

Indexing Land Surface for Efficient kNN Query

Cyrus Shahabi

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
shahabi@usc.edu

Lu-An Tang

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
lvantang@usc.edu

Songhua Xing

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
sxing@usc.edu

ABSTRACT

The class of k Nearest Neighbor (kNN) queries is frequently used in geospatial applications. Many studies focus on processing kNN in Euclidean and road network spaces. Meanwhile, with the recent advances in remote sensory devices that can acquire detailed elevation data, the new geospatial applications heavily operate on this third dimension, i.e., land surface. Hence, for the field of databases to stay relevant, it should be able to efficiently process spatial queries given this constrained third dimension. However, online processing of the surface k Nearest Neighbor (skNN) queries is quite challenging due to the huge size of land surface models which renders any accurate distance computation on the surface extremely slow. In this paper, for the first time, we propose an index structure on land surface that enables exact and fast responses to skNN queries. Two complementary indexing schemes, namely Tight Surface Index (TSI) and Loose Surface Index (LSI), are constructed and stored collectively on a single novel data structure called Surface Index R-tree (SIR-tree). With those indexes, we can process skNN query efficiently by localizing the search and minimizing the invocation of the costly surface distance computation and hence incurring low I/O and computation costs. Our algorithm does not need to know the value of k a priori and can incrementally expand the search region using SIR-tree and report the query result progressively. It also reports the exact shortest surface paths to the query results. We show through experiments with real world data sets that our algorithm has better performance than the competitors in both efficiency and accuracy.

1. INTRODUCTION

Two recent technological advances are giving rise to a new class of exciting applications marrying the interactivity and engagements of computer games with the information richness and practicality of geospatial information systems. These include advances in: 1) sensory devices that can acquire imagery, elevation and other types of data of Earth surface at a very high resolution from distance, and 2) computer graphics software and hardware that can quickly render complicated high resolution 3D graphics on the computer display. This new class of *geo-realistic games* has a wide range of applications in the areas of decision making, entertainment and realistic simulation systems for

Permission to make digital or hard copies of portions 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. Copyright for components of this work owned by others than VLDB Endowment must be honored.

Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

training to name a few.

One fundamental technical challenge that prevents the realization of these applications is in fact in the area of data management. The challenge is that real-world large geospatial datasets residing on disk drives need to be queried and accessed as if they are synthetically rendered data in memory. Unfortunately, most disk-based data structures are designed to expedite the rendering of this geo-realistic data (e.g., Google Earth) rather than its querying and access. This paper is our initial attempt to design an index structure on a subset of this data set to expedite one class of spatial queries.

The k Nearest Neighbor (kNN) query is an important class of queries in geospatial databases. The kNN query searches for *sites* (e.g., hotels, gas stations or other points of interest) with minimum distance with reference to one or more query points [1]. Various studies focused on the problems of kNN and its variations such as reverse kNN [2], continuous kNN [3] and network kNN [7]. Most of those queries are carried out in 2-D geospatial database and not take the elevation information into consideration. However, incorporating land terrain information becomes critical to make the aforementioned applications as realistic as possible. Thus the new generation of geo-databases should be able to support the kNN query on 3-D data of land surface. In this so called *surface k Nearest Neighbor* queries, the objects can only move on the terrain surfaces and the actual distance depends on the shortest surface path between two points. Besides the applications mentioned in [8] [13], skNN query is also useful for tourism applications, evacuation planning and military operations. Furthermore, different variations of skNN such as the continuous skNN query where *sites* (e.g., vehicles) are moving and the visible skNN query enable many other interesting applications.

Example 1: Figure 1 shows a sample application for skNN query. In a virtual environment based on the elevation data of Yosemite National Park, a trekker wants to know how far the nearest camp site is. Note that, p_1 is the nearest site in 3-D Euclidean distance; however, since its surface path need to go across the valley, it is actually farther than site p_2 on the other side.



Figure 1. The Surface k Nearest Neighbor Query

Unlike other types of kNN queries, there are few studies on skNN queries [8] [13], largely due to the following three challenges:

- The huge size of surface model: A surface is usually represented as the TIN (Triangular Irregular Network) model, a mesh generated from the sampled ground positions with 3-D coordinates. A region of $10\text{km} \times 10\text{km}$ using a 10m sampling interval has about 1 million triangles. The standard TEC (Topographic Engineering Center) LIDAR data sets are at 1m resolution (resulting in 100,000,000 triangles to represent a 100 km^2 area), with some even higher resolution data collected for special purposes, e.g., for military applications.
- The high computation complexity to find the shortest surface path: The state-of-the-art algorithm [10] needs $O(N^2)$ time to compute the shortest surface path between only a single pair of points, where N is the size of the surface model. In [13], it is reported that this operation will take tens of minutes on a modern PC machine. Note that the surface distance computation is fundamental and frequent in the skNN query.
- There is no existing surface index structure to expedite spatial querying. Classic indexes for traditional kNN query, such as R-tree and Voronoi diagrams are difficult to apply directly due to the complexity of the land surface. The previous studies on surface data are mostly focused on the compression, rendering and visualization of this data [9].

Even though the land surface model of TIN can be considered as a network with triangles' sides and vertices as edges and nodes of the network, respectively, the proposed techniques for network kNN are not suitable for skNN either. First, the road network distance on TIN is only an approximation of the actual surface distance, thus the shortest surface path may pass through the faces of the triangles and not along the edges. Hence, the network kNN algorithms may not provide exact answers to skNN queries. Second, TIN is much larger and more complex than a traditional road network. Many of the existing algorithms cannot scale to such huge and complex datasets.

We are only aware of one study addressing skNN [8] [13], in which the authors employed a filter and refinement strategy to process skNN queries. The idea is to answer skNN queries by ranking the sites based on the range of their upper and lower bound distances to the query point. The main drawback is that the reported k nearest neighbors is not accurate and may be out of order. Our experiments (see Section 6.3.2) show that even if we ignore the out-of-sequence results, still the accuracy of this approach drops to below 50% for $k > 5$. Moreover, this approach cannot report the actual shortest surface path between the query point and the result set and require knowing the value of k a priori.

In this paper, we propose two novel spatial indexes for skNN query, namely *Tight Surface Index* (TSI) and *Loose Surface Index* (LSI). These indexing schemes partition the land surface in the neighborhoods around the sites similar to the concept of Voronoi diagram. Since the computation of Voronoi cells on land surface is more challenging due to the irregularity of surface paths. We create two approximations of these cells, the *tight* and *loose* cells. Any point inside the tight cell of a site is immediately reported as its nearest neighbor and those outside the loose cell of the site are others' nearest neighbors. For those points in the buffer between the two cells (which is a small area relative to the entire space),

we need to incur the complexity of computing the actual surface distance. However, the cells localize this computation to save significantly on both the I/O and CPU processing. Our techniques can handle obstacles by modeling obstacles as part of TIN and setting their elevation to be infinite.

The remainder of the paper is organized as follows. Section 2 discusses some related research. In Section 3, we define the problem and provide some background materials. Section 4 describes the two surface indexes, their spatial properties, and the data structure for their implementation. Section 5 provides the algorithms to process skNN query and Section 6 reports on the results of extensive experiments. Finally, Section 7 summarizes the paper and discusses the future work.

2. RELATED WORK

2.1 kNN Query Processing Techniques

Depending on the assumed distance metric, kNN queries can be divided into the following two categories.

With the first category, the distance computation is performed in constant time (e.g., the computation of Euclidean distance using the point's coordinates). The proposed algorithms and techniques mainly focus on pruning the site set P to identify kNN. Roussopoulos et al. [1] propose an R-tree based kNN algorithm that prunes in a branch-and-bound manner; Korn et al. [2] study the influence set (reverse nearest neighbors) to the sites; Tao et al. focus on continuous kNN with moving query points [3] and aggregate kNN on multiple query points [4].

With the second category, the computation of the distance metrics is computationally complex due to the underlying constrained space. One important class is the road network kNN where the space is constrained due to an underlying graph structure. Shahabi et al. [5] introduce an embedding technique to transfer the road network to a constraint-free high dimensional space. Papadias et al. [6] introduce techniques for network kNN queries by integrating network and Euclidean information and capturing pragmatic constraints. Kolahdouzan et al. [7] propose a Voronoi-based algorithm, VN^3 , for spatial network databases.

The only previous approach to the skNN problem is discussed in [8], with an extended version in [13]. Deng et al. propose a distance range ranking method for the skNN query on the multi-resolution terrain model. Based on a terrain visualization model called *Direct Meshes* [9], the authors design lower and upper bound metrics for surface distance. Instead of computing the surface distance directly, the sites are selected out and ranked by their distance range between lower and upper bounds. Even though this is an efficient approach to process skNN query, it only provides approximate answers. That is, the algorithm neither guarantees the correctness of the computed distance nor provides the actual shortest surface path between the query point and target sites. In our experiments (Section 6), we compare the accuracy of our approach with that of [8].

2.2 Computational Geometry Studies

The shortest path computation on polyhedron surfaces has been extensively studied by the computational geometry community. The idea is to first unfold all the faces of the polyhedron in one plane, and then the surface path will be a straight line in that plane

connecting the two points. However, there are several combinatorial ways to unfold the polyhedron depending on the unfolding order of each face. The shortest path is the straight line with minimum distance connecting two points in all possible unfoldings. The state-of-the-art algorithm, Chen-Han algorithm (CH) [10], uses a tree to store and traverse all the possibilities. This algorithm costs $O(N^2)$ time and $\Theta(N)$ space, where N is the number of polyhedron faces. There are several improvements on Chen-Han algorithm: Kanai et al. [11] propose an approximate algorithm for surface distance computation. Deng et al. [12] design an expansion based algorithm to find the shortest path on surface. However, Deng et al. [13] point out, even the best algorithm still takes several minutes for the distance computation between a single pair of points on the complex land surface. Note that in skNN queries, the surface distance computation is a fundamental and frequent operation. In our experiments, we compare the response time of our approach with that of [10].

3. BACKGROUND

Before explaining the approach for surface kNN query, let us first formally define the metrics used in our approach and the problem.

3.1 Metrics and Problem Definition

TIN (Triangular Irregular Network) is the most popular model to construct a land surface. TIN is generated from DEM (Digital Elevation Model) of sampled ground positions at regularly spaced intervals. Based on these samples, TIN constructs the surface triangles by connecting the points as non-overlapping triangles. Note that, the sites and the query points may be located on the face of triangles. For computational convenience, in such cases, the system will connect the site/point with the three vertices of that triangle, making it as one vertex of the TIN model.

Given the real-world application scenarios, we formalize the definition of surface distance as follows:

Definition 1 (Surface Distance): Let T be the surface model, the surface distance between two points p and q , $D_S(p, q)$, is the length of the shortest path connecting the two points on T .

Definition 2 (Euclidean Distance): The 3-D Euclidean distance between two points p and q , $D_E(p, q)$, is the length of the straight line connecting the two points.

Since the land surface is a constrained 3-D space, $D_E(p, q)$ is the lower bound for $D_S(p, q)$. Another related distance metric which is frequently used as the upper bound is the network distance over the triangles.

Definition 3 (Network Distance): Let T be the surface model, the network distance between two points p and q , $D_N(p, q)$, is the length of the shortest path between the two points on the graph of T .

Note that the network distance is computed by accumulating the length of the triangle sides that exist between two points. Hence, it is always larger than the surface distance since it does not pass over the face of the triangles. It is important to note that the network path is not always an appropriate approximation of the actual surface path because even though the actual distance values are close, the actual paths may be quite different. To illustrate, consider Figure 2 that shows the three types of distances in an

area within the Yosemite National Park. The value of network distance is close to the surface distance but they are on two different paths.

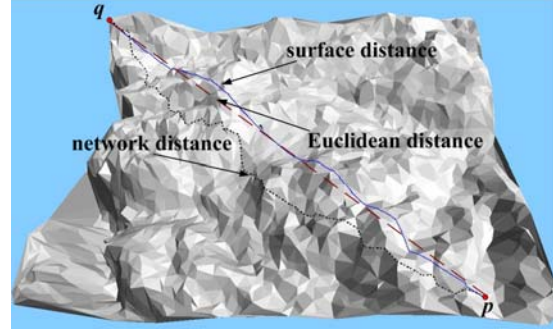


Figure 2. Surface, Network and Euclidean Distances

Now we give a formal description of the skNN problem:

Problem Definition: Let T be the surface model and P be the site set, given a query point q , an skNN query identifies the k nearest neighbor sites to q based on the metric of surface distance on T .

Evaluating an skNN query consists of two steps: 1) Computing the shortest surface path from q to a set of candidate sites in P ; 2) Identifying the k nearest neighbors. Without loss of generality, in the following sections we first address the nearest neighbor query ($k = 1$), then later in Section 5 we extend our approach to the kNN ($k > 1$) problem.

3.2 Shortest Surface Path Computation

Chen-Han (CH) algorithm [10] is the state-of-the-art algorithm and widely used in computing the shortest path on polyhedron surfaces. We explain it here using an example.

Example 2: Figure 3 shows the process of computing the surface distance between A and B on a tetrahedron. The triangular face 1, 2, 3 and 4 are unfolded to a plane with different unfolding orders (Case 1--3). The surface distance is the length of the shortest straight lines connecting A and B. The algorithm will compare the unfolding results and output Case 2 as the shortest surface path.

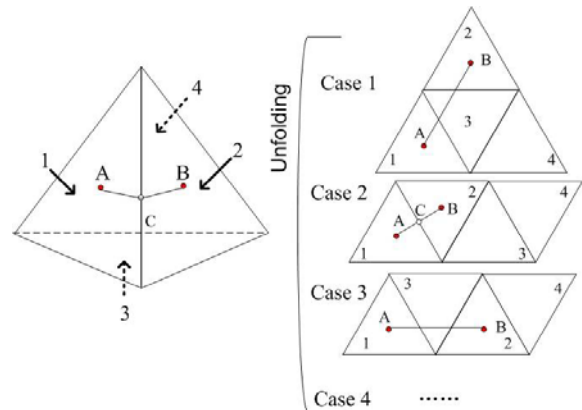


Figure 3. The Unfolding Process of Chen-Han Algorithm

The main drawback of CH algorithm is the expensive costs in time and space. It computes the surface distance between one pair of points in $O(N^2)$ time, where N is the number of the surface

faces. Even worse, the algorithm needs $\Theta(N)$ space to store the unfolding results. Hence, it is too costly to apply it directly on the fly to compute surface distances in skNN queries.

3.3 Voronoi Diagram

A common method to partition space is by using Voronoi diagrams. A Voronoi diagram [14] divides a space into disjoint cells according to the sites. For any query point that falls inside a cell, the nearest neighbor is the cell's generator (the site corresponding to that cell). Voronoi diagrams have been widely used to process continuous kNN [3] and network kNN queries [7].

Various approaches have been proposed to compute Voronoi diagrams in 2-D Euclidean spaces [15]. The common boundary edge between two neighboring Voronoi cells is the perpendicular bisector of the line connecting the two corresponding sites. Voronoi diagrams have also been used to partition network spaces; Kolahdouzan et al. employ network Voronoi diagrams to support network kNN queries [7]. However, to the best of our knowledge, none of the proposed algorithms can be extended to compute Voronoi diagrams on complex land surfaces. The challenges are:

- 1) The dividing edges of the Voronoi cells on land surface are not only determined by the locations of the sites, but also influenced by the surface itself. Thus simply finding the perpendicular bisector line between two sites, which is the foundation of many existing algorithms, is no longer sufficient or even necessary.
- 2) The naïve way of constructing Voronoi diagram on polyhedron [10] has to compute the shortest surface path to all the vertices. This algorithm needs $O(mN^3)$ time and $\Theta(mN)$ space, where m is the total number of sites and N is the number of polyhedron's vertices. This cost may be acceptable in the case of a polyhedron where m and N are relatively small, but in skNN query cases there are usually thousands of sites and millions of TIN vertices, rendering the surface distance computation impossible.

Since constructing the Voronoi diagram on land surface is not feasible. Thus, we will propose two novel indexes to help processing skNN query.

4. INDEXING THE LAND SURFACE

This section presents our two proposed spatial index schemes: *Tight Surface Index* (TSI) and *Loose Surface Index* (LSI) to process skNN queries. We also analyze their spatial properties.

4.1 Tight Surface Index

To explain the core idea of Tight Surface Index, recall that the surface distance has a lower bound of Euclidean distance and an upper bound of network distance. Hence, if the upper bound distance between the query point q and a site p , $D_N(p, q)$, is less than any of the lower bound distances from q to any other site, then we can safely conclude that p is the nearest neighbor of q . In such cases, the locations of query points are in an area close to p . Tight Surface Index (TSI) is devised to specify such a neighborhood area for each and every site p .

Definition 4 (Tight Cell): Let T be a surface model, tight cell $TC(p_i)$ is a polygon area around site p_i , defined by $TC(p_i) = \{q: q \in T \text{ and } D_N(p_i, q) < D_E(p_j, q) (\forall p_j \in P, p_j \neq p_i)\}$, p_i is called the generator of $TC(p_i)$.

Property 1: For any query point $q \in TC(p_i)$, the nearest neighbor of q in surface distance is p_i .

Proof: $q \in TC(p_i) \Rightarrow D_N(p_i, q) < D_E(p_j, q) (\forall p_j \in P, p_j \neq p_i)$.

Since $D_S(p_i, q) \leq D_N(p_i, q)$ and $D_E(p_j, q) \leq D_S(p_j, q)$ then

$D_S(p_i, q) < D_S(p_j, q)$. Hence, p_i is the nearest neighbor of q in surface distance. \square

We can now formally define the Tight Surface Index as follows.

Definition 5 (TSI): Let P be the site set $P = \{p_1, p_2, \dots, p_m\}$, and T be a surface model, Tight Surface Index (TSI) is a set of tight cells generated from P , defined by $TSI(P) = \{TC(p_1), TC(p_2), \dots, TC(p_m)\}$.

Note that with TSI, the cells do not cover the entire area; there exist some *unclassified areas* between the cells. Figure 4 shows the TSI on the land surface of Yosemite. For the purpose of clarity, only seven of them with their generators are drawn in the figure.

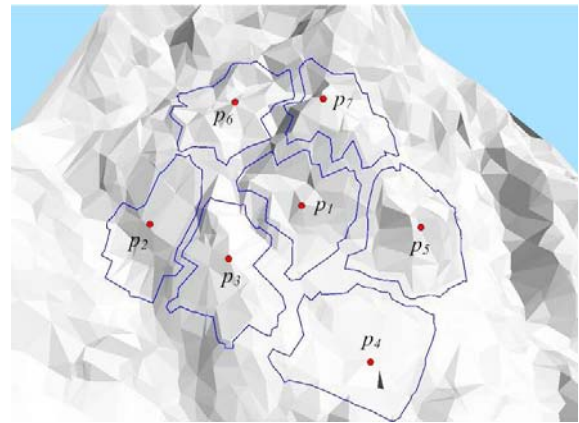


Figure 4. Tight Surface Index

TSI can be used as a filter step for skNN query processing in the following way: If the query point q is inside a certain tight cell, the system will immediately report the generator as q 's nearest neighbor; but if q locates in the unclassified area (outside any tight cell), then we have to carry out more computations.

4.2 Loose Surface Index

TSI defines a tight area around site p in which any point is guaranteed to have p as its nearest neighbor in surface distance. Opposite to TSI, Loose Surface Index (LSI) defines a loose area around p outside which any point is guaranteed **not** to have p as its nearest neighbor in surface distance. Consequently, any query point q that is outside a site p 's tight cell but inside its loose cell, require further processing. Any other query point can be immediately decided upon whether it has p as nearest neighbor or not using LSI and TSI, respectively. We now formally define LSI and prove its properties. We also formally show the relationships between the two indexes.

Definition 6 (Loose Cell): Let T be a surface model, loose cell $LC(p_i)$ is a polygon area around site p_i , defined by $LC(p_i) = \{q: q \in T \text{ and } D_E(p_i, q) < D_N(p_j, q) (\forall p_j \in P, p_j \neq p_i)\}$, p_i is called the generator of $LC(p_i)$.

It is easy to see that the loose cell of each site fully contains its tight cell, because every point in a tight cell also satisfies the conditions of a loose cell.

Property 2: Site p_i is guaranteed not to be the nearest neighbor of q if q is outside $LC(p_i)$.

Proof (by contradiction): If q is outside $LC(p_i)$, then $\exists p_j \in P (p_j \neq p_i)$ such that $D_E(p_i, q) > D_N(p_j, q)$.

Thus $D_S(p_i, q) \geq D_E(p_i, q) > D_N(p_j, q) \geq D_S(p_j, q)$. That is, the surface distance from q to p_i is larger than its distance to p_j . Hence p_i cannot be the nearest neighbor of q . \square

Definition 7 (LSI): Let P be a site set $P = \{p_1, p_2, \dots, p_m\}$ and T be a surface model, Loose Surface Index (LSI) is a set of loose cells generated from P , defined by $LSI(P) = \{LC(p_1), LC(p_2), \dots, LC(p_m)\}$.

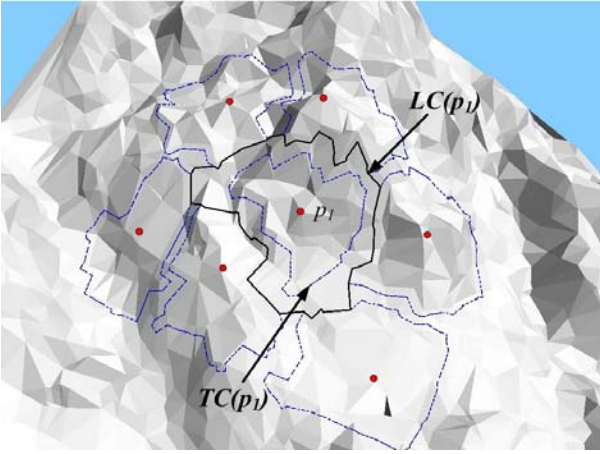


Figure 5. Loose Surface Index

Figure 5 illustrates an example of Loose Surface Index. Unlike TSI, the loose cells cover the entire map and have some overlap areas. Moreover, the overlap areas of LSI are precisely the same as the unclassified areas in TSI. This is not a coincidence. In fact, one of the most important contributions of this paper is that we prove that loose cells and tight cells actually share the same edges.

Property 3: Given that TSI and LSI are generated for the same site set P , the tight and loose cells have common edges; more specifically, all the tight cell's edges are also the edges of loose cells.

Proof: To prove the above property, we only need to prove that for any edge e of the tight cell $TC(p_i)$, e is also the edge of another loose cell $LC(p_j)$.

By definition of a tight cell $TC(p_i)$, for any point q on e , $\exists p_j (p_j \neq p_i)$ that $D_N(p_i, q) = D_E(p_j, q)$.

Meanwhile, by definition of the loose cell $LC(p_j)$, q is also a point on its edge, since $D_E(p_j, q) = D_N(p_i, q)$. Thus we can safely conclude that e is also an edge of $LC(p_j)$. \square

Since the tight cells have unclassified areas in the middle, and the loose cells overlap with each other, it is a little tricky to determine their neighbors. Fortunately, since they have common edges we can define the neighbors in the following manner.

Definition 8 (Neighbor): Given a site p , the neighbors of p are defined as $NL(p) = \{p_i | TC(p_i) \text{ and } LC(p) \text{ have common edges}\}$.

LSI will be used as a complementary index for TSI: if the query point q is in the unclassified area of TSI, then we will investigate the loose cells that contain q , they are the candidates as q 's nearest neighbor. Hence, we only need to compute the surface distance between q and those candidates. This is much more efficient as compared to the naïve way of computing surface distances from q to all the sites.

Another usage of LSI is to improve the efficiency of surface distance computation. Chen-Han algorithm needs to unfold the triangles of the entire surface area to find the shortest path. However, assuming q 's nearest neighbor is p , if we show that not only q cannot be outside $LC(p)$ but also the shortest surface path from q to p is fully contained in $LC(p)$, then we can limit the unfolding only within $LC(p)$.

Property 4: If p_i is the nearest neighbor of q , then the shortest surface path from q to p_i is inside the loose cell $LC(p_i)$.

Proof (by contradiction): Suppose that a portion of the shortest surface path from q to p falls outside $LC(p)$, then suppose a point m is on that outside portion of the path (Figure 6).

Since m is outside $LC(p)$, then $\exists p_j \in P (p_j \neq p_i)$ that $D_E(p_i, m) > D_N(p_j, m)$ and hence $D_S(p_i, m) > D_S(p_j, m)$.

Then $D_S(p_i, q) = D_S(p_i, m) + D_S(m, q) > D_S(p_j, m) + D_S(m, q) = D_S(p_j, q)$, which contradict the fact that p_i is the nearest neighbor of q . \square

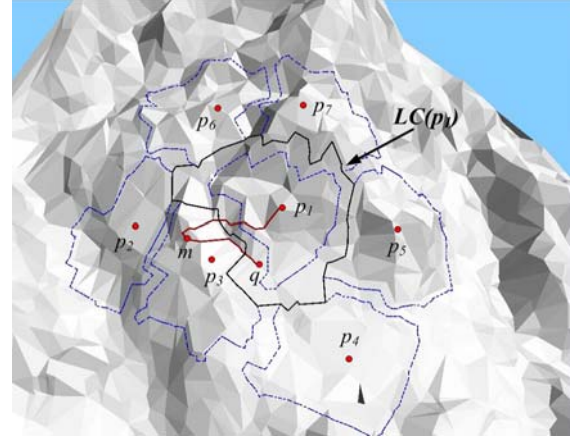


Figure 6. Shortest Surface Path and Loose Cell

Therefore, Property 4 indicates that we only need to unfold the triangles inside the candidate loose cells containing q to find the nearest neighbor. Thus the high computation cost is reduced since the area of loose cell is on average $1/m$ of the original map. Subsequently, the time complexity of surface distance computation is reduced from $O(N^2)$ to $O((N/m)^2)$ and the space complexity is reduced to $\Theta(N/m)$, increasing the algorithm's efficiency for large number of sites (m is the total number of sites, N is the size of surface model).

We need to point out that property 4 only holds for the first nearest neighbor. For kNN with $k > 1$, the shortest surface path from q to the generator will not only stay inside its own loose cell. We will discuss this issue later in Section 5.

4.3 Index Construction

Even though the cells of TSI and LSI serve almost the same purpose as those of a Voronoi diagram on 2-D Euclidean space, their construction is much more challenging. In this section, we discuss our proposed process to construct the cells of TSI and LSI. We will mainly focus on the construction of tight cells as the loose cells construction follows a similar process. We start by describing a naïve approach and then we explain our fast index construction algorithm.

4.3.1 Naïve Index Construction

A naïve approach to generate the tight cells is as follows. First, for each surface triangle of the TIN mesh, we need to make a decision to identify the tight cell to which it belongs.

The simple case is when all the three vertices of the triangle belong to the same site p_i . In this case, we immediately consider the triangle to be a member of the p_i 's cell (e.g., Triangle 1 in Figure 7). In contrast, if none of the vertices belongs to p_i , the triangle is excluded (e.g., Triangle 2 in Figure 7). A more complicated case is that only one or two vertices of the triangle belong to site p_i (e.g., Triangle 3, 5 in Figure 7). Then we need to compute the exact location of the two or three *transition points* on the edges of such triangle. A *transition point* is defined as a point on an edge whose network distance to its nearest site p_N is equal to the minimal of its Euclidean distances to all the other sites (e.g., p_i in Figure 7 that $D_N(p_b, p_i) = D_E(p_b, p_2)$). We could term this *transition point* as p_N 's *transition point*. Therefore, a *transition point* breaks an edge into two pieces, one belonging to a certain tight cell and the other one belonging to the unclassified area. Connecting all the transition points across surface triangles would eventually generate the border of the tight cells. For each vertex of the triangle, we need to compute both the Euclidean and the network distance to all of the m sites in order to decide to which site's cell the vertex belongs. Since network distance computation costs $O(N \log N)$ time, the total time complexity to this naïve approach is $O(mN^2 \log N)$, where m is the total number of sites and N is the size of the surface model. Hence, in the following section we introduce a novel approach that would result in a much lower time complexity for most practical scenarios.

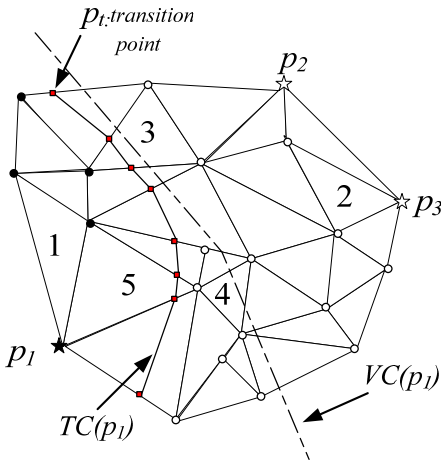


Figure 7. Tight Surface Index Construction

4.3.2 Fast Index Construction

We can significantly reduce the complexity of the naïve algorithm, if we could successfully identify the triangles that overlap with the edges of the tight cells. Then there would be no need to examine the vertices of any other triangle. Unfortunately, there is no way to know these *edge triangles* in advance. However, we can start with a rough prediction of those triangles by first drawing the Voronoi diagram¹ of the sites in the Euclidean space. Those triangles that overlap with the edges of this Voronoi diagram, termed *candidate triangles*, are potential *edge triangles*. The intuition of using Voronoi diagram is based on the following observation on the relationship between one site's tight cell and its Voronoi cell.

Property 5: Given any site p_i , its tight cell $TC(p_i)$ is inside its Voronoi cell $VC(p_i)$.

Proof: To prove the above property, we only need to prove that all p_i 's *transition points* are inside $VC(p_i)$.

By the definition of *transition point*, for any of p_i 's *transition points* p_t , $D_N(p_b, p_i) \leq D_E(p_b, p_i) (\forall p_i \in P, p_i \neq p_i)$. Meanwhile, $D_E(p_b, p_i) \leq D_N(p_b, p_i)$. Therefore, $D_E(p_b, p_i) \leq D_E(p_b, p_i) (\forall p_i \in P, p_i \neq p_i)$. By the definition of Voronoi diagram, p_t must be inside $VC(p_i)$. \square

The final step is to investigate each of the *candidate triangles*, using the same process we discussed for the naïve approach, to decide whether this triangle is in fact an *edge triangle* or not. Towards this end, we need to compute the Euclidean and network distance between each of the vertices of the candidate *edge triangle* to all the m sites in order to decide the tight cell each vertex belongs to. There are two possible cases: 1) only one or two vertices of the triangle belong to site p_i (e.g., Triangle 3 in Figure 7), then it is an *edge triangle*, the *transition points* are computed and the TSI edge is generated. 2) None of the vertices belong to site p_i (Triangle 4 in Figure 7); therefore, this triangle is not an *edge triangle* and we need to find another *candidate triangle*. Fortunately the area of tight cell is tighter than Voronoi cell, we only need to examine whether the *candidate triangle*'s closer neighbors are *edge triangles* or not (e.g., Triangle 5 in Figure 7). Once an *edge triangle* is determined, the *transition point* on the corresponding edge can be computed by solving quadratic equations. In the worst case, for each generator, we only need to investigate the triangles inside the corresponding Voronoi cell as a result of Property 5. Hence, the total time to generate TSI is brought down to $O(N^2 \log N)$.

Algorithm 1 depicts the algorithm of TSI construction approach. A stack is maintained to filter and refine the *candidate triangles*. First we compute the Voronoi diagram and push the triangles that overlap with the Voronoi edges into the candidate stack (Line 2-5); then examine each triangle in the stack to identify the *edge triangles*. The algorithms have to search the triangle's neighbors if the one is not an *edge triangle* (Line 7-10); else we can generate the TSI edges on the triangles (Line 11-15).

¹ Note that this Voronoi diagram is built in 3-D Euclidean space. A simpler but rougher alternative is the Voronoi diagram on the projected 2-D Euclidean space. However, the following Property 5 may not hold for 2-D Voronoi diagram resulting in more *candidate triangles*.

In the case of LSI construction, when the *candidate triangle* is found not to be an *edge triangle*, the search area is expanded to farther neighbors, because loose cell is larger than its corresponding Voronoi cell. The algorithm of LSI construction is similar to Algorithm 1; however, the farther neighbor instead of the closer one is pushed into the stack (Line 9).

```

Algorithm 1: Fast TSI Construction (Site set  $P$ , Surface  $T$ )
1  initialize stack  $C$ , result set  $TSI$ ;
2  compute the Voronoi diagram  $V(P)$  in Euclidean space;
3  for each surface triangle  $f$  overlapping with  $V(P)$  edges
4    push ( $C, f$ );
5  end for
6  while ( $C$  is not empty)
7     $t \leftarrow \text{pop}(C)$ ;
8    if ( $t$  is not edge triangle) then
9      push the closer neighbor triangles into  $C$ ;
10   end if
11   else //  $t$  is edge triangle
12     compute the edge  $e$  on  $t$ ;
13     add  $e$  to  $TSI$ ;
14   end else
15   end while
16   return  $TSI$ ;

```

Figure 8. TSI Construction Algorithm

4.4 Surface Index R-tree

TSI and LSI are ways to partition the land surface space. To be practical for skNN query processing as indexes, they need to be stored in an appropriate data structure.

Many data structures are proposed and employed to facilitate kNN processing. *R-tree* [16] is the most prominent index structure widely used in geospatial data management area. In this section, we show how we use the R-tree to store the TSI/LSI indexes. We refer to this index structure as Surface Index R-tree (SIR-tree).

As shown in Figure 9, SIR-tree has the basic structure of an R-tree that is generated on the site set P . The difference lies in that the SIR-tree's leaf nodes not only store the sites inside the corresponding MBR, but also record the pointers to the vertex lists of tight and loose cells. Since TSI and LSI are both generated from the same site set, they share one SIR-tree. To facilitate kNN query processing, we also store the site's neighbor list in SIR-tree. Similar to Voronoi diagram in 2-D Euclidean space, where the average number of neighbors is bounded by six [14], the tight/loose cell has a constant average number of neighbors as shown in Figure 19. Therefore, the overhead for maintaining such neighbor list is constant per site.

The SIR-tree index will be constructed only once before query processing. Suppose that we have built an R-tree on the site set, and the TSI and LSI are also constructed. To incorporate the two index schemes into the R-tree, we need to traverse the R-tree and add the pointers to the cells and neighbor list in each leaf node.

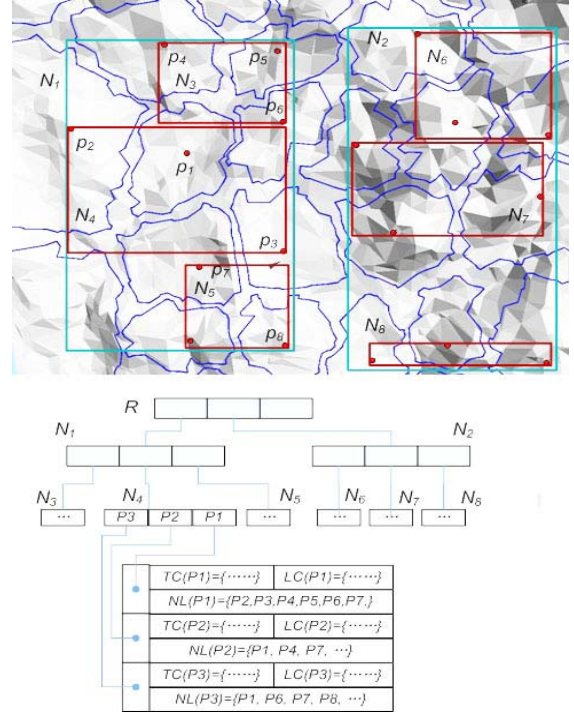


Figure 9. Surface Indexes R-tree

To dynamically maintain SIR-tree, we have implemented the insert, delete and update operations. A good property of LSI and TSI is the local updatability: when a site changes, only its neighbor's cells need updating. However, TSI has unclassified areas in the middle and LSI allows the cells to overlap with each other, which make it difficult to determine the cell edges. Fortunately, Property 3 shows that, each tight cell shares the same edges with neighbor site's loose cells and vice-versa. Hence, we can update LSI with the new tight cell and TSI with the new loose cell. The algorithm for SIR-tree insertion is shown in Figure 10 (the delete and update operations are similar and hence omitted).

```

Algorithm 2: SIR-tree Insertion (SIR-tree  $I$ , site  $p$ )
1.  locate  $p$  in  $I$ , find out the loose cell  $LC(r)$  containing  $p$ ;
2.   $p.\text{neighbor} \leftarrow LC(r)$ 's neighbor;
3.  compute  $TC(p)$  and  $LC(p)$ ;
4.  for each site  $p_i$  in  $p.\text{neighbor}$ 
5.    update  $LC(p_i)$ 's edges according to  $TC(p)$ ;
6.    update  $TC(p_i)$ 's edges according to  $LC(p)$ ;
7.  insert  $p$  into  $I$ ;
8.  return  $I$ ;

```

Figure 10. SIR-tree Insertion Algorithm

Example 3: Figures 11 shows the insertion process for TSI: when a new site p_i is inserted into SIR-tree, first we compute the loose cell $LC(p_i)$. Since the neighbor tight cells share the same edge with $LC(p_i)$, we can easily adjust them according to $LC(p_i)$. Finally we insert $TC(p_i)$ to the index.

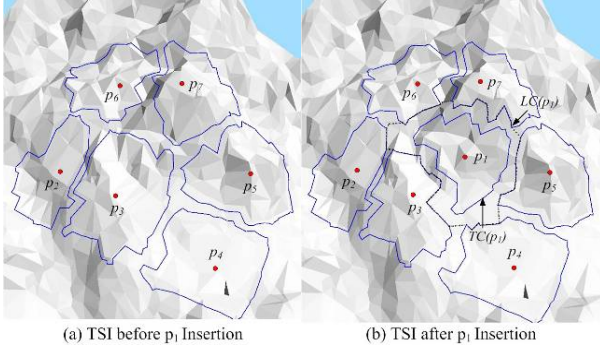


Figure 11. SIR-tree Insertion Example

5. QUERY PROCESSING

In this section, we explain our algorithms for skNN query processing using SIR-tree. First, we discuss our method for the nearest neighbor query processing ($k = 1$), and then we extend it to the kNN ($k > 1$) case.

5.1 Nearest Neighbor Query

Given a query point q , to process the surface nearest neighbor query, we have to traverse in the SIR-tree to locate q with the TSI/LSI indexes. First we carry out a depth-first search from the SIR-tree root to the last node N_l that contains q . Then we get all the sites inside N_l , check out the tight/loose cells containing q . If none of the cells contains q , we have to go to N_l 's parent node N_p and continue search. As soon as we find a certain tight cell containing q , the cell's generator is returned as the nearest neighbor. If q is in the unclassified area of TSI, we will take the sites whose loose cells contain q as *candidate NNs*. As a consequence of Property 3, we know that if we find a certain loose cell p containing q in such case, all the other *candidate NNs* are in the neighbor cells of $LC(p)$. Subsequently, each *candidate NN's* loose cell is unfolded to compute the shortest surface path and the one with the minimum distance is selected as the nearest neighbor. The algorithm to process the nearest neighbor query is depicted in Figure 12.

Proposition 1: Let N be the size of the surface model T and m be the total number of sites, Algorithm 3's time complexity is $O((N/m)^2 + \log(m))$.

Proof: The major time consuming steps are the searching step (Line 2-15) and the surface distance computation step (Line 17). The algorithm needs $O(\log(m))$ time to locate q in the SIR-tree and $O((N/m)^2)$ time to compute the surface distance (if necessary) by using Chen-Han's unfolding technique. Thus the total time complexity is $O((N/m)^2 + \log(m))$. \square

However our experiments show that in most cases, we can locate q quickly in a certain tight cell and output the generator as the result without any surface distance computation.

Example 4: In Figure 13, by a depth-first search, query point q_1 is located in the node N_1 . Then the algorithm have to check whether q_1 is contained in the tight/loose cells whose generators are inside N_1 . (e.g. $p_1, p_2 \dots p_8$). When the algorithm find q_1 is in the tight cell $TC(p_2)$, thus p_2 is immediately returned as the query result and the algorithm terminates. For query point q_2 , first we search in N_4 but could not find any tight cell containing q_2 , only a loose cell $LC(p_3)$ contains q_2 . Hence it is in an unclassified area. We check p_3 's

neighbors and select p_6 as another *candidate NN*. Consequently, we have to compute the surface distance of *candidate NNs* for further processing. $LC(p_6)$ and $LC(p_3)$ are unfolded to compute the shortest surface paths. Finally, $D_S(p_3, q)$ is identified as the shortest distance and p_3 is returned as q_2 's nearest neighbor.

Algorithm 3: NN Query (SIR-tree I , point q , surface T)

1. initialize candidate set C , stack *nodelist*;
2. for the node N in I that contains q // depth-first search
3. push(N , *nodelist*)
4. end for
5. while *nodelist* is not empty
6. $N \leftarrow \text{Pop}(\textit{nodelist})$;
7. for each site p in N
8. if $TC(p)$ contains q then return p ;
9. if $TC(p)$ doesn't contain q but $LC(p)$ contains q
10. add p to C ;
11. check p 's neighbors, add the one containing q to C ;
12. break;
13. end if
14. end for
15. end while
16. for each p in C // the candidate NNs
17. unfold $LC(p)$ in T to compute $D_S(p, q)$;
18. end for
19. select p with the minimum $D_S(p, q)$ in C ;
20. return p ;

Figure 12. NN Query Algorithm

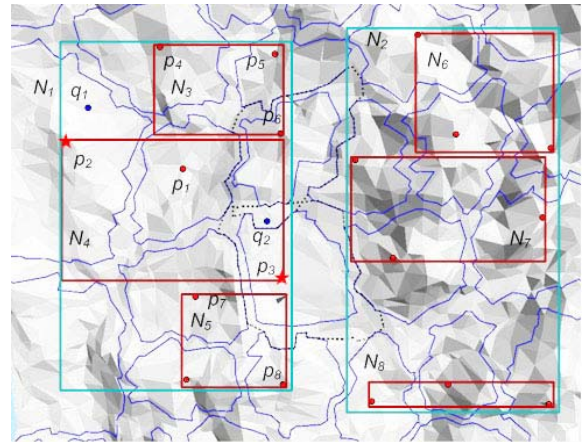


Figure 13. Nearest Neighbor Query Example

5.2 kNN Query

Our algorithm can be extended to process the kNN query in an expanding manner. We start from the first NN and expand the search area by including the neighbor loose cells of the first nearest neighbor. Once we identify the second nearest neighbor in these cells, we continue by investigating the neighbor loose cells of the second nearest neighbor to find the third nearest neighbor and so on. Therefore, we can report the results incrementally.

In order for this approach to work, we need to prove that the next nearest site at each step of the process is the generator of one of the neighbor cells of those nearest neighbors found so far.

Property 6: The next nearest site is the generator of one of the neighbors of the NNs found so far.

Proof (by induction): First, we prove the basis, that is, we need to prove that the second nearest site is the neighbor of the first NN.

This proposition can be proved by contradiction. In Figure 14, suppose p_3 is the first NN of q , and the second NN is p_7 , which is not the neighbor of p_3 .

Since p_3 and p_7 are not neighbors, a point m on the shortest surface path between q and p_7 can be found that is outside both $LC(p_3)$ and $LC(p_7)$. And by Property 2, p_7 is not a candidate nearest neighbor to m since m is not in $LC(p_7)$.

Thus, there exists another site, p_1 for instance, which is closer to m than p_7 . That is, $D_S(m, p_7) > D_S(m, p_1)$, then $D_S(q, p_7) = D_S(q, m) + D_S(m, p_7) > D_S(q, m) + D_S(m, p_1) = D_S(q, p_1)$. Therefore p_7 is farther to q than both p_3 and p_1 , which contradicts the assumption that p_7 is the second nearest neighbor.

Now we prove the inductive step, that is, suppose the inductive hypothesis holds for $k-1$, we prove that it also holds true for k . Let $G = \{p_1, p_2, \dots, p_{k-1}\}$ be the $k-1$ nearest neighbors of the query point q , we prove that the k -th nearest neighbor p_k is among the neighbor cells of G .

The proof is similar to the above prove by contradiction. Consider a k -th NN point p_k which is not the neighbor of G . Then on the surface path from q to p_k , there exists a point m which belongs neither to $LC(p_i)$ nor to $LC(p_1) \dots LC(p_{k-1})$. Thus, p_k is not the nearest neighbor of m . Suppose the nearest neighbor of m is p_i , where $p_i \neq p_1, p_2, \dots, p_k$. Hence, $D_S(q, p_k) = D_S(q, m) + D_S(m, p_k) > D_S(q, m) + D_S(m, p_i) = D_S(q, p_i)$. Therefore, Thus p_k is farther to q than p_1, p_2, \dots, p_{k-1} and p_i , which contradicts the assumption that p_k is the k -th nearest neighbor. \square

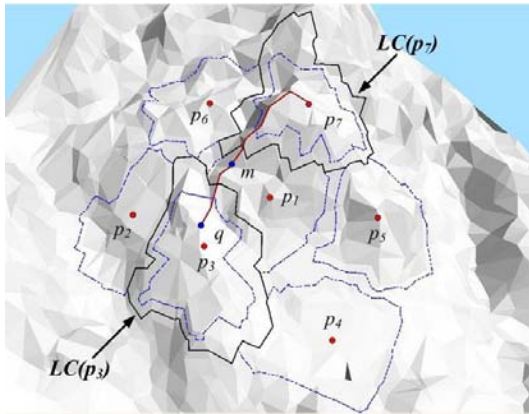


Figure 14. Surface Path to Second NN

As a consequence of Property 6, we also know that the shortest surface path from q to the k -th NN p_k will lie in the area of

$$LC(G) \cup LC(p_k) = LC(p_1) \cup LC(p_2) \cup \dots \cup LC(p_k).$$

Now we are ready to explain our algorithm to process kNN queries: we start by adding all the neighbors of the current NN set G to a candidate set. For each candidate p_i , we unfold the area of $LC(p_i)$ and $LC(G)$ to compute surface distance, then we select the

one with minimum distance as the next NN result. The algorithm is depicted in Figure 15. Note that, in this algorithm, we use a heap to store the candidates. Hence, even without a pre-specified value for k , the algorithm can still report the results incrementally.

Algorithm 4: kNN Query (SIR-tree I , point q , surface T)

1. $p \leftarrow$ Nearest Neighbor Query(I, q, T); //Algorithm 3
2. add p to kNN set G ;
3. initialize minimum heap H ;
4. while($G.size < k$)
5. for each neighbor site p_i of G ;
6. unfold $LC(G) \cup LC(p_i)$ to compute surface distance;
7. add p_i to H ;
8. end for
9. $p \leftarrow$ deheap H ;
10. add p to G ;
11. end while;
12. return G ;

Figure 15. kNN Query Algorithm

Example 5: Figure 16 shows the process to query third nearest neighbor on the land surface of Yosemite National Park. p_1 and p_2 are the first and second nearest neighbor found so far. The neighbors of them: $p_3, p_4 \dots p_{10}$ are selected as the candidate sites. To compute the shortest surface path from query point q to a site p_3 , we only need to unfold the area of $LC(p_1) \cup LC(p_2) \cup LC(p_3)$. When all the candidates are pushed in the heap, the one with minimum surface distance will be selected out as the third nearest neighbor.

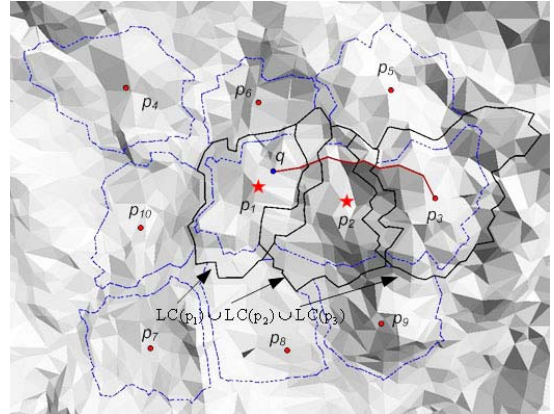


Figure 16. kNN Query Example

Proposition 2: Let N be the surface model size, m be the number of sites, and s be the average number of each site's neighbors, Algorithm 4's time complexity is $O(sk^3(N/m)^2 + \log(m))$.

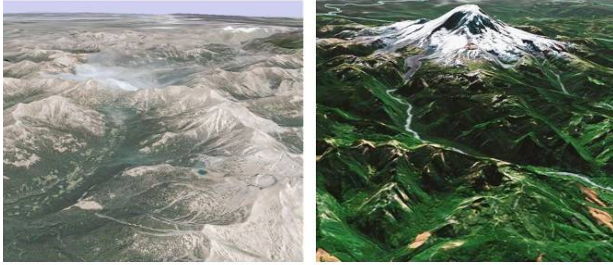
Proof: Calling Algorithm 3 has the time complexity of $O((N/m)^2 + \log(m))$. The major time consuming step of Algorithm 4 is the surface distances computation (Line 6). In the worst case, there are $s \times k$ candidates, with each candidate's surface distance computation requiring $O((kN/m)^2)$ time. The total time complexity is $O(sk^3(N/m)^2 + \log(m))$. Our experiments will show that s is a relatively small number less than 10 in real applications. \square

6. PERFORMANCE EVALUATION

6.1 Experimental Setup

As the skNN problem is motivated by real-world applications, two large scale land surface datasets in real world are used in the experiments. They are downloaded from USGS [17] and the same as the datasets used in previous studies [8].

1. Eagle Peak (EP) area at Wyoming State: The data set covers an area around $10.7\text{km} \times 14\text{km}$.
2. Bearhead (BH) area at Washington State: The area is almost the same size as EP, but has more mountains.



(a) Eagle Peak

(b) Bearhead

Figure 17. EP and BH Areas

Figure 17 shows an aircscape of the two areas from Google Earth. The sites are uniformly distributed on the surface with varying densities from 1% to 5%.

As mentioned in previous sections, three types of distance metrics are frequently used in the skNN query: Euclidean distance, surface distance and network distance. Euclidean distances are computed directly using the coordinates. We use the most recent implementation of CH algorithm [18], although localized because of our technique, to compute surface distances, and employ the classic Dijkstra algorithm [19] for network distance computation.

The experiments were conducted on a PC with Intel 6420 Dual CPU 2.13G Hz and 3.50 GB RAM. The operating system is Windows XP SP2. All the algorithms are implemented in Microsoft Visual Studio 2005.

6.2 Parameters of the Surface Index

As mentioned in Sections 4.1 and 5.1, if the query point q is located in a certain tight cell $TC(p)$, our algorithm can immediately report p as the nearest neighbor. Figure 18 shows the coverage ratio of the TSI with varying site densities on the two data sets. The average TSI coverage is about 75% of the whole area (for both EP and BH), which means that 3/4 NN queries could be processed immediately with no further computation.

Another parameter influencing the skNN query is the average number of neighbor cells for each site. In the Voronoi diagram of 2D Euclidean space, each site on average has only six neighbors [15]. This is a good property: when expanding the search area for k nearest neighbors, a relatively small number of neighbors means lower I/O cost. Unfortunately, this property does not hold in the cases of TSI and LSI, because the cells are no longer convex polygons. However, the statistics on real data sets (Figure 19) show that TSI/LSI's average number of neighbors is still less than 10 for uniformly distributed sites. Thus it incurs a relatively low I/O cost when we expand the search area for kNN queries.

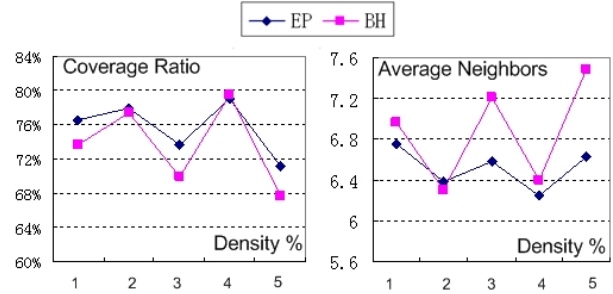


Figure 18. TSI Coverage Figure 19. Average Neighbors

6.3 Competing Approaches

The main contribution of our Surface Index (SI) approach is that it provides the exact k nearest neighbors with their corresponding surface paths to the query point in a relatively short time. The two competitions are the Range Ranking (RR) method [8] that provides approximate answers quickly and the original CH algorithm [10] that generates exact results but slowly. In this section, we use variations of both algorithms to show that our results are more accurate than the most accurate variation of RR and faster than the fastest variation of CH.

With RR method [8], a four-step filter and refinement framework is designed to process the skNN queries: 1) Perform an Euclidean kNN query in the 2D projection plane; 2) Rank the selected k sites based on the upper and lower bounds of surface distance; 3) Employ the k -th site's upper bound as the threshold, perform a range query using query point q as the center and the threshold as the radius; 4) Rank the candidates generated in step 3 to find skNN, using the same algorithm as in step 2.

In [8], the authors designed two upper/lower bounds (DMTM and MSDN) based on network/Euclidean distance. The two new metrics are generated from a multi-resolution terrain model called *Direct Mesh* (DM). In our variation of their method, termed RR for the rest of this section, we do not consider the DM structure and directly use the network distance as the upper bound and Euclidean distance as the lower bound at the highest resolution (10 meters DEM). This variation of RR results in the most accurate computation of surface kNN (since it is on the highest resolution), which is the main purpose of our comparison².

To compare our response time, we use a variation of the original CH algorithm which has similar filter and refinement framework as the RR method, but in steps 2 and 4 the precise surface distances are computed to rank the sites. This approach generates the same accurate results as ours but as we show has a much worse response time.

² Note that the response time of this variation may be worse than the response time of the original approach with multi-resolution. Unfortunately, we did not have access to the implementation of DM to compare the actual response time. From the numbers reported in [10], our results are comparable but we make no claim on the comparison between our response times and those of the original approach, and we only focus on the accuracy comparison given this most accurate variation, i.e., RR, on the highest resolution.

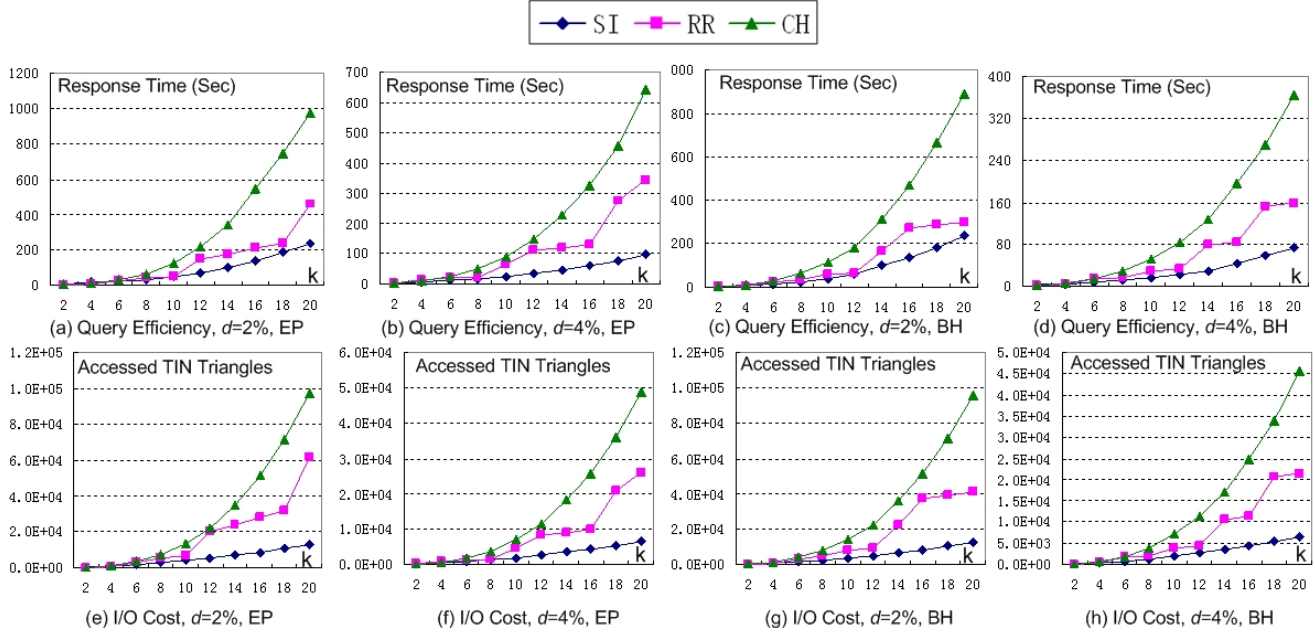


Figure 20. Query Efficiency, I/O Cost vs. Value of k

6.3.1 Efficiency

First we compare the performance of the three algorithms by varying the value of k from 2 to 20. The experiments are carried out for two site densities (2% and 4%) on the EP and BH data sets. We performed 500 skNN queries with uniformly distributed query points in each case. Figure 20 shows the average query processing time and number of I/O operations (as a function of number of TIN triangles retrieved).

The results indicate that CH algorithm performs the worst. The accessed TIN triangles and response time grow rapidly as the value of k increases, thus rendering this approach infeasible for frequent skNN queries on land surface. Our SI algorithm has an obvious predominance over other competitors from $k > 6$. In the case of $k = 20$, SI only requires half the time of RR, and 20% of CH. With its two spatial indexes, SI avoids accessing unnecessary surface triangles and yields the best performance in I/O. By limiting the number of candidates, SI reduces the number of costly distance computations; by partitioning the area to surface index cells, SI brings down the size of the areas in which it performs the distance computation and hence achieves a better performance.

We also study the effect of site density d , by fixing k at 10. Figure 21 shows the response time and number of I/O operations on the two data sets. In general, the cost of all three algorithms decreases as the site density increases. This is because a higher density leads to a smaller search area for a fixed value of k , thus less number of surface triangles accessed. The RR's filter threshold is strongly influenced by the land surface, especially when the sites are sparse. Our SI consistently outperforms the other two in both metrics, the average costs are about half of RR and 30% of CH.

6.3.2 Accuracy

Although the CH algorithm is the slowest option, it is an exact technique that guarantees the precision of surface distances and

query results. In this set of experiments, the generated kNN results ($k = 1$ to 10) of RR and SI are compared with that of CH. We carry out 500 skNN queries with uniformly distributed query points in the two data sets for 4% site density.

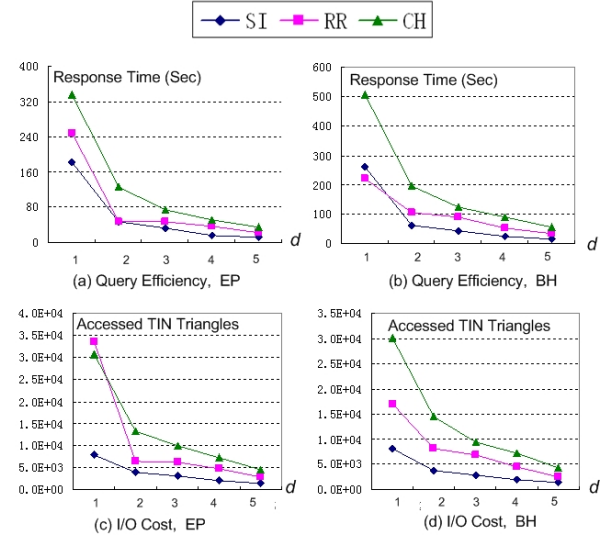


Figure 21. Query Efficiency, I/O cost vs. Density ($k=10$)

Since RR does not report the result incrementally, we use a relax standard of precision for RR in these experiments. We assume the query result of RR is precise if RR reports the same kNN sites as those of CH, even if the sites are in wrong order. And if the two sites cannot be identified by only checking the range between the upper and lower bound distances, we use the network distance as the approximate metric to rank them. Figure 22 shows the accuracy ratio.

As expected, the results show that SI is a precise algorithm with 100% accuracy. It not only finds the skNN correctly, but also lists

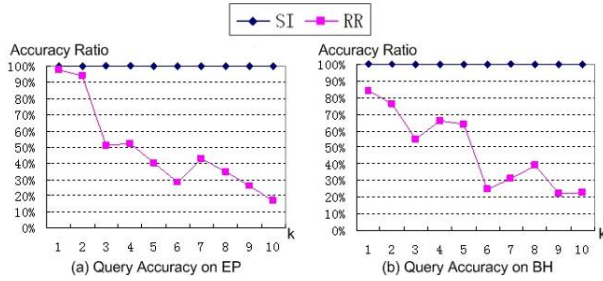


Figure 22. Accuracy Ratio of RR and SI

them in the right order and provides the shortest surface paths. On the other hand, even with the relax standard of precision; RR's accuracy is still not acceptable. The accuracy drops quickly as the value of k increases. With $k=10$, only 20% of the results are valid.

This drawback is due to the fundamental assumption of using network distance as the upper bound/approximation for surface distance. Figure 23 shows the average distance values in the skNN queries. Although the network distances are close to surface distance with less than 10% relative error, we observed that the average network distance of 8th site is even larger than the average surface distance of 10th site, which means that the RR fails to rank the sites because the surface distance of 10th site is in the range of 8th site. As the size of the query result grows (larger values of k), the probability to encounter sites with discrepancy between their actual surface distance and their threshold bounds increases, thus the algorithm is more likely to fail.

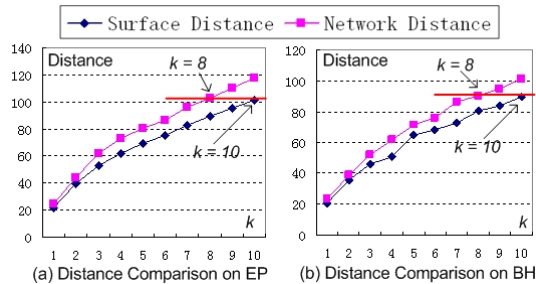


Figure 23. Surface and Network Distance Values

Some improved lower bounds are discussed in [8], however, no matter how tight the lower bound is, RR's accuracy cannot improve with the fixed upper bound as network distance. Furthermore, the computation cost also increases for computing more complex distance metrics. Finally, the precise distance and surface path still cannot be provided by RR.

7. CONCLUSION AND FUTURE WORK

In this paper, we introduced an efficient skNN processing method that provides: 1) exact answers to the queries, 2) the actual shortest surface paths and 3) incremental results. We also compared our approach in accuracy with the range ranking method [8] and in response time with the Chen-Han algorithm [10]. We showed that while our results are 100% accurate (vs. lower than 50% accuracy for the most accurate variation of [8] when $k > 5$) its response time is 4 to 5 times better than an efficient variation of [10] for most cases.

Our future plan includes further evaluation of the performance of our approach with synthetic datasets where we can vary the values of several parameters (e.g., coverage, average number of

neighbors). We also plan to study variations of the skNN query such as the continuous skNN query, visible skNN query and aggregate skNN query.

8. ACKNOWLEDGMENTS

This research has been funded in part by NSF grants IIS-0238560 (PECASE), IIS-0324955 (ITR), IIS-0534761 and IIS-0742811 (SGER), the NSF Center for Embedded Networked Sensing (CCR-0120778) and in part under JPL SURP program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

- [1] N. Roussopoulos, S. Kelley, and F. Vincent: Nearest neighbor queries. SIGMOD 1995.
- [2] F. Korn, S. Muthukrishnan: Influence Sets Based on Reverse Nearest Neighbor Queries. SIGMOD 2000.
- [3] Y. Tao, D. Papadias, Q. Shen: Continuous Nearest Neighbor Search. VLDB 2002.
- [4] D. Papadias, Y. Tao, K. Mouratidis, C.K. Hui: Aggregate nearest neighbor queries in spatial databases. TODS 2005.
- [5] C. Shahabi, M. R. Kolahdouzan, M. Sharifzadeh: A road network embedding technique for k-nearest neighbor search in moving object databases. ACM-GIS 2002.
- [6] D. Papadias, J. Zhang, N. Mamoulis, Y. Tao: Query Processing in Spatial Network Databases. VLDB 2003.
- [7] M. Kolahdouzan, C. Shahabi: Voronoi-based k nearest neighbor search for spatial network databases. VLDB 2004.
- [8] K. Deng, X. Zhou, H. T. Shen, K. Xu, X. Lin: Surface k-NN Query Processing. ICDE 2006.
- [9] K. Xu, X. Zhou, and X. Lin. Direct mesh: A multiresolution approach to terrain visualisation. ICDE 2004.
- [10] J. Chen and Y. Han. Shortest paths on a polyhedron. 6th ACM Symp. Comput. Geometry, pages 360–369, 1990.
- [11] T. Kanai and H. Suzuki. Approximate shortest path on polyhedral surface based on selective refinement of the discrete graph and its applications. Geometric Modeling and Processing, pages 241–250, 2000.
- [12] K. Deng and X. Zhou. Expansion-based algorithms for finding single pair shortest path on surface. W2GIS 2004.
- [13] 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. The VLDB Journal, June 2007.
- [14] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf: Computational Geometry: Algorithms and Applications. 2nd edition, 2000.
- [15] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu: Spatial Tessellations, Concepts and Applications of Voronoi Diagrams. 2nd edition, 2000.
- [16] A. Guttman: R-trees: a Dynamic Index Structure for Spatial Searching. SIGMOD 1984.
- [17] [Http://data.geocomm.com](http://data.geocomm.com)
- [18] B. Kaneva and J.O'Rourke: An implementation of Chen & Han's shortest paths algorithm. Proc. of 12th Canadian Conf. on Comput. Geom, pages 139–146, 2000.
- [19] E. W. Dijkstra: A note on two problems in connection with graphs. Numerische Mathematik, 1:269–271, 1959.