DST then
14:         Insert E.OBJ into A
15:      else if E.DST is not infinity then
16:         Insert E back into MainPQ
17:      end if
18:   else
19:      MinCov ← MIN_{i=1}^{m} Cov_i
20:      i ← the index i such that Cov_i = MinCov
21:      if X_i is not in L then
22:         Insert X_i into L
23:         D ← Calculate AGGMINDIST(Q, X_i)
24:         Create an entry E from X_i and D
25:         Insert E into MainPQ
26:      end if
27:      (X_i, Cov_i) ← GETNEXTVNN(Tree, q_i, PQ_i, B_i)
28:      Insert X_i into B_i
29:   end if
30: until k objects in A or dataset exhausted
31: return A
```

steps (Lines 1 to 8), Z is discovered as the first VNN of $\boldsymbol{q_1}$ and Y is discovered as the first VNN of $\boldsymbol{q_2}$. The minimum coverage ($MinCov$) of each step is illustrated as two circles, each corresponding to one query point. Each pair of circles are labelled according to its step number. A solid circle denotes the case where an object is discovered via its corresponding query point, and a dotted circle denotes the opposite case. For example, $MinCov$ at Step 1 is denoted as two circles with the labels of "1". The circle centered at $\boldsymbol{q_1}$ is solid because the discovered object Z is retrived via $\boldsymbol{q_1}$. The execution steps are as follow:

Step 1: Since $MainPQ$ is still empty, we go to Line 19 and calculate $MinCov$; then we select the index i such that Cov_i is equal to $MinCov$, i.e., i is one in this case. The VNN of $\boldsymbol{q_1}$, Z, is inserted into $MainPQ$ (Line 25). We retrieve the next VNN of $\boldsymbol{q_1}$ to replace Z and the corresponding coverage Cov_i is updated (Line 27).

Step 2: The priority queue $MainPQ$ has only one object, Z, in it ($MainPQ = \langle Z \rangle$). We still cannot determine the SUMMINVIDIST of Z because Z is not yet contained by all SRs. As a result, we need to retrieve more objects to expand the SRs. Object Y which is the next VNN of $\boldsymbol{q_2}$, is inserted into $MainPQ$. We then retrieve the next VNN of $\boldsymbol{q_2}$ to replace Y.

Step 3: [$MainPQ = \langle Y, Z \rangle$.] Object X is discovered via $\boldsymbol{q_2}$ and is inserted into $MainPQ$.

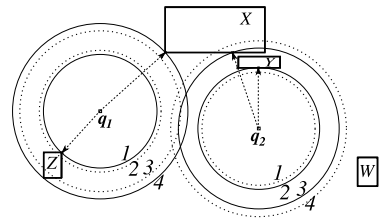
Step 4: [$MainPQ = \langle X, Y, Z \rangle$.] Object X is discovered via $\boldsymbol{q_1}$ but it is discarded because it is a duplicate.

Step 5: [$MainPQ = \langle X, Y, Z \rangle$.] Object W is retrieved via $\boldsymbol{q_2}$ and inserted into $MainPQ$.

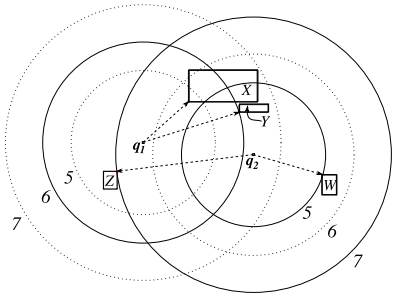
Step 6: [$MainPQ = \langle X, Y, Z, W \rangle$.] Object Y is discovered via $\boldsymbol{q_1}$ but discarded.

Step 7: [$MainPQ = \langle X, Y, Z, W \rangle$.] Object Z is discovered via $\boldsymbol{q_2}$ but discarded.

Step 8: [$MainPQ = \langle X, Y, Z, W \rangle$.] At this point, we have obtained enough obstacles to calculate the SUMMINVIDIST of X, the head object of $MainPQ$, based on the obstacle list of $\langle X, Y, Z, W \rangle$. Object X is retrieved and its SUMMINVIDIST is calculated (Lines 11 and 12). Since the SUMMINVIDIST of X is smaller than the next nearest item (Line 13), X is added to $\mathcal{A}$ (Line 14).



(a) Steps 1 to 4



(b) Steps 5 to 7

Fig. 11. MRF example

Algorithm 6 MRF-MIN-AVkNN($Tree$, $\mathcal{Q}$, k)

```
1: Create an empty set A of answers
2: for q_i in Q do
3:    Create a list B_i of obstacles
4:    Create PQ_i with Tree.ROOT as the first entry
5:    (X_i, Cov_i) ← GETNEXTVNN(Tree, q_i, PQ_i, B_i)
6:    Insert X_i into B_i
7: end for
8: MinCov ← 0
9: repeat
10:   MinCov ← MIN_{i=1}^{m} Cov_i
11:   i ← the index i such that Cov_i = MinCov
12:   if X_i is not in A then
13:      Insert X_i into A
14:   end if
15:   (X_i, Cov_i) ← GETNEXTVNN(Tree, q_i, PQ_i, B_i)
16:   Insert X_i into B_i
17: until k objects in A or dataset exhausted
18: return A
```

For the MIN-AVkNN query, we can improve the algorithm by removing the post-processing part. This is because, if X is a VNN of $\boldsymbol{q_i}$ and has never been previously discovered as a VNN of any $\boldsymbol{q_j}$ in $\mathcal{Q}$ (where i is not equal to j), MINVIDIST($\boldsymbol{q_i}, X, \mathcal{S}$) must be smaller than or equal to any MINVIDIST($\boldsymbol{q_j}, X, \mathcal{S}$). That is, MINVIDIST($\boldsymbol{q_i}, X, \mathcal{S}$) is equal to MINMINVIDIST($\mathcal{Q}, X, \mathcal{S}$). This improved algorithm is shown in Algorithm 6. In order to

find the next AVNN, it is sufficient to always look for X_i that has the smallest MINVIDIST to $\boldsymbol{q_i}$ (Lines 10 and 11) and ensure that its not a duplicate (Lines 12 to 14). After that, we replace the current X_i by the next VNN of $\boldsymbol{q_i}$ and then update Cov_i (Line 15). Object X_i is then inserted into its corresponding obstacle list $\mathcal{B}_i$ (Line 16). The loop (Lines 9 to 17) repeats until k neighbors are discovered or the dataset is exhausted for all query points.

B. Single Retrieval Front (SRF)

In this section, we present two *single retrieval front (SRF)* algorithms: (i) Algorithm 8 for the SUM and MAX aggregate functions; (ii) Algorithm 9 for the MIN aggregate function.

Algorithm 8 accesses the database via a single *filtered incremental aggregate NN* algorithm (FILTERED-IANN, Algorithm 7), which adapts a similar strategy to BF-kNN (Algorithm 1). However, Algorithm 7 has following differences from Algorithm 1: (i) visibility filtering (Lines 3 and 4) is applied to avoid needlessly processing entries (nodes/objects) invisible to all query points, and (ii) MINDIST is replaced by AGGMINDIST.

Although Algorithm 7 contains visibility filtering, objects retrieved via the algorithm are still ranked according to the AGGMINDIST metric. The post-processing component is used to re-rank objects according to the AGGMINVIDIST metric.

Algorithm 7 FILTERED-IANN($Tree$, $\mathcal{Q}$, PQ, $\mathcal{B}$)

```
1: while PQ is not empty do
2:    E ← PQ.POPHEAD()
3:    if E is blocked by B for all q in Q then
4:       Discard E
5:    else if E contains an object then
6:       return (E.OBJ, E.DST)
7:    else if E contains a node then
8:       Children ← Tree.GETCHILDNODES(E.NODE)
9:       for all C in Children do
10:         D ← Calculate AGGMINDIST(Q, C)
11:         Create NewEntry from C and D
12:         Insert NewEntry into PQ
13:      end for
14:   end if
15: end while
16: return (null, infinity)
```

The initialization steps (Lines 1 to 3) of Algorithm 8 involves: (i) creating a priority queue PQ for the FILTERED-IANN query (Line 1), $MainPQ$, an obstacle list $\mathcal{B}$ and an answer set $\mathcal{A}$ for post-processing of the retrieved objects (Line 2); (ii) initializing the coverage Cov to zero (Line 3).

Similar to the MRF counterpart, the query-processing loop (Lines 4 to 21) of the algorithm consists of the data retrieval and the post-processing components. When $MainPQ$ is not empty (Line 5), we process the retrieved object by calculating the AGGMINVIDIST of the head object of $MainPQ$ if the following two criteria are satisfied.

(i) The head object $MainPQ$.HEAD().OBJ is confined in AGGSR($\mathcal{Q}, Cov$) (Definition 5).

(ii) All query points $\boldsymbol{q}$ in $\mathcal{Q}$ are contained by AGGSR($\mathcal{Q}, Cov$). Specifically, Cov is greater than or equal to the minimum coverage bound c_b that makes AGGSR($\mathcal{Q}, c_b$) confine all query points in $\mathcal{Q}$. The value of c_b calculated as MAX{AGGMINDIST($\mathcal{Q}, \boldsymbol{q}$) : $\boldsymbol{q} \in \mathcal{Q}$}.

For the SUM and MAX aggregate functions, the AGGSRs are convex (Lemma 1). By imposing these two criteria, we ensure that any object that may obscure any part of the head object is discovered. Therefore, the AGGMINVIDIST of an object is calculated only when all relevant obstacles are known.

Each iteration of the repeat-until loop (Lines 4 to 21) can be one of the two cases:

(i) **The AGGMINVIDIST of the head object can be calculated (Lines 6 to 12).** For this case, we first retrieve the object E at the head of $MainPQ$ and calculate its AGGMINVIDIST (Lines 6 and 7). Second we check if the AGGMINVIDIST of E is still smaller than the distance of the current head object (Line 8). The object becomes the next NN if that is the case (Line 9). The object is otherwise inserted back into $MainPQ$ if the distance is not infinity (Line 11).

(ii) **More objects need to be retrieved (Lines 14 to 19).** For this case, we retrive a new object X and update Cov via FILTERED-IANN (Line 14). If the object X is not $null$ then X is inserted into $\mathcal{B}$ (Line 16) and an entry is created according to X and Cov, which is AGGMINDIST($\mathcal{Q}, X$) (Line 17). The new entry is inserted into $MainPQ$ (Line 18).

The loop (Lines 4 to 21) repeats until k of AVNNs are retrieved or the dataset is exhausted.

Algorithm 8 SRF-AVkNN($Tree$, $\mathcal{Q}$, k)

```
1: Create PQ with Tree.ROOT as the first entry
2: Create MainPQ, an obstacle list B and an answer set A
3: Cov ← 0
4: repeat
5:    if MainPQ is not empty and
      ∀q ∈ Q, q ∈ AGGSR(Q, Cov) and
      MainPQ.HEAD().OBJ ⊆ AGGSR(Q, Cov) then
6:       E ← MainPQ.POPHEAD()
7:       E.DST ← AGGMINVIDIST(Q, E.OBJ, B)
8:       if E.DST ≤ MainPQ.HEAD().DST then
9:          Insert E.OBJ into A
10:      else if E.DST is not infinity then
11:         Insert E back into MainPQ
12:      end if
13:   else
14:      (X, Cov) ← FILTERED-IANN(Tree, Q, PQ, B)
15:      if X is not null then
16:         Insert X into B
17:         Create an entry E from X and Cov
18:         Insert E into MainPQ
19:      end if
20:   end if
21: until k objects in A or dataset exhausted
22: return A
```

Figure 12 shows how Algorithm 8 runs on the example in Figure 11. The aggregate function is SUM. The execution steps are as follow:

Step 1: Since $MainPQ$ is initially empty, we skip to Line 14. Object X is retrieved via an FILTERED-IANN call and inserted into $\mathcal{B}$ and $MainPQ$ with the distance of SUMMINDIST($\mathcal{Q}, X$) (Lines 14 to 19).

Step 2: [$MainPQ = \langle X \rangle$.] We cannot yet calculate the SUMMINVIDIST of X because a part of X is still outside the current AGGSR (Ellipse 1). We continue to retrieve the next ANN, Y, and insert it into $MainPQ$.

Step 3: [$MainPQ = \langle X, Y \rangle$.] Object Z which is the next aggregate NN to $\mathcal{Q}$ is retrieved and inserted into $MainPQ$.

Step 4: [$MainPQ = \langle X, Y, Z \rangle$.] Object W is retrieved and inserted into $MainPQ$.

Step 5: [$MainPQ = \langle X, Y, Z, W\rangle$.] The SUMMINVIDIST of X, the current head object, can be calculated because X is inside the AGGSR. We calculate the SUMMINVIDIST based on the four obstacles we have retrieved (X, Y, Z and W). The SUMMINVIDIST of X is smaller than the SUMMINDIST of Y, the next head of $MainPQ$, so X is the first AVNN of $\mathcal{Q}$.

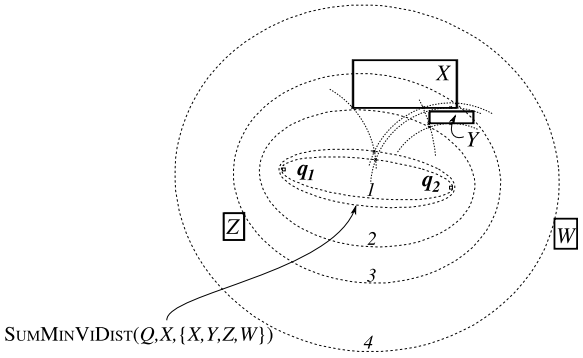


Fig. 12. SRF example

For the MIN aggregate function, MINSR is concave. Algorithm 8, which relies on the AGGSR convexity, is thus no longer applicable. In this case, we have formulated an alternative algorithm which exploits a special property of the MINMINVIDIST function. The distance MINMINVIDIST between X and $\mathcal{Q}$ is the minimum of MINVIDIST$(\boldsymbol{q}, X, \mathcal{S})$ for all query points in $\mathcal{Q}$, where $\mathcal{S}$ is the set containing all objects in the dataset. It is therefore sufficient to use only objects nearer to $\mathcal{Q}$ to determine the MINMINVIDIST between an object and $\mathcal{Q}$. In other words, the query processing can be done in the same manner as PREPRUNING (Algorithm 3). Specifically, the AVkNN algorithm for MIN (Algorithm 9) is obtained by replacing: (i) MINVIDIST by MINMINVIDIST (Line 5), and (ii) MINDIST by MINMINDIST (Line 15).

Algorithm 9 SRF-MIN-AVkNN($Tree$, $\mathcal{Q}$, k)

```
1: Create PQ with Tree.ROOT as the first entry
2: Create an empty set A of answers
3: while PQ is not empty and |A| is less than k do
4:    E ← PQ.POPHEAD()
5:    E.DST ← Calculate MINMINVIDIST(Q, E, A)
6:    if E.DST > PQ.HEAD().DST then
7:       if E.DST is not infinity then
8:          Insert E back into PQ
9:       end if
10:   else if E contains an object then
11:      Insert E.OBJ into A
12:   else if E contains a node then
13:      Children ← Tree.GETCHILDNODES(E.NODE)
14:      for all C in Children do
15:         D ← Calculate MINMINDIST(Q, C)
16:         Create NewEntry from C and D
17:         Insert NewEntry into PQ
18:      end for
19:   end if
20: end while
21: return A
```

C. Analysis on the MRF and SRF Approaches

We analyze the two approaches using three parameters: (i) the number m of query points; (ii) the number k of AVNNs required; (iii) the sparsity of the query points (defined as the span s of the $s \times s$ square that confines the query points).

Our analysis includes both I/O and CPU costs. The I/O cost is the cost for accessing nodes in the R-Tree. The CPU cost is dominated by the visibility computation.

The number m of query points has a positive correlation to the I/O cost of MRF because MRF executes a VkNN query for each query point. Since SRF uses a single query to retrieve objects, m should have no effect on the I/O cost of SRF. The CPU costs of both SRF and MRF are proportional to m, because the cost of AGGMINVIDIST computation is proportional to m.

A larger k means more nodes to retrieve and distances to compute. Hence, both I/O and CPU costs increase as k increases regardless of whether the algorithm is MRF or SRF based. The incremental I/O and CPU costs for retrieving the next VNN also has a positive correlation with k. This is because there are more obstacles involved in the MINVIDIST computation and more invisible objects or nodes to prune due to more obstacles as k increases.

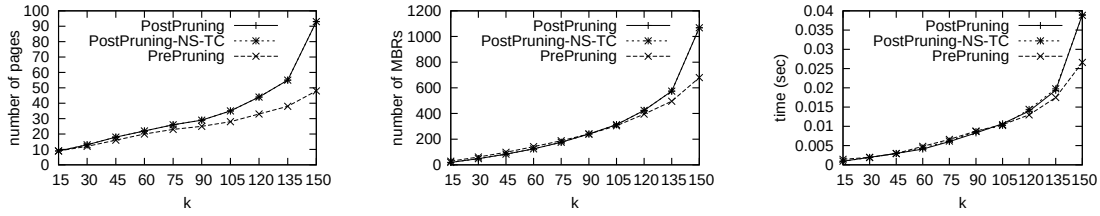
The effect of the sparsity of the query points depends on the aggregate function. For SUM and MAX aggregate functions, the query has to consider more objects in order to obtain k AVNNs for a more scattered $\mathcal{Q}$. The effect is opposite for MIN-AVkNN, i.e., the query has to consider fewer objects in order to obtain k AVNNs for a more scattered $\mathcal{Q}$. This is because more scattered query points means that there are less common objects in the sets of visible objects from different query points. According to Definition 8, X being visible to $\mathcal{Q}$ requires: (i) X to be completely visible to $\mathcal{Q}$ for SUM and MAX; and (ii) X to be partially visible to $\mathcal{Q}$ for MIN. As $\mathcal{Q}$ becomes more scattered, it is harder for an object to be visible to all query points in $\mathcal{Q}$ but easier to be visible to at least one of query points in $\mathcal{Q}$. This affects MRF and SRF algorithms in the same manner.

VI. EXPERIMENTAL STUDY

In this section, we report the result of our experimental study. We use both synthetic and real datasets. We generate datasets with different cardinalities. The default cardinality we use in the experiments is 150,000. Each dataset contains rectangles that are distributed uniformly at random in a space of $10,000 \times 10,000$ square units. The width and height of each rectangle vary from 0.5 to 10 units randomly. The real dataset has $556,696$ census blocks from Iowa, Kansas, Missouri and Nebraska in a space of $10,000 \times 10,000$ square units. Each dataset is stored in a disk-based R*-tree with a disk page size of 4 KB. Each R*-tree has the buffer capacity of 5% of its size. Each experiment is conducted on 20 randomly located queries and the reported result is the average result of the 20 queries.

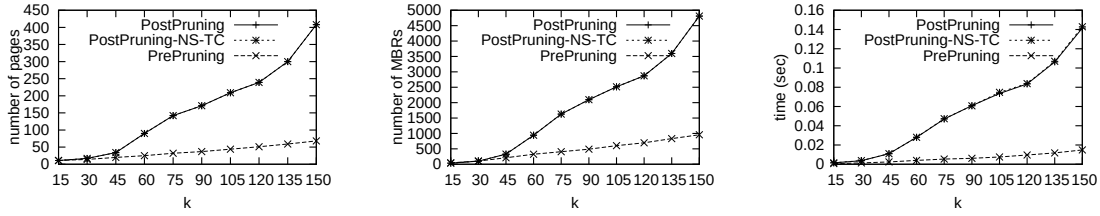
A. Experiments on the VkNN Algorithms

This subsection presents a performance comparison between the two POSTPRUNING variants (Section IV-A) and the PREPRUNING algorithm (Section IV-B). The two POSTPRUNING variants are the (standard) POSTPRUNING algorithm described in Algorithm 2 and the modification of the NS ripple algorithm, POSTPRUNING-NS-TC. We vary two parameters, the number k of VNNs and the cardinality n of the dataset.



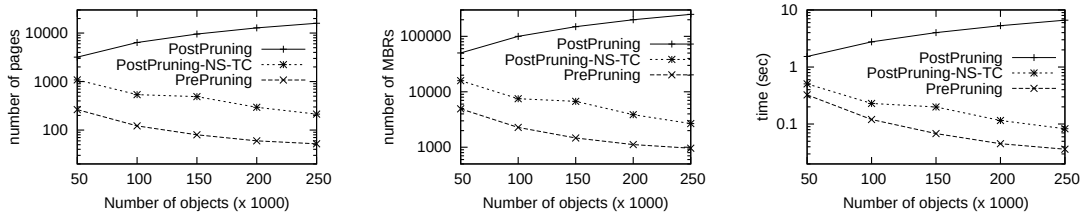
(a) I/O cost (b) Visibility computation (CPU) cost (c) Total response time

Fig. 13. The effect of k on a synthetic dataset of $150,000$ rectangles



(a) I/O cost (b) Visibility computation (CPU) cost (c) Total response time

Fig. 14. The effect of k on a real dataset containing $556,696$ census blocks from Iowa, Kansas, Missouri and Nebraska



(a) I/O cost (b) Visibility computation (CPU) cost (c) Total response time

Fig. 15. The effect of n on synthetic datasets

1) Effect of k: In this experiment, we study the effect of k on the I/O cost, CPU cost and total response time. For both datasets, we vary the k value from 15 to 150 with an increment of 15. Figure 13 shows the result for the synthetic dataset with the default cardinality. For all cost measures, POSTPRUNING and POSTPRUNING-NS-TC do not produce any noticeable difference when k is smaller than the number of VNNs. The NS termination check provides benefit only when we use the VkNN query to rank all visible objects. We therefore focus our comparison on POSTPRUNING and PREPRUNING in this experiment.

For all cost measures, POSTPRUNING and PREPRUNING perform similarly when k is small. As k increases, the cost of POSTPRUNING increases more rapidly than that of PREPRUNING. This is because, as more VNNs are retrieved, the ratio between visible and invisible nodes becomes greater. These invisible nodes are pruned by PREPRUNING but not by POSTPRUNING.

In terms of the I/O cost (Figure 13(a)), PREPRUNING always performs better than POSTPRUNING because PREPRUNING is optimal in terms of the I/O cost (Theorem 1).

In terms of the CPU (visibility computation) cost (Figure 13(b)), for k values under 90 PREPRUNING has a slightly higher cost than POSTPRUNING. This is because PREPRUNING applies the MINVIDIST function to nodes as well as objects while the MINVIDIST function is applied to only objects for POSTPRUNING. As more VNNs are retrieved, POSTPRUNING has more entries to consider than PREPRUNING because many nodes are pruned by PREPRUNING.

The total response time is shown in Figure 13(c). We observe that the two algorithms perform similarly when k is small. When k is greater than 135 the benefit of pruning invisible nodes becomes notable and PREPRUNING outperforms POSTPRUNING more and more. In summary, PREPRUNING has a better performance and scales better than POSTPRUNING.

The same experiment is conducted on the real dataset and the result is shown in Figure 14. Similar to the results from the synthetic dataset, PREPRUNING scales better than POSTPRUNING for all measures. The cost difference between the two algorithms is much larger than that of the synthetic dataset. This is because the real dataset has a greater density than the synthetic dataset. The higher density consequently accents the difference between the results produced by the MINDIST and MINVIDIST distance functions.

2) Effect of n: In this experiment, we study the effect of n by using POSTPRUNING, POSTPRUNING-NS-TC and PREPRUNING to rank all visible objects for each n value. We vary n from 50,000 to 250,000 with an increment of 50,000. Figure 15 shows that the I/O cost, CPU cost and total response time of POSTPRUNING increase as n increases, while the costs for POSTPRUNING-NS-TC and PREPRUNING decrease. This is because POSTPRUNING visits every node. Increasing the number of objects means a larger R*-Tree and more nodes for POSTPRUNING to visit. POSTPRUNING-NS-TC has lower costs than POSTPRUNING because POSTPRUNING-NS-TC terminates the search when all possible VNNs candidates are considered.

For PREPRUNING, although the NS-TC is not applied, the algorithm achieves lower costs than POSTPRUNING-NS-TC. This

is because, PREPRUNING visits only nodes overlapped with the visibility region. The costs of POSTPRUNING-NS-TC and PREPRUNING reduce as n increases because of the negative correlation between the number of VNNs and n as shown in Figure 16. In summary, PREPRUNING visits fewer nodes, performs less visibility computation and has a smaller total response time than the two POSTPRUNING variants. Specifically, PREPRUNING has a threefold smaller response time than POSTPRUNING-NS-TC.

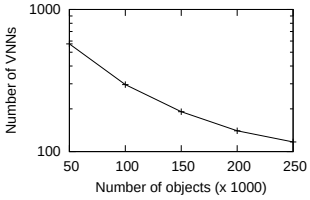


Fig. 16. Number of VNNs vs dataset size

3) Summary: PREPRUNING has a better performance than POSTPRUNING and POSTPRUNING-TC-NC. When the number of obstacles is small, the two POSTPRUNING variants may have a smaller total response time than PREPRUNING, however, the cost difference is negligible.

B. Experiments on the AVkNN Algorithms

This subsection presents performance comparisons between two sets of AVkNN algorithms, MRF and SRF, in terms of the I/O cost and total response time. For both MRF and SRF, the total response time is significantly dominated by the CPU cost, and thus the CPU cost can be deduced from the total response time. Therefore, in this section, we only present the total response time but not the CPU cost.

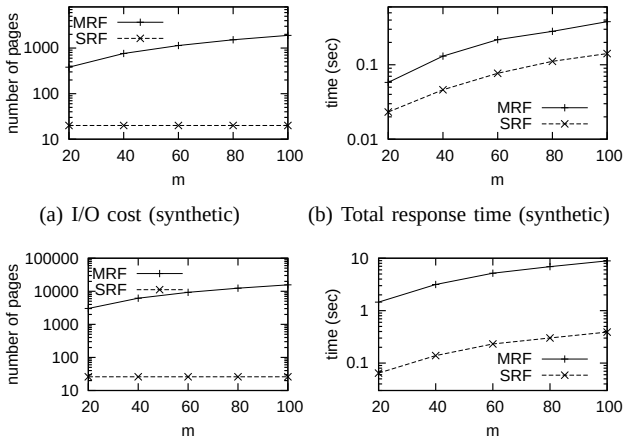
In the experiments, we vary the following parameters: (i) the number m of query points; (ii) the value of k; (iii) the sparsity of the query points (defined as the span s of the $s \times s$ square that confines the query points). The default values of m, k and s are 40, 60 and 1 respectively.

We omit the result on the effect of n due to the fact that n affects both MRF and SRF in the same way. This is because the pre-pruning strategy is applied in all MRF and SRF algorithms.

For MRF, we use Algorithm 5 for SUM-AVkNN and MAX-AVkNN, and Algorithm 6 for MIN-AVkNN. For SRF, we use Algorithm 8 for SUM-AVkNN and MAX-AVkNN, and Algorithm 9 for MIN-AVkNN. For both MRF and SRF, SUM-AVkNN and MAX-AVkNN only differ in the aggregate function.

1) Effect of m: We vary m from 20 to 100 with an increment of 20. Figures 17(a) and 17(c) show the result in terms of the I/O cost. The I/O cost of MRF increases as m increases, while the I/O cost of SRF remains stable. MRF has a higher I/O cost than SRF. This is because MRF executes a VkNN query on each query point while SRF executes a single query. Figures 17(b) and 17(d) show the result in terms of the total response time. The total response time of SRF increases as m increases and SRF outperforms MRF. This is because changes in the value of m affect the AGGMINVIDIST calculation costs. The I/O cost of MRF is always higher than the I/O cost of SRF so we omit presenting I/O costs for the rest of the experiments.

The result for the MAX-AVkNN query is shown in Figure 18. The total response times of MRF and SRF increase as m increases and SRF outperforms MRF, which are similar to the results

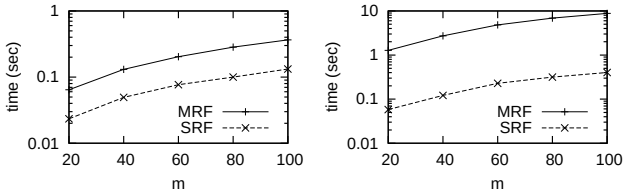


(a) I/O cost (synthetic) (b) Total response time (synthetic)

(c) I/O cost (real) (d) Total response time (real)

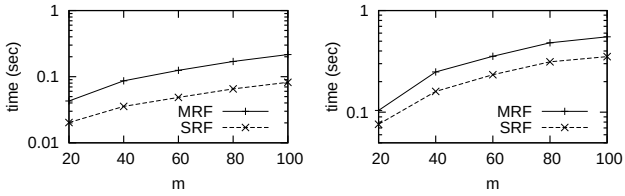
Fig. 17. Effect of m on sum-aggregate VkNN query

from the SUM-AVkNN query. As discussed earlier, the MAX-AVkNN and SUM-AVkNN queries use the same algorithm and only differ in the aggregate distance function. Consequently, they both produce similar results for all settings in our experiments. We thus omit MAX-AVkNN results from the rest of the experiments. The result for the MIN-AVkNN query is shown in Figure 19. SRF continues to perform better than MRF.



(a) Total response time (synthetic) (b) Total response time (real)

Fig. 18. Effect of m on max-aggregate VkNN query

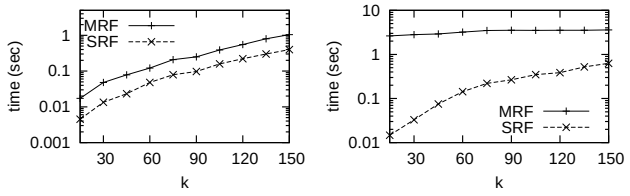


(a) Total response time (synthetic) (b) Total response time (real)

Fig. 19. Effect of m on min-aggregate VkNN query

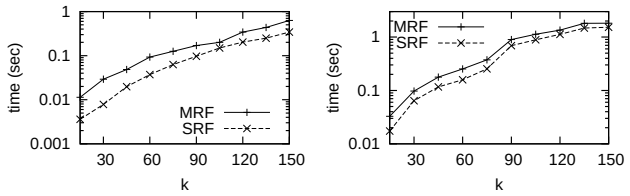
2) Effect of k: We vary k from 15 to 150 with an increment of 15. According to the SUM-AVkNN and MIN-AVkNN query results in Figures 20 and 21, respectively, the total response time increases as k increases for both algorithms, and SRF performs better than MRF for both datasets. However, the increase in the total response time for MRF-SUM-AVkNN on the real dataset (Figure 20(b)) is slower than the others. It is recorded that the total response time was increased from 2.635 to 3.593 seconds as the value of k increased from 15 to 150. In this setting, the slow increase is due to the fact that a large number of objects (functioning as obstacles) has to be retrieved before the first AVNN can be returned. This effect is apparent in the real dataset

because the distribution and sizes of data objects are less uniform than those of the synthetic dataset.



(a) Total response time (synthetic) (b) Total response time (real)

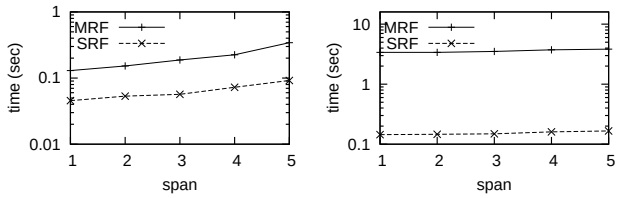
Fig. 20. Effect of k on sum-aggregate VkNN query



(a) Total response time (synthetic) (b) Total response time (real)

Fig. 21. Effect of k on min-aggregate VkNN query

3) Effect of sparsity of query points: In this experiment, we study the effect of the sparsity of query points by varying the span s of the query set from 1 to 5 units with an increment of 1 unit. Figures 22 and 23 show that SRF continues to outperform MRF for the SUM-AVkNN and MIN-AVkNN queries. Figure 22(a) shows that the total response time gradually increases as s increases for the SUM-AVkNN on the synthetic dataset. This is because a greater value of s produces a greater difference between sets of visible objects of the query points. Consequently, we need to retrieve more objects and nodes in order to find the k nearest ones visible to all query points. The result for the real dataset is shown in Figure 22(b). The increase in total response time is less than that of the synthetic dataset.

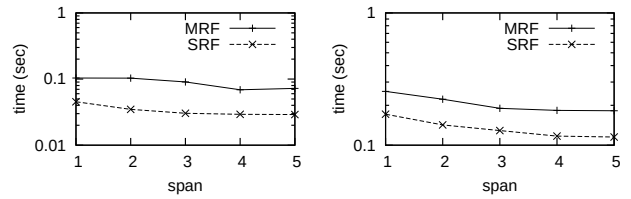


(a) Total response time (synthetic) (b) Total response time (real)

Fig. 22. Effect of sparsity of query points on sum-aggregate VkNN query

The result for the MIN-AVkNN query is shown in Figure 23. The span s has a negative correlation with the total response time for both algorithms and both datasets. An increase in s provides a greater difference in perspectives between query points. A greater difference in perspectives provides more objects visible to $\mathcal{Q}$. This is because an object needs to be to visible to only one of the m query points to be visible to $\mathcal{Q}$ for the MIN-AVkNN query. Therefore, for both MRF and SRF, the number of objects and nodes required to be considered in order to find k visible objects is reduced.

4) Summary: SRF is superior to MRF in terms of the I/O cost. The difference between the total response times of the two approaches is smaller than that of the I/O cost. The total response



(a) Total response time (synthetic) (b) Total response time (real)

Fig. 23. Effect of sparsity of query points on min-aggregate VkNN query

time for processing AVkNN queries increases as k or m increases for all aggregate functions. The total response time decreases as s increases for the MIN function and increases as s increases for SUM and MAX. We conclude that SRF is a better method for AVkNN query processing than MRF.

VII. Conclusions

In this article, we investigated the *visible k nearest neighbor (VkNN)* problem and a distance function called *minimum visible distance* (MinViDist), which is the distance between a query point to the nearest visible point of an object. Furthermore, we presented two VkNN algorithms, PrePruning and PostPruning. Both algorithms build up the visibility knowledge incrementally as the visible nearest objects are retrieved. PostPruning uses MinViDist for result ranking and MinDist for branch ordering. PrePruning uses MinViDist for both. It is shown in the experimental results that PrePruning scales better than PostPruning in terms of the CPU and I/O costs as k becomes larger or the density of the dataset increases.

We also proposed a multiple query point generalization to the VkNN query according to three aggregate distance functions: SUM, MAX and MIN of the visible distances from an object to the query points. We proposed two approaches, *multiple retrieval front (MRF)* and *single retrieval front (SRF)*. MRF issues a VkNN query at each query point to retrieve objects, whereas SRF issues just one aggregate query to retrieve objects from the database. Both approaches use a separate priority queue to re-rank the retrieve objects according to the aggregate visible distance metric. We showed that SRF consistently performs better than MRF.

VIII. Future Work

Moving query points form our current research direction for VkNN. Our approach is to adapt the safe-region concept, which is widely used in variants of NN problems with moving queries [15], [17], [19], [31], to formulate a region that the visible k NNs do not change (VkNN safe region). In order to solve this problem, the first subproblem to address is maintenance of a visibility region of a moving query point. This subproblem was addressed by Aronov et al. [1]. Their technique is however not suitable for regions with holes/obstacles in the middle (which is commonly the case for VkNN). The second subproblem to address is maintenance of the MinViDist between an object and a moving query point. These two subproblems will be investigated in order to derive a safe-region solution for moving VkNN queries.

Another possible research direction involves deriving an alternative distance measure to MinViDist. In some applications, it could be more meaningful to rank visible objects based on how large they appear according to the perspective of the user at the query point $\boldsymbol{q}$. For example, a distant mountain would be more

prominent than a flower right next to the user. An alternative measure could be formulated based on the size of the projected image of each visible object on a unit-circle (or a unit-sphere in 3D) centered at $\boldsymbol{q}$. Using this measure, the object with the largest projected image is considered to be the most preferred or the nearest.

REFERENCES

[1] B. Aronov, L. J. Guibas, M. Teichmann, and L. Zhang. Visibility queries and maintenance in simple polygons. *Discrete & Computational Geometry*, 27(4):461–483, 2002.

[2] T. Asano, T. Asano, L. J. Guibas, J. Hershberger, and H. Imai. Visibility-polygon search and Euclidean shortest paths. In *FOCS*, pages 155–164, 1985.

[3] T. Asano, T. Asano, L. J. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, 1(1):49–63, 1986.

[4] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, 1975.

[6] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.

[7] E. W. Dijkstra. A note on two problems in connection with graphs. *Numeriche Mathematik*, 1:269–271, 1959.

[8] K. Engel, M. Hadwiger, C. Rezk-Salama, and J. M. Kniss. *Real-time volume graphics*. A K Peters Ltd., 2006.

[9] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. El Abbadi. Constrained nearest neighbor queries. In *SSTD*, pages 257–278, 2001.

[10] Y. Gao, B. Zheng, G. Chen, W.-C. Lee, K. Lee, and Q. Li. Visible reverse k-nearest neighbor queries. In *ICDE*, 2009.

[11] S. K. Ghosh and D. M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.*, 20(5):888–910, 1991.

[12] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[13] P. J. Heffernan and J. S. B. Mitchell. An optimal algorithm for computing visibility in the plane. *SIAM J. Comput.*, 24(1):184–201, 1995.

[14] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[15] L. Kulik and E. Tanin. Incremental rank updates for moving query points. In *GIScience*, pages 251–268, 2006.

[16] K. C. K. Lee, W. C. Lee, and H. V. Leong. Nearest surrounder queries. In *ICDE*, pages 85–94, 2006.

[17] K. C. K. Lee, J. Schiffman, B. Zheng, W.-C. Lee, and H. V. Leong. Round-eye: A system for tracking nearest surrounders in moving object environments. *Journal of Systems and Software*, 80(12):2063–2076, 2007.

[18] S. Nutanong, E. Tanin, and R. Zhang. Visible nearest neighbor queries. In *DASFAA*, pages 876–883, 2007.

[19] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The V*-Diagram: A query dependent approach to moving kNN queries. In *VLDB*, pages 1095–1106, 2008.

[20] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576, 2005.

[21] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.

[22] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[23] H. Samet. Depth-first k-nearest neighbor finding using the MaxNearestDist estimator. In *ICIAP*, pages 486–491, 2003.

[24] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, San Francisco, CA, 2006.

[25] S. Suri and J. O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *Symposium on Computational Geometry*, pages 14–23, 1986.

[26] Y. Tao, D. Papadias, X. Lian, and X. Xiao. Multidimensional reverse *k* nn search. *VLDB J.*, 16(3):293–316, 2007.

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

[28] B. R. Vatti. A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63, 1992.

[29] A. Zarei and M. Ghodsi. Efficient computation of query point visibility in polygons with holes. In *Symposium on Computational Geometry*, pages 314–320, 2005.

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

[31] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, pages 443–454, 2003.