Nearest Surrounder Queries

Ken C.K. Lee[†] Wang-Chien Lee[†] Hong Va Leong[‡]

[†] Pennsylvania State University, University Park, PA16802, USA. {cklee,wlee}@cse.psu.edu
[‡] The Hong Kong Polytechnic University, Hung Hom, Hong Kong. cshleong@comp.polyu.edu.hk

Abstract

In this paper, we study a new type of spatial query, Nearest Surrounder (NS), which searches the nearest surrounding spatial objects around a query point. NS query can be more useful than conventional nearest neighbor (NN) query as NS query takes the object orientation into consideration. To address this new type of query, we identify angle-based bounding properties and distance-bound properties of R-tree index. The former has not been explored for conventional spatial queries. With these identified properties, we propose two algorithms, namely, Sweep *and* Ripple. Sweep *searches surrounders according to their orientation, while* Ripple *searches surrounders ordered by their distances to the query point. Both algorithms can deliver result incrementally with a single dataset lookup. We also consider the multiple-tier NS (mNS) query that searches multiple layers of NSs. We evaluate the algorithms and report their performance on both synthetic and real datasets.*

1 Introduction

Over the last decade, much research effort on spatial query processing has been put forth on Nearest Neighbor (NN) queries which play an important role in decision making. For instance, a tourist information system supporting nearest neighbor queries may assist tourists to find the nearest attractive points of their interest. However, in many cases, an NN query may not be the right question to ask. Reconsider the above tourist information system example. A query that returns surrounding attractive points could provide the tourists good pictures of their surroundings. Take a digital battlefield as another example. To ensure a clear firing path, a query that can find all surrounding enemies to a soldier is more critical than the one that solely reports the nearest enemies. These motivating examples foster the needs of a new type of spatial query – "*Nearest Surrounder*".

Given a set of objects O and a query point q, a *nearest surrounder (NS) query* retrieves the nearest neighbors (NNs) from a query point q at different angles. The result set, denoted by $NS(q)$, contains a set of tuples in form of $\langle o, [\alpha, \beta)\rangle$, where $o \in O$ and $[\alpha, \beta)$ is a range of angles in which o is the NN of q. Figure 1 shows an NS query point, q, surrounded by 10 objects, labeled $o_1, o_2, \cdots o_{10}$. Then, $NS(q) = \{\langle o_1, [\alpha_f, \alpha_a)\rangle, \langle o_3, [\alpha_a, \alpha_b)\rangle, \langle o_6, [\alpha_b, \alpha_c)\rangle, \langle o_7, [\alpha_c, \alpha_d)\rangle, \langle o_8, [\alpha_d, \alpha_e)\rangle, \langle o_9, [\alpha_e, \alpha_f)\rangle\}$ where α_a through α_f are distinct angles. The objects in the set are the nearest surrounders to q in the associated range of angles, since no other object is located between them and the query point in these angles.

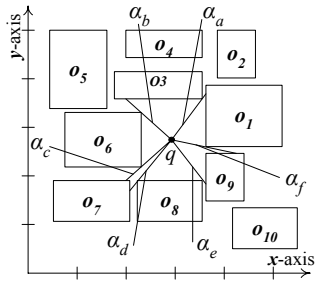


Figure 1. Nearest surrounder queries

The notion of NS queries can be extended to *multi-tier NS (mNS)* queries which search m layers of nearest surrounders to a query point. Informally, mNS provides a result set in which at a particular range of angles, m objects in order of their distances to the query point are retrieved. For $m = 2$, extending the above example (in Figure 1), the first-tier and second-tier NS objects corresponding to q are $\{o_1, o_3, o_6, o_7, o_8, o_9\}$ and $\{o_2, o_4, o_5, o_{10}\}$[1], respectively.

Our study presented in this paper addresses these interesting NS queries. In this study, we use R-tree as the underlying index structure for its efficiency and wide acceptance by the industry and research community. We first identify a set of angle-based bounding properties that serve as the basis for development of efficient algorithms. Then we propose two novel algorithms, namely, *Sweep* and *Ripple*, that perform branch-and-bound traversal on R-tree based on two different searching strategies. Since *surroundings* and *nearness* are two important facets of NS queries, *Sweep* algorithm takes an *angular view* to search the space while *Ripple* algorithm employs a *distance-based* strategy. Both of

[1] Associated angles are omitted for easy illustration.

our algorithms need only *a single dataset lookup* and they deliver result progressively such that partial results can be delivered to users as soon as they are available. These algorithms are applicable to many applications. For instance, a tourist looking for a nearest attractive point may prefer one in front of his moving direction. *Sweep* algorithm may render the result starting from his preferred direction so that the tourist can decide how much deviation her journey has to be made. Moreover, for a digital battlefield scenario, a soldier may be concerned with not only his immediate surrounding enemies (say tanks) but also other soldiers behind the tanks. Thus, *Ripple* algorithm can help progressively visualizing the enemies layer by layer.

We experimentally evaluated these algorithms with both synthetic and real datasets. In particular, we studied the impact of *object density* on the performance of our proposed algorithms in term of I/O cost, runtime memory usage and CPU time. Object density is governed by two parameters, i.e., number of objects and object size. From the experimental result, we observed that the higher the object density (larger object size or larger dataset size), the better the query performance, because surrounders can be found within a small area. Generally speaking, *Ripple* algorithm performs better than *Sweep* algorithm in term of CPU time, while *Sweep* algorithm outperforms *Ripple* algorithm in term of runtime memory consumption.

The remainder of this paper is organized as follows. Section 2 reviews existing works and relevant spatial query problems. Section 3 discusses the properties of R-tree that help NS search. Section 4 discusses Sweep and Ripple algorithms for processing NS queries. Section 5 reports performance study and finally Section 6 concludes this paper.

2 Related Work

Our algorithms for NS query processing are based upon R-tree [2, 5] for its simplicity, efficiency and popularity. Each R-tree node is associated with a Minimum Bounding Rectangle (MBR). Based on the MBR boundary, two important *distances* from an MBR to a query point, MINDIST and MINMAXDIST, are derived in [8]. The best-first NN search proposed by Hjaltason at el. [7] is often used for distance browsing. The algorithm uses only MINDIST values to determine the traversal order and access termination. However, for processing of NS queries, using distances alone is not sufficient to identify the nearest surrounding objects. Angular aspects of NS queries also need to be considered. This motivates us to exploit *angle-based bounding* properties of MBR for NS query processing (details will be discussed in Section 3).

A related work on NN search is constrained NN (CNN) [4]. A CNN query limits the scope of NN search within a specified region. Since partial MBRs are considered, the semantics of MINDIST is revised. Consider Region 1 shown in Figure 2(a). The revised MINDIST of g corresponding to Region 1 is now referred to as the distance from the query point to the nearest point on g within the intersected region. With the revised MINDIST, the CNN search algorithm is the same as the original NN algorithm.

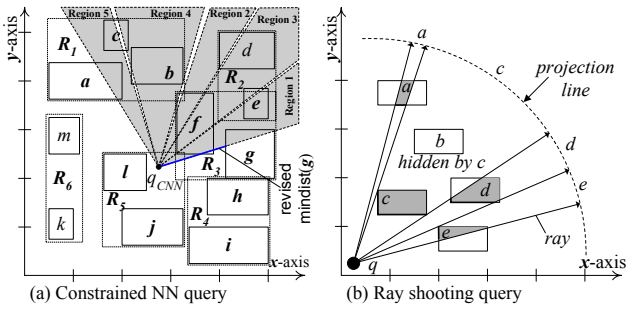


Figure 2. CNN and ray shooting queries

An NS query can be approximately evaluated by issuing multiple CNN queries on non-overlapped regions. As illustrated in Figure 2(a), a portion of NS query is evaluated as five individual CNN queries over Region 1, 2, 3, 4 and 5. In this example, the first four of these queries report f to be the nearest while the last one reports a. This discloses two deficiencies in using CNN queries to evaluate an NS query: 1) adjacent CNN queries are likely to redundantly access a similar set of index nodes and objects, e.g., CNN queries on Region 1, 2 and 3 access both R_2 and R_3; 2) the answer set approximated by CNN queries may not be correct, e.g., b and c should be parts of the NS result but they are not taken. Even worse, there is no ideal solution addressing these two deficiencies at the same time.

The *spherical projection* operations [9] in image rendering have some similarity to NS queries. As shown in Figure 2(b), edges of objects a, b, c, d and e are projected with respect to a query point, q, on a projection line placed behind all objects. On the projection line, b is not shown since it is hidden by c. To achieve spherical projection, ray shooting queries [1] are often used. A ray shooting query is an infinite line segment originated from a query point in search of the first hit object. However, NS search and image rendering are conceptually and functionally different. Our work on NS query focuses heavily on the mechanism on fetching index pages containing possible surrounding objects from the disk storage to the main memory in a very large spatial database, while rendering algorithms focus on generating a display from a relatively smaller number (in the order of hundreds to thousands [3]) of objects resident in main memory. Our work does not assume a projection line as the backdrop but projection does. The project operations initiate a huge number of ray shooting queries, while our algorithm performs only a single query. To reduce the number of ray queries, radial subdivisions are maintained for each scene object [6]. Although the use of radial subdivisions is similar to our *Sweep* algorithm, *Sweep* uses angles to order the access of MBRs. Another approach to save the num-

ber of ray queries is a sweep plane algorithm [3] that uses a projection line which is a straight line perpendicular to the viewer vector moves toward the backdrop and determines the intersection points between the line and scene objects in memory. However, the plane is limited to a restricted angular range and the plane has to traverse all objects in the scene. Our *Ripple* algorithm scans index nodes and explores spatial objects along the distance order for all directions.

3 Preliminaries

In this section, we exploit the angle-based bounding properties of MBR that can be used to derive heuristics for efficient NS search. In addition, we analyze how to determine an NS for two given objects.

3.1 Angle-Based Bounding Properties

Given a query point, q, and an MBR, R, we first define some terminologies that help to clarify important concepts, notations and identify some *angle-based bounding properties* and *angular distance properties* that are useful to our searching algorithms. In our discussion, *polar angles* measured from the positive x-axis (ranging in $[0, 2\pi)$) are used[2].

Definition 1 Angular bound of an MBR. *Taking q as an origin in the NS search space, the angular bound of R is denoted by $[\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}]$ where $\theta^{\vdash}_{q,R}$ and $\theta^{\dashv}_{q,R}$ are respectively the minimum angle and the maximum angle of R, and $\theta^{\vdash}_{q,R} \leq \theta^{\dashv}_{q,R}$ (See Figure 3(a)). If q is located inside R, the angular bound of R to q is assigned to $[0, 2\pi)$.* ■

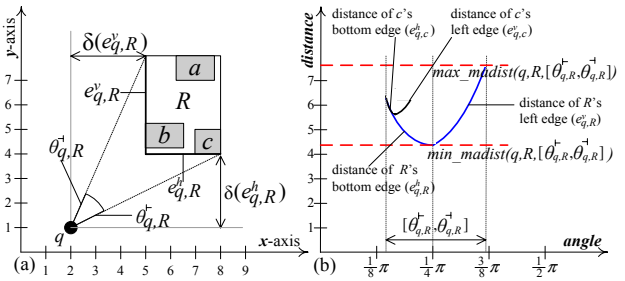


Figure 3. Angle-based Bounding Properties

Note that the above definition does not hold when R intersects with the positive x-axis in the search space. In this case, we partition R horizontally along the x-axis into R_1 (above and touching x-axis) and R_2 (below x-axis) such that the above definition remains valid for each partition.

Property 1 *The angular bound of an MBR associated with an R-tree index node, R_p, must be wide enough to cover each of the child MBRs, R_c with respect to q, i.e. $[\theta^{\vdash}_{q,R_c}, \theta^{\dashv}_{q,R_c}] \subseteq [\theta^{\vdash}_{q,R_p}, \theta^{\dashv}_{q,R_p}]$.* ■

If q stays within both R_p and R_c, the angular bounds, $[\theta^{\vdash}_{q,R_p}, \theta^{\dashv}_{q,R_p}] = [\theta^{\vdash}_{q,R_c}, \theta^{\dashv}_{q,R_c}] = [0, 2\pi)$ (By Definition 1). If q is inside R_p but outside R_c, certainly $[\theta^{\vdash}_{q,R_c}, \theta^{\dashv}_{q,R_c}] \subset [\theta^{\vdash}_{q,R_p}, \theta^{\dashv}_{q,R_p}] = [0, 2\pi)$ holds. As shown in Figure 3(a), R is an MBR enclosing other MBRs a, b and c. The angular bound of R covers all of its enclosing MBRs.

The angular bound suggests the R-tree traversal order (e.g., nodes and objects are visited according to their MBRs' minimum angles (also called *starting angle*[3])). This facilitates our *Sweep* algorithm to fetch the nodes in an angular order (e.g., from 0 to 2π). Further, the application of Property 1 enables branch-and-bound search in our *Sweep* algorithm (see Section 4.1).

Definition 2 Minimum angular distance of an MBR. *The minimum angular distance between q and R at an angle α (denoted by $madist(q, R, \alpha)$) refers to the Euclidean distance from q to a nearest boundary point on R at α. If q stays inside R, $madist(q, R, \alpha) = 0$. If $\alpha \notin [\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}]$, $madist(q, R, \alpha) = \infty$.* ■

From the definition, Property 2 below is observed. Next, we extend Definition 2 to define the minimum angular distance bound, which further leads to Property 3.

Property 2 *Given an angle α, the minimum angular distance of an index node's MBR, R_p, is less than or equal to that of any of its enclosing MBRs, R_c, i.e., $madist(q, R_p, \alpha) \leq madist(q, R_c, \alpha)$.* ■

Definition 3 Minimum angular distance bound. *Given an angular range $[\vartheta^{\vdash}, \vartheta^{\dashv}]$, a contiguous range of minimum angular distances is determined. Both $max\_madist(q, R, [\vartheta^{\vdash}, \vartheta^{\dashv}])$ and $min\_madist(q, R, [\vartheta^{\vdash}, \vartheta^{\dashv}])$ refer to the maximum (i.e., $max_{\alpha \in [\vartheta^{\vdash}, \vartheta^{\dashv}]} madist(q, R, \alpha)$) and minimum (i.e., $min_{\alpha \in [\vartheta^{\vdash}, \vartheta^{\dashv}]} madist(q, R, \alpha)$) limits of the distance range, respectively.* ■

Property 3 *The MBR of R_p always provides the lower bound on the minimum angular distance with respect to q than to all of its child MBRs.* ■

Based on Property 3, if the min_madist of an index node is greater than the max_madist of a found nearest surrounder object for a same angular range, all child nodes of the index node do not constitute any NS so that they can be safely ignored from detailed examination. This property helps pruning the search space in our algorithms.

3.2 Object Comparison

An essential issue in NS query processing is to determine which objects among all candidates are NSs for an angular range. We treat the object comparison mechanism as a binary function. Every time the function compares two object MBRs and returns the nearer one in the entire angular range, or both the objects, each associated with a divided angular range in which the object is nearer.

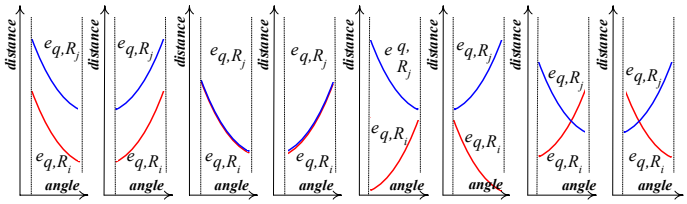
We compare two objects by comparing their edge distances to a query point, q. The edge is a line segment on

[2]In this paper, polar angle and angle are used interchangeably when no confusion is caused.

[3]We use angle bound minimum (maximum) angles and starting (ending) angles interchangeably.

a MBR boundary. An MBR R has at most two nearest edges facing the query point q, which, based on their orientations, are denoted by $e^h_{q,R}$ (horizontal edge) and $e^v_{q,R}$ (vertical edge). As shown in Figure 3(a), the distance between q and $e^h_{q,R}$ is denoted by $madist(q, e^h_{q,R}, \alpha_h)$ (i.e., $\delta(e^h_{q,R})/\sin\alpha_h$) and that between q and $e^v_{q,R}$ is denoted by $madist(q, e^v_{q,R}, \alpha_v)$ (i.e., $\delta(e^v_{q,R})/\cos\alpha_v$), where $\delta(e^h_{q,R})$ and $\delta(e^v_{q,R})$ are the *perpendicular distances* from $e^h_{q,R}$ to the x-axis and from $e^v_{q,R}$ to the y-axis of the search scope, respectively; and α_h and α_v are angles from q to any points on $e^h_{q,R}$ and $e^v_{q,R}$, respectively. Figure 3(b) plots the distance curves for MBR R and MBR c over a range of angles corresponding to the query point q according to the scenario in Figure 3(a). For notational convenience, we use $e_{q,R}$ to represent an edge subject to any orientation. We also reuse max_madist and min_madist to represent their maximum and minimum distance limits of an edge, respectively.

We can use the distance curves to determine the nearer object and the corresponding angular range. Figure 4 shows four cases (where R_i's edge (i.e. e_{q,R_i}) is nearer than R_j's (i.e. e_{q,R_j}), the cases for R_j's nearer than R_i's are symmetric and omitted for space saving) categorized by whether the edges are in parallel or orthogonal to each other and whether they intersect with each other.



(a) Case 1 (b) Case 2 (c) Case 3 (d) Case 4

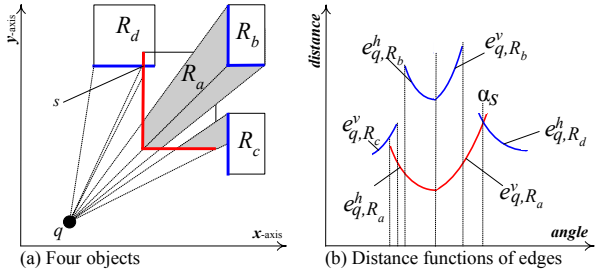
Figure 4. Edge relationships

Case 1. Parallel edges with no intersection: This case (Figure 4(a)) occurs when two edges are in parallel within a given angular range. For instance, we compare R_a and R_b (in Figure 5(a)) whose edges are parallel. From the corresponding distance curves shown in Figure 5(b), we can observe that both e^h_{q,R_b} and e^v_{q,R_b} do not contribute any nearer edge than e^h_{q,R_a} and e^v_{q,R_a} correspondingly. It can be quickly determined by checking their edge perpendicular distance, i.e., $\delta(e^h_{q,R_a})$ and $\delta(e^h_{q,R_b})$, or $\delta(e^v_{q,R_a})$ and $\delta(e^v_{q,R_b})$, due to sharing the same angular range. ■

Case 2. Parallel edges with intersection: This case (Figure 4(b)) occurs when two edges are in parallel and intersecting, that means these edges are overlapped for the entire intersected angular range. In this case, we can use other criteria (e.g. object ID) to decide their priority as an NS. ■

Case 3. Orthogonal edges with no intersection: This case (Figure 4(c)) occurs when two edges are perpendicular to each other and they have no intersection. To quickly identify this case, we examine their minimum angle distance bounds. If min_madist of an edge is greater than max_madist of another, we can assert that the former object is not nearer than the latter to q for the entire angular range. As shown in Figure 5(a), e^h_{q,R_a} covers a part of e^v_{q,R_c} and the corresponding distance functions are shown in Figure 5(b). ■

Case 4. Orthogonal edges with intersection: This case (Figure 4(d)) is the result of two perpendicular edges intersecting or touching each other at a point. In this case, we have to determine the angular range for which of the two edges is nearer to a query, i.e., we determine a split angle that divides the angular range into two. Consider Figure 5(a), where e^v_{q,R_a} intersects e^h_{q,R_d}. Let s denote the intersecting point. The split angle from q to s, α_s is $\arctan(\delta(e^h_{q,R_d})/\delta(e^v_{q,R_a}))$, thus e^v_{q,R_a} (i.e., R_a) appears nearer until α_s. After that e^h_{q,R_d} (i.e. R_d) is nearer. ■



(a) Four objects (b) Distance functions of edges

Figure 5. Examples

Function EdgeCompare(q, $e_{q,i}$, $e_{q,j}$, $[\vartheta^\vdash, \vartheta^\dashv]$)
Input. *a query point (q), edges of MBRs of objects i and j ($e_{q,i}$) and ($e_{q,j}$), a common angular range ($[\vartheta^\vdash, \vartheta^\dashv]$).*
Output. *a set of ⟨object:angular range⟩ tuples.*

1. **if** (both $e_{q,i}$ and $e_{q,j}$ are parallel) **then** /* *Case 1 and 2* */
2. **if** ($\delta(e_{q,i}) < \delta(e_{q,j})$) **then** return $\{\langle i : [\vartheta^\vdash, \vartheta^\dashv]\rangle\}$;
3. **else if** ($\delta(e_{q,j}) < \delta(e_{q,i})$) **then** return $\{\langle j : [\vartheta^\vdash, \vartheta^\dashv]\rangle\}$;
4. **else** return $\{\langle e : [\vartheta^\vdash, \vartheta^\dashv]\rangle\}$; /* $e = i$ *or* j */
5. **else** /* *Case 3 and 4* */
6. **if** ($max\_madist(q, e_{q,i}, [\vartheta^\vdash, \vartheta^\dashv]) <$ $min\_madist(q, e_{q,j}, [\vartheta^\vdash, \vartheta^\dashv])$) **then** return $\{\langle i : [\vartheta^\vdash, \vartheta^\dashv]\rangle\}$;
7. **else if** ($max\_madist(q, e_{q,j}, [\vartheta^\vdash, \vartheta^\dashv]) <$ $min\_madist(q, e_{q,i}, [\vartheta^\vdash, \vartheta^\dashv])$) **then** return $\{\langle j : [\vartheta^\vdash, \vartheta^\dashv]\rangle\}$;
8. **else** /* *Case 4 only* */
9. Let α_s be a split angle;
10. **if** ($e_{q,i}$ is horizontal) **then** $\alpha_s \leftarrow \arctan(\delta(e_{q,i})/\delta(e_{q,j}))$;
11. **else** $\alpha_s \leftarrow \arctan(\delta(e_{q,j})/\delta(e_{q,i}))$; /* $e_{q,j}$ *is horizontal* */
12. **if** ($madist(q, e_{q,i}, \vartheta^\vdash)) < madist(q, e_{q,j}, \vartheta^\vdash)$) **then** return $\{\langle i : [\vartheta^\vdash, \alpha_s)\rangle, \langle j : [\alpha_s, \vartheta^\dashv]\rangle\}$;
13. **else** return $\{\langle j : [\vartheta^\vdash, \alpha_s)\rangle, \langle i : [\alpha_s, \vartheta^\dashv]\rangle\}$;

Figure 6. Function EdgeCompare

Function EdgeCompare implementing the edge comparison logic is listed in Figure 6. With EdgeCompare, algorithm ObjectCompare, outlined in Figure 7, analyzes the nearness of two input objects. The comparison result, $\mathcal{O}$, initially empty, will contain a set of $\langle o : [\vartheta^\vdash_o, \vartheta^\dashv_o]\rangle$ tuples, where o is the nearest object at an angular range $[\vartheta^\vdash_o, \vartheta^\dashv_o]$. A common angular range, $[\vartheta^\vdash, \vartheta^\dashv]$ is first identified in line 2. Outside $[\vartheta^\vdash, \vartheta^\dashv]$, individual objects at non-overlapping angular ranges are set to be part of $\mathcal{O}$ since they are incomparable. Next, we compare their edges with the common

angular range by invoking EdgeCompare. The EdgeCompare result is added to $\mathcal{O}$. Our proposed algorithms, *Sweep* and *Ripple* to be discussed in the next section, utilize this ObjectCompare algorithm to decide what objects are NSs.

Algorithm ObjectCompare(q, i, $[\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}]$, j, $[\vartheta^{\vdash}_{q,j}, \vartheta^{\dashv}_{q,j}]$)
Input. *a query point (q), two objects (i) and (j) with respective angular range ($[\vartheta^{\vdash}_i, \vartheta^{\dashv}_i]$) and ($[\vartheta^{\vdash}_i, \vartheta^{\dashv}_i]$).*
Output. *a set of ⟨object:angular range⟩* tuples.
1. Let $\mathcal{O}$ be the comparison result, initialized $\emptyset$;
2. Let $[\vartheta^{\vdash}, \vartheta^{\dashv}]$ be the common angular range, i.e., $[\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}] \cap [\vartheta^{\vdash}_{q,j}, \vartheta^{\dashv}_{q,j}]$;
3. **if** $(([\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}] - [\vartheta^{\vdash}, \vartheta^{\dashv}]) \neq \emptyset)$ **then** $\mathcal{O} \leftarrow \mathcal{O} \cup \{\langle i : ([\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}] - [\vartheta^{\vdash}, \vartheta^{\dashv}])\rangle\}$;
4. **if** $(([\vartheta^{\vdash}_{q,j}, \vartheta^{\dashv}_{q,j}] - [\vartheta^{\vdash}, \vartheta^{\dashv}]) \neq \emptyset)$ **then** $\mathcal{O} \leftarrow \mathcal{O} \cup \{\langle j : ([\vartheta^{\vdash}_{q,j}, \vartheta^{\dashv}_{q,j}] - [\vartheta^{\vdash}, \vartheta^{\dashv}])\rangle\}$;
5. **foreach** e_i **in** $\{e^h_{q,i}, e^v_{q,i}\}$
6. **foreach** e_j **in** $\{e^h_{q,j}, e^v_{q,j}\}$
7. **if** ($e_{q,i}$ and $e_{q,j}$ have overlapped angular range, $[\vartheta'^{\vdash}, \vartheta'^{\dashv}]$)
8. **then** $\mathcal{O} \leftarrow \mathcal{O} \cup$ EdgeCompare(q,e_i,e_j,$[\vartheta'^{\vdash}, \vartheta'^{\dashv}]$);
11. Output $\mathcal{O}$;

Figure 7. Algorithm ObjectCompare

4 Nearest Surrounder Search Algorithms

In the section, we describe basic operations for *Sweep* and *Ripple* algorithms, followed by possible extension: progressive result delivery and handling of multi-tier NS queries. By the end of this section, we compare and contrast the algorithms and suggest the type of scenarios each one would be expected to perform better.

4.1 Sweep Search Algorithm

4.1.1 Basic *Sweep* Operation

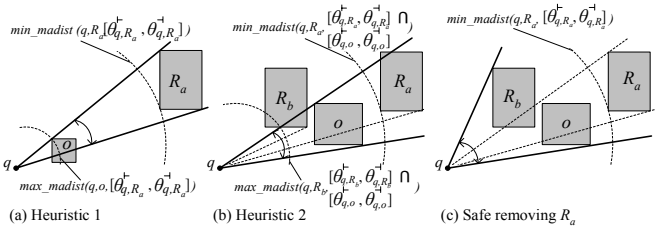
Sweep operates in the same fashion as the best-first NN algorithm [7]. It maintains a priority queue with the index nodes and objects ordered according to their starting angles. Collectively, the index nodes and objects inside the priority queue are termed *entries*. In addition, *Sweep* maintains a NS result initialized to $\{\langle \perp : [0, 2\pi)\rangle\}$ where $\perp$ represents a dummy NS that is replaceable by any real object. The dummy NS bears two purposes in our NS algorithms: 1) it pads the total angular range of entire NS result such that the whole angular range $[0, 2\pi)$ is covered; 2) it catches a "*hole*" at the associated angular range, where no NS has yet been found. Its minimum angular distance is ∞.

At the very beginning, the root index node is inserted to the priority queue. Next, the head entry of the queue is de-queued and studied. If a de-queued entry is an index node, all its children are placed back to the priority queue. If an de-queued entry is an object, it is compared with currently found NS objects maintained in an NS result for the same angular range using ObjectCompare (see Section 3.2) and the comparison result is incorporated to the NS result. This routine continues until all queue entries are examined.

4.1.2 Enhancements of *Sweep*

Scanning the entire priority queue will require exploring all index nodes and examining all objects. In other words, full index lookup will be resulted. In fact, not all queue entries can contribute to an NS result. To improve the search performance, the search space must be effectively pruned. In the following, we consider two important heuristics to avoid exploring unnecessary index nodes, considerably reducing I/O cost and save the number of ObjectCompare calls which incur most computation cost in the algorithm.

Heuristic 1. Given the angular bound of R, $[\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}]$, currently found NS objects whose angular bounds intersecting $[\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}]$ are selected and the maximum of their max_madist called *conservative upper bound* is computed. If $min\_madist(q, R, [\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}])$ is greater than this conservative upper bound, R is asserted not an NS or not containing NS objects. Thus, R needs not to explore or examined. ■



(a) Heuristic 1 (b) Heuristic 2 (c) Safe removing R_a

Figure 8. Heuristics

As shown in Figure 8(a), there is an MBR R_a (representing either an object or an index node) is examined and there is an object o already taken as NS. Since $min\_madist(q, R_a, [\theta^{\vdash}_{q,R_a}, \theta^{\dashv}_{q,R_a}])$ exceeds the conservative upper bound, $max\_madist(q, o, [\theta^{\vdash}_{q,R_a}, \theta^{\dashv}_{q,R_a}])$, R_a need not be explored or examined.

Owing to the access order defined only on the starting angles, an entry with a smaller starting angle always comes before the other possibly closer objects. Thus, it fosters the need of Heuristic 2 which slightly revises the access order by considering additional factors so that entries which are most likely to contain NS objects are retrieved first.

Heuristic 2. Suppose there exists an angular range $[\vartheta^{\vdash}, \vartheta^{\dashv}]$ covering all currently found NS objects, an MBR R is most likely to contain an NS object if it can provide the smallest minimum angular distance in the common angular range, i.e., $[\vartheta^{\vdash}, \vartheta^{\dashv}] \cap [\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}]$ among all the other queue entries. ■

Figure 8(b) shows one currently found object o and two MBRs, R_a and R_b. Following the starting angle order, R_a is examined immediately after o. However, as anticipated, R_a does not contribute any NS within the common angular range, $[\theta^{\vdash}_{q,o}, \theta^{\dashv}_{q,o}] \cap [\theta^{\vdash}_{q,R_a}, \theta^{\dashv}_{q,R_a}]$. Even worse it is hidden by R_b which will be accessed next to R_a. Instead of R_a, R_b with a smaller minimum angular distance is picked according to Heuristic 2.

As R_b is picked instead of R_a, R_a will remain in the priority queue, occupying extra runtime memory. It leaves a question when R_a is suitably fetched from the priority

queue. Recall Heuristic 1. An entry can be safely discarded when its minimum angular distance is greater than the maximum angular distance of NS objects within the entry's angular range. Thus an entry could be fetched from the queue when its angular range is already explored. Based on Heuristic 1 and 2, the access order of the queue is revised. To manage this queue retrieval operation, function Lookahead is implemented (in Figure 9). It looks over the queue content and returns the best entry. It takes on three parameters, namely, a query point, q, a priority queue, Q, and an angular range $[\vartheta^{\vdash}, \vartheta^{\dashv}]$ which covers all currently found NS objects. Lookahead does not re-order the positions of queue entries but every time when it is invoked, it examines the front part of the queue to pick one entry according to the revised priority. The highest priority is given to an entry that will be fully covered within the parameter angular range. Next, an entry with the smallest min_madist in the intersected angular range is chosen. At last, the head entry is returned.

Function Lookahead(q, Q, $[\vartheta^{\vdash}, \vartheta^{\dashv}]$)
Input. *a query point* (q), *a priority queue* (Q), *an angular range given by* Sweep *algorithm* ($[\vartheta^{\vdash}, \vartheta^{\dashv}]$).
Output. *an element*.
1. Let $[\epsilon_i | 1 \leq i \leq |Q|]$ be the content of Q;
2. **if** (ϵ_i such that $[\theta^{\vdash}_{q,\epsilon_i}, \theta^{\dashv}_{q,\epsilon_i}] \subseteq [\vartheta^{\vdash}, \vartheta^{\dashv}]$ exists) **then** remove ϵ_i from Q; return ϵ_i; /* *flush remaining* */
3. Let $Q' = [\epsilon_j | 1 \leq j \leq |Q| \wedge [\theta^{\vdash}_{q,\epsilon_j}, \theta^{\dashv}_{q,\epsilon_j}] \cap [\vartheta^{\vdash}, \vartheta^{\dashv}] \neq \emptyset]$ be prefix of Q;
4. **if** ($Q' \neq \emptyset$) **then**
5. pick ϵ_i from Q' with least $min\_madist(q, \epsilon_i, [\theta^{\vdash}_{q,\epsilon_i}, \theta^{\dashv}_{q,\epsilon_i}] \cap [\vartheta^{\vdash}, \vartheta^{\dashv}])$;
6. remove ϵ_i from Q; return ϵ_i; /* *least distant entry* */
7. remove ϵ_1 from Q; return ϵ_1; /* *default de-queue operation* */

Figure 9. Function Lookahead

Progressive result delivery is an important property of our NS algorithms. The completed part of an intermediate result is delivered as soon as it is ready. With traversal order based on the starting angles, the following heuristic can guarantee the correctness of partial *Sweep* result delivery during the query execution.

Heuristic 3. If the starting angle of the head entry of the priority queue is α, all the remaining entries do not affect the existing NS result associated with angles $\leq \alpha$. ■

This heuristic enables *Sweep* to deliver a partial result up to the starting angle of the head entry of the priority queue.

4.1.3 *Sweep* Algorithm and Example

With all discussed enhancements, we list *Sweep* algorithm in Figure 10. A priority queue, Q, is first initialized to contain the root of an R-tree; an angle α representing the maximum ending angle of examined objects is initialized to 0; and the NS result, $\mathcal{N}$, which is set to $\{\langle \perp : [0, 2\pi) \rangle\}$ (line 1–3). Later, the priority queue is examined iteratively until the empty queue is encountered that is the sign of termination (line 4–12). In line 6, function Lookahead is invoked to pick an entry from Q. If the entry which min_madist is greater than the conservative upper bound which is determined as the maximum among max_madists of all overlapped NS result objects, it is safe to ignore the entry in the rest of examination. In line 9, when the entry is an index node, its children are put into Q for later investigation, but when the entry is an object, function NSIncorporate (to be discussed in next paragraph) is invoked to update the NS result. After all, α, is updated to the maximum ending angle observed so far. At last, the final NS result stored in $\mathcal{N}$ is output.

Algorithm *Sweep* (q, $root$)
Input. a query point (q), and an R-tree root index node ($root$).
Output. an NS result, i.e., a set of ⟨object:angular range⟩ tuples.
1. Let Q be priority queue and be initialized with $root$;
2. Let α be max. ending angle of examined objects, initialized to 0;
3. Let $\mathcal{N}$ be the NS result and be initialized as $\{\langle \perp : [0, 2\pi] \rangle\}$;
4. **while** (Q is not empty)
5. Let ϵ be an element (it could be either a node or an object);
6. $\epsilon \leftarrow$ Lookahead(q, Q, $[0, \alpha]$);
7. **if** ($min\_madist(q, \epsilon, [\theta^{\vdash}_{q,\epsilon}, \theta^{\dashv}_{q,\epsilon}]) >$ *conservative upper bound* $\wedge\, [\theta^{\vdash}_{q,\epsilon}, \theta^{\dashv}_{q,\epsilon}] \subseteq [0, \alpha]$) **then** skip;
8. **else**
9. **if** (ϵ is node) **then** explore ϵ and put all its child entries to Q;
10. **else**
11. $\mathcal{N} \leftarrow$ NSIncoporate(q,$\mathcal{N}$,ϵ);
12. $\alpha \leftarrow \max(\alpha, \theta^{\dashv}_{q,\epsilon})$;
13. Output $\mathcal{N}$;

Figure 10. Pseudo-code of *Sweep* algorithm

Function NSIncorporate (in Figure 11) updates the NS result. It extracts $\langle i, [\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}] \rangle$ from an NS result, $\mathcal{N}$, where $[\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}]$ intersects the angular bound of an object o, $[\theta^{\vdash}_{q,o}, \theta^{\dashv}_{q,o}]$. It invokes ObjectCompare (in Section 3.2) to determine the nearest objects for appropriate angular ranges. It then replenishes the result by adding result tuples, $\langle r, [\vartheta^{\vdash}_{q,r}, \vartheta^{\dashv}_{q,r}] \rangle$, into $\mathcal{N}$.

Function NSIncorporate(q, $\mathcal{N}$, o)
Input. a query point (q), an NS Result ($\mathcal{N}$), and an object MBR (o).
Output. the updated NS Result.
1. **foreach** ($\langle i, [\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}] \rangle \in \mathcal{N} \,|[\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}] \cap [\theta^{\vdash}_{q,o}, \theta^{\dashv}_{q,o}] \neq \emptyset$)
2. $\mathcal{N} \leftarrow \mathcal{N} - \{\langle i, [\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}] \rangle\}$;
3. **foreach** ($\langle r, [\vartheta^{\vdash}_{q,r}, \vartheta^{\dashv}_{q,r}] \rangle \in$ ObjectCompare(q,i,$[\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}]$, o,$[\vartheta^{\vdash}_{q,i}, \vartheta^{\dashv}_{q,i}] \cap [\theta^{\vdash}_{q,o}, \theta^{\dashv}_{q,o}]$))
4. $\mathcal{N} \leftarrow \mathcal{N} \cup \{\langle r, [\vartheta^{\vdash}_{q,r}, \vartheta^{\dashv}_{q,r}] \rangle\}$;
5. return $\mathcal{N}$;

Figure 11. Function NSIncorporate

We conclude the section by studying a running example based on a scenario depicted in Figure 12(a) where 10 objects ('a' through 'j') are included in 4 index nodes. The radial lines show the starting angles of touched MBRs. Our trace shown in Figure 13 starts at where a priority queue contains the four index nodes, i.e., $[R_1, R_2, R_3, R_4]$, ordered by their starting angles[4]. R_1, the head entry of the priority queue is firstly explored so its children a, b, and

[4]Since R_1 intersects x-axis, we consider its starting angle as 0.

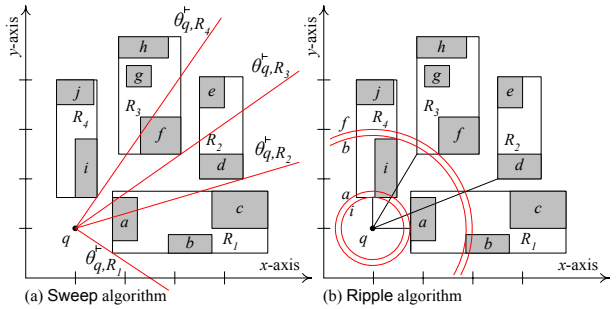


Figure 12. *Sweep* **and** *Ripple* **examples**

c are inserted into the queue. Crossing the x-axis of the search space, a is divided into a_1 (above and on the x-axis) and a_2 (below the x-axis). Since a_1 has smaller minimum angular distance than c at the same starting angle, a_1 is placed before c in the queue. The queue content becomes $[a_1, c, R_2, R_3, R_4, a_2, b]$. a_1 is retrieved and compared with $\perp$ in the current NS result set. Then the NS result is changed to $\{\langle a_1, [0, \theta^{\dashv}_{q,a_1})\rangle, \langle \perp, [\theta^{\dashv}_{q,a_1}, 2\pi)\rangle\}$. α, is updated to $\theta^{\dashv}_{q,a_1}$. Next, c is examined. Behind a_1, c is safely discarded.

ϵ	Q	$\mathcal{N}$ (Remarks)
-	$[R_1, R_2, R_3, R_4]$	$\{\langle \perp : [0, 2\pi)\rangle\}$
R_1	$[a_1, c, R_2, R_3, R_4, a_2, b]$	ditto (a is divided into a_1 and a_2.)
a_1	$[c, R_2, R_3, R_4, a_2, b]$	$\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle \perp : [\theta^{\dashv}_{q,a_1}, 2\pi)\rangle\rangle\}$
c	$[R_2, R_3, R_4, a_2, b]$	ditto
R_3	$[R_2, f, R_4, g, h, a_2, b]$	ditto (By Heu. 2, R_3 is chosen, not R_2.)
f	$[R_2, R_4, g, h, a_2, b]$	$\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle f : [\theta^{\dashv}_{q,a_1}, \theta^{\dashv}_{q,f})\rangle\rangle, \langle \perp : [\theta^{\dashv}_{q,f}, 2\pi)\rangle\rangle\}$ (By Heu. 2, f is chosen.)
R_2	$[R_4, g, h, h, a_2, b]$	ditto (R_2 is safely discarded.)
R_4	$[i, g, h, j, a_2, b]$	ditto
i	$[g, h, j, a_2, b]$	$\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle f : [\theta^{\dashv}_{q,a_1}, \theta^{\vdash}_{q,i})\rangle\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle\rangle, \langle \perp : [\theta^{\dashv}_{q,i}, 2\pi)\rangle\rangle\}$
g, h	$[j, a_2, b]$	ditto (g and h are safely discard.)
j	$[a_2, b]$	$\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle f : [\theta^{\dashv}_{q,a_1}, \theta^{\vdash}_{q,i})\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle j : [\theta - q, i^{\dashv}, \theta^{\dashv}_{q,j})\rangle, \langle \perp : [\theta^{\dashv}_{q,j}, 2\pi)\rangle\rangle\}$
a_2	$[b]$	$\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle f : [\theta^{\dashv}_{q,a_1}, \theta^{\vdash}_{q,i})\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle j : [\theta^{\dashv}_{q,i}, \theta^{\dashv}_{q,j})\rangle, \langle \perp : [\theta^{\dashv}_{q,j}, \theta^{\vdash}_{q,a_2})\rangle, \langle a_2 : [\theta^{\vdash}_{q,a_2}, 2\pi)\rangle\rangle\}$
b	[ ]	ditto

Figure 13. The trace of *Sweep* **algorithm**

R_2 becomes the head entry of the queue. Due to Heuristic 2, R_3, overlapping the angular range of a_1 and providing a shorter minimum angular distance than R_2, is picked instead. Then R_3's children f, g and h are enqueued. For the same reason, f is taken rather than R_2, and the NS result is updated to $\{\langle a_1, [0, \theta^{\dashv}_{q,a_1})\rangle, \langle f, [\theta^{\dashv}_{q,a_1}, \theta^{\dashv}_{q,f})\rangle, \langle \perp, [\theta^{\dashv}_{q,f}, 2\pi)\rangle\}$. Now α is updated to $\theta^{\dashv}_{q,f}$. R_2 is retrieved from the queue since now its angular range is entirely covered by the currently explored scope $[0, \theta^{\dashv}_{q,f}]$. As R_2's minimum angular distance is greater than the conservative upper bound formed by both a_1 and f, R_2 is ignored to examine.

R_4 is next explored and its children i and j are inserted to the queue. i is then examined and the NS result is revised into $\{\langle a_1, [0, \theta^{\dashv}_{q,a_1})\rangle, \langle f, [\theta^{\dashv}_{q,a_1}, \theta^{\vdash}_{q,i})\rangle, \langle i, [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle \perp, [\theta^{\dashv}_{q,i}, 2\pi)\rangle\}$. Further, g and h are examined but they are not closer than i to q. j and a_2 are examined in sequence. Finally b is examined but it is hidden by a_2. The queue is empty and the algorithm terminates.

4.1.4 Extension of *Sweep* for m-tier NS

To extend *Sweep* algorithm to handle multi-tier NS, we need to extend the notation of NS result. For example, to handle a 2-tier NS query, we associate a pair of NSs for each angular range. This initial result for a 2NS query is initialized as $\{\langle(\perp, \perp) : [0, 2\pi)\rangle\}$. As in the example of Figure 12(a), a_1 and c are examined in order and the NS result is $\{\langle(a_1, c) : [0, \theta^{\dashv}_{q,c})\rangle, \langle(a_1, \perp) : [\theta^{\dashv}_{q,c}, \theta^{\vdash}_{q,a_1})\rangle, \langle(\perp, \perp) : [\theta^{\vdash}_{q,a_1}, 2\pi)\rangle\}$. Other than the extended NS slots in the NS result, the logic of *Sweep* algorithm for m-tier NS queries is pretty much the same as that already discussed for single tier NS queries.

4.2 Ripple Search Algorithm

4.2.1 Basic *Ripple* Operation

Similar to *Sweep*, *Ripple* maintain an NS result (a set of $\langle$object:angular range$\rangle$ tuples) and a priority queue containing index nodes and objects. However, the traversal order of *Ripple* is in ascending distance order. By arranging entries in distance order, the exploration of the search space by *Ripple* goes from a query point outward. *Ripple* can terminate before the queue is completely scanned according to the following heuristic.

Heuristic 4. Because of distance ordering of all queue entries, the examination of a priority queue can be completely skipped as long as two conditions are satisfied, namely, 1) the NS result has no dummy NS object; and 2) all queue entries have longer distances to the query point than the *conservative upper bound* of the entire NS result. ■

The condition 1) guarantees that the NS result is complete, while the condition 2) asserts that the rest of the priority queue does not have any object that will appear nearer than any currently found NS object. This heuristic provides a safe condition that *Ripple* terminates correctly and a faster response of the algorithm will be resulted.

To easily point out the difference from *Sweep*, let us consider the running example using *Ripple* depicted in Figure 12(b) where ripple lines indicate the distance of selected objects from a query point, q. The trace of *Ripple* is in Figure 14. At first, the NS result is set to $\{\langle \perp : [0, 2\pi]\rangle\}$ and the priority queue is initialized to $[R_4, R_1, R_3, R_2]$ according to their distance order. First, R_4 is explored and its children i and j are inserted into the priority queue, which becomes $[i, R_1, R_3, j, R_2]$. Then i is the first object to be examined and found to be an NS. The NS result is updated to $\{\langle \perp : [0, \theta^{\vdash}_{q,i})\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle \perp : [\theta^{\dashv}_{q,i}, 2\pi)\rangle\}$. Next,

R_1 is explored and its children a, b and c are inserted into the queue. Now a is picked and incorporated to the result (which becomes $\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle \perp : [\theta^{\dashv}_{q,a_1}, \theta^{\vdash}_{q,i})\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle \perp : [\theta^{\dashv}_{q,i}, \theta^{\vdash}_{q,a_2})\rangle, \langle a_2 : [\theta^{\vdash}_{q,a_2}, 2\pi)\rangle\}$)[5]. Next, R_3 is explored and its children f, g and h are inserted to the queue. Then, b is picked but found to be hidden by a_2. Besides, f is examined and it fills a hole between a_1 and i. j is examined and the result is further updated to $\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle f : [\theta^{\dashv}_{q,a_1}, \theta^{\vdash}_{q,i})\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle j : [\theta^{\dashv}_{q,i}, \theta^{\dashv}_{q,j})\rangle, \langle \perp : [\theta^{\dashv}_{q,j}, \theta^{\vdash}_{q,a_2})\rangle, \langle a_2 : [\theta^{\vdash}_{q,a_2}, 2\pi)\rangle\}$. The remaining entries of the priority queue are examined but none of them is found to be NS objects. At last, *Ripple* terminates.

ϵ	Q	$\mathcal{N}$ (Remarks)
-	$[R_4, R_1, R_3, R_2]$	$\{\langle \perp : [0, 2\pi)\rangle\}$
R_4	$[i, R_1, R_3, j, R_2]$	ditto
i	$[R_1, R_3, j, R_2]$	$\{\langle \perp : [0, \theta^{\vdash}_{q,i})\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle \perp : [\theta^{\dashv}_{q,i}, 2\pi)\rangle\}$
R_1	$[a, R_3, b, j, R_2, c]$	ditto
a	$[R_3, b, j, R_2, c]$	$\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle \perp : [\theta^{\dashv}_{q,a_1}, \theta^{\vdash}_{q,i})\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle \perp : [\theta^{\dashv}_{q,i}, \theta^{\vdash}_{q,a_2})\rangle, \langle a_2 : [\theta^{\vdash}_{q,a_2}, 2\pi)\rangle\}$ (a is divided into a_1 and a_2.)
R_3	$[b, f, j, R_2, c, g, h]$	ditto
b	$[f, j, R_2, c, g, h]$	ditto
f	$[j, R_2, c, g, h]$	$\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle f : [\theta^{\dashv}_{q,a_1}, \theta^{\vdash}_{q,i})\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle \perp : [\theta^{\dashv}_{q,i}, \theta^{\vdash}_{q,a_2})\rangle, \langle a_2 : [\theta^{\vdash}_{q,a_2}, 2\pi)\rangle\}$ (f fills the hole between a_1 and i.)
j	$[R_2, c, g, h]$	$\{\langle a_1 : [0, \theta^{\dashv}_{q,a_1})\rangle, \langle f : [\theta^{\dashv}_{q,a_1}, \theta^{\vdash}_{q,i})\rangle, \langle i : [\theta^{\vdash}_{q,i}, \theta^{\dashv}_{q,i})\rangle, \langle j : [\theta^{\dashv}_{q,i}, \theta^{\dashv}_{q,j})\rangle, \langle \perp : [\theta^{\dashv}_{q,j}, \theta^{\vdash}_{q,a_2})\rangle, \langle a_2 : [\theta^{\vdash}_{q,a_2}, 2\pi)\rangle\}$
The rest of the queue i.e., R_2, c, g, and h are scanned, but none of them are taken in the result.		

Figure 14. The trace of *Ripple* algorithm

4.2.2 Extension of *Ripple* for m-Tier NS

The extension of *Ripple* for m-tier NS is slightly different from that of *Sweep*. Instead of maintaining m NS slots in each NS result tuple, we maintain an array of m NS results. Each array element represents a tier of NS result. This arrangement of an m-tier NS result resembles onion rings. The NS result of lower tiers (inner rings) are filled prior to the NS result of higher tiers (outer rings). When an object is examined, we first study how it contributes to the lower tier(s). There are two possible outcomes in this examination. The first possible outcome is that if an entire object does not appear closer to q at the current tier, then we examine the object with respect to the next immediate tier. This examination will repeat until it can contribute NS result at a certain tier or all m tiers are visited. The second possible outcome is that if the object appears nearer than a part of any existing found NS objects (except the dummy object), we need to examine the rest of the object (if any) and the replaced parts for the next immediate tier.

[5] a is divided along x-axis into a_1 and a_2.

While *Early Termination* (Heuristic 4) discussed in the previous subsection is still applicable, it is also extended with the following heuristic.

Heuristic 5. Tier Completion: As long as 1) the NS result of a tier contains no dummy NS; and 2) the head entry of the queue to be examined is farther than the *conservative upper bound* of the NS result of the tier under consideration; the remaining entries have no effect on the current tier. ■

Based on Heuristic 5, three possible enhancements for *Ripple* can be achieved, namely, skip examination of completed tier NS result, progressive result delivery, and early m-tier termination. Here, progressive result delivery refers to delivering NS result in tier-wise fashion once a tier is completed. To summarize and generalize *Ripple* algorithm for m-tier NS query, we outline the pseudo-code of algorithm *Ripple* in Figure 15.

Ripple takes three parameters, namely the root of a R-tree, $root$, a query point, q, and the number of tiers, m. It operates according to the content of a priority queue, Q, initialized with the $root$. It maintains an integer $CurrTier$ set to 1 and an array of NS results, $\mathcal{N}$. Line 7 checks whether the result of the current tier is completed according to Heuristic 5. If it is completed, the result of current tier is then delivered (line 8) and $CurrTier$ is then incremented (line 9). Next, line 10 terminates the algorithm if all m tiers are completed. Line 11 explores the index node represented by entry ϵ. Line 12 updates the result when ϵ is an object with procedure UpdateTier that incorporates the result with an object o at the current tier and the replaced part of currently found NS object or the remaining of o as $remain$. UpdateTier is recursively called until all tiers are visited or dummy object at a certain tier is replaced.

4.3 Discussion

Ripple looks more efficient than *Sweep* because its early termination condition enables the algorithm to stop without scanning the remaining queue entries. However, the *Ripple*'s early termination conditions do not necessarily guarantee a better performance. The strict conditions require that all nearest surrounders for all angles, i.e., $[0, 2\pi)$, be found. If there is no NS in certain angles, *Ripple* will blindly continue to examine the queue entries, incurring a higher computation and I/O overhead. More importantly, holes may be scattered in the NS result, leading *Ripple* to continuously explore entries intersecting those holes which in turn results in excessive expansion of the queue. For the same case, *Sweep* would perform better because it visits R-tree nodes in the order of starting angle of nodes/objects. Thus *Sweep* can jump over the holes.

5 Performance Evaluation

5.1 Experiment Settings

In this section, we conduct an experimental evaluation of the NS searching algorithms in terms of 1) node/page accesses in R-tree index (i.e., I/O cost), 2) maximum memory

Algorithm *Ripple*($root$,q,m)
Input. a query point (q), and a R-tree root index node ($root$), and the number of tiers (m)
Output. a m-tier NS result.

1. Let Q be the priority queue and be initialized with $root$;
2. Let $CurrTier$ be an integer and be initialized to 1;
3. Let $\mathcal{N}[1..m]$ be an NS result array and each is set to $\{\perp : [0, 2\pi)\}$;
4. Let ϵ be entry of Q;
5. **while** (Q is not empty)
6. $\epsilon \leftarrow dequeue(Q)$;
7. **if** ($\mathcal{N}[CurrTier]$ is completed, implied by ϵ) **then**
8. Output $\mathcal{N}[CurrTier]$;
9. $CurrTier \leftarrow CurrTier + 1$;
10. **if** ($CurrTier > m$) **then terminate**; */* early termination */*
11. **if** (ϵ is node) **then** explore ϵ and put all its child to Q;
12. **else if** (ϵ is object) **then** UpdateTier(q,$\mathcal{N}$,m,$CurrTier$,ϵ);

Procedure UpdateTier(q,$\mathcal{N}$,m,c,o)
Input. a query point(q), an array of m NS result ($\mathcal{N}$), and an object (o) to be inserted into NS result from c-th tier up.
Output. Updated NS result ($\mathcal{N}$).

1. **if** ($c \leq m \wedge o \neq \perp$) **then**
2. NSIncorporate(q,$\mathcal{N}[c]$,o), find $remain$; */* Figure. 11 */*
3. UpdateTier(q,$\mathcal{N}$,m,$c+1$,$remain$); */* recursive call */*

Figure 15. Pseudo-code of *Ripple* algorithm

size used to maintain a priority queue (i.e., the major source of main memory consumption, measured in unit of number of entries in the queue), and 3) CPU time (the measurement of computational overhead). We implemented with C++ the *Sweep* and *Ripple* algorithms as well as a sampling approach based on shooting query (denoted by *Sample* in this section) in which shooting queries are evaluated for discrete sample angles at 1^o, 0.1^o and 0.01^o and they can only provide an approximated result. We perform experiments upon Red Hat Linux 9.0 on Intel Celeron 2.0GHz computers. We use R*-tree [2] as the underlying object index structure supporting all the evaluated algorithms with the node/page size of $1k$ bytes and the maximum node capacity is 50. The cache is set to 25 pages for all experiments.

To evaluate the performance of these algorithms, we use both real and synthetic datasets. All datasets are normalized in a fixed square space of [1000,1000]. Three real datasets obtained from [10] representing landscapes in different states, namely California, Pennsylvania and Rhode Island (labeled as CA, PA and RI respectively), are used in our experiments. The number of objects (i.e., dataset size denoted by ds) in CA, PA and RI are 1225k, 744k and 50k, respectively, while the average object sizes (denoted by os) in CA, PA and RI are $[0.26, 0.24]$, $[0.33, 0.54]$ and $[1.44, 1.06]$, respectively. Note that those object sizes are relative to the fixed evaluation space[6]. In addition, the object distribution of RI and PA is relatively uniform while that of CA is highly skewed. Synthetic datasets, generated based on uniform object distribution, contain 10k, 100k, 1000k objects and object sizes are set in range of $[0.5, 0.5]$, $[1, 1]$, and $[2, 2]$.

The query point distribution conforms to the data distribution. For real datasets, we use landmark points from the same data sources [10] as query points. For synthetic dataset, query points are uniformly generated. The number of tiers, m, for each query is ranged from 1 to 5. For each setting, we conducted 30 NS queries and the result is the average over the 30 measurements.

We evaluate the performance of *Sweep*, *Ripple* and *Sample* in terms of page accesses, maximum queue length and CPU time under various settings of dataset size and object size. We experiment both synthetic and real datasets. As both dataset size and object size are the two factors affecting the object density, we evaluate the dataset size and the object size independently based on synthetic datasets.

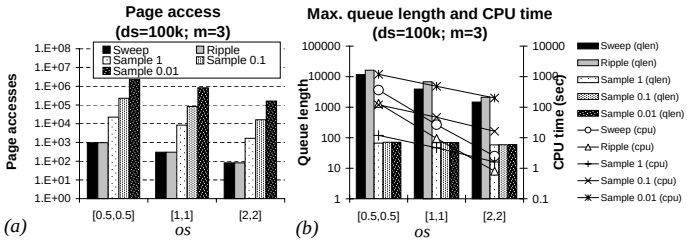


Figure 16. I/O, MEM, CPU (synthetic, varying dataset sizes)

Figure 16 shows the performance (all in log scale) of the algorithms in a setting where object size is fixed at [1,1] and the number of objects is ranged from 10k, 100k to 1000k. The number of tiers, m, is set to 3. The results for other values of m were experimented and they provided similar results. For space saving, they are not included in this paper. The performance for all algorithms deteriorates with smaller object set experiments, because the search checks the farther objects for nearest surrounders; making the search scope larger to cover additional candidate nodes or objects. In term of page accesses (in Figure 16(a)), both *Sweep* and *Ripple* perform equally well and they are better than *Sample*s since *Sample*s access similar sets of pages in different runs. It is observed that *Sample 0.01* incurs most page accesses owing to its narrow search scope per run. In term of memory consumption (in Figure 16(b)) expressed as maximum queue length, *Sample*s are the best because of its narrow sample search scope for each individual run. Meanwhile, for large datasets, *Sweep* saves more space than *Ripple* since the *Sweep*'s lookahead effectively selects objects/index nodes to explore. However, *Sweep*'s lookahead takes more processing time. It is observed that both *Sweep* and *Ripple* provide shorter CPU time when large datasets are used. From the trend, we can expect that all performance metrics for our proposed algorithm will remain low for any larger datasets.

Figure 17 shows the performance (all in log scale) of all

[6]Due to scaling to same square space in this evaluation, on the average, RI's objects are the largest since RI is the smallest state among three; in contrast, CA's objects are the smallest as CA is the largest state.

algorithms in a setting that dataset size is fixed at 100k while the object size is varied from [0.5,0.5], [1,1] to [2,2]. The observation from this figure is similar to that in Figure 16. It can be explained similarly.

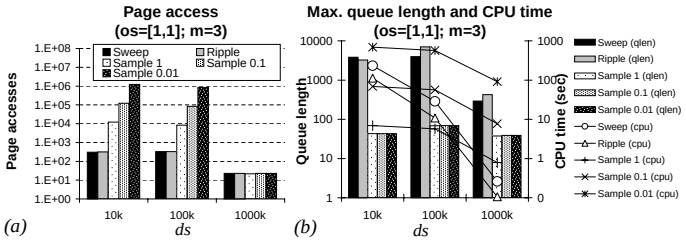


Figure 17. I/O, MEM, CPU (synthetic, varying object sizes)

Using the same set of performance metrics, we evaluate the real datasets: CA, PA and RI. Figure 18(a) depicts the page accesses. *Ripple* and *Sweep* are generally superior to *Sample*s as explained before. As shown in Figure 18(b), *Ripple* provides almost equal CPU time as *Sweep* while *Sweep* generally consumes less memory than *Ripple* except for CA dataset which is a highly skewed dataset. *Sweep* suffers in consuming more memory for priority queues because *Sweep*'s lookahead accumulates entries in the queue. *Sample*s provide the minimal readings.

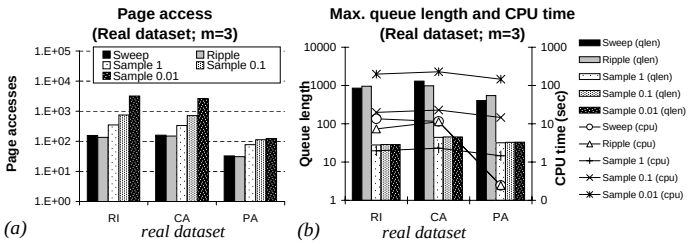


Figure 18. I/O, MEM, CPU (real dataset)

From the experiment results, we have the following observations. Both *Ripple* and *Sweep* can efficiently look up the index with least page accesses. In general, *Ripple* saves in CPU time. *Sweep* is memory saving except in the condition that skewed object distribution or low data sets are experimented.

6 Conclusion

This paper presents a study on a new class of spatial queries: the Nearest Surrounder (NS) queries and its variant, the multi-tier NS queries (mNS). These queries, searching the nearest spatial objects surrounding a query point, are different from the conventional NN queries which have been extensively studied in the past decade. In this paper, we identify these new queries and study their associated query processing issues such as angle-based bounding properties and different search strategies. Based on these we proposed *Sweep* and *Ripple* algorithms.

In this paper, we considered NS queries covering $[0, 2\pi)$ in the search space. In many cases, users would be more interested in finding nearest surrounders in a certain angular range, i.e., $\subset [0, 2\pi)$. Also, an NS query can be executed as a collection of finer NS queries that search surrounders in disjointed angular ranges. This allows us to employ appropriate algorithms in different angular ranges based on different object distribution. Reasonably, a finer NS query with smaller search space incurs less runtime memory in maintaining a queue. We plan to proceed to investigate the feasibility of the NS queries with angular limits and performance improvement by partitioning an NS query.

Acknowledgements

In this research, Wang-Chien Lee and Ken C.K. Lee were supported in part by US National Science Foundation grant IIS-0328881.

References

[1] P. K. Agarwal and J. Matousek. Ray Shooting and Parametric Search. In *Proceedings of the ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada*, pages 517–526, 1992.

[2] N. Backmann, H.-P. Kriegel, R. Schneider, and B. Seegar. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the ACM SIGMOD Conf., Atlantic City, NJ, May 23-25*, pages 322–331, 1990.

[3] L. Downs, T. Moller, and C. H. Sequin. Occlusion Horizons for Driving Through Urban Scenery. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 121–124, 2001.

[4] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. E. Abbadi. Constrained Nearest Neighbor Queries. In *Proceedings of the SSTD Conf., Redondo Beach, CA, Jul 12-15*, pages 257–278, 2001.

[5] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD Conf., Boston, MA, Jun 18-21*, pages 47–57, 1984.

[6] O. Hall-Holt and S. Rusinkiewicz. Visble Zone Maintenance for Real-Time Occlusion Culling. In *DIMACS Workshop on Computational Geometry, Nov 14-15*, 2002.

[7] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.

[8] N. Roussopoulos, S. Kelly, and F. Vincent. Nearest Neighbor Queries. In *Proceeding of the 1995 ACM SIGMOD Conf., San Jose, CA, May 22-25*, pages 71–79, 1995.

[9] D. Salomon. *Computer Graphics and Geometric Modeling*. Springer-Verlag New York, Inc., 1999. ISBN: 0-387-98682-0.

[10] U.S. Census Bureau. *2002 TIGER/Line Files (Website)*. http://www.census.gov/geo/www/tiger/tiger2002/tgr2002.html.