Range-based Obstructed Nearest Neighbor Queries

Huaijie Zhu[†], Xiaochun Yang[†], Bin Wang[†], Wang-Chien Lee[‡]
[†]Northeastern University, China
[‡]Pennsylvania State University, University Park, PA 16802, USA
zhuhjneu@gmail.com, {yangxc, binwang}@mail.neu.edu.cn, wlee@cse.psu.edu

ABSTRACT

In this paper, we study a novel variant of obstructed nearest neighbor queries, namely, *range-based obstructed nearest neighbor* (RONN) search. A natural generalization of *continuous obstructed nearest-neighbor* (CONN), an RONN query retrieves the *obstructed nearest neighbor* for every point in a specified range. To process RONN, we first propose a *CONN-Based* (CONNB) algorithm as our baseline, which reduces the RONN query into a range query and four CONN queries processed using an R-tree. To address the shortcomings of the CONNB algorithm, we then propose a new *RONN by R-tree Filtering* (RONN-RF) algorithm, which explores effective filtering, also using R-tree. Next, we propose a new index, called *O-tree*, dedicated for indexing objects in the obstructed space. The novelty of O-tree lies in the idea of *dividing the obstructed space into non-obstructed subspaces*, aiming to efficiently retrieve highly qualified candidates for RONN processing. We develop an O-tree construction algorithm and propose a space division scheme, called *optimal obstacle balance* (OOB) scheme, to address the tree balance problem. Accordingly, we propose an efficient algorithm, called *RONN by O-tree Acceleration* (RONN-OA), which exploits O-tree to accelerate query processing of RONN. In addition, we extend O-tree for indexing polygons. At last, we conduct a comprehensive performance evaluation using both real and synthetic datasets to validate our ideas and the proposed algorithms. The experimental result shows that the RONN-OA algorithm outperforms the two R-tree based algorithms significantly. Moreover, we show that the OOB scheme achieves the best tree balance in O-tree and outperforms two baseline schemes.

Keywords

Nearest neighbor; Range-based nearest neighbor; Obstacle

1. INTRODUCTION

With the growing popularity of smart mobile devices and the rapid advance in wireless communication and positioning technologies, various spatial queries have been proposed to support the needs of mobile users. Range nearest neighbors (RNN) query [1, 3, 5, 10, 11, 14, 15] is a popular spatial query that retrieves the nearest neighbors to a spatial region where the mobile user is possibly located. It has been regarded as a fundamental query in support of various real applications, which have been discussed widely in the literatures [1, 3, 5, 10] and can be categorized into the following four use scenarios.

- Privacy-preserving. Mobile users are not willing to reveal their exact location information to location-based service providers, due to the concern of location privacy leaking. Spatial cloaking, a well-received privacy preserving technique, proposes to blur the user's exact location into a spatial region in order to satisfy the user's specified privacy requirements [9, 11, 12, 13, 14, 15, 16, 24, 27]. RNN facilitates search of the nearest neighbors with respect to the spatial region.
- Continuous queries. For services that facilitate continuous lookup of nearest neighbors while a user is moving around an area, it is inefficient to process individual NN queries against the whole data set at the server. To submit an RNN query to prefetch all possible nearest neighbors for this area as a candidate set is a better choice. As a result, NN queries issued for objects in this area can be processed locally against the candidate set, saving both computation and communication cost significantly [14, 15, 16].
- Batch processing. Consider the situation where very close NN queries are issued from many different users. Instead of processing these queries separately, it's more efficient to group spatially adjacent NN queries and process them in a batch. An RNN query with a spatial range bounding all the NN query points in the batch can be issued to obtain a candidate set and then the individual NN queries can be resolved with the candidate set returned by RNN [14].
- Location uncertainty. Due to the limitation of underlying positioning techniques (especially those network-based techniques relying on cellular and wifi access points) as well as the sampling imprecision caused by continuous motion, network delays, and frequent location updates (even for devices equipped with GPS), the location uncertainty of a mobile user remains an issue. To support such a user, an RNN query with a spatial region where the user location is guaranteed (to cope with the location uncertainty) can be issued to retrieve the NNs to the user [15, 16].

SIGMOD '16 June 26–July 1, 2016, San Francisco, USA

DOI: http://dx.doi.org/10.1145/2882903.2915234

Nevertheless, these prior RNN works mostly assume a plain space with no obstacles, and thus use Euclidean distance, the shortest distance between a data object and a query range, as the distance measure. While their contributions are significant to the field, there exists a gap to real-life applications, as the space we live in usually has obstacles instead of being plain. For example, a traveler drives around mountains and lakes; troops take detours to avoid enemy's ground; wireless communication messages may avoid routing through an area that suffers electromagnetic interference. Indeed, a recent trend in research of spatial queries is to consider the obstructed space where the *obstructed distance* [23], i.e., the shortest path between two points without crossing any obstacle, is adopted as the distance measure. For example, research studies on the *obstructed nearest neighbor* (ONN) query, which retrieves k ($k \geq 1$) NNs of a given query point in the obstructed space, are reported [23, 25]. Moreover, to support mobile applications, continuous obstructed nearest neighbor (CONN) query for finding nearest neighbors to a given line segment in the obstructed space is proposed [6]. The idea of *visibility graph* (VG) is also proposed to simplify the calculation of obstructed distance [4]. In addition, the problem of spatial clustering in the presence of obstacles is studied [21, 22]. The realistic needs and these research works for supporting mobile users in the obstructed space inspired our research presented in this paper.

Considering the problem of RNN under the context of obstructed space, in this paper, we propose a new spatial query, namely, *range-based obstructed nearest neighbor* (RONN) query.[1] Given a set of data objects P, and a set of obstacles O, an RONN query, specified by a query range R, retrieves all the nearest neighbors corresponding to every point in R based on obstructed distance.[2] An example of the RONN query is illustrated in Fig. 1. As shown, the circled points denote the data objects, the red line segments logically represent the obstacles and the rectangle at the center specifies a query range.[3] The result of this RONN search consists of the blue points. Notice that point p_7 is not in the result set owing to obstacle o_1. p_3 is not in the result set since p_4 is nearer to the query range than p_3. A major challenge faced in processing RONN lies in the adoption of the obstructed distance as a measure, which makes the existing techniques for conventional RNN inapplicable.

In this paper, we first propose a *CONN-Based* (CONNB) algorithm as the baseline to tackle the RONN problem. By reducing an RONN query to a range query and four CONN queries [6], we exploit an R-tree of data objects and obstacles to answer the query. Moreover, observing some shortcomings of the CONNB algorithm, we propose a new *RONN by R-tree Filtering* (RONN-RF) algorithm, which explores

[1]To the best knowledge of the authors, the problem of RNN has not yet been addressed under the context of the obstructed space.

[2]Following the convention in spatial database research, we use a rectangle to represent the query range, which is assumed to be in parallel to x-axis or y-axis. Alternative representations of the query range will be explored in future works.

[3]Although an obstacle can be in any shape (e.g., triangle, pentagon, etc.), line segment is the basic unit of shapes, e.g., rectangular and polygon. for simplicity, we first follow the common assumption adopted in the literature of obstructed spatial queries to use *line segment* for obstacles. Later we also consider the polygons as an extension.

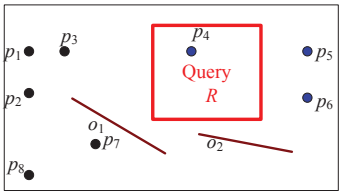


Figure 1: RONN search

effective filtering strategies to facilitate efficient query processing. Nevertheless, to process RONN, there exist two issues in R-tree due to the presence of obstacles. First, it requires multiple traversals of the R-tree for both RONN-RF and CONNB, i.e., as to be detailed later, RONN-RF needs to traverse R-Tree twice and CONNB requires five scans of the R-tree. Second, R-tree indexes both data objects and obstacles in the same Euclidean space, resulting in expensive online computation of obstructed distance during query processing.

To address the above two issues, we propose a new index, namely, *O-tree*, which organizes data objects and obstacles in a way that *non-obstructed subspaces* can be produced to alleviate the computation of obstructed distance. The novelty of O-tree lies in the idea of *dividing the obstructed space into non-obstacle subspaces*, aiming to efficiently retrieve highly qualified candidates for accelerating RONN processing. O-tree is constructed by sequentially selecting obstacles to find pivot points for space division recursively. Eventually, R-trees are embedded in the O-tree to handle the non-obstructed subspaces.

In order to build a balanced O-tree, we propose a notion of tree balance, namely, *obstacle balance*. Based on our analysis, we observe that the tree balance is primarily affected by the orders of obstacles being selected to build the O-index. Accordingly, we propose a space division scheme, namely, *optimal obstacle balance* (OOB). Moreover, we propose an efficient algorithm, called *RONN by O-tree Acceleration* (RONN-OA), which exploits O-tree to accelerate query processing of RONN. In addition, we extend O-tree for indexing polygons. At last, we conduct a comprehensive performance evaluation using both real and synthetic datasets to validate our ideas and evaluate the proposed algorithms.

The contributions made in this research are five-fold:

- We formalize the *Range-based Obstructed Nearest Neighbor* (RONN) query, a new variant of spatial queries in the obstructed space. To the best of our knowledge, this work is the first attempt to tackle the RONN problem.
- We propose two original RONN algorithms based on R-tree. The first one (CONNB) answers an RONN query with one range query and four CONN queries. The second one (RONN-RF) explores effective heuristics to facilitate efficient filtering by R-tree.
- As an alternative to R-tree, we propose a novel index (O-tree) by dividing the obstructed space into non-obstructed subspaces. This new index prevails R-tree in supporting efficient RONN query processing. In addition, we propose a space division scheme (OOB) for constructing the O-tree and study the tree balancing issue as an optimization problem. Moreover, we extend the O-tree to index obstacles as polygons (see Appendix).
- Based on O-tree, we propose a new algorithm (RONN-OA) for efficient query processing of RONN.

- We conduct extensive experiments to evaluate the proposed algorithms and schemes. The experimental result shows that the RONN-OA and RONN-RF algorithms significantly outperform the baseline algorithm (CONNB), with 360% and 300% improvement, respectively. Meanwhile, our study shows that OOB is the scheme of choice for space division in O-tree.

2. RELATED WORK

A lot of research effort related to our work, including *range based spatial queries*, *ONN queries*, and *spatial cloaking query* (for location privacy preservation), have been made in the field of spatial databases.

2.1 Range-based Spatial Queries

Many spatial queries have been studied over the years, e.g., range queries, nearest neighbor (NN) queries, continuous nearest neighbor (CNN) queries [18, 26, 20], *range nearest neighbor* (RNN) queries [10], and *range based skyline* (RSky) queries [16]. Among them, the "range-based" spatial queries are the most relevant to our work.

The notion of kRNN queries has been proposed to retrieve k nearest neighbors for every point in a specified range [10]. New algorithms are developed based on either planar geometry in two-dimensional space or linear programming in high-dimensional spaces. In addition, a solution-based auxiliary index, called *EXO-tree*, is developed to support a variety of NN queries, including traditional NN, CNN, and RNN searches. However, the design of EXO-tree does not consider the obstructed space and thus is not applicable to RONN.

Another relevant work is the RSky queries in mobile environment [16]. Two algorithms, namely, *I-SKY* and *N-SKY*, are proposed to find skyline objects in static and highly dynamic datasets, respectively. The authors also further consider the problem under the contexts of probabilistic and continuous queries. To process probabilistic RSky queries, they propose new pruning techniques to improve the computational efficiency. Moreover, they compute the valid scope for each skyline object to process continuous RSky queries efficiently. Again, the above RNN and RSky queries are considered only in non-obstructed space. In this work, we design a novel index for obstructed space to enhance query processing of RONN.

2.2 ONN Queries

Due to the impact of obstacles, a growing number of research lines considering the obstructed spatial space, including obstructed spatial queries, visible spatial queries, and obstructed spatial clustering, have been explored.

In obstructed spatial queries, Zhang et al. [25] propose algorithms for answering classical spatial queries, such as range queries, NN search, distance join queries, closest pair queries and distance semi-join queries, in the obstructed space. Xia et al. study the *obstructed nearest neighbor* (ONN) query which finds the kNNs of a given query point based on the obstructed distance [23]. This work adopts R-tree to index the data objects and obstacles in order to incrementally compute the obstructed distance by filtering out a large number of obstacles. Gao and Zheng study the continuous obstructed nearest neighbor (CONN) search [6]. As our baseline algorithm is to reduce an RONN to a range query and four CONN queries, we discuss some key ideas of this work in Section 3. RONN is fundamentally different from CONN as the RONN query is based on a range rather than a line segment.

Research works on visible spatial queries also consider the existence of obstacles. However, these queries adopt Euclidian distance instead of obstructed distance as the distance measure, i.e., two objects are visible to each other iff the straight line segment connecting them does not cross any obstacle. A visible nearest neighbors query returns the NNs visible to a given query point [19]. Gao explores the continuous visible nearest neighbor query which finds the visible NN of every point along a query line segment [7].

In addition, the problem of spatial clustering in the presence of obstacles has attracted considerable attention in recent years. It divides a set of two-dimensional data objects into smaller homogeneous groups (i.e., clusters) by taking into account the impact of obstacles or obstructed distance. A number of clustering algorithms with obstacle constraints have been reported in the literature [21, 22].

Although these works have considered the obstructed space or the impact of the obstacles, the RONN query has not been studied previously.

2.3 Spatial Cloaking Query

In location privacy preservation, several works propose to perform spatial cloaking for protecting users' exact location from leaking by specifying a *cloaked range*, instead of the exact query location. The spatial cloaking query calculates a safe region and provides all data objects inside the safe region (i.e. candidates) instead of a single exact answer [17].

Casper [17] blurs a user's exact location into a cloaked spatial range, in form of a rectangle, based on privacy requirements specified by the user. Accordingly, the server processes the query based on the cloaked spatial ranges. Gruteser, et al. [9] propose a Quadtree-based algorithm for spatial cloaking. It achieves identity anonymity in LBS by spatio-temporal cloaking based on a k-anonymity model, i.e., the cloaked location is made indistinguishable from the location information of at least k-1 other users. Gedik and Liu [8] propose a novel spatio-temporal cloaking algorithm, called CliqueCloak, which provides k-anonymity in LBSs for mobile users. The most close work to ours is [27]. Zhu et al. [27] construct a cloaked range corresponding to the exact location using a pyramid tree to address the obstructed nearest neighbor problem under the context of obstructed space. Different from the above cloaked ranged query techniques, RONN returns the nearest neighbors to every possible point in the cloaked range. We show the comparison results in Section 7.

3. PRELIMINARIES

In this section, we first introduce some background and definitions for our research. Then we formulate the RONN problem.

3.1 Background and Definitions

In this work, we focus on the *obstructed space*. We assume that there exists a set of data objects P and a set of obstacles O in the space. We assume that people moving from one point to another in the obstructed space follow the shortest path while avoiding the obstacles. In this space, a proper distance measure (other than Euclidean distance) is needed for obstructed spatial queries. Thus, we define the notion of

obstacle-free path and use it to define the *obstructed distance* between two points.

Definition 1. (OBSTACLE-FREE PATH [23]). Consider two points p, p' in the obstructed space. An *obstacle-free path*, $\mathsf{P}(p, p') = \langle d_0, d_1, ..., d_n \rangle$ (where d_i is a point in the space and $d_0 = p, d_n = p'$), consists of n line segments of straightline subpaths $\langle d_0, d_1 \rangle, \langle d_1, d_2 \rangle, ..., \langle d_{n-1}, d_n \rangle$ that connect p and p'.[4]

Definition 2. (OBSTRUCTED DISTANCE [23]). Let the distance of an obstacle-free path $\mathsf{P}(p, p')$ be $|\mathsf{P}(p, p')| = \sum_{i \in [0, n-1]} dist(d_i, d_{i+1})$ where $dist(d_i, d_{i+1})$ is the Euclidean distance between d_i and d_{i+1}. The obstructed distance between p and p', denoted by $||p, p'||$, is the length of the *shortest obstacle-free path* (*shortest path* for short) from p to p', i.e., $\nexists \mathsf{P}(p, p'), |\mathsf{P}(p, p')| < ||p, p'||$.

Based on the notion of obstructed distance, we now define the *obstructed nearest neighbor* to a query point.

Definition 3. (OBSTRUCTED NEAREST NEIGHBOR). For a given query point q in the obstructed space, the data object $p \in P$ is the obstructed nearest neighbor (ONN) of q iff $\forall p' \in P - p, ||p, q|| \leq ||p', q||$.

By capturing the anticipated moving trajectory of a mobile user as a line segment, the idea of ONN can be extended to CONN as defined below.

Definition 4. (CONTINUOUS OBSTRUCTED NEAREST NEIGHBOR). Given a query line segment $\overline{q}$, the continuous obstructed nearest neighbors corresponding to $\overline{q}$, denoted by $CONN(\overline{q})$, is a subset of objects in P such that $\forall$ point $q \in \overline{q}$, $ONN(q) \in CONN(\overline{q})$.

Notice that the CONN of a line segment defined above is the union of ONN objects with respect to the (infinite number of) points on the line segment $\overline{q}$. Alternatively, $CONN(\overline{q})$ can be compactly represented as the union of CONNs corresponding to a sequence of line segments $\overline{q}_1$, $\overline{q}_2$, ..., $\overline{q}_t$ such that (i) $\overline{q}$ is the concatenation of $\overline{q}_1, \overline{q}_2, ..., \overline{q}_t$, (ii) there exists only one CONN, i.e., the ONN, for each $\overline{q}_i$ (where $i = 1..t$), and (iii) $ONN(\overline{q}_i) \neq ONN(\overline{q}_{i+1})$ (where $i = 1..t-1$).

Accordingly, the points $s_1, s_2, ..., s_{t-1}$ that split $\overline{q}$ as $\overline{q}_1$, $\overline{q}_2$, ..., $\overline{q}_t$, termed as *split points*, are defined below.

Definition 5. (SPLIT POINTS FOR CONN). Let the query line segment $\overline{q}$ be denoted by $\overline{ab}$, where a and b are the two end points of $\overline{q}$. Consider the $\overline{q}_1, \overline{q}_2, ..., \overline{q}_t$ associated with $CONN(\overline{q})$ where $\overline{q}_1 = \overline{as_1}, \overline{q}_2 = \overline{s_1 s_2}, ..., \overline{q}_t = \overline{s_{t-1} b}$. The points $s_1, s_2, ..., s_{t-1}$ are split points of $\overline{ab}$. Moreover, let p_i and p_{i+1} be the ONNs to all the points along $\overline{s_{i-1} s_i}$ and $\overline{s_i s_{i+1}}$, respectively. Then the obstructed distance from s_i to p_i and p_{i+1} are the same, i.e., $||p_i, s_i|| = ||p_{i+1}, s_i||$.

EXAMPLE 1. Fig. 2 *illustrates CONN. As shown,* $\{p_1, p_2, ..., p_6\}$ *are data objects,* $\{o_1, o_2, ..., o_4\}$ *are obstacles, and* $\overline{ab}$ *is the query line segment. The output of the CONN search is* $\{\langle p_1, \overline{as_1} \rangle, \langle p_2, \overline{s_1 s_2} \rangle, \langle p_6, \overline{s_2 s_3} \rangle, \langle p_3, \overline{s_3 b} \rangle\}$*, where* s_1*,* s_2 *and* s_3 *are the split points. The split point* s_1 *for point* p_1 *and* p_2 *can be computed by a quadratic algorithm* [6].

The notion of CONN and split points are used to describe some of our algorithms built upon CONN.

[4]Note that there exists a straightline path between two points if and only if there is no obstacle blocking the path.

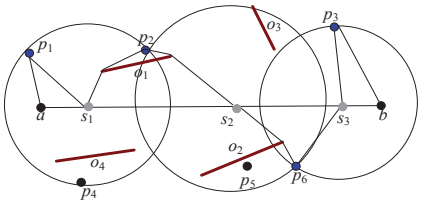


Figure 2: CONN search

3.2 Problem Formulation

In this section, we formally define the RONN query.

Problem Statement. *Given a set of data objects* P *and a set of obstacles* O*, a range-based obstructed nearest neighbor (RONN) query* Q *(specified by a rectangular range* R*), retrieves all the obstructed nearest neighbors of points in the query range, i.e.,* $RONN(Q) = \{ONN(p) | p \in R\}$.

4. R-TREE BASED QUERY PROCESSING

In this section, we propose two R-tree based algorithms for RONN query processing.

Given an RONN query Q, specified by a query range R, any object located inside R is automatically an RONN to the query. Thus, an idea to answer an RONN is to first issue a range query to find all "internal" RONNs located within R and then find the "external" RONNs located outside R. Intuitively, as indicated in Lemma 1, these external RONNs of R are the ONNs to the boundary of R.

LEMMA 1. *An object* p *is an external RONN of* R *if and only if 1)* p *is not located in* R*; and 2)* p *is an ONN for at least one point on the boundary of* R.

PROOF. For the necessary condition, it is inherent from the RONN definition that if p is an external RONN to R, then (1) and (2) above are satisfied.

Next, we prove the sufficient condition by contradiction. *Assume that there exists an RONN* p *outside* R *which is not the ONN to any point on the boundary of* R. Since p is an RONN, p is the ONN for at least one point m in R (see Fig. 3 for illustration). Let n be the intersection point of line segment $\overline{pm}$ and the boundary of R where n is located. Based on the assumption, as p is not the ONN of n (which is a point on the boundary), there must be another data object p' such that $||n, p'|| < ||n, p||$. Thus, $||n, p'|| + ||n, m|| < ||n, p|| + ||n, m||$. As a result, we have $||m, p'|| < ||m, p||$, which contradicts our assumption that p is the ONN of m. Therefore, the sufficient condition must hold.

Combine the cases of necessary and sufficient conditions, the proof completes. □

4.1 CONN-Based Algorithm

According to Lemma 1, this RONN query can be answered by issuing four CONN queries (using each edge of R as the query line segment) and one range query. The union set of data objects returned from the above five queries answers the RONN query. This *CONN-based* (CONNB) algorithm, serves as a baseline for comparison in our evaluation.

In CONNB, R-tree has been employed to index both data objects and obstacles. Accordingly, a quadratic algorithm proposed in [6] can be used to return the CONNs for each edge of R. As a result, CONNB is also quadratic.

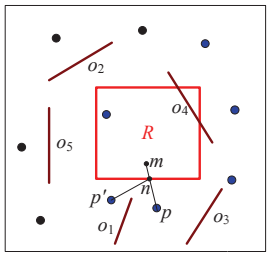


Figure 3: Illustration for Lemma 1

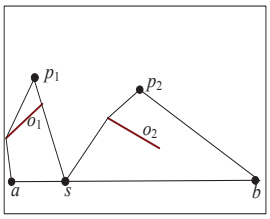


Figure 4: Maximum Obstructed Distance

4.2 R-tree Filtering Algorithm

The *CONNB* algorithm directly invokes range and CONN queries and thus does not explore some potential filtering strategies during query processing. Notice that some intermediate RONNs acquired during query processing may potentially disqualify certain RONN candidates located far away from the query range R. Thus, an idea to facilitate efficient filtering is to explore the RONNs using a distance-browsing approach to obtain RONNs in accordance with their minimal Euclidean distance to R. Accordingly, in this section, we propose a new RONN algorithm, called *R-tree Filtering Algorithm* (RONN-RF), which built upon R-tree to facilitate distance-based browsing and filtering.

To achieve our goal, RONN-RF maintains a number of *maximum obstructed distances*, one for each edge of the query range, from the set of intermediate RONNs to the query range R in order to filter out unqualified candidates. While using R-tree facilitates browsing of data objects in ascending order of their minimal Euclidean distance to R, a major challenge faced in answering the RONN query is the use of obstructed distance as a measure. Here we introduce the notion of *max obstructed distance of an object set to a line segment*, as defined below, for discussion of RONN-RF.

Definition 6. (MAXIMUM OBSTRUCTED DISTANCE OF AN RONN SET TO LINE SEGMENT). Given a set of intermediate RONNs P and a line segment $\overline{ab}$ on the obstacle space, we say that $||m, p_m||$ is the *maximum obstructed distance* from P to $\overline{ab}$, denoted by $max\_odis(P, \overline{ab})$, iff there exist a point m on $\overline{ab}$ such that $\forall$ point i on $\overline{ab}$, $||i, ONN_P(i)|| \leq ||m, ONN_P(p)||$ where $ONN_P(i)$ denotes the point in P serving as the ONN of point i.

EXAMPLE 2. *Consider* Fig. 4, *where* $O = \{o_1, o_2\}$ *is an obstacle set,* $P = \{p_1, p_2\}$ *is a set of intermediate ONNs obtained during query processing, and* $\overline{ab}$ *is a side edge of the query range. As shown,* p_1 *and* p_2 *are the RONNs for all points located on* $\overline{as}$ *and* $\overline{sb}$*, respectively. In this example, we have* $max\_odist(p_1, \overline{as}) = ||p_1, s|| < ||p_2, b|| = max\_odist(p_2, \overline{sb})$*, which thus is* $max\_odis(P, \overline{ab})$.

In RONN-RF, four *maximum obstructed distances* (MODs), one for each edge of the query range, are maintained to filter out unqualified objects from examination and to determine the terminating condition for query processing. RONN-RF first issues a range query to obtain the internal RONNs, which are used to initialize the four MODs.[5] Next, RONN-RF traverses the R-tree to visit the data objects in ascending order of their minimal Euclidean distances to query range R in order to determine whether they are ONNs to each edge. During the traversal of R-tree, a heap H is used to store the R-tree entries (including *minimum bounding boxes* (MBBs), data objects, and obstacles) in order to process them in ascending order of their minimum distances to R. Initially, H contains only the root entry. By de-heaping the entry e with the shortest distance to R, RONN-RF checks whether the minimal Euclidean distances from e to R is greater than all the current MODs. If this is true, e will be eliminated and a new entry is de-heaped. Otherwise, if e is an MBB (i.e., internal index node), all its child entries are pushed into H for further processing. On the other hand, if e is a leaf node of R-tree, there are two cases: (i) e is an obstacle: it is inserted into the visibility graph VG, which is used for computing the split points on edges for RONNs; and (ii) e is a data object: it updates RONN results and MODs corresponding to each edge by finding split points on edges of the range boundary (to be discussed further later). The above query processing steps (i.e., de-heaping a new entry from H and so on) are repeated until H becomes empty.

The operation of finding split points (called $FindSP$) plays an important role in query processing of RONN-RF. We follow the idea of CONN algorithm [6] to compute the split points for the query boundary. Notice that, instead of sequentially finding RONNs corresponding to each edge (as CONNB does), RONN-RF examines a data object corresponding to all edges simultaneously. This results in significant performance improvement, in addition to the major saving obtained from filtering out unqualified RONN candidates, because RONN-RF needs to scan R-tree only once instead of four times. Let $r_i (i = 1..4)$ denote the corner points of the query range R and $\overline{E}_{ij}$ denote the edge $\overline{r_i r_j}$. Given the intermediate RONN result set (RS_{ij}) and the visibility graph (VG) of obstacles visited so far, we are able to decide whether the data object under examination p is an RONN to $\overline{E}_{ij}$. If it is, we are able to obtain a list of split points that divide $\overline{E}_{ij}$ into line segments being corresponding to RONNs in RS_{ij}. Moreover, the MOD for $\overline{E}_{ij}$ is updated.

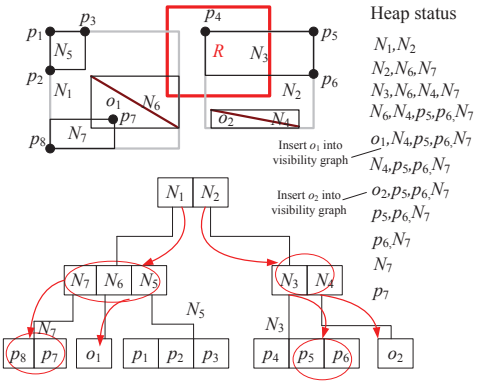


Figure 5: An example of RONN-RF

Fig. 5 shows an example for illustration of the RONN-RF algorithm. As shown, there are eight data objects $p_1, p_2, \ldots, p_8$ and two obstacles o_1 and o_2, maintained in an R-tree index. In the index, the root node contains two internal nodes N_1 and N_2 which cover N_5, N_6, N_7, and N_3, N_4, respectively. In turn, N_5 contains data objects p_1, p_2, and p_3, N_7 contains data objects p_7 and p_8, and N_3 contains

[5]These MODs are incrementally updated as the RONN answer set is refined during query processing.

data objects p_4, p_5, and p_6. On the other hand, N_6 and N_4 contains o_1 and o_2, respectively. Given a query range R (i.e., the red rectangle), RONN-RF first issues a range query to obtain the internal RONN p_4. Accordingly, it initializes the four MODs corresponding to each edge of R using p_4. Next, to identify external RONNs by traversing R-tree, we initialize H with N_1 and N_2, sorted in ascending order of their Euclidean distances to R.[6] We proceed to pop the top entry N_1 from H. As N_1 is an internal node, we compare the Euclidean distances of its child entries N_5, N_6 and N_7 with the maintained MODs and insert N_6 and N_7 back to H. Notice here that N_5 is filtered by the maintained MODs. Similarly, N_2 is de-heaped from H and then N_3, N_4 are inserted back. Continue with the process, N_3 is de-heaped and data objects p_5 and p_6 are inserted back. Here, since p_4 is already in the RONN result set, it is not put back to H. After that, we pop N_6 which contains an obstacle o_1. Thus, we insert o_1 into the visibility graph. Similarly, we handle N_4 and insert o_2 into visibility graph. Then p_5 is de-heaped as a data object and found to be an RONN (by updating current split points via VG). MODs corresponding to each edge of R are updated. Similarly, p_6 are included in the result set. Finally, we pop N_7 and insert p_7 into H because the Euclidean distance of p_8 is greater than some of the maintained MODs. p_7 is de-heaped as a data object but it can not update the current RONN results due to the blocking of obstacle o_1. At this stage, H becomes empty so the query processing ends. The final RONN result set includes p_4, p_5 and p_6.

As shown in the above example, we maintain four MODs, one for each edge. An RONN candidate is filtered only if all its Euclidean distances to the edges of query boundary are greater than the corresponding MODs. The filtering condition, as specified in Theorem 1, effectively reduces the search space of RONN solutions and thus alleviates significant computation overhead. Algorithm 1 presents the pseudo-code of RONN-RF. Notice that the main issue on R-tree is that the minimal Euclidian distance of accessing order is not very effective due to the affecting of obstacles, i.e., the minimal Euclidian distance of N_7 is very small but this is not the RONN result.

THEOREM 1. *Given a query range R and the current RONN result set RS, an entry e of R-tree is filtered iff for each edge $\overline{ab}$ of R, $min\_dist(e, \overline{ab}) > max\_odis(RS, \overline{ab})$.*

In Theorem 1, notice that $min\_dist(e, \overline{ab})$ is Euclidean distance, which is easy to compute. Moreover, it can filter distant data objects effectively because the Euclidean distance represents the lowerbound of obstructed distance from an entry e to the query range R.

Comparing to CONNB algorithm, the RONN-RF algorithm has two advantages: 1) RONN-RF needs only one traversal of R-tree to find the external RONNs while CONNB requires four scans of the R-tree; and 2) In traversal of the R-tree, RONN-RF explores effective heuristics to prune the RONN search space, while CONNB do not exploit intermediate (internal and external) RONNs for filtering in query processing.

[6]Notice that both N_1 and N_2 have Euclidean distances to some edges of R shorter than the current RONNs' MODs to these edges.

Algorithm 1: R-tree Filtering Algorithm (RONN-RF)

Input: Data objects and obstacles R-tree T, min-heap $H = (root(T), 0)$, and query range R
Output: RONN results list RS

```
1  inRes ← Rangequery(T, R);/* internal RONNs*/
2  RS ← inRes; RS_k ← inRes ; k = 1,2,3,4 /*storing ONNs for four edges*/
3  over_k ← 0; k = 1,2,3,4 /*Terminating signal for four edges, 0 means query processing for the corresponding edge is not over yet*/
4  while true do
5      k ← 1;/*Starting from edge 1*/
6      for Each edge Ē_ij of R do
7          RS_k ← FindSP(Ē_ij, RS_k); /*Finding split points*/
8          MOD_k ← max_odis(Ē_ij, RS_k);
9          if H ≠ ∅ and H.head.key > MOD_k then
10             over_k = 1. /* Processing for edge k is over*/
11         if over_1&over_2&over_3&over_4 or H ≐ ∅ then
12             return; /*There is no entry which will be an ONN for any edge*/
13         k++;
14     while H ≠ ∅ do
15         de-heap the top entry (e, key) of H;
16         if e is a data object then
17             k ← 1;
18             for key < MOD_k do
19                 insert e into RS_k;
20                 k++;
21             break;
22         else
23             for each child entry e_i ∈ e do
24                 if min_dist(e_i, R) < MAX(MOD_k) then
25                     insert e_i into H based on min_dist(e_i, R).
26     RS ← RS ∪ RS_k;
27 return RS;
```

5. O-TREE FOR OBSTRUCTED SPACE

As shown earlier, R-tree is used in CONNB and RONN-RF to index both data objects and obstacles in the obstructed space. While R-tree is a well-received index structure for spatial databases, however, it faces two issues in processing RONN queries due to the presence of obstacles. *One is the need for multiple traversals of R-tree in both RONN-RF and CONNB.* When doing internal RONN (range query) and external RONN query, RONN-RF needs to traverse R-tree twice and CONNB requires five scans of the R-tree. This inspires us to explore processing ideas that requires only one scan of the (indexed) data to process internal and external RONN queries all together. *The other issue is the expensive online computation of obstructed distance.* Note that R-tree is built based on the Euclidean space, i.e., Euclidean distance is used as the measure among data objects, obstacles and minimum bounding boxes (MBBs) in the index. As a result, R-tree-based RONN algorithms have to

deal with expensive online computation of obstructed distance. Recall the example in Fig. 5. While p_1, p_2 and p_3 (actually N_5) have been effectively pruned early by RONN-RF, p_7 and p_8 have to be accessed since their Euclidean distances to R is small. This incurs expensive computation of their obstructed distances to R before they are eventually excluded from the answer. Thus, we aim to reduce access of data objects in online computation of obstructed distance.

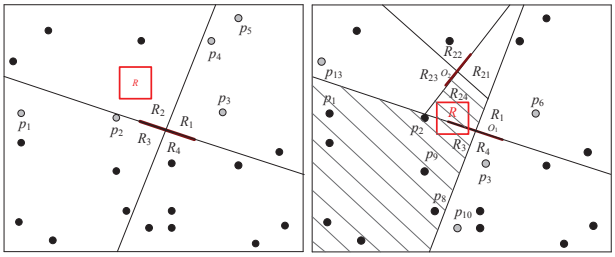
To address these issues, we propose a novel index, called *O-tree*, to support efficient RONN query processing in the obstructed space. The novelty of O-tree lies in the idea of *dividing the obstructed space into non-obstructed subspaces* such that we can use R-tree to index the data objects associated with these subspaces. Our idea is to divide the space into four subspaces along the line of a selected obstacle and some perpendicular lines, e.g., its bisector. Recursively we divide the subspaces into more subspaces to build an index structure. The indexed subspaces may ideally consist of non-obstructed spaces which can facilitate efficient computation and pruning in query processing. While the idea is simple and elegant, the construction of O-tree still faces some design challenges, e.g., tree balance.

In the following, we first discuss the issues of space division in O-tree construction and propose a scheme to address this issue. At last, we present the O-tree construction algorithm.

5.1 Space Division

As mentioned earlier, we aim to construct the O-tree by dividing an obstructed space into less obstructed subspaces (towards non-obstructed subspaces). Without loss of generality, we first consider an obstructed subspace with only one obstacle. Fig. 6(a) shows an example of such obstructed space. By dividing the space along the lines of the obstacle and its perpendicular bisector, we have four non-obstructed subspaces.

Following the design of conventional space-division based spatial indexes, e.g., Quad-tree or k-d tree, we use a subtree to cover the data objects located in each subspace (e.g., three black circles are covered in Region R_2). Nevertheless, this design inherently faces an issue in spatial query process, i.e., for a query point (a query range in the case of RONN) located close to the boundary of a region, the data objects located in neighbor regions need to be accessed. For example, in Fig. 6(a), not only the black circles in R_2 but also all the other grey circles in R_1, R_3 and R_4 need to be accessed to find the RONNs, which would obviously defeat the ultimate goal of our O-tree. To address this issue, we trade some storage overhead for processing efficiency in O-tree by including, for a giving region (subspace), all data objects that should be considered to answer any RONN query specified in the region. This set of additional data objects can be obtained by precomputing CONNs to the boundary line segments of the region during O-tree construction. Therefore, for the example in Fig. 6(a), O-tree stores the data objects denoted by both black circles in R_2 and the grey circles, $p_1, p_2, ..., p_5$. As such, all the candidate data objects for a given query range R can be quickly retrieved by accessing the regions intersected with R. This feature of O-tree is also applicable to the general case of a space with multiple obstacles. As shown in Fig. 6(b), only data objects associated with R_{24} and R_3 need to be examined in query processing (to be detailed in Section 6).



(a) One obstacle (b) Multiple obstacles

Figure 6: An example of space division

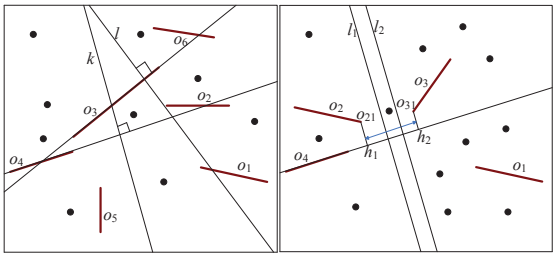


Figure 7: No strict obstacle balance

Figure 8: Illustration for equal perpendicular line

5.2 Optimal Obstacle Balance Scheme

Two key questions that may naturally arise during the discussion of space division earlier are:

- Which obstacle should be first selected to divide the space?
- For a given obstacle, is its perpendicular bisector the best choice to divide the space?

To answer these two questions, a fundamental yet closely related question is "How the divided spaces and thus the resultant O-tree are structured, if we choose obstacles in different orders with different "pivot points" to divide the space?" We argue that a balanced O-tree is desirable and thus propose a space division strategy to achieve tree balance. In O-tree, as obstacles are used to determine the pivot points for space division, we aim to maintain the *obstacle balance*, which is defined as follows:

Definition 7. (OBSTACLE BALANCE) Selecting one obstacle (obstacle line and its perpendicular line) to divide the space into four subspaces. In each dividing step, the obstacle balance is satisfied by the condition $|n_i - n_{i+1}| <= 1$ (n_i denotes the number of obstacles in each subspace), i.e., the difference of height of each branch is less than 1.

According to the above strict obstacle balance definition, can we always construct an obstacle balance tree? The answer is not. In fact, we can not always construct a balanced tree based on selection of obstacles and their perpendicular lines. (Sometimes there is no answer for the problem). Fig. 7 illustrates an example. From Fig. 7, we can see that if we choose o_4 (or o_3) to divide the space, there is no suitable perpendicular line to satisfy strict obstacle balance. The same situation still exists when choosing other obstacles to divide the space.

Since we can not always build a strictly balanced tree, we aim to build an optimal approximate balanced tree. Ac-

cordingly, in order to construct an optimal balanced tree, we propose a new space division scheme, called *optimal obstacle balance* (OOB) scheme, for O-tree. This scheme corresponds to an optimization problem which can be solved using enumeration.

Definition 8. (OPTIMIZATION PROBLEM) Select one obstacle (i.e., obstacle line) and one of its perpendicular lines to divide the space into four subspaces in each space dividing step. The goal is to minimize the objective function $\sum(|n_i - n_j|^2)$ $(i < j, i, j = 1, 2, 3, 4)$ where n_i, n_j denote the number of obstacles into two different subspaces.

The OOB scheme to choose an obstacle and a perpendicular line:

(1) there are n choices of obstacles to divide the space. This can be tried one at a time, so it takes $O(n)$ time.

(2) for a chosen obstacle, there are infinite choices to select a perpendicular line, which is obviously infeasible to enumerate one by one. So we aim to try some finite (a constant m) choices in selecting the best perpendicular line. Let's show an example in Fig. 8 for illustration. We project the endpoints of each obstacle on the chosen obstacle line. The projection of the endpoint o_{21} on obstacle line o_4 is h_1, while the projection of the endpoint o_{31} is h_2. We can see that there are also infinite perpendicular lines (such as l_1 and l_2) to choose from between h_1 and h_2. However, the objective score of choosing l_1 (or l_2) is the same as that of choosing perpendicular line through h_1 (or h_2). In other words, a certain range of perpendicular lines always equal to one perpendicular line through the projection of endpoints. We only need to compute the objective scores of the perpendicular lines through the projections of all endpoints to find the minimal score. As there are $2(n-1)$ perpendicular lines through all the endpoints, this step takes $O(2(n-1))$ time.

In total, this problem can be solved in $O(n * 2(n-1)) = O(n^2)$ time.

5.3 O-tree Index Structure

Based on the above discussion on space division, *O-tree* index first exploits obstacles to divide the obstructed space into a number of non-obstructed subspaces, which cover only data objects. Accordingly, we use R-tree to index the data objects in these Euclidean subspaces.

In summary, the O-tree index is structured as follows. The root node, denoting the whole obstructed space, points to four child nodes. Each of these child nodes denotes a region in the obstructed space. The internal nodes of *O-tree* maintain entries of $(reg, obs, optr_1, optr_2, optr_3, optr_4)$, while the leaf nodes contain entries of the form $(reg, rptr)$ where reg denotes a 2-dimensional polygon region, obs is the obstacle dividing the space, and $optr_i$ and $rptr$ denote pointers to O-tree and R-tree nodes, respectively. Fig. 9 illustrates the space divisions and the corresponding O-tree. Obstacle o_1 firstly divides the region into four subregions and the root has four child nodes (R_1, R_2, R_3, R_4). Next, R_3 is divided into four subregions (R_{31}, R_{32}, R_{33}, R_{34}) utilizing obstacle o_2, and the other three subregions directly form the leaf node. Internal node R_3 stores obstacle o_2, $optr$ pointing to four O-tree children R_{31}, R_{32}, R_{33}, and R_{34}. Leaf node R_{34} stores $rptr$ to an R-tree which contains N_1 and N_2. In turn, N_1 contains data objects p_3 and p_4, while N_2 contains p_5 and p_6. Similarly, R_{33} stores $rptr$ to another R-tree which contains N_2.

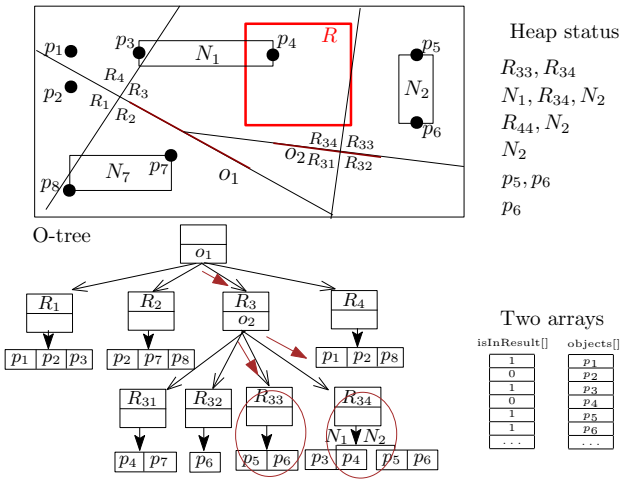


Figure 9: An example of O-tree

Reducing Data Redundancy Using Arrays for O-tree. As shown in Fig. 9, data objects p_1 and p_2 are stored twice in R-trees of nodes R_2 and R_3, respectively. The same case is for data objects p_5 and p_6. As there exists some data redundancy for O-tree index, two potential issues arise: (i) index size increases and (ii) possible redundant traversals of the same data object in query processing.

To reduce the index size, we store all the data objects in an array *objects* and store the array index like *id* for each object in our O-tree index. As shown in Fig. 9, there is an array *objects* for all the data objects. This reduces index size to some extent. As for traversal of data objects in the O-tree, if data object p is examined, we can check if its duplicate p' is already in the result set. We also build an array $isInResult$ corresponding with *objects* array in Fig. 9. $isInResult$ marks whether a data object is in result. As we can obtain the *id* of data object p from O-tree index, whether p is already in the result can be checked quickly. If p is not already in result, we mark it with $isInResult[id] = true$.

5.4 Building O-tree

In this section, we present our O-tree construction algorithm that performs space division recursively and fills its leaf nodes with R-trees containing data objects. The pseudo code of space division algorithm is shown in Algorithm 2.

Algorithm 2: Space division Algorithm

Input: A set of obstacles O, An O-tree node denoted by $Node$ (which contains information of its covered region, denoted by reg)
Output: The constructed O-tree denoted by $Node$

```
1 if O ≠ ∅ then
2     Use OOB scheme to decide one obstacle obs;
3     Divide Region into four subregions and create four
      children optrs;
4     for each optr do
5         Find all the obstacles O' inside the region of
          optr;
6         Space_division(O', optr);
```

At first, we initialize the *root* node of O-tree with *reg* that covers the whole obstructed space. That is, the region of *root* covers all the points and obstacles. Then we recursively invoke the space division algorithm with Space_division(O, *root*) to build up the O-tree. As shown in Algorithm 2, if the current *O-tree* node has obstacles in its covered region, i.e., $O \neq \varnothing$, we utilize OOB scheme to choose one obstacle *obs* (Line 2). Then we divide the current node's region into four subregions and create four children nodes (Line 3). At last, we recursively invoke the Space_division algorithm to build up subtrees for the subregions. Finally, the subtrees are put together as an O-tree (Lines 4-6). After the O-tree is constructed, we then fill the leaf nodes of *O-tree* index with corresponding R-trees by invoking CLN algorithm.

We analyze the time complexity of *Space division* Algorithm. For each region, if there are n obstacles to divide the space, utilizing OOB scheme to do the optimal space division takes $O(n^2)$, and for its four children regions, it takes $O(4(\frac{n}{4})^2)$. So for all the space divisions, it takes $O(n^2 + 4(\frac{n}{4})^2 + 4(\frac{n}{4^2})^2 + ... + 4(\frac{n}{4^h})^2)$=$O(n^2)$, where h is the height of the tree.

Algorithm 3: Complete Leaf Nodes Algorithm (CLN)

Input: A set of points P, a set of obstacles O and O-tree node denoted by $Node$
Output: A *O-tree*

```
1 if Node is not Leaf Node then
2     for each child optr of Node do
3         CLN(P, O, optr);
4 else
5     Create an R-tree root node pointed by rptr for Node;
6     Find all the data objects inside region of ONode and insert them into the R-tree via rptr;
7     for each edge E_ij of the region of Node do
8         res ← CONN(E_ij, O, P);
9         insert res into the R-tree via rptr;
```

The pseudo code of the CLN algorithm is presented in Algorithm 3. The basic idea is to hang corresponding R-tree which contains ONNs of boundary and points inside with each leaf node. To complete the leaf node of O-tree with R-tree, we recursively traverse O-tree from the root. For each O-tree node, we check whether it is a leaf node. If not, we move on its children (Lines 1-3). If it is a leaf node, we create an R-tree for the O-tree node by inserting all the points inside this leaf node's region into the R-tree (Lines 5-6). At last, we compute the ONNs for each edge of the node's region [6] and insert them into the R-tree (Lines 7-9).

6. QUERY PROCESSING ON O-TREE

In this section, we first introduce a new RONN query processing algorithm, called *RONN-OA*, based on O-tree. Next, we analyze the time complexity of CONNB, RONN-RF and RONN-OA algorithms.

6.1 RONN-OA Algorithm

Similar to RONN-RF, RONN-OA maintains four maximum obstructed distances (MODs), one for each edge of the query range R, to filter out unqualified objects and to determine the terminating condition. RONN-OA first issues a range query to obtain the internal RONNs, which are used to initialize the four MODs. It is worth noting that the range query also returns the leaf nodes of O-tree that intersects with the query range. So the result of range query of RONN-OA Algorithm is different from that of range query of RONN-RF. Meanwhile, the obstacles visited in the process are retrieved to build a visibility graph (VG) for later RONN processing. Next, with the leaf nodes of O-tree obtained, RONN-OA continues to traverse the corresponding R-trees to access the data objects to find the external RONNs. During the traversal of R-trees, a heap H is used to store the R-tree entries. Initially, H contains only *the root entries of R-trees associated with the retrieved O-tree leaf nodes*. By de-heaping the entry e with the minimal distance to R, RONN-OA checks whether the minimal Euclidean distances from e to R is greater than all the current MODs. If this is true, e will be eliminated and a new entry is de-heaped. Otherwise, if e is an MBB (i.e., internal node of R-tree), all its child entries are examined for filtering based on Theorem 1 (as RONN-RF does) and inserted back to H for later processing. On the other hand, if e is a data object, RONN-OA examines it as an RONN candidate, and then updates RONN results and MODs as appropriate (corresponding to each edge by finding split points on edges of the range boundary, as discussed earlier for RONN-RF). The above query processing steps (i.e., de-heaping a new entry from H and so on) are repeated until H becomes empty. The pseudo-code of RONN-OA algorithm can be found in the Appendix A.

Recall the example in Figure 9. Given a query range R as specified by the red rectangle, RONN-OA first issues a range query to obtain the internal RONN p_4. At the same time, it will obtain the corresponding leaf nodes R_{11} and R_{14} that intersect with R. So we only need to traverse R_{11} and R_{14} for external RONN query. This filters a lot of data objects to accelerate RONN query processing. Obstacles o_1 and o_2 are also inserted into the visibility graph VG during the process of performing the range query. Accordingly, RONN-OA initializes the four MODs corresponding to each edge of R using p_4. Next, to traverse the R-trees corresponding to R_{11} and R_{14}, we initialize H with N_1 and N_2, sorted in ascending order of their Euclidean distances to R. We proceed to pop the top entry N_1 from H. As N_1 consists of p_3 and p_4, we check and filter p_3 by the maintained MODs. Meanwhile, we find p_4 is already the RONN result so it is not put back to H. Similarly, N_2 is de-heaped from H and then N_3, N_4 are inserted back. Continue with the process, N_3 is de-heaped and data objects p_5 and p_6 are inserted back. Then we pop p_5 and find it to be an RONN. Thus, we update MODs corresponding to each edge of R. Finally, p_6 is examined and included in the result set. At this stage, H becomes empty so the query processing ends. The final RONN result set includes p_4, p_5 and p_6. For the example, it is worth noting that p_7 and p_8, which need to be examined before eventually being excluded from the result set in RONN-RF, are not even considered in the processing due to the strength of O-tree.

THEOREM 2. *The RONN-OA algorithm retrieves exactly the ONN of every point on a given query range, i.e., the algorithm has no false misses and no false hits.*

Please see the proof in Appendix B.

6.2 Complexity Analysis

In this section, we analyze the time complexity of the three algorithms proposed in this paper. Let $|OT|$ be the size of O-tree, $|T|$ be the size of R-tree used in RONN-RF, $|T'|$ be the total size of R-trees maintained in the leaf nodes of O-tree, and $|VG|$ be the number of nodes in visibility graph. Moreover, let $n_{CONNB}, n_{RONN-RF}$ and $n_{RONN-OA}$ denote the number of data objects accessed during the external RONN search in CONNB, RONN-RF, and RONN-OA, respectively. The time complexity of these algorithms can be derived as follows.

Based on [6], the time complexity of the CONN algorithm is $O(n_{CONNB} \times \log|T| \times |VG| \times \log|VG|)$. Thus, we can easily deduce that the time complexity of our CONNB is $O(\log|T|+4\times n_{CONNB}\times\log|T|\times|VG|\times\log|VG|)$=$O(n_{CONNB}\times log|T| \times |VG| \times log|VG|)$.

The time complexity of RONN-RF is $O(\log|T|+n_{RONN-RF}\times \log|T| \times |VG| \times \log|VG|)$= $O(n_{RONN-RF} \times log|T| \times |VG| \times log|VG|)$ where $O(\log|T|)$ is for range query. Notice that $n_{RONN-RF}$ is the number of data objects accessed during external RONN query, which is much smaller than $4 \times n_{CONNB}$ because RONN-RF traverses R-tree only once and filters a lot of unqualified results during traversal of R-trees.

Finally, for RONN-OA, it takes $O(\log|OT|)$ for obtaining the internal RONNs while obtaining the obstacles to construct the visibility graph VG for the query. Additionally, it traverses some R-trees to visit data objects in order to find external RONNs. For the $n_{RONN-OA}$ data objects accessed, RONN-OA takes $O(n_{RONN-OA} \times \log|T'| \times |VG| \times \log|VG|)$. Therefore, the time complexity is $O(\log|OT|+n_{RONN-OA} \times \log|T'| \times |VG| \times \log|VG|)$= $O(n_{RONN-OA} \times log|T'| \times |VG| \times log|VG|)$. From the above analysis, we may observe that the three algorithms have about the same order of complexity. However, the number of data objects accessed during query processing is a key factor contributing to the actual computational cost which is to be evaluated experimentally in Section 7.

7. EXPERIMENTS

We evaluate the performance of the proposed algorithms for the RONN query. All the algorithms were implemented in C++, while the experiments were conducted on an Intel Core 2 Duo 3.40 GHz PC with 4GB RAM.

We conduct experiments on three real datasets.

- ***Greece***. It consists of 22,650 MBRs of rivers (polylines) as obstacles and 5,921 locations of cities and villages (as the data objects).[7]
- ***Tiger Census Blocks***. It contains 556,696 census blocks (polygons) of Iowa, Kansa, Missouri, and Nebraska.[7]
- ***California Roads***. It contains the MBRs of 2,249,727 streets (polylines) of California.[7]

In addition, we construct synthetic datasets using SpatialDataGenerator.[7] We generated a synthetic dataset ***Uniform*** that contains 80,000 data objects and 20,000 line segments and rectangles as obstacles, respectively, following uniform distribution. Also we generate another 80,000 uniformed distributed data objects as the data of ***Tiger Census Blocks*** dataset. Finally, we randomly extract 500,000 streets from ***California Roads*** and generate two sets of 1 million data objects following uniform and zipf distributions to form two large-scale datasets ***UniformCA*** and ***ZipfCA***, respectively. All datasets are normalized to a $10,000 \times 10,000$ space in order to fit the search range.

Table 1: Parameter ranges and defaults values

Parameters	Range
♯ of obstacles ($\|O\|$)	1,000, 2,000, 4,000, **8,000**, 16,000, 500,000
♯ of data objects ($\|P\|$)	5,000, 20,000, 10,000, **40,000**, 80,000, 1,000,000
query size ($qsize$)	**100**, 200, 300, 400, 500

We conduct the performance evaluation in two aspects: (1) evaluating the efficiency of RONN algorithms – we compare the latency of the proposed RONN algorithms under various parameters (summarized in Table 7, numbers in bold are the default settings); and (2) evaluating the efficiency of O-tree – we compare the height, number of leaf nodes, index size and construction time of O-tree with respect to the space division strategies under comparison. Three different kinds of obstacles are indexed: (1) *line segment* obstacles (i.e., polylines in datasets); (2) *rectangle* obstacles (i.e., MBRs in the datasets); and (3) *polygon* obstacles. We randomly generate five groups of RONN queries with different query sizes where each group of queries consists of 50 RONN queries using SpatialDataGenerator. [8] In each experiment, we test one parameter at a time (other parameters are fixed at their default values). The reported experimental results are obtained by averaging the processing time of queries.

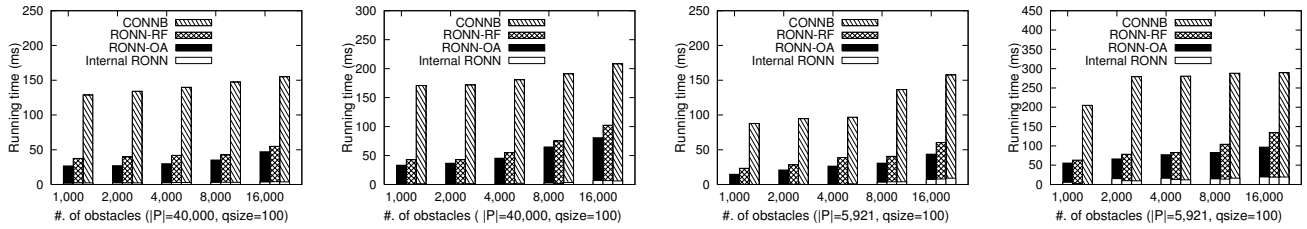
7.1 Efficiency of RONN Algorithms

In this section, we evaluate the efficiency of the proposed RONN algorithms, including CONNB, RONN-RF and RONN-OA. For CONNB and RONN-RF algorithms, data and obstacle sets are indexed with an R*-tree [2], where the page size is fixed at 4KB. For RONN-OA, we construct the O-tree index by adopting the proposed optimal obstacle balance (OOB) scheme for space division. We measure the average latency as the performance metric corresponding to three different parameters: (a) numbers of obstacles $|O|$; (b) numbers of data objects $|P|$; and (c) query size $qsize$. Accordingly, we test the three RONN algorithms using different datasets as described above.

Effect of the number of obstacles. In this experiment, we compare the average running time of three algorithms for processing RONN queries by increasing the number of obstacles. The average running time for finding internal RONNs and external RONNs, are depicted in Fig. 10. Fig. 10(a) shows the result on Uniform Segment dataset. As shown, it takes very little time to find the internal RONNs (too small to be seen), as the main cost in answering RONN queries is spent on external RONN search. RONN-OA and RONN-RF outperform CONNB significantly, with 360% and 300% improvement, respectively. On average, RONN-OA takes 50ms for each query. The excellent performance was achieved by the effective pruning facilitated by O-tree. As a result, RONN-OA searches a small number of R-trees for only once. RONN-RF also performs much better than CONNB because it traverses R-tree only once (instead of four times as CONNB does). The filtering in RONN-RF also helps. As expected, the performance becomes worse when the number of obstacles increases, especially in Greece Segment dataset, but all three algorithms scale well. The results obtained using the other three datasets are shown in Figs. 10(b)-(d). The observed trend and conclusion are consistent with

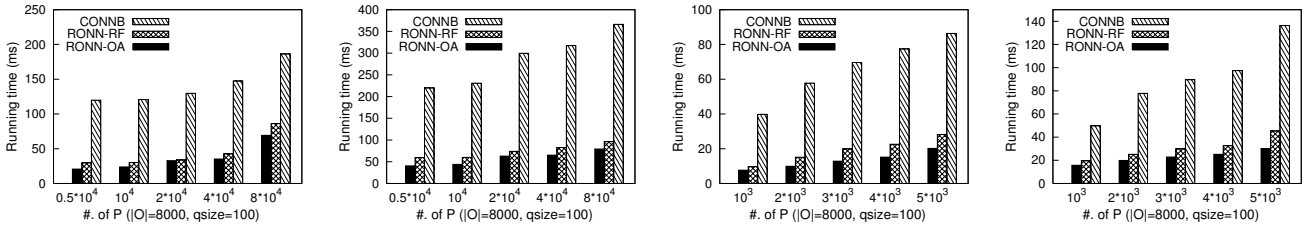
[7]http://www.chorochronos.org.

[8]Query size denotes the size of edges in query rectangle.



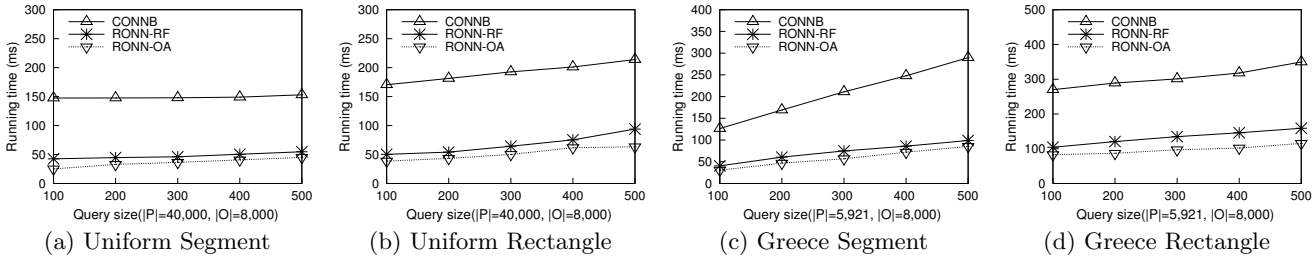
(a) Uniform Segment (b) Uniform Rectangle (c) Greece Segment (d) Greece Rectangle

Figure 10: Performance vs. $|O|$



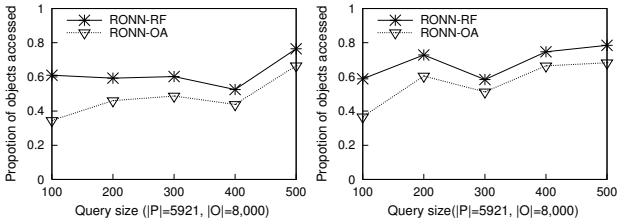
(a) Uniform Segment (b) Uniform Rectangle (c) Greece Segment (d) Greece Rectangle

Figure 11: Performance vs. $|P|$



(a) Uniform Segment (b) Uniform Rectangle (c) Greece Segment (d) Greece Rectangle

Figure 12: Performance vs. query size



(a) Greece Segment (b) Greece Rectangle

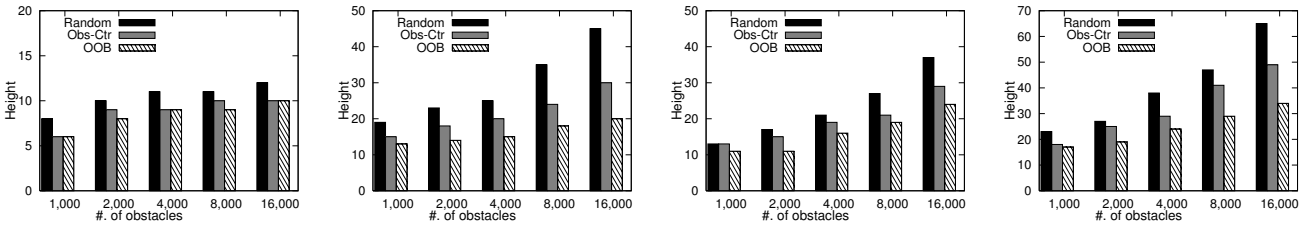
Figure 13: Proportion of data objects accessed

Fig. 10(a). Compare with the obstacles represented as line segments, all the three algorithms spent longer time when obstacles are represented as rectangles. This is because more vertices of obstacles need to be inserted to their corresponding visibility graphs.

Effect of the number of data objects. Fig. 11 compares the performance of RONN algorithms by varying the number of data objects. For the Uniform dataset with segment and rectangle obstacles, respectively, all the three algorithms answer the query within half second. Both RONN-RF and RONN-OA are very efficient, cost less than 0.1 second when $|P|$ was 80,000. RONN-OA outperforms the two R-tree based algorithms significantly under various number of data objects because O-tree index provides good filtering ability. Moreover, the processing time of all three algorithms increases as the number of data objects increases. Also comparing Fig. 11(a) and Fig. 11(b), the three algorithms spent longer time when obstacles were represented as rectangles. The results on Greece dataset show that the three algorithms take less time compared with the Uniform dataset because the number of data objects in Uniform dataset is smaller.

Effect of query size. We then compare the RONN algorithms by varying query sizes and show the result in Fig. 12. For *Uniform* dataset, the results in Fig. 12(a) shows the superiority of RONN-OA over the two R-tree based algorithms. The search time of RONN-OA only takes about 35 ms, but the CONNB algorithm takes more than 150 ms, while RONN-RF takes almost 45 ms, because O-tree effectively divides the obstructed space into non-obstacle space. As a result, points in the non-obstacle spaces can be effectively filtered using R-trees. As Figs. 12(a) and 12(b) show, it takes more time to process rectangle dataset than line segment dataset. Similar results on *Greece* dataset are observed in Figs. 12(c) and 12(d).

Proportion of the number of data objects accessed. As mentioned in the complexity analysis earlier, the number of data objects accessed during query processing is a



(a) Uniform Segment (b) Uniform Rectangle (c) Greece Segment (d) Greece Rectangle

Figure 14: The height of O-Tree vs. ♯ of O

key factor. We compare our RONN-OA and RONN-RF algorithm on proportion of the number of data objects accessed (dividing the number of data objects accessed by CONNB algorithm) by varying query size in Fig. 13. In Fig. 13(a), RONN-OA and RONN-RF, significantly outperforms CONNB, and RONN-OA outperforms RONN-RF because O-tree index can filter some data objects whose Euclidean distance to query range is small but obstructed distance to query range is big while R-tree can not filter such kind of data objects. The similar results are shown in on *Rectangle* dataset in Fig. 13(b).

7.2 Construction of the O-tree Index

As discussed in Section 5, space division is an essential issue in construction of O-tree. To achieve tree balance, we propose the optimal obstacle balance (OOB) scheme. In this section, we evaluate the performance of the OOB scheme, in comparison with the following two baseline schemes:

- *Random.* This scheme randomly chooses an obstacle for space division in construction of O-tree.
- *Obs-Ctr.* Obs-Ctr is to simply use the perpendicular bisector of selected obstacle as the perpendicular line for space division. It chooses the obstacle with the best score to divide the current space.

To compare these space division schemes, we use the heights and leaf nodes of the corresponding O-trees as performance metrics, as they reflect how balanced these O-trees are, i.e., the smaller the height and the number of leaf nodes are, the better the tree balance is. Additionally, we compare the size of the resulted index and their construction time.

Fig. 14 depicts the tree height of the space division schemes by varying the numbers of obstacles in four datasets. We can see that the OOB scheme outperforms the other two schemes in all circumstances using different datasets with different sizes of obstacle sets. The result also shows that Obs-Ctr is better than Random. When increasing the number of obstacles, the tree height for all schemes increases slowly in the experiment on uniform obstacle sets. However, for skewed datasets, i.e., *Greece* dataset, the impact of the number of obstacles is obviously more significant, especially on the Random and Obs-Ctr schemes. OOB performs the best because it achieves better obstacle balance in each space division step. In addition, the Obs-Ctr scheme performs better than the Random scheme in general, as they do achieve certain degree of obstacle balance. Moreover, by comparing Fig. 14(a) and Fig. 14(b), we observe that the tree height resulted from various schemes are higher on Rectangle dataset than on the Segment dataset, because these schemes deal with more obstacles on Rectangle dataset.

To evaluate those schemes more comprehensively, we further compare the number of leaf nodes in their resulted O-trees. With the growing numbers of obstacles, the number of leaf nodes in the tree is larger. And the number of leaf nodes using Random scheme increases faster than that using OOB or Obs-Ctr, showing OOB schemes achieving better tree balance. The corresponding experiment results and additional experimental results about index size and construction time can be found in Appendix D.

In summary, the height and the number of leaf nodes show that the proposed OOB scheme achieves the best tree balance among these three schemes.

8. CONCLUSION

In this paper, we formulate a new query, namely, *range-based obstructed nearest neighbor* (RONN) query, for finding nearest neighbors to a given spatial range in the obstructed space. We carry out a systematic study on the RONN query. First, we propose the CONNB algorithm as a baseline for the tackled RONN problem. To address the shortcomings of the CONNB algorithm, we further propose a new RONN-RF algorithm, that explores heuristics and filtering strategies via seamless integration of range and CONN queries to facilitate efficient query processing. Next, we propose a new index, called *O-tree*, dedicated for indexing objects in the obstructed space. The novelty of O-tree lies in the idea of *dividing the obstructed space into non-obstacle space*, aiming to exploit efficient query processing in non-obstacle subspaces. We propose a space division scheme OOB to address the tree balance problem. Moreover, we propose an efficient algorithm, called RONN-OA, which exploits O-tree to accelerate query processing of RONN. At last, a comprehensive performance evaluation is conducted to validate the proposed ideas and demonstrate the efficiency and effectiveness of the proposed index and algorithms.

This work may lead towards several new directions for future work, e.g., different shapes of query range, mobile obstacles, as well as parallel processing techniques.

9. ACKNOWLEDGMENTS

This work is partially supported by the NSF of China for Outstanding Young Scholars under grant No. 61322208, the National Basic Research Program of China (973 Program) under grant No. 2012CB316201, the NSF of China for Key Program under grant No. 61532021, and the NSF of China under grant Nos. 61272178 and 61572122. We are grateful to the anonymous reviewers for their constructive comments on this paper. Xiaochun Yang is the corresponding author of this work.

10. REFERENCES

[1] J. Bao, C.-Y. Chow, M. F. Mokbel, and W.-S. Ku. Efficient evaluation of k-range nearest neighbor queries in road networks. *MDM*, pages 115–124, 2010.

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

[3] N. Chan and T. Dang. On efficient processing of complicated cloaked region for location privacy aware nearest-neighbor queries. *International Conference on Information Communication Technology*, pages 101–110, 2013.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O.Schwarzkopf. *Computational Geometry: Algorithms and Applications, Second Edition.* Springer-Verlag, Reading, Massachusetts, 2000.

[5] P. Galdames and Y. Cai. Efficient processing of location-cloaked queries. *IEEE INFOCOM*, 131(5):2480–2488, 2012.

[6] Y. Gao and B. Zheng. Continuous obstructed nearest neighbor queries in spatial databases. *SIGMOD*, pages 577–558, 2009.

[7] Y. Gao, B. Zheng, W.-C. Lee, and G. Chen. Continuous visible nearest neighbor queries. *EDBT 09*, pages 144–155, 2009.

[8] B. Gedik and L. Liu. A customizable k-anonymity model for protecting location privacy. *Distributed Computing Systems (ICDCS)*, pages 763–774, 2005.

[9] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. *ACM MobiSys*, pages 763–774, 2003.

[10] H. Hu and D. L. Lee. Range nearest-neighbor query. *TKDE*, 18(1):78–91, 2006.

[11] C. H. J and J. R. Efficient processing of moving k-range nearest neighbor queries in directed and dynamic spatial networks. *Mobile Information Systems*, 2016.

[12] P. Kalnis, G. Ghinita, K. Mouratidis, and D. Papadias. Preventing location based identity inference in anonymous spatial queries. *TKDE*, 26(3):1719–1733, 2007.

[13] A. Khoshgozaran, C. Shahabi, and H. Shirani-Mehr. Enabling location privacy; moving beyond k-anonymity, cloaking and anonymizers. *Knowledge Information Systems*, 26(3):435–465, 2011.

[14] H.-I. Kim and J.-W. Chang. k-nearest neighbor query processing algorithms for a query region in road networks. *Journal of Computer Science and Technology*, 28(4):585–596, 2013.

[15] E. Lee, H. Cho, T. Chung, and K. Ryu. Moving range k nearest neighbor queries with quality guarantee over uncertain moving objects. *Information Sciences*, 325:324–341, 2015.

[16] X. Lin, J. Xu, and H. Hu. Range-based skyline queries in mobile environments. *TKDE*, 25(4):835–849, 2013.

[17] M. F. Mokbel, C. Y. Chow, and W. G. Aref. The New Casper: Query processing for location services without compromising privacy. *VLDB*, pages 763–774, 2006.

[18] K. Mouratidis, M. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. *VLDB*, pages 43–54, 2006.

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

[20] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. *VLDB*, pages 287–298, 2002.

[21] A. K. Tung, J. Hou, and J. Han. Spatial clustering in the presence of obstacles. *ICDE*, pages 322–331, 1990.

[22] X. Wang and H. J. Hamilton. Clustering spatial data in the presence of obstacles. *International Journal on Artificial Intelligence Tools*, 14(1-2):177–198, 2005.

[23] C. Xia, D. Hsu, and H. Tung. A fast filter for obstructed nearest neighbor queries. *BNCOD*, pages 203–215, November 2004.

[24] C. Zhang and Y. Huang. Cloaking locations for anonymous location based services: a hybrid approach. *Geoinformatica*, pages 159–182, 2009.

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

[26] B. Zheng, W.-C. Lee, and D. L. Lee. On searching continuous k nearest neighbors in wireless data broadcast systems. *IEEE Transactions on Mobile Computing*, 6(7):748–761, 2007.

[27] H. Zhu, J. Wang, B. Wang, and X. Yang. Location privacy preserving obstructed nearest neighbor queries. *Journal of Computer Research and Development*, 51(1):115–125, 2014.

APPENDIX

A. PSEUDO-CODE OF RONN-OA

Algorithm 4 depicts the pseudo-code of the RONN-OA Algorithm. It first issues an internal RONN query (see Algorithm 5) (Line 1). The corresponding leaf nodes intersect with the query range are returned. Notice that this is done simultaneously while traversing O-tree to perform the range query. With support of O-tree, CONN-OA avoids traversing O-tree from the root again (unlike what RONN-RF does to process the external RONN query). Lines 7-27 are very similar to the RONN-RF Algorithm. The main difference is that we initiate the min-heap with the root entries of R-trees associated with the found leaf nodes of O-tree in Line 1.

The range query used in RONN-OA is depicted in Algorithm 5. We recursively traverse O-tree to the leaf nodes. For each node visited, we first check whether its region intersects with the query range R (Line 1). If yes, we continue to check whether the node is a leaf node. There are two conditions: (i) if the node is a leaf node, we will traverse its associated R-tree to find data objects located in the query range and add them into the result set *Res*. Meanwhile, the leaf node itself is added to a leaf node list, namely, *LNodeList* (Lines 2-4). (ii) if the node is not a leaf node, we will recursively search each of its child node (Lines 5-7).

B. PROOF OF THEOREM 2

PROOF. We prove Theorem 2 by showing that the RONN-OA algorithm returns result with no false misses and no false hits.

(1) The result returned by the RONN-OA algorithm has no false misses. According to the RONN-OA algorithm, we know that an object p could be an answer to RONN only if (i) it is inside a region overlapped with the query range, or

Algorithm 4: RONN-OA Algorithm

Input: A O-tree OT, query range R, and min-heap H
Output: RONN results list RS

RQ(R, OT, $InternalRes$, $LNodeList$);
for $k = 1$ to 4 **do**
  $RS_k \leftarrow InternalRes$; $over_k \leftarrow 0$;
$RS \leftarrow InternalRes$;
for Each node $n \in LNodeList$ **do**
  insert($n.rptr.root$, $mindist(n, R)$) into H;
while *true* **do**
  $k \leftarrow 1$;
  for Each edge $\overline{E}_{ij}$ of R **do**
    $RS_k \leftarrow$ FindSP($\overline{E}_{ij}, RS_k$);
    $MOD_k \leftarrow max\_odis(RS_k)$;
    if $H.head.key > MOD_k$ **then**
      $over_k = 1$.
    if $over_1 \& over_2 \& over_3 \& over_4$ *or* $H \doteq \varnothing$ **then**
      return;
    $k++$;
  while $H \neq \varnothing$ **do**
    de-heap the top entry (e, key) of H;
    if e is a data object **then**
      check whether e is in result RS; /* using $isInResult$ array*/
      $k \leftarrow 1$;
      for $key < MOD_k$ **do**
        insert e into RS_k; $k++$;
      break;
    else
      for each child entry $e_i \in e$ **do**
        if $mindist(e_i, R) < MAX(MOD_k)$ **then**
          insert$(e_i, mindist(e_i, R))$ into H.
$RS \leftarrow RS \cup RS_k$;
return RS;

Algorithm 5: O-tree::Range Query Algorithm (RQ)

Input: A query region R, a O-tree Node $Node$, and a Node list $LNodeList$.
Output: A set of data objects and $LNodeList$

if isIntersect($Node.reg, R$) **then**
  if $Node$ is leaf node **then**
    $Res \leftarrow$ rangequery(R,$Node.rptr$);
    insert $Node$ into $LNodeList$;
  else
    for Each $optr$ of $Node$ **do**
      RQ(R, $optr$, Res,$LNodeList$);
return Res and $LNodeList$;

(ii) it belongs the ONNs to the boundary line segments of a overlapped regions. Assume data object p does not satisfy the above two conditions but p is an answer to RONN. Then p is the nearest neighbor to any point inside the query range. Let q is a such point inside the query range. Moreover, q must exists inside a certain region R_i that are overlapped with the query range. From Lemma 1, we know that the ONN to q must be inside R_i or belong to the ONNs to the boundary of R_i. This contradicts with the assumption. Therefore, the returned results by RONN-OA has no false misses.

(2) The RONN-OA algorithm inserts all the data objects that satisfy the above two conditions into the heap for checking if it is an RONN answer. Hence, the algorithm has no false hits.

From (1) and (2) above, the proof completes. □

C. EXTENSION ON O-TREE

In this section, we extend the O-tree index for *polygons*. Fig. 15 shows an example of indexing a pentagon o_1. For space division, we can take each edge of the polygon as a separate obstacle line to divide the space. Then we can use the division schema for obstacles represented as line segments to divide the space. Firstly, we use the edge $\overline{ab}$ to divide the region into four subregions and the root has four child nodes (R_1, R_2, R_3, R_4). As R_1 can be divided into four subregions (R_{11}, R_{12}, R_{13}, R_{14}) utilizing the edge $\overline{ae}$, R_4 can be divided into four subregions using $\overline{cd}$, and the other two subregions can directly form the leaf node. At last, R_{11} and R_{42} can be divided into four subregions by $\overline{de}$ and $\overline{bc}$, respectively. An example of space division is shown on the top side of the figure, while the corresponding O-tree is depicted on the bottom side. Internal node R_1 stores obstacle $\overline{ae}$, and $optr$ pointing to three O-tree children R_{11}, R_{12}, and R_{13}. Notice that O-tree does not store R_{14} since R_{14} locates inside the pentagon o_1. Internal node R_{11} stores obstacle $\overline{de}$, and $optr$ pointing to two O-tree children R_{111} and R_{114}. Leaf node R_{12} stores $rptr$ pointing to an R-tree which contains data objects p_3 and p_4. The other nodes is similar to Internal node R_{11} and leaf node R_{12}. Notice that when we use $\overline{ab}$ to divide the whole space, we choose the perpendicular bisector through the endpoint of $\overline{de}$ and $\overline{cd}$ to avoid splitting $\overline{de}$ and $\overline{cd}$ into more obstacles.

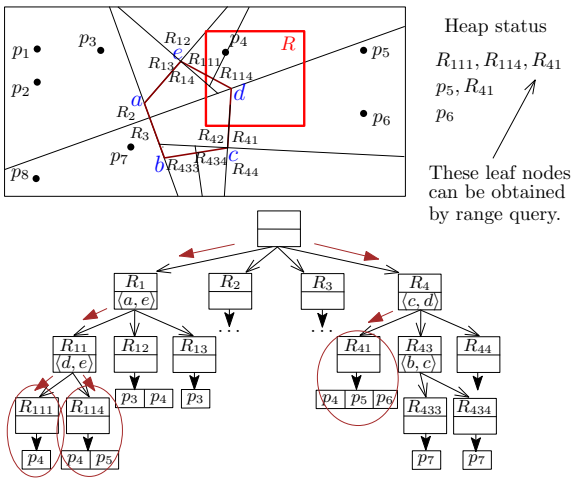


Figure 15: Index about polygon

Next, we introduce the RONN processing for polygon-shaped obstacles based on O-tree. Firstly, we issue a range query to obtain the interval RONN p_4. At the same time, it obtains the corresponding leaf nodes R_{111}, R_{114} and R_{44} that intersect with R. Then we follow the similar steps of RONN-OA shown in the heap status part of Fig. 15 until H

becomes empty so the query processing ends. Actually, the other data objects are filtered by internal RONN p_4 except p_5 and p_6 while searching O-Tree. Finally, the RONN result set includes p_4, p_5 and p_6.

While computing the ONNs with polygon obstacles, we observe a problem. As shown in Fig. 16, there is a rectangle obstacle o, and data objects p_1 and p_2. In our O-tree index of this rectangle obstacle, we take each edge of the rectangle as a separate line segment obstacle to perform space division and build O-tree index. However, in query processing, when we compute the obstructed distance of p_2 to p_1, we choose the distance of shortest path of $\langle p_2, b, d, p1\rangle$ instead of actual shortest path $\langle p_2, a, p1\rangle$. Actually, we can not go across from b to d. This is the *wrong crossing* problem.

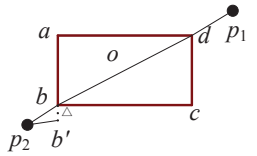


Figure 16: Wrong crossing

To avoid wrong crossing, we adopt an idea of Δ *shifting* to address the problem. As shown in Fig. 16, when going through p_2 to b (ending points of obstacle), we assume it goes through p_2 to b' which is obtained by shifting b with Δ distance (Note that Δ is relatively small here).

D. EXTRA EXPERIMENTAL RESULTS

To evaluate the experiments comprehensively, we list some extra experimental results.

D.1 Number of Leaf Nodes of O-tree

To evaluate the three space division schemes for building an O-tree more comprehensively, we further compare the number of leaf nodes in O-tree. As depicted in Fig.17(a), when increasing the numbers of obstacles, the number of leaf nodes in the tree become larger. In addition, the number of leaf nodes using Random scheme increases faster than that using OOB and Obs-Ctr. OOB scheme achieves the best tree balance. As expected, more division needed to be considered in *Uniform Rectangle* dataset, so the number of leaf nodes is much larger than that on *Uniform Segment* dataset. Figs.17(c) and 17(d) show the similar results on Greece datasets.

D.2 Index Size and Construction Time

Then, we show the index size and construction time of O-tree in Figs. 18(a)-(b) with respect to different numbers of obstacles on Uniform Segment dataset. For index size in Fig.18(a), O-tree generated using the OOB scheme only takes less than 10 MB, much smaller than the other two schemes. This result is consistent with the results on the height and the number of leaf nodes of O-trees. Additionally, the Random scheme results in largely varied sizes of O-trees. The construction times of index resulted from all space division schemes are shown in Fig. 18(b). Random is faster than the other two schemes because it takes no time to decide how to divide the space. However, this results an unbalanced tree which does not support RONN processing well. The other schemes take under 12 minutes to finish the index construction. In the last index comparison part of experiments, we compare O-tree performance of space division schemes when we assume each obstacle is an rectangle on Uniform dataset, and the experimental results are shown in Figs. 18(c)-(d) . O-tree index with line segment obstacles is more efficient for space division and building index in most cases. This is because if we take each rectangle obstacle as four line segment obstacles, many obstacles will be divided into more parts, and thus more difficult to deal with than the line segment obstacles.

D.3 Overlap of data objects stored in leaf nodes

We further evaluate the number of overlapped data objects among leaf nodes in O-tree. We compare the three space division schemes to see their effects on the number of overlapped data objects. Fig. 19 shows the comparison results. When increasing the number of obstacles, the number of overlapped objects increases. When the number of obstacles is less than 4,000, using OOB division scheme only produces less than 0.3 overlap. Random division scheme generates more overlap since it does not consider the density of the obstacles.

D.4 Comparison with Cloaking Query

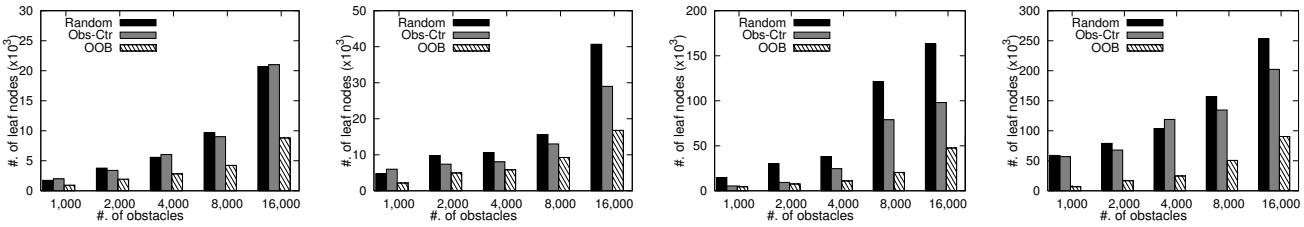
Cloaking query in obstructed space [27] is the most related work to ours. Cloaking query is mainly proposed for protecting the query point so that the untrusted server cannot infer the correct query point among queries in the cloaked region. Due to the lack of the exact location information at the server, the anonymous query processor calculates a safe region and provides all data objects inside the safe region (i.e. candidates) instead of a single exact answer [17].

Different from cloaking query, an RONN query returns nearest neighbors to every query in the query region. A good point is that RONN can also protect the real query point and guarantees its returned result covering the exact answer to the protected query point in the query range. RONNA returns less number of points than the candidates returned by the cloaked query, therefore saving the communication cost as well as the post-processing time at client.

Fig. 18 shows the comparison between RONN-OA and cloaking query in obstructed space [27]. We use the number of returned points to represent communication cost. Fig. 20(a) shows that RONN incurs much less communication cost than cloaking query does. Fig. 20(b) shows that cloaking query runs a little faster than RONN-OA since it directly returns those candidates within the safe region without any examination. Notice that, more candidates returned from server require more cost for post-processing at client since the final answer should be refined and returned to the user. Fig. 20(c) shows the processing time at client.

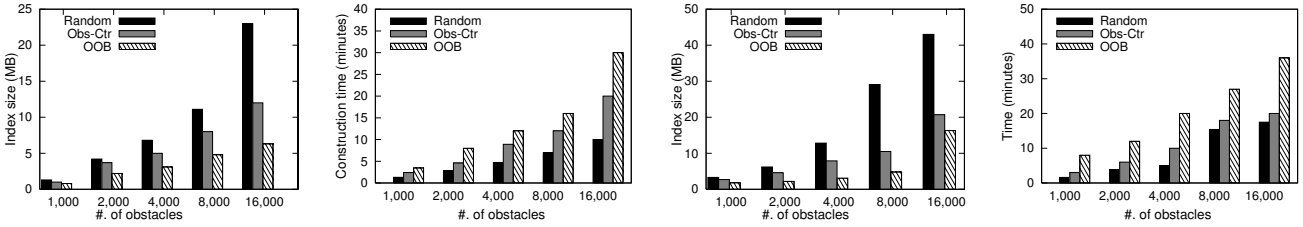
D.5 Results on Polygon Obstacles

We also test the running time of our algorithms on *Tiger Census Blocks* datasets with polygon obstacles. They contain uniform and zipf distributed data objects datasets, respectively. Fig. 21 shows that RONN-OA and RONN-RF outperform CONNB significantly. The running time on uniform distributed dataset is faster than that on zipf distributed dataset, because the skewed data distribution makes the maximum obstructed distances (MODs) (i.e. filtering distance) longer than uniform data. Moreover, more data objects need to be examined.



(a) Uniform Segment (b) Uniform Rectangle (c) Greece Segment (d) Greece Rectangle

Figure 17: The number of leaf nodes



(a) Uniform Segment (b) Uniform Segment (c) Uniform Rectangle (d) Uniform Rectangle

Figure 18: Index size and construction time on Uniform Segment obstacles

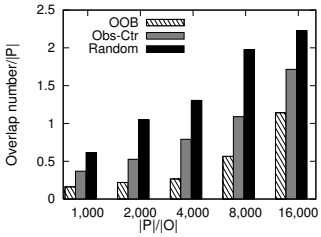
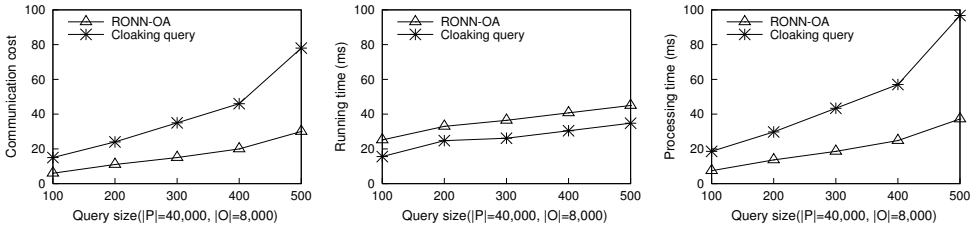
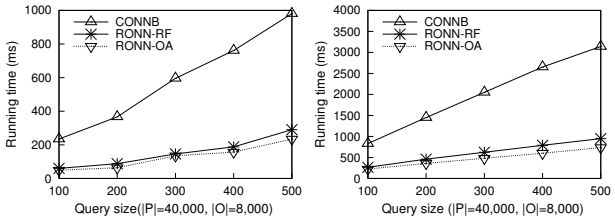


Figure 19: Data overlap



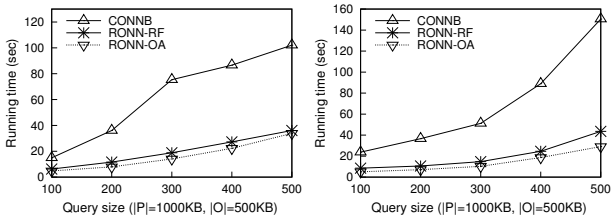
(a) Communication cost (b) Running time at server (c) Running time at client

Figure 20: Comparison with Cloaking query



(a) Uniform Tiger (b) Zipf Tiger

Figure 21: Performance on polygon obstacles



(a) (UniformCA) dataset (b) (ZipfCA) dataset

Figure 22: Performance on large datasets

D.6 Results on Large Datasets

We also evaluate the three algorithms on large datasets *UniformCA* and *ZipfCA* and show the experimental results in Fig. 22. The running time for the three algorithms were within 2 minutes on *UniformCA* dataset, faster than the time on *ZipfCA*.

Our RONN-OA and RONN-RF outperform the baseline algorithm CONNB with 405% and 300% improvement, respectively, which is consistent to the result on the other tested datasets. For example, when query size was 300, the running time was 75.3 seconds, 18.9 seconds, and 14 seconds for CONNA, RONN-RF, and RONN-OA, respectively. We can see that RONN-OA improves RONN-RF by 35% on the large dataset *UniformCA*.