Monochromatic and Bichromatic Reverse Nearest Neighbor Queries on Land Surfaces

Da Yan, Zhou Zhao and Wilfred Ng
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
{yanda, zhaozhou, wilfred}@cse.ust.hk

ABSTRACT

Finding *reverse nearest neighbors* (RNNs) is an important operation in spatial databases. The problem of evaluating RNN queries has already received considerable attention due to its importance in many real-world applications, such as resource allocation and disaster response. While RNN query processing has been extensively studied in Euclidean space, no work ever studies this problem on land surfaces. However, practical applications of RNN queries involve terrain surfaces that constrain object movements, which rendering the existing algorithms inapplicable.

In this paper, we investigate the evaluation of two types of RNN queries on land surfaces: *monochromatic RNN* (MRNN) queries and *bichromatic RNN* (BRNN) queries. On a land surface, the distance between two points is calculated as the length of the shortest path along the surface. However, the computational cost of the state-of-the-art shortest path algorithm on a land surface is quadratic to the size of the surface model, which is usually quite huge. As a result, surface RNN query processing is a challenging problem.

Leveraging some newly-discovered properties of Voronoi cell approximation structures, we make use of standard index structures such as an R-tree to design efficient algorithms that accelerate the evaluation of MRNN and BRNN queries on land surfaces. Our proposed algorithms are able to localize query evaluation by accessing just a small fraction of the surface data near the query point, which helps avoid shortest path evaluation on a large surface. Extensive experiments are conducted on large real-world datasets to demonstrate the efficiency of our algorithms.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

Keywords

Reverse nearest neighbor, land surface, terrain

1. INTRODUCTION

Recent technological advances in remote sensing have made available high resolution terrain data of the entire Earth surface. As a result, many online Earth visualization platforms emerge, such as "Google Earth™" and "Bing Maps for Enterprise". Other examples involving geo-realistic rendering of surfaces include computer games such as "Counter-Strike" and "Call of Duty". However, these applications focus mainly on the rendering of land surfaces rather than the processing of spatial queries on them.

Due to the ubiquity of high resolution terrain data, it is becoming more and more important to develop geographic information systems (GISs) that support efficient query processing on land surfaces. Such systems would enable many novel and useful applications in the terrain context.

The shortest path problem has been studied on land surfaces by many works such as [5, 6, 7, 8], and the best-known exact algorithm is *Chen and Han's algorithm*, which takes $O(n^2)$ time on a surface model of size n. Recently, several works [1, 2, 3, 4] begin to study k-nearest neighbor (k-NN) queries on land surfaces, which is termed *surface k-NN* (SkNN) queries.

We call a *reverse nearest neighbor* (RNN) query on land surfaces as a *surface RNN* (SRNN) query, which, to our knowledge, has not been studied before. There are two popular types of RNN queries in the literature:

- Given a data point set O and a query point q, a *monochromatic reverse nearest neighbor* (MRNN) query finds all the data points $o \in O$ that have q as their nearest neighbor (NN).
- Given a site point set S, a data point set O and a query point $q \in S$, a *bichromatic reverse nearest neighbor* (BRNN) query finds all the data points $o \in O$ that are closer to q than any other point in S.

SRNN queries have many real world applications. For example, *bichromatic* SRNN (BSRNN) queries are important in the domain of disaster response: when a disaster (e.g. earthquake/tsunami) happens, the transportation system may stop functioning, and thus rescue teams have to find ways within the disaster area, instead of using the blocked/damaged roads, to save the lives of the suffered. In this case, a victim should be reached by the rescue team nearest to him/her, and therefore, it is important for the rescue teams to keep track of their RNNs among the victims. Other applications of BSRNN queries include supply distribution during military operations, and wild animal rescue in nature reserves.

Besides BSRNN queries, *monochromatic* SRNN (MSRNN) queries are also useful in many application domains, such as outdoor activities and military operations. Consider the activity of mountaineering: in order to keep the mountaineers safe, it is important for each member to keep track of his/her RNNs among all the other mountaineers, so that if a member encounters an accident such as a landslide, he/she can get help from his/her nearest neighbor who tracks him/her (e.g. through GPS devices). Similar situations happen in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'12, October 29–November 2, 2012, Maui, HI, USA.
Copyright 2012 ACM 978-1-4503-1156-4/12/10 ...$15.00.

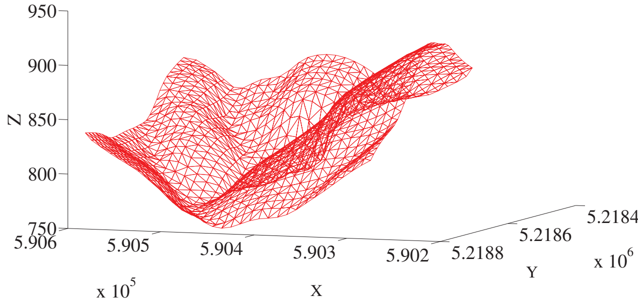


Figure 1: Triangular Irregular Network (TIN)

military operations, where a troop would reinforce one of its reverse nearest neighbor troop if that troop suffers severe casualty.

In this paper, we study RNN queries on land surfaces, including both MSRNN and BSRNN queries. Unlike traditional RNN queries, the evaluation of SRNN queries is more challenging due to the following two reasons:

- **The size of the surface model data is huge.** Unlike query processing in the Euclidean space, queries on land surfaces also involve the environmental data (i.e. the surface model) besides the object data. A land surface is usually represented by the *Triangulated Irregular Network* (TIN) model [1, 4], a mesh derived from the *Digital Elevation Model* (DEM) of sampled ground positions at regularly spaced interval (cf. Figure 1). The surface data are usually collected with high resolution, e.g. terrain data with 10m sampling interval can be easily accessed from [25]. As a result, a typical land surface data contains at least millions of TIN vertices.

- **Finding shortest surface path is computationally expensive.** The best-known algorithm for exact shortest surface path computation is *Chen and Han's algorithm*, which takes $O(n^2)$ time on a terrain data with n triangular faces. It is reported in [3] that, for some moderately large areas (a few square kilometers), *Chen and Han's algorithm* may take tens of minutes on a modern PC machine to compute a shortest surface path. The poor scalability of *Chen and Han's algorithm* poses a challenge for SRNN query processing, since the definition of RNNs on land surfaces is based on the shortest surface path distances.

As a result, efficient SRNN query evaluation requires that (1) the shortest surface path computation should be avoided whenever possible, by efficient pruning methods; and that (2) only a small fraction of the surface data near the query point should be accessed, so that shortest surface path computations are done on a small surface.

To achieve these two goals, we adopt the Voronoi cell [24] approximation structures (i.e. tight/loose cells) to accelerate SRNN query processing. The concepts of tight/loose cells are first proposed in [1] for processing SkNN queries. The algorithms of [1] achieve significant pruning power when answering SkNN queries, by using several properties of the relationship between a cell and a query points, which we call the *cell-point properties*.

However, the *cell-point properties* along are not sufficient for answering SRNN queries. We discover several (non-trivial) properties of the relationship between two cells, termed the *cell-cell properties*, which is essential in SRNN query processing. Furthermore, our MSRNN algorithm requires the loose cell of the query point, but [1] only gives an offline algorithm that computes the loose cells of all data points together. Therefore, we also study how to efficiently compute the loose cell of the query point online.

While [1] designs a cell index dedicated to SkNN query processing, we find that its algorithm may not return the exact kNNs, due to a flawed cell-cell property it assumes. Therefore, we do not follow that indexing methods, but rather use standard index structures such as R-tree in our algorithms. Extensive experiments on large real-world datasets have demonstrated the efficiency of our algorithms for MSRNN and BSRNN queries.

The rest of the paper is organized as follows. We formally define MSRNN and BSRNN queries in Section 2. In Section 3, we review the concept of tight/loose cells, as well as the *cell-point properties*. Our cell indexing method is explained in Section 4, where we also present the algorithm of surface NN query processing using the index. In Section 5, we prove two new *cell-cell properties* and present our algorithm for NN queries among the data objects (rather than the query point), which is a fundamental operation in our MSRNN algorithm. Our online algorithm for constructing tight/loose cells is introduced in Section 6, and our algorithms for MSRNN and BSRNN queries are described in Section 7. Extensive experiments are conducted in Section 8 on large real-world datasets to verify the efficiency of our algorithms. Finally, we review the related work in Section 9, and conclude the paper in Section 10.

2. MSRNN & BSRNN QUERIES

A land surface can be regarded as a continuous function that assigns every point (x, y) on a horizontal plane to a unique z-coordinate value. The TIN model is the most popular model to represent a land surface, which is constructed from the sampled ground positions at regularly spaced intervals, by Delaunay triangulation [24], to form a set of non-overlapping triangles whose vertices are these sampled points. We call the graph defined by the sampled vertices and the edges of the triangles as the *model network*. Figure 1 shows a piece of land surface represented by the TIN model, where the red lines correspond to the edges of the *model network*.

Before presenting the formal definition of SRNN queries, let us first define some distance metrics between two points on the TIN model, as well as the corresponding notations.

DEFINITION 1. *Let p_1 and p_2 be two points on a land surface.*

- *The **surface distance** between p_1 and p_2, denoted $d_S(p_1, p_2)$, is the length of the shortest path connecting p_1 and p_2 along the surface.*
- *The **Euclidean distance** between p_1 and p_2, denoted $d_E(p_1, p_2)$, is the length of the straight line connecting p_1 and p_2.*
- *The **network distance** between **vertices** p_1 and p_2, denoted $d_N(p_1, p_2)$, is the length of the shortest path connecting p_1 and p_2 along the model network.*

It is obvious that for any two given vertices p_1 and p_2 on a land surface,

$$d_E(p_1, p_2) \le d_S(p_1, p_2) \le d_N(p_1, p_2), \quad (1)$$

where the second inequality is because the shortest network path is also a path along the surface. Therefore, the *Euclidean distance* and the *network distance* are the lower bound and the upper bound of the *surface distance*, respectively.

Now, we are ready to formally define SRNN queries:

DEFINITION 2 (MSRNN). *Given a data point set O and a query point q on a land surface, a **monochromatic surface reverse nearest neighbor** (MSRNN) query finds all the data points $o \in O$ such that $\forall o' \in O - \{o\}, d_S(o, q) \le d_S(o, o')$.*

DEFINITION 3 (BSRNN). *Given a site point set S, a data point set O and a query point $q \in S$ on a land surface, a* ***bichromatic surface reverse nearest neighbor*** *(BSRNN) query finds all the data points $o \in O$ such that $\forall s \in S - \{q\}, d_S(o,q) \le d_S(o,s)$.*

3. TIGHT CELL & LOOSE CELL

Voronoi diagram [24] is a powerful tool for processing NN queries. Given a set O of data objects, a Voronoi diagram divides the space into disjoint cells, where each cell belongs to one object. If a query point q falls into the Voronoi cell of object $o \in O$, then o is guaranteed to be the NN of q among all objects in O.

Many algorithms have been proposed for computing Voronoi diagrams in 2D Euclidean space, among which the most commonly used one is Fortune's plain-sweep algorithm [24], whose time complexity is $O(n \log n)$ with $n = |O|$.

However, as pointed out by [1], it is very expensive to construct the exact Voronoi diagram on a land surface. Therefore, [1] proposes two approximation structures of Voronoi cells that can be computed without the necessity of shortest surface path evaluation:

DEFINITION 4 (TIGHT CELL). *Given a data point set O on a land surface S, the tight cell of a data object $o \in O$, denoted as $TC(o)$, is a polygon area around o, defined by $TC(o) = \{q \in S \mid d_N(o,q) \le d_E(o',q),\ \forall o' \in O - \{o\}\}$.*

DEFINITION 5 (LOOSE CELL). *Given a data point set O on a land surface S, the loose cell of a data object $o \in O$, denoted as $LC(o)$, is a polygon area around o, defined by $LC(o) = \{q \in S \mid d_E(o,q) \le d_N(o',q),\ \forall o' \in O - \{o\}\}$.*

Note that we can always assume an object $o \in O$ to be on a vertex, since otherwise, we can split the face that the object is on as three faces by connecting the three face vertices to o.

We can rewrite the definitions of tight and loose cells in the following equivalent forms:

$$TC(o_i) = \{q \in S \mid d_N(o_i,q) \le d_E(o_k,q),\ o_k = \arg\min_{o_j \in O - \{o_i\}} d_E(o_j,q)\} \quad (2)$$

$$LC(o_i) = \{q \in S \mid d_E(o_i,q) \le d_N(o_k,q),\ o_k = \arg\min_{o_j \in O - \{o_i\}} d_N(o_j,q)\}. \quad (3)$$

Figure 2 shows the tight and loose cells of 6 objects (o_1 to o_6) on a surface, where the blue cells are the tight cells of the objects in them, and the green (purple) cell is the loose cell of o_1 (o_6).

Next, we review the cell-point properties proposed in [1], which are necessary in both the SkNN and our SRNN query processing.

Summary of cell-point properties. Let us use $NN_S(q \mid O)$ to denote the surface nearest neighbor of q among the data objects in O. Note that if $q \in O$, then $q = NN_S(q \mid O)$. Theorem 1 summarizes the cell-point properties that correspond to Properties 1 to 4 in [1], the proof of which can be found therein.

THEOREM 1. *Given a set of objects $O = \{o_1, o_2, \ldots, o_n\}$ on a land surface, the following statements hold:*

1. *If a query point $q \notin O$ is within $TC(o_i)$, then $o_i = NN_S(q \mid O)$.*
2. *If a query point $q \notin O$ is outside $LC(o_i)$, then o_i is guaranteed not to be $NN_S(q \mid O)$.*
3. *All the edges of the tight cells are also the edges of the loose cells.*
4. *If $o_i = NN_S(q \mid O)$, then the shortest surface path from q to o_i is within $LC(o_i)$.*

4. CELL INDEXING

In this section, we first point out a problem with the existing cell index for SkNN query processing [1], which explains why we do not follow that framework for indexing. Instead, we propose to index the loose cells of all data objects by an R-tree, which is used to answer surface NN and SRNN queries. Finally, we present an algorithm of surface NN query processing using the index.

Problem with the existing SkNN algorithm. Based on Statement 3 in Theorem 1, [1] defines the neighbors of an object $o \in O$ as $NL(o) = \{o' \in O - \{o\} \mid TC(o')$ and $LC(o)$ have common edges$\}$. For example, in Figure 2, $NL(o_1) = \{o_2, o_3, o_4, o_5\}$.

For each data object $o \in O$, its neighbor list $NL(o)$ is stored with o in the cell index of [1], and used to find the k NNs incrementally on the surface. However, the underlying property of that incremental algorithm (Property 6 in [1]) is based on the following statement:

If o_i and o_j are not neighbors, then $LC(o_i) \cap LC(o_j) = \emptyset$, and thus a point m on the shortest surface path between o_j and a point q within $LC(o_i)$ should exist outside both $LC(o_i)$ and $LC(o_j)$.

Nevertheless, this claim is not proved in [1], and in fact, it is not correct. One counterexample is given by Figure 2: although $o_6 \notin NL(o_1)$, we have $LC(o_6) \cap LC(o_1) \neq \emptyset$; moreover, the shortest surface path between o_6 and o_1 is totally within $LC(o_6) \cup LC(o_1)$.

Therefore, the incremental method is not guaranteed to be correct even when k goes from 1 to 2. This result is not surprising since Voronoi cells are dedicated to 1-NN queries.

In order to study how frequently the claim is breached in general, we randomly generate an object set O on a fraction of the Eagle Peak land surface dataset [25], and check how many objects $o_i \in O$ has its loose cell $LC(o_i)$ overlapping with that of a non-neighbor object $o_j \notin NL(o_i)$, by using the following approach.

We bulk-load an R-tree T_{LC} from the loose cells $\{LC(o) \mid o \in O\}$ using the Sort-Tile-Recursive (STR) algorithm [10]. For each object $o_i \in O$, we find the set of objects $C = \{o_j \in O - \{o_i\} \mid LC(o_j) \cap LC(o_i) \neq \emptyset\}$ by an intersection query on T_{LC} with query window $LC(o_i)$. Then, for each $o_j \in C$, we check whether $LC(o_i) \cap TC(o_j) \neq \emptyset$. If $o_j \in NL(o_i)$, we know that $LC(o_i)$ and $TC(o_j)$ have common edges and thus $LC(o_i) \cap TC(o_j) \neq \emptyset$. Otherwise, we find a breach of the claim.

The experimental results are surprising: we find that 70% to 80% of the objects in O breach of the claim (which is very high), and the larger the object set size $|O|$, the smaller the breach frequency.

Cell index. While the tight/loose cells of the data objects provide great pruning power in surface NN and SRNN query processing, cell construction is time-consuming. As a result, they are usually pre-computed offline, and organized by a spatial index.

We have shown that the neighbor information maintained by the cell index of [1] is no longer useful for correctly answering SkNN queries. Fortunately, only 1-NN queries are necessary in SRNN query processing, and therefore, we do not adopt the index structure of [1], but rather use the R-tree T_{LC} over all the object loose cells as the cell index, which has already been used in breach checking.

The R-tree T_{LC} is sufficient for answering surface NN and SRNN queries. We now present our algorithm for surface NN queries, namely Algorithm 1, which makes use of T_{LC}.

Algorithm for finding $\mathbf{NN_S(q \mid O)}$. According to Statements 2 and 4 in Theorem 1, we can use Algorithm 1 to find $NN_S(q \mid O)$ for a query point $q \notin O$. Specifically, we first issue a point intersection query on the R-tree T_{LC} to find the objects whose loose cells contain q (Line 1). If there is only one such object o (Line 2), this implies that $\forall o' \in O - \{o\}$, $LC(o')$ does not contain q, and according to Statement 2 in Theorem 1, only o has chance to be $NN_S(q \mid O)$, which is returned in Line 4. In fact, we have

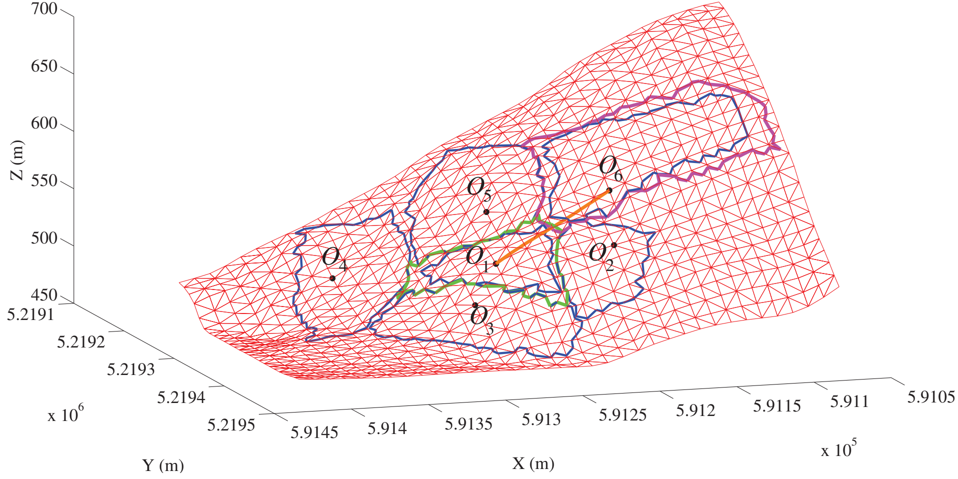


Figure 2: Tight Cells & Loose Cells

Algorithm 1 *QO-NN*(q, O, T_{LC})

Input: query point $q \notin O$; object set O; R-tree T_{LC} bulk-loaded from $\{LC(o) \mid o \in O\}$
Output: $NN_S(q \mid O)$

1: Find the object set $C = \{o \in O \mid q$ is within $LC(o)\}$ by a point intersection query on T_{LC} with query point q
2: **if** $|C|$=1 **then**
3: $\quad \{C = \{o\}\}$
4: $\quad$ **return** o
5: $min \leftarrow \infty$, $o_{min} \leftarrow NULL$
6: **for each** $o \in C$ **do**
7: $\quad$ Invoke *Chen and Han's algorithm* to compute $d_S(o, q)$ on $LC(o)$
8: $\quad$ **if** $d_S(o, q) < min$ **then**
9: $\quad\quad min \leftarrow d_S(o, q)$
10: $\quad\quad o_{min} \leftarrow o$
11: **return** o_{min}

$q \in TC(o)$ in this case. Otherwise, we check all the candidates $o \in C$ and return the one with the smallest shortest surface path length (Lines 5 to 11). Note that we compute $d_S(o, q)$ on $LC(o)$ rather than on the whole surface, and therefore the shortest surface path between q and o found on $LC(o)$ may not be the global shortest surface path. However, according to Statement 4 in Theorem 1, we are able to find the shortest surface path to $NN_S(q \mid O)$, which guarantees the correctness of Algorithm 1.

5. OBJECT NN QUERIES

In this section, we study object NN queries: Given an object $o \in O$, find its nearest neighbor in O on the surface, i.e. $NN_S(o \mid O - \{o\})$. As we shall see, an object NN query is a fundamental operation in our MSRNN algorithm in Section 7.

Cell-cell properties. Before describing our algorithm for processing object NN queries, we first present two underlying *cell-cell properties* which guarantee the correction of our algorithm.

Theorem 2 prunes all the objects whose loose cells do not intersect with $LC(o)$, from being $NN_S(o \mid O - \{o\})$.

THEOREM 2. *Any object $o' \in O$ that satisfies $LC(o) \cap LC(o') = \emptyset$ cannot be $NN_S(o \mid O - \{o\})$.*

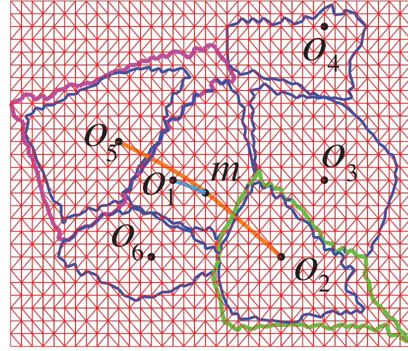


Figure 3: Illustration of the Proof of Theorem 2

PROOF. Let us illustrate the idea by Figure 3, where o_2 corresponds to o in Theorem 2, and o_5 corresponds to o'. In Figure 3, the green (purple) cell is $LC(o_2)$ ($LC(o_5)$).

Since $LC(o) \cap LC(o') = \emptyset$, a point m on the shortest path between o and o' (the orange path in Figure 3) should exist outside both $LC(o)$ and $LC(o')$.

Since m is outside of $LC(o')$, according to Definition 5, we know that $\exists o'' \in O - \{o'\}$, $d_E(o', m) > d_N(o'', m)$. We now prove that $\exists o'' \in O - \{o', o\}$, $d_E(o', m) > d_N(o'', m)$.

We prove by contradiction. If o is the only object in $O - \{o'\}$ that satisfies $d_E(o', m) > d_N(o, m)$, then $\forall o'' \in O - \{o', o\}$, $d_E(o', m) \leq d_N(o'', m)$. Since $d_E(o', m) > d_N(o, m) \geq d_E(o, m)$, we have $d_N(o', m) \geq d_E(o', m) > d_E(o, m)$ and $\forall o'' \in O - \{o', o\}$, $d_E(o, m) < d_E(o', m) \leq d_N(o'', m)$. Therefore, we obtain $\forall o'' \in O - \{o\}, d_E(o, m) < d_N(o'', m)$. According to Definition 5, this implies that m is inside $LC(o)$, which contradicts our assumption of m.

Now that we know $\exists o'' \in O - \{o', o\}$, $d_E(o', m) > d_N(o'', m)$ (in Figure 3, o_1 corresponds to o''), we have $d_S(o', m) \geq d_E(o', m) > d_N(o'', m) \geq d_S(o'', m)$. Thus, $d_S(o', o) = d_S(o', m) + d_S(m, o) > d_S(o'', m) + d_S(m, o) \geq d_S(o'', o)$, which implies that o'' is closer to o than o', i.e. $o' \neq NN_S(o \mid O - \{o\})$. □

Theorem 2 provides the candidates of $NN_S(o \mid O - \{o\})$ for further refinement. To find the exact $NN_S(o \mid O - \{o\})$ among the candidates, we have to invoke *Chen and Han's algorithm* to compute the lengths of shortest surface paths from o to these candidates.

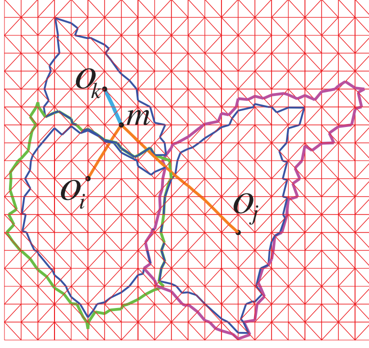


Figure 4: Illustration of the Proof of Theorem 3

Since the time cost of *Chen and Han's algorithm* is quadratic to the number of triangular faces, we would like to evaluate the shortest surface paths on a small fraction of the surface, which is made possible by the following theorem:

THEOREM 3. *If* $o_j = NN_S(o_i \mid O - \{o_i\})$*, then the shortest surface path from* o_j *to* o_i *is within* $LC(o_j) \cup LC(o_i)$*.*

PROOF. Let us prove by contradiction. Note that $LC(o_j)$ must intersect with $LC(o_i)$ according to Theorem 2. Suppose that a point m on the shortest path between o_j and o_i exists that is outside both $LC(o_j)$ and $LC(o_i)$ (see Figure 4).

Since m is outside of $LC(o_j)$, according to Definition 5, we have $\exists o_k \in O - \{o_j\},\ d_E(o_j, m) > d_N(o_k, m)$. We now prove that $\exists o_k \in O - \{o_i, o_j\},\ d_E(o_j, m) > d_N(o_k, m)$.

We prove by contradiction. If o_i is the only object in $O - \{o_j\}$ that satisfies $d_E(o_j, m) > d_N(o_i, m)$, then $\forall o_k \in O - \{o_i, o_j\},\ d_E(o_j, m) \leq d_N(o_k, m)$. Since $d_E(o_j, m) > d_N(o_i, m) \geq d_E(o_i, m)$, we have $d_N(o_j, m) \geq d_E(o_j, m) > d_E(o_i, m)$ and $\forall o_k \in O - \{o_j, o_i\},\ d_E(o_i, m) < d_E(o_j, m) \leq d_N(o_k, m)$. Therefore, we have $\forall o_k \in O - \{o_i\}, d_E(o_i, m) < d_N(o_k, m)$. According to Definition 5, this implies that m is inside $LC(o_i)$, which contradicts our assumption of m.

Now that we know $\exists o_k \in O - \{o_i, o_j\},\ d_E(o_j, m) > d_N(o_k, m)$ (see Figure 4), we have $d_S(o_j, m) \geq d_E(o_j, m) > d_N(o_k, m) \geq d_S(o_k, m)$. Thus, $d_S(o_j, o_i) = d_S(o_j, m) + d_S(m, o_i) > d_S(o_k, m) + d_S(m, o_i) \geq d_S(o_k, o_i)$, which implies that o_k is closer to o_i than o_j, contradicting the assumption $o_j = NN_S(o_i \mid O - \{o_i\})$ in the theorem. ☐

Algorithm for finding $\mathbf{NN_S(o \mid O - \{o\})}$. Our MSRNN query processing only requires to compute the surface distance between an object $o \in O$ and its surface nearest neighbor in O, i.e. $NN_S(o \mid O - \{o\})$. According to Theorems 2 and 3, we can use Algorithm 2 to compute this distance. Note that Algorithm 2 can be easily extended to find $NN_S(o \mid O - \{o\})$ by tracking the object with minimum current distance.

Since the computation of $d_S(o, NN_S(o \mid O - \{o\}))$ for objects $o \in O$ is a basic operation in our MSRNN query processing, one method is to pre-compute the value for each object $o \in O$. However, unlike Algorithm 1 which may not require any shortest surface path computation, Algorithm 2 requires to perform this expensive computation for $|C|$ times. Thus, the pre-computation phase takes a long time and is only worthwhile for those applications where the object set O is static and they are frequently queried. In our implementation, $d_S(o, NN_S(o \mid O - \{o\}))$ is computed online.

Intersection judgement. Algorithm 1 requires a point intersection query on R-tree T_{LC}, and if the query point q is found to be within the minimum bounding box (MBR) of a leaf node entry corresponding to $LC(o)$, a refinement step is required to check whether q is within the polygon $LC(o)$. Note that all the intersection judgements can be done simply on the xy-plane, without considering the z-coordinate values. We regard q as being within a polygon P even if q is on the boundary of P.

Algorithm 2 *OO-NN*(o, O, T_{LC})

Input: query point $o \in O$; object set O; R-tree T_{LC} bulk-loaded from $\{LC(o) \mid o \in O\}$
Output: $d_S(o, NN_S(o \mid O - \{o\}))$

1: Find the set $C = \{o' \in O - \{o\} \mid LC(o') \cap LC(o) \neq \emptyset\}$ by an intersection query on T_{LC} with query window $LC(o)$
2: $min \leftarrow \infty$
3: **for each** $o' \in C$ **do**
4: Invoke *Chen and Han's algorithm* to compute $d_S(o, o')$ on $LC(o') \cup LC(o)$
5: **if** $d_S(o, o') < min$ **then**
6: $min \leftarrow d_S(o, o')$
7: **return** min

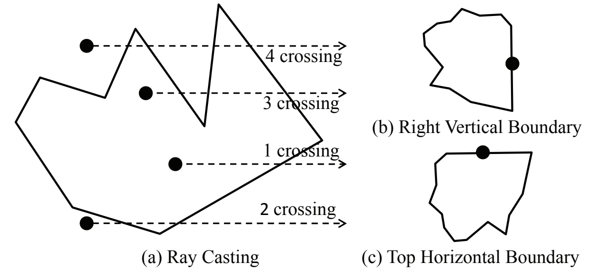


Figure 5: Illustration of the Ray Casting Algorithm

To determine whether a point q is within a polygon P, we use the *ray casting algorithm*. The method counts the number of times the horizontal ray extending to the right of q crosses a polygon boundary edge. If this number is even, the point is outside; otherwise, the point is inside. Figure 5(a) illustrates the idea of the *ray casting algorithm*. However, due to the *edge crossing rules* of the algorithm, a point on a right or top edge is considered outside, and therefore we check these special cases (e.g. the cases shown in Figures 5(b) and (c)) to ensure the correctness of intersection judgement.

Algorithm 2 requires an intersection query on R-tree T_{LC} with a polygon window win, and if the MBR of win is found to intersect with the MBR of a leaf node entry corresponding to $LC(o)$, a refinement step is needed to check whether polygon $LC(o)$ intersects with win.

The general problem of determining whether two polygons P_1 and P_2 intersect cannot be answered by simply checking whether there exists a vertex of P_1 that is within P_2 (see Figure 6(a)), and requires more advanced techniques such as the *Bentley-Ottmann algorithm* [24].

However, both win and $LC(o)$ in our problem are loose cells, whose boundary must have a vertex on an edge of the *model network* whenever it crosses the edge (see Section 6). Therefore, we have the following theorem:

THEOREM 4. *Two loose cells* LC_1 *and* LC_2 *intersect with each other, if and only if, there exists a vertex of* LC_1 *that is within* LC_2*.*

PROOF. Suppose that part of the edge de of LC_1 is inside LC_2 but both d and e are outside of LC_2, then de crosses at least two

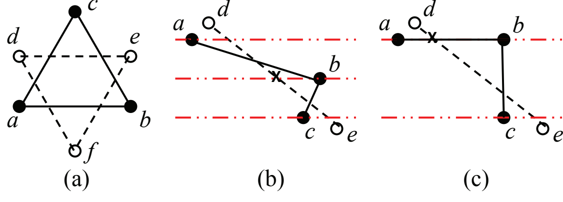


Figure 6: Polygon Intersection

edges of LC_2, such as edges ab and bc in Figures 6(b) and (c). Let us call a horizontal or vertical line of model network edges as a model line. Note that there are two possible cases: (1) If b is not collinear with a and c on a model line, then there exist at least 3 model lines between a and c (see Figure 6(b)) and de must cross the model line in between. (2) If b is collinear with a (or c) on a model line as shown in Figure 6(c), then de must cross the model line of ab (or bc). Thus, in either case, there is another vertex on de, contradicting the fact that de is an edge of LC_1. $\square$

Note that we can use the *ray casting algorithm* described before to determine whether a vertex of LC_1 is within LC_2.

6. CELL CONSTRUCTION

In this section, we present our approach for cell construction, which is important due to the following reasons. (1) In order to accelerate SRNN query processing, we need to pre-compute $TC(o)$ and $LC(o)$ for all $o \in O$. Recall that we also need to bulk-load an R-tree T_{LC} from the loose cells $LC(o)$. (2) Our MSRNN algorithm requires to obtain the loose cell of the query point online.

The work of [1] proposes to compute the tight/loose cells by contracting/expanding the Voronoi cells of the object set O in 3D Euclidean space. However, it is not easy to compute the 3D Voronoi cells and then to map them onto the surface. Besides, this approach constructs the cells of all objects in O at once, and cannot support efficient cell evaluation for an individual point q, which is a fundamental operation in our MSRNN algorithm.

Therefore, we propose to construct the cells of a point online, by a breadth-first traversal of the faces starting from that point. We first describe the data structures relevant cell construction.

Vertex inverted index. We build an inverted index I_{faces} where each entry corresponds to a vertex v of the TIN model. Each entry of I_{faces} has format $\langle v, L(v)\rangle$, with $L(v)$ being the list of all the faces whose vertices include v. For example, in Figure 7, the list $L(b)$ contains six faces marked by 1 to 6.

One usage of I_{faces} is to get the seeding faces for our breadth-first cell construction algorithm, and the other usage is to obtain the gluing relationships of faces, which are required as part of the input to *Chen and Han's algorithm*. Referring to Figure 7 again, suppose that we want to obtain the face next to Face 6 by the edge ab. This can be done by first computing $L(a) \cap L(b) = \{5, 6\}$ and then picking the face that is not itself (i.e. Face 5). For a face on the

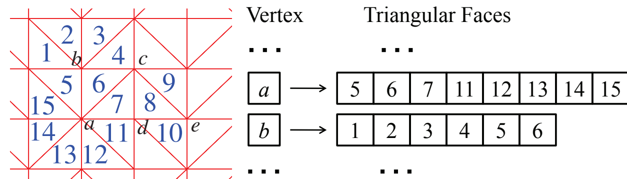


Figure 7: Vertex Inverted Index for Faces

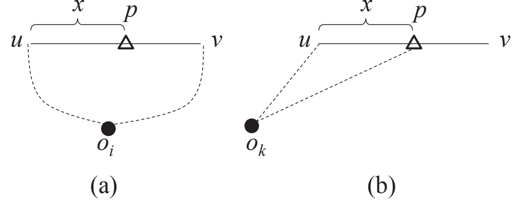


Figure 8: Exact Cell Vertex Computation

boundary of the land surface, its neighboring face along a boundary edge does not exist, and we denote it by $NULL$.

The above approach requires two I_{faces} look-ups (i.e. two I/O operations) to get one neighboring face of a given face. An alternative method is to pre-compute the three neighboring faces of each face in the TIN model, so that only one I/O operation is required to obtain all the three neighboring faces for a given face. We do not adopt this alternative to avoid the additional space overhead.

Tight cell vertex computation. Let us consider how to find the vertices of $TC(o_i)$. Given a vertex v of the model network, let o_k be the NN of v among $O - \{o_i\}$ in the Euclidean space, i.e. $o_k = \arg\min_{o_j \in O-\{o_i\}} d_E(o_j, v)$. We bulk-load an R-tree T_O from the object points in O beforehand, so that o_k can be found by a best-first NN search [12] using query point v, where object o_i is filtered out during the R-tree traversal.

To decide whether v is within $TC(o_i)$, according to Equation (2), we need to compute and compare $d_N(o_i, v)$ and $d_E(o_k, v)$. If $d_N(o_i, v) \le d_E(o_k, v)$, we say that vertex v is on o_i's side, denoted as $v \dashv o_i$; otherwise, we say that $v \not\dashv o_i$.

For an edge uv of the model network, if $u \dashv o_i$ and $v \not\dashv o_i$, then $d_N(o_i, u) \le d_E(o_k, u)$ and $d_N(o_i, v) > d_E(o_k, v)$, and therefore there must exist at least one *split point* s on uv such that $d_N(o_i, s) = d_E(o_k, s)$. The split point s is a vertex of $TC(o_i)$.

There can be more than one *split point* on an edge uv, which can be found by solving a quartic equation. Consider a point p on edge (u, v) with $|up| = x$. From Figure 8(a) we can see that $d_N(o_i, p) = \min\{d_N(o_i, u) + x, d_N(o_i, v) + (|uv| - x)\}$ is a piecewise linear function of x with at most 2 pieces. In Figure 8(b), by Cosine law, we can obtain $d_E(o_k, p)^2 = d_E(o_k, u)^2 + x^2 - 2 \cdot d_E(o_k, u) \cdot x \cdot \cos\angle o_k uv$. Thus, we can solve the equation $d_N(o_i, s)^2 = d_E(o_k, s)^2$ to compute split points s.

The work of [1] mistakenly assumes that at most one split point s exists on an edge uv. To avoid the complication of solving quartic equations, we follow their assumptions by finding a split point inside the exact $TC(o_i)$. Although cells computed in this way are not the exact ones, they do not influence the correctness of SRNN evaluation, but just slightly reduce the pruning power of the cells, due to the following intuitive principle:

Making the tight cells smaller and the loose cells larger than the exact ones does not influence the correctness of the cell properties.

We compute the split point s on an edge as follows. Consider face $\triangle def$ in Figure 9, where $e \dashv o_i$ and $f \not\dashv o_i$. We compute the split point s on ef as follows: Let us denote $x = |fs|$, we compute s by solving the equation

$$d_E(o_k, f) + x = (|ef| - x) + d_N(o_i, e) \qquad (4)$$

The solution is $x = (|ef| + d_N(o_i, e) - d_E(o_k, f))/2$, and the coordinate of s is computed as $\overrightarrow{Os} = \overrightarrow{Of} + x \cdot \overrightarrow{fe}$. We now prove that all the *split points* we compute is inside $TC(o_i)$: the R.H.S. of Equation (4) equals $d_N(o_i, s)$, and the L.H.S. is at least $d_E(o_k, s)$, which implies that $d_N(o_i, s) \le d_E(o_k, s)$, or s is inside $TC(o_i)$.

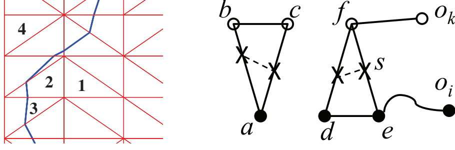


Figure 9: Relaxed Cell Vertex Computation

The work of [1] differentiates 4 possible cases for a face: (1) all three vertices are on o_i's side (e.g. Face 1 in Figure 9), in which case the face is totally contained in $TC(o_i)$; (2) all three vertices are not on o_i's side (e.g. Face 4 in Figure 9), in which case the face is totally outside of $TC(o_i)$; (3) only one vertex is on o_i's side (e.g. Face 3 in Figure 9), in which case two split points can be computed as illustrated by face $\triangle abc$ in Figure 9; (4) two vertices are on o_i's side (e.g. Face 2 in Figure 9), in which case two split points can be computed as illustrated by face $\triangle def$ in Figure 9. In the last two cases, a cell edge of $TC(o_i)$ is defined by the two split points. We also adopt this approach in our implementation.

Loose cell vertex computation. The vertices computation for $LC(o_i)$ is similar but more tricky. Given a vertex v of the model network, let $o_k = \arg\min_{o_j \in O-\{o_i\}} d_N(o_j, v)$. According to Equation (3), to decide whether v is within $LC(o_i)$, we need to compute and compare $d_E(o_i, v)$ and $d_N(o_k, v)$.

Due to the large size of the TIN *model network* (typically with millions of vertices) and the object set O, it is not practical to find o_k by evaluating $d_N(o_j, v)$ for all $o_j \in O - \{o_i\}$. Besides, since the computation of $LC(o_i)$ works in a breath-first manner, the network distance computation for different vertices v can be shared rather than done individually, so as to utilize the locality property.

Our approach of network distance computation when evaluating $LC(o_i)$ is as follows. We maintain a pool of partial results obtained from the Dijkstra algorithm for different source vertices $o_k \in O - \{o_i\}$ during the computation of $LC(o_i)$. Whenever a network distance computation $d_N(o_k, v)$ is required, we check whether the partial result of o_k exists in the pool: (1) if the partial result exists, we check whether $d_N(o_k, v)$ is already available. If it is available, it is returned directly; otherwise, we continue to run the Dijkstra algorithm until $d_N(o_k, v)$ is computed. (2) if the partial result does not exist, we create and initialize a new partial result for o_k, run Dijkstra until $d_N(o_k, v)$ is computed, and add the partial result to the pool.

Our approach is efficient since only a small number of partial results are kept in the pool. To see this, consider the example in Figure 10, where vertex c of face $\triangle abc$ is being evaluated. Note that o_{k5} is so far from o_i that $d_E(o_{k5}, c) > d_N(o_{k3}, c)$, thus $d_N(o_{k5}, c)$ is even larger and o_{k5} can be safely pruned. In general, if $d_E(o_\ell, v)$ is found to be larger than $\min_{o_j}\{d_N(o_j, v)\}$ (named *upper bound*) for all checked objects o_j, then o_ℓ can be pruned.

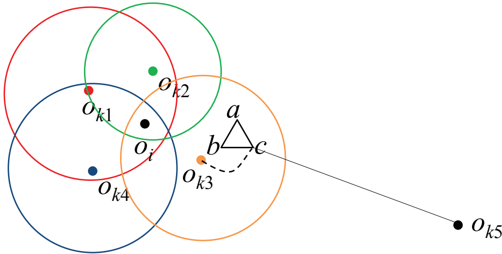


Figure 10: Pruning Candidates of o_k

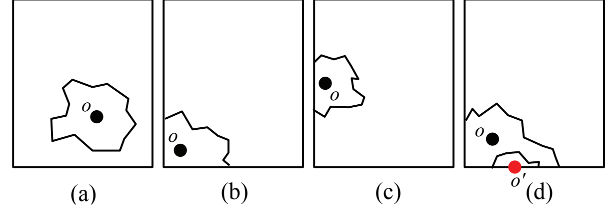


Figure 11: Boundary Cells

To get a tight *upper bound* in the very beginning, we find the nearest neighbor of v among objects in $O - \{o_i\}$ in the Euclidean space, say o_n, using the object R-tree index T_O mention before, and initialize the upper bound as $d_N(o_n, v)$.

The split point computation is similar to that of $TC(o_i)$.

Breath-first cell edge collection. To collect all the edges of $TC(o_i)$ and $LC(o_i)$, we check the faces in a breadth-first manner, starting from o_i. Specifically, we first initialize a queue Q with the faces whose vertices contain o_i, which can be obtained by looking up $L(o_i)$ from I_{faces}. Then, whenever Q is not empty, we fetch the next face in Q, check the three face vertices, and evaluate split points for $TC(o_i)$ and $LC(o_i)$ on the face. If at least one vertex is on o_i's side in terms of $LC(o_i)$, we add the non-visited neighboring faces (if exists) to Q; otherwise, we know that the face is totally outside of $LC(o_i)$ and we do not further expand faces.

Recall that during this process, we maintain a pool of partial Dijkstra results for different source vertices o_k. Since $LC(o_i)$ covers only a very small fraction of the whole surface, we can expect that objects far away from o_i will never require a network distance computation to a vertex on a traversed face. Thus, the pool is kept small (usually less than 10 source objects), and even for those source objects in the pool, the Dijkstra algorithm is run on just a small fraction of the whole surface, as illustrated by the circles in Figure 10.

Boundary cells. For most of the cells, the edges collected form a closed curve (see Figure 11(a)), and thus the polygon cell can be easily constructed. However, there are cases where the edges form one open curve (see Figures 11(b) and (c)), and additional edges or vertices from the boundary of the surface are required to form the polygon cell. To decide which side of the curve is inside, we check whether object o is in the resulted polygon. In our experiments, we also find other rare cases such as the one shown in Figure 11(d). Therefore, for all the other cases, we set $TC(o)$ to be the point o and $LC(o)$ to be the MBR of the collected split points.

7. SRNN QUERY PROCESSING

7.1 MSRNN Query Processing

Given an object set O and a query point q on a land surface, a MSRNN query returns the set of objects $\{o \in O \mid NN_S(o \mid O \cup \{q\} - \{o\}) = q\}$. Equivalently, an object o is the MSRNN of q, if and only if

$$d_S(o, NN_S(o \mid O - \{o\})) \geq d_S(o, q). \quad (5)$$

To obtain the MSRNN candidates, we compute $LC(q)$ among the set $O \cup \{q\}$ online using our breadth-first cell construction algorithm described in Section 6, with an important additional operation: whenever we obtain a split point s for the loose cell $LC(q)$ on edge uv, according to Equation (3) we have

$$d_E(q, s) = d_N(o, s),\ o = \arg\min_{o_i \in O} d_N(o_i, s), \quad (6)$$

and we collect o into the MSRNN candidate set C. The following theorem guarantees that the MSRNNs of q must be in C.

Algorithm 3 Finding the MSRNNs of Query Point q

1: $MRNN \leftarrow \emptyset$
2: Compute $LC(q)$ among $O \cup \{q\}$ and collect candidates to set C
3: **for each** $o \in C$ **do**
4: **if** $LC(o) \cap LC(q) \neq \emptyset$ **then**
5: $d_S(o, NN_S(o \mid O - \{o\})) \leftarrow$*OO-NN*$(o, O, T_{LC})$
6: Invoke *Chen and Han's algorithm* to compute $d_S(o, q)$ on $LC(o) \cup LC(q)$
7: **if** $d_S(o, NN_S(o \mid O - \{o\})) \geq d_S(o, q)$ **then**
8: $MRNN \leftarrow MRNN \cup \{o\}$
9: **return** $MRNN$

THEOREM 5. *Any object $o \notin C$ in O cannot be q's MSRNN.*

PROOF. Note that the polygon $LC(q)$ computed in the context of $O \bigcup \{q\}$ is the same as the polygon $LC(q)$ computed in the context of $C \bigcup \{q\}$. Therefore, for any point v outside $LC(q)$, it holds that $\exists o_c \in C,\ d_E(q, v) > d_N(o_c, v)$.

For an object $o \notin C$ in O, it must be outside of $LC(q)$ according to the definition of loose cells, and therefore $\exists o_c \in C,\ d_E(q, o) > d_N(o_c, o)$. Thus $d_S(q, o) \geq d_E(q, o) > d_N(o_c, o) \geq d_S(o_c, o) \geq d_S(o, NN_S(o \mid O - \{o\}))$, and according to Equation (5), o cannot be an MSRNN. □

Algorithm 3 shows our algorithm for MSRNN queries. In Line 2, we first compute the MSRNN candidate set C by evaluating $LC(q)$ among $O \cup \{q\}$ using the breadth-first approach. We filter out those candidates o that satisfy $LC(o) \cap LC(q) = \emptyset$ in Line 4. This is because, according to Theorem 2, any point $p \in O \cup \{q\}$ that satisfies $LC(o) \cap LC(q) = \emptyset$ cannot be $NN_S(o \mid O \cup \{q\} - \{o\})$, where $LC(o)$ is computed in the context of $O \cup \{q\}$, denoted as $LC(o \mid O \cup \{q\})$. Note that the $LC(o)$ in Algorithm 3 is the pre-computed one in the context of O, i.e. $LC(o \mid O)$. According to Definition 5, it can be easily proved that $LC(o \mid O \cup \{q\}) \subseteq LC(o \mid O)$. Therefore, if Line 4 finds that $LC(o \mid O) \cap LC(q) = \emptyset$, then $LC(o \mid O \cup \{q\}) \cap LC(q) = \emptyset$ and q cannot be $NN_S(o \mid O \cup \{q\} - \{o\})$.

For each non-pruned candidate o, we compute $d_S(o, NN_S(o \mid O - \{o\}))$ in Line 5 using the object NN query described in Section 5, compute $d_S(o, q)$ in Line 6, and then determine whether o is an MSRNN of q by checking Equation (5) in Line 7. Note that in Line 6, we only need to evaluate the shortest surface path on $LC(o \mid O) \cup LC(q)$ ($\supseteq LC(o \mid O \cup \{q\}) \cup LC(q)$) according to Theorem 3.

If $d_S(o, NN_S(o \mid O - \{o\}))$ is pre-computed for all $o \in O$, Algorithm 3 only require one shortest surface path computation. Otherwise, it needs to compute several shortest surface paths.

7.2 BSRNN Query Processing

To process BSRNN queries, we pre-compute the cells for all sites $s \in S$ and bulk-load an R-tree T_{LC} from $\{LC(s) \mid s \in S\}$. During cell construction, an R-tree T_S is bulk-loaded for nearest neighbor queries when computing the cells. Besides, we also bulk-load an R-tree T_O for range queries used in the BSRNN query processing.

Algorithm 4 shows our algorithm for BSRNN queries. According to Statement 2 in Theorem 1, only those objects within $LC(q)$ have chance to be BSRNNs, and they are obtained as the candidate set C in Line 2. For each candidate o, if it is within $TC(q)$, it is guaranteed to be a BSRNN of q according to Statement 1 in Theorem 1. Otherwise, we find the nearest site to o in Line 7 and check whether it is q in Line 8 to determine whether o is a BSRNN.

The algorithm can also support BSRNN queries on a moving object set O, by maintaining T_O as a TPR*-tree [23].

Algorithm 4 Finding the BSRNNs of Query Site q

1: $BRNN \leftarrow \emptyset$
2: Perform a range query on T_O with query window $LC(q)$ to find the object set $C = \{o \in O \mid o \text{ is within } LC(q)\}$
3: **for each** $o \in C$ **do**
4: **if** o is within $TC(q)$ **then**
5: $BRNN \leftarrow BRNN \cup \{o\}$
6: **else**
7: $nn \leftarrow$*QO-NN*(o, S, T_{LC})
8: **if** $nn = q$ **then**
9: $BRNN \leftarrow BRNN \cup \{o\}$
10: **return** $BRNN$

8. EXPERIMENTS

In this section, we evaluate the performance of our algorithms for MSRNN and BSRNN queries on two large real-world datasets downloaded from [25]: Eagle Peak (Eagle) and Bearhead (BH), which have also been used in previous studies such as [1] and [2]. The statistical information of the datasets is given in Table 1.

Table 1: Statistics of Real Datasets

	Number of Vertices	Number of Faces
Eagle Peak	1,381,481	2,762,693
Bearhead	1,318,844	2,637,538

We use the most recent implementation of *Chen and Han's algorithm* [9] for surface shortest path computation. All our programs are written in JAVA, and the executable program of [9] is called on the localized surface regions for surface shortest path computation.

The pre-computation of the index structures, such as the vertex inverted index I_{faces} and the loose cell R-tree T_{LC}, are carried out on a public Linux server with eight 3GHz Intel Xeon X5450 CPU and 32GB memory. All the experiments on queries are done on a computer with 3GHz Intel Core2 Duo E8400 CPU and 2GB memory, where the huge *model network* and all the index structures are disk-based.

Let $G = (V, E)$ be the model network, we define object (site) density as $|O|/|V|$ ($|S|/|V|$). We test our algorithms with different object densities. For each density configuration, we randomly generate 100 query points and run our algorithms with each query point. All the reported measures are averaged on the 100 runs.

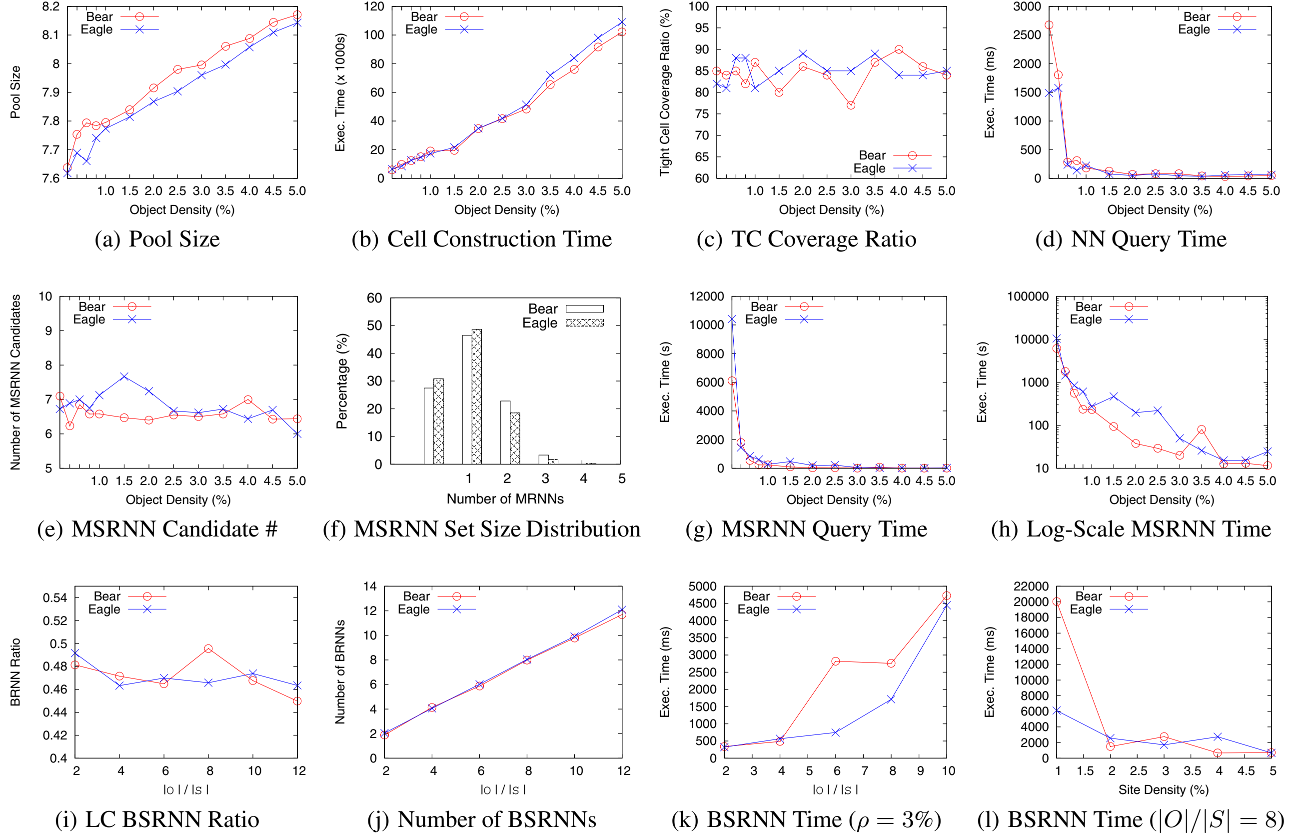
8.1 Results of Index Construction

The operation of breadth-first cell construction for all objects $o \in O$ dominates the time cost of the index construction phase. Figure 12(a) shows the number of source objects whose partial Dijkstra results is in the pool when computing $TC(o)$ and $LC(o)$ averaged over all the objects in O, with varying object density. We can see from Figure 12(a) that, as object density increases, the pool size also increases. The pool size is usually around 7 to 8, which verifies the effectiveness of our Euclidean distance based pruning of network distance computation.

Figure 12(b) shows the time of constructing cells for all the objects $o \in O$, with varying object density. We can see from the figure that the cell construction time increases almost linearly with the increment of object density. This is because, given a specific surface data, object number is proportional to object density.

8.2 Results of NN Queries

In this set of experiments, we evaluate the performance of our algorithm for surface NN queries (Algorithm 1). Recall that no



(a) Pool Size (b) Cell Construction Time (c) TC Coverage Ratio (d) NN Query Time

(e) MSRNN Candidate # (f) MSRNN Set Size Distribution (g) MSRNN Query Time (h) Log-Scale MSRNN Time

(i) LC BSRNN Ratio (j) Number of BSRNNs (k) BSRNN Time ($\rho = 3\%$) (l) BSRNN Time ($|O|/|S| = 8$)

Figure 12: Experimental Results

shortest surface path computation is necessary if query point q falls within the tight cell of some object. For each object density, we count the number of such queries, denoted N, among the 100 NN queries tested. The ratio $N/100$ is called the *Tight Cell Coverage Ratio* (TCCR). Figure 12(c) shows the TCCR with varying object density, where around 80% to 90% NN queries do not need to invoke *Chen and Han's algorithm*. Figure 12(d) shows the average NN query processing time with varying object density. The query processing time is long (1 to 3 seconds) for sparse object distribution, due to the large areas of loose cells which act as input to *Chen and Han's algorithm*.

8.3 Results of MSRNN Queries

In this set of experiments, we evaluate the performance of our algorithm for MSRNN queries (Algorithm 3). Recall that Algorithm 3 performs MSRNN candidate filtering in the beginning in Line 2. Figure 12(e) shows the average number of candidates with varying object density, where we can see that there are usually 6 to 7 candidates and the trend is not sensitive to object density. We also find that the MSRNN count distribution is not sensitive to object density. Figure 12(f) shows the MSRNN count distribution over all the MSRNN queries we tested, where almost half of the query points have exactly one MSRNN, very few of the query points have four MSRNNs, and no query point has over four MSRNNs.

Figure 12(g) shows the average MSRNN query processing time with varying object density. The query processing time is quite long for sparse object distribution, due to the large areas of loose cells which act as input to *Chen and Han's algorithm*.

To further clarify the trend of query time, we plot the curves in log-scale in Figure 12(h), where we can see that the performance of our algorithm is good for reasonably large object densities ($\geq$3%).

8.4 Results of BSRNN Queries

In this set of experiments, we evaluate the performance of our algorithm for BSRNN queries (Algorithm 4). For each site density, we generate object sets of different sizes, and for each object set, we perform 100 BSRNN queries.

The objects that fall in $TC(q)$ are guaranteed to be q's BSRNNs. Let the percentage of BSRNNs among the objects in $LC(q) - TC(q)$ be termed the *BSRNN ratio*. Figure 12(i) shows the *BSRNN ratio* when $|O|/|S|$ varies. The results are averaged over experiments with different site densities, since *BSRNN ratio* is insensitive to site density. We can see that around 45% to 50% of the objects that fall in region $LC(q) - TC(q)$ are q's BSRNNs.

The average number of BRNNs of a query point is also found to be insensitive to site density. On the other hand, it is sensitive to $|O|/|S|$, as Figure 12(j) shows. This is within the expectation, since each site $s \in S$ has to serve $|O|/|S|$ objects on average.

Figure 12(k) shows the average BSRNN query processing time on the site set S with site density 3%, when $|O|/|S|$ varies. The query processing time increases when $|O|/|S|$ increases. This is because, as $|O|/|S|$ becomes larger, more objects tend to fall in the region $LC(q) - TC(q)$, and therefore more NN queries are required to check whether each candidate object is q's BSRNN.

Figure 12(l) shows the average BSRNN query processing time for different site densities, when $|O|/|S| = 8$. The query time increases as site density decreases. This is because smaller site densities implies larger areas of loose cells, which not only implies

larger input region to *Chen and Han's algorithm*, but also higher chance that an object falls in region $LC(q) - TC(q)$.

9. RELATED WORK

The concept of RNN is first introduced in [13]. Since then, many algorithms have been proposed for finding RNNs in the Euclidean space. Algorithms for MRNN query processing include SAA [14], TPL [15], Finch [20], Influence Zone [21]. Algorithms for BRNN query processing include [18] and [19]. [22] studies RNN queries in road networks.

Studies on finding shortest surface paths date back to the 80s to 90s, when a series of exact and approximation algorithms are proposed. The most efficient exact algorithms include the one proposed in [7] with $O(n^2 \log n)$ time complexity, and *Chen and Han's algorithm* [8] with $O(n^2)$ time complexity, where n corresponds to the number of triangles in the TIN model. *Chen and Han's algorithm* is later implemented in [9].

Recently, [5] proposes an approximate algorithm with $O(\log n + \sqrt{n})$ time complexity, by treating the surface paths on the *non-rough* areas as straight lines, and [6] studies the problem of finding shortest paths with slope constraint and develops the *surface simplification* technique to reduce the model complexity, so as to reduce the time complexity of finding shortest paths.

As for the research on surface k-NN queries, [2] and [3] first propose a filter and refinement strategy to answer SkNN queries on multi-resolution terrain models. However, this approach can neither guarantee accuracy nor provide the shortest surface paths to the nearest neighbors. An exact solution to the SkNN problem is proposed in [1], where approximations of *Voronoi cells* are constructed to prune unnecessary shortest surface path computation and to localize the shortest path computation. In this paper, we also make use of these *Voronoi Cell* approximation structures for the same purposes. Although the concept of *Voronoi diagram* is only related to 1-NN, [1] claims that its approximation structures can be used in an incremental manner to answer k-NN queries for arbitrary k. Unfortunately, as we have discussed in Section 4, the correctness of the algorithm relies on an unproved statement, which is, however, flawed. [4] studies the problem of continuously monitoring the k nearest neighbors of a fixed query point when the objects on the surface are moving.

10. CONCLUSION

In this paper, we investigate how to process RNN queries on land surfaces. Specifically, we study *monochromatic SRNN* queries and *bichromatic SRNN* queries. Our SRNN algorithms are based on several newly-discovered properties of the Voronoi cell approximation structures, and are able to localize the query evaluation by accessing just a small fraction of the surface data near the query point. The majority of the objects that cannot be the SRNNs of the query point are pruned by our cell properties. Furthermore, we propose a new online cell construction algorithm which is essential to our MSRNN algorithm, and is efficient due to our smart approach for computing network distances. Extensive experiments on large real-world datasets demonstrate the efficiency of our algorithms.

11. ACKNOWLEDGEMENTS

This work is partially supported by RGC GRF under grant number HKUST 618509.

12. REFERENCES

[1] C. Shahabi, L.-A. Tang and S. Xing. "Indexing Land Surface for Efficient kNN Query". In *VLDB*, 2008.

[2] K. Deng, X. Zhou, H. T. Shen, K. Xu and X. Lin. "Surface k-NN Query Processing". In *ICDE*, 2006.

[3] K. Deng, X. Zhou, H. T. Shen, Q. Liu, K. Xu and X. Lin. "A Multi-Resolution Surface Distance Model for k-NN Query Processing". In *VLDB Journal*, 17(5):1101–1119, 2008.

[4] S. Xing, C. Shahabi and B. Pan. "Continuous Monitoring of Nearest Neighbors on Land Surface". In *VLDB*, 2009.

[5] S. Xing and C. Shahabi. "Scalable Shortest Paths Browsing on Land Surface". In *ACM GIS*, 2010.

[6] L. Liu and R. C.-W. Wong. "Finding Shortest Path on Land Surface". In *ACM SIGMOD*, 2010.

[7] J. S. B. Mitchell, D. M. Mount and C. H. Papadimitriou. "The Discrete Geodesic Problem". In *SIAM J. Comput.*, 16(4):647–668, 1987.

[8] J. Chen and Y. Han. "Shortest Paths on a Polyhedron". In *SoCG*, 1990.

[9] B. Kaneva and J. O'Rourke. "An Implementattion of Chen & Han's Shortest Paths Algorithm". In *CCCG*, 2000.

[10] S. T. Leutenegger, J. M. Edgington and M. A. Lopez. "STR: A Simple and Efficient Algorithm for R-tree Packing". In *Technical Report*, *Institute for Computer Applications in Science and Engineering*, 1997.

[11] N. Roussopoulos, S. Kelly and F. Vincent. "Nearest Neighbor Queries". In *SIGMOD*, 1995.

[12] G. R. Hjaltason and H. Samet. "Distance Browsing in Spatial Databases". In *ACM TODS*, 24(2), June 1999, pp. 265–318.

[13] F. Korn and S. Muthukrishnan. "Influence Sets Based on Reverse Nearest Neighbor Queries". In *SIGMOD*, 2000.

[14] I. Stanoi, D. Agrawal and A. El Abbadi. "Reverse Nearest Neighbor Queries for Dynamic Databases". In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2000.

[15] Y. Tao, D. Papadias and X. Lian. "Reverse kNN Search in Arbitrary Dimensionality". In *VLDB*, 2004.

[16] M. L. Yiu, D. Papadias, N. Mamoulis and Y. Tao. "Reverse Nearest Neighbors in Large Graphs". In *TKDE*, 2006.

[17] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang and X. Li. "Continuous Reverse k Nearest Neighbors Queries in Euclidean Space and in Spatial Networks". In *VLDB Journal*, 2011.

[18] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. "Discovery of Influence Sets in Frequently Updated Databases". In *VLDB*, 2001.

[19] T. Xia, D. Zhang, E. Kanoulas and Y. Du. "On Computing Top-t Most Influential Spatial Sites". In *VLDB*, 2005.

[20] W. Wu, F. Yang, C.-Y. Chan and K.-L. Tan. "Finch: Evaluating Reverse k-Nearest-Neighbor Queries on Location Data". In *PVLDB*, 2008.

[21] M. A. Cheema, X. Lin, W. Zhang and Y. Zhang. "Influence Zone: Efficiently Processing Reverse k Nearest Neighbors Queries". In *ICDE*, 2011.

[22] D. Taniar1, M. Safar, Q. T. Tran, W. Rahayu and J. H. Park. "Spatial Network RNN Queries in GIS". *The Computer Journal*, 2010.

[23] Y. Tao, D. Papadias and J. Sun. "The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries". In *VLDB*, 2003.

[24] M. D. Berg, O. Cheong, M. V. Kreveld and M. Overmars. "Computational Geometry: Algorithms and Applications". *Springer-Verlag New York Inc.*, 2008.

[25] `http://data.geocomm.com`