New Lower and Upper Bounds for Shortest Distance Queries on Terrains

Manohar Kaul†, Raymond Chi-Wing Wong†, Christian S. Jensen‡
† IIT Hyderabad ‡ The Hong Kong University of Science and Technology
† mkaul@iith.ac.in ‡ raywong@cse.ust.hk

ABSTRACT
The increasing availability of massive and accurate laser data enables the processing of spatial queries on terrains. As shortest-path computation, an integral element of query processing, is inherently expensive on terrains, a key approach to enabling efficient query processing is to reduce the need for exact shortest-path computation in query processing. We develop new lower and upper bounds on terrain shortest distances that are provably tighter than any existing bounds. Unlike existing bounds, the new bounds do not rely on the quality of the triangulation. We show how use of the new bounds speeds up query processing by reducing the need for exact distance computations. Speedups of of nearly an order of magnitude are demonstrated empirically for well-known spatial queries.

1. INTRODUCTION
With the increasing availability of accurate and massive laser data, 3D mesh representations of terrains are increasingly being used to simulate and study natural phenomena [1].

Applications
Queries on terrains are important in diverse applications, and the database community has recently studied the efficient support for, e.g., distance, kNN, and range queries on terrains [4, 6, 12, 19, 24, 29, 30]. We proceed to cover some applications.

In computer graphics and computer vision, the similarity between 3D shapes is used for object recognition [13] and image segmentation [26]. A main challenge is the mapping of 3D objects into compact canonical representations referred to as shape contexts or feature vectors, which can serve as search keys in image retrieval. A shape context can be defined as the distribution of selected points from a reference point. In order to compute these distances, surface paths play a vital role as they are invariant to transformations such as rotation and translation and are robust to slight deformations. Similarity is then defined by comparing the shape contexts of similar points on different 3D shapes. In graphics, the kNN classifier is heavily used to execute kNN queries, so that the query object’s class label can be set to the majority class label of its kNNs. In some settings, objects change shape dynamically, calling for frequent similarity re-computation [31]. In these settings, the performance of surface kNN computation is essential.

3D models of scientific data are becoming increasingly popular in areas such as biology, chemistry, anthropology and archeology to name a few. In neuroimaging, kNN queries are central to tumor classification using magnetic resonance imaging (MRI) images [2]. Here, getting the exact kNN set is important, and not getting it right can have serious consequences. In [MRI] [8] and diffusion MRIs that build very high resolution 3D models (many triangles) of organs as a function of time, it is also important to generate accurate kNN results as efficiently as possible. The above scenario rules out the use of approximation algorithms. Surface range queries also play an important role in neuroscience. For example, neuroscientists conduct spatial range queries on brain mesh simulations to study the neuron density and number of branches in an area [27].

In kinodynamics, motion planning problems on 3D surfaces is important. For example, collision avoidance [22], moving query objects must constantly keep track of their kNN sets of static or moving obstacles. Here, accurate kNN sets must be computed efficiently. Collision problems are also studied in particle physics, where the underlying terrain can be a 3D energy/force field. In robotics, robots must navigate 3D terrains with obstacles. Doing so, they may rely on an existing triangulation of their environment, or they may generate a triangulation on-the-fly [16, 20].

In military tactical analysis [15], computing shortest paths and spatial queries on terrains plays an important role. Factors such as the movement of troops and equipment over rough terrains can affect a military unit’s success in a battle exercise. Computing exact results of surface spatial queries is important.

In each of the above applications, surface spatial queries, including kNN and range queries, are prevalent [4, 6, 19, 24, 29, 30]. Every such spatial query uses as a distance metric the shortest surface distance to compute the distances between objects on the surface. Since computing the exact shortest surface distance is computationally expensive, we must instead find tighter upper and lower bounds for the shortest surface distance and use them to improve the efficiency of all surface spatial queries.

State-of-the-art Method
Several recent works in academia [6, 19, 24, 29, 30] extend well-known spatial queries from Euclidean space to terrains. Surface shortest path computation underlies all the resulting terrain queries and is identified to be extremely expensive and time-consuming. The best existing exact surface shortest path algorithm is proposed by Chen and Han [4] and has a time complexity of $O(N^2)$, where $N$ is the number of vertices in the triangulated surface. Spatial queries on terrains employ distance bounds in order to prune unnecessary objects before refinement, where expensive shortest path computations are performed.

Let the source and destination points on a terrain be denoted by $s$ and $t$, respectively. Let $\mathcal{E}$, $\Pi(s, t)$, and $\Pi_C(s, t)$ denote the Euclidean line-segment, shortest surface path, and shortest network
path between \( s \) and \( t \) on the terrain. Then, the corresponding path lengths can be denoted as \( |E'| \), \( |\Pi(s, t)| \), and \( |\Pi_C(s, t)| \). \( \Pi(s, t) \) is allowed to cut across the faces of the triangulation, while \( \Pi_C(s, t) \) is restricted to the edges of the faces. The motivation behind using distance bounds is that they are expensive to compute in comparison to the exact surface shortest path \( \Pi(s, t) \). Kaul et al. [12] show that a surface shortest path computation on a triangulation with 20K vertices, takes nearly 7.2 hours, while computing the shortest network path on the same triangulation takes only 0.04 seconds in comparison. However, the lower bound \( |E'| \) was shown to be nearly 9 times smaller than \( |\Pi(s, t)| \) [12], which was a very loose bound.

Kaul et al. [12] propose a tighter lower bound that captures the information about the surface of the underlying terrain. Both lower and upper bounds are derived from the shortest network distance \( |\Pi_C(s, t)| \). More specifically, they set the upper bound to \( |\Pi_C(s, t)| \) and the lower bound to \( \lambda \cdot |\Pi_C(s, t)| \), where \( \lambda = \min \{ \sin \theta_{\text{min}} , \sin \theta_{\text{min}} \cos \theta_{\text{min}} \} \) and \( \theta_{\text{min}} \) is the minimum interior angle of all faces in the terrain. The application of these bounds to \( knN \), reverse \( knN \), and range queries on surfaces, yields substantial improvements in execution times.

The quality of the state-of-the-art bounds proposed in [12] depends on the minimum interior angle \( \theta_{\text{min}} \), of all faces in the terrain. Observe that a larger \( \theta_{\text{min}} \) improves the tightness of the lower bound, while decreasing \( \theta_{\text{min}} \) results in a looser lower bound. Thus, for low values of \( \theta_{\text{min}} \), the lower bound quality deteriorates to be worse than even the Euclidean distance \( \ell \). Thus, the authors use a constrained Delaunay triangulation to ensure that \( \theta_{\text{min}} \) always exceeds 45°.

To illustrate, Figure 1 shows a triangulation with two degenerate triangles, \( \Delta AsB \) and \( \Delta AtB \), that share a common base and where \( s \) and \( t \) are the source and target vertices, respectively.

![Figure 1: Terrain with degenerate](image)

Figure 1: Terrain with degenerate

This example shows that in the presence of such degenerate triangles, for source vertex \( s \) and target vertex \( t \), the shortest surface distance \( |\Pi(s, t)| \) for path \( \Pi(s, t) \) (i.e., path \( st \) shown in dotted line in Figure 1) that is allowed to cut across a face is much smaller than the shortest network distance \( |\Pi_C(s, t)| \) (i.e., length of path \( sBt \) shown in bold lines in Figure 1), that is restricted to the edges of the faces. Thus, \( |\Pi_C(s, t)| \) is not a good approximation of \( |\Pi(s, t)| \) in this case.

**New Tighter Bound** Motivated by the above observations, we propose tighter upper and lower bounds for the surface shortest distance \( |\Pi(s, t)| \) that are independent of \( \theta_{\text{min}} \) and are thus unaffected by degenerate skinny triangles (smaller \( \theta_{\text{min}} \)) in a triangulation.

We adopt an approach where an approximation path is allowed to cut across faces, like \( \Pi(s, t) \) does, and this turns out to yield tighter approximations to \( \Pi(s, t) \). We term this new approximation a shortest surface face-crossing path \( \Pi_{FC}(s, t) \). A path \( \Pi_{FC}(s, t) \) cuts across a face, via cut vertices that are placed strategically on the edges of the face. We only introduce cut vertices on edges of faces being visited, and no cut vertices are introduced on edges of unvisited faces. This feature enables efficiency because there is no need to introduce useless cut vertices for unvisited faces.

Since \( \Pi(s, t) \) is the shortest surface path and the shortest surface face-crossing path \( \Pi_{FC}(s, t) \) can never exceed the network path \( \Pi_C(s, t) \), by definition, we know that \( |\Pi(s, t)| \leq |\Pi_{FC}(s, t)| \leq |\Pi_C(s, t)| \). Thus, \( |\Pi_{FC}(s, t)| \) forms our tighter upper bound. In our experiments, for a default setting of \( \theta_{\text{min}} \), \( |\Pi_{FC}(s, t)| \) is 0.92 times smaller than \( |\Pi_C(s, t)| \). Note that \( |\Pi_C(s, t)| \) is the best existing upper bound.

Our constant-factor lower bound is then computed as \( \lambda' \cdot |\Pi_{FC}(s, t)| \), where \( \lambda' \) is a constant fraction, whose value is at least 0.9. The significance of a constant-factor bound is that its quality has no dependence on the shape of a face (e.g., the minimum interior angle of the face). In our experiments, we study the effect on the tightness of our lower bound by varying the amount of cut-vertices introduced. This causes the values of \( \lambda' \) and \( |\Pi_{FC}(s, t)| \) to change accordingly. In our experiments, for a default setting of \( \theta_{\text{min}} \), \( \lambda' \cdot |\Pi_{FC}(s, t)| \) is 1.86 times longer than \( \lambda \cdot |\Pi_C(s, t)| \).

The new bounds are based not only on the vertices in the triangulated surface, but also on the introduced cut vertices. Therefore, they are more expensive to compute than the existing bounds [12]. However, the new bounds are tighter, and for \( knN \) and range queries, more objects are pruned, resulting in fewer expensive exact shortest distance computations.

In our experimental results, compared to the state-of-the-art bounds, algorithms using our tighter constant-factor bounds can prune up to 45% objects more, resulting in nearly 82 times faster execution time, for a surface \( knN \) query.

**Contributions and Organization** Our contributions can be outlined as follows. First, to the best of our knowledge, we are the first to propose tighter constant-factor lower and upper bounds for the surface shortest path that are always tighter than the best existing lower and upper bounds [12]. Second, we prove the tightness of the new bounds and show how they are always tighter than the existing bounds [12] for all possible values of \( \theta_{\text{min}} \). Third, we introduce a user-defined error parameter \( \epsilon \) in our experiments to enable a trade-off between the bound tightness and computation time. By studying the effect of the new bounds on existing surface spatial queries, we provide an understanding of how our new bounds affect the performance of such queries. Fourth, we present a comprehensive empirical study that offers insight into the accuracy, efficiency, and scalability properties of the new bounds. The study reveals speedups of nearly an order of magnitude for well-known terrain spatial queries.

The remainder of the paper is organized as follows. Section 2 formulates the problem. Section 3 describes our lower and upper bounds. Section 4 presents our algorithm. Section 5 covers the empirical study of the bounds and proposed algorithms. Section 6 describes related work. Finally, Section 7 concludes the paper.

### 2. PRELIMINARIES

The surface of a natural terrain is too irregular and is hence approximated by digital 3D models, which are based on a set of elevation values measured on the physical terrain. The most common such digital model used is the Triangulated Irregular Network (TIN) [7, 14, 23] which is displayed as a network of non-overlapping triangular faces. A TIN can therefore also be defined as a Delaunay graph \( G = (V, E) \) where \( V \) and \( E \) denote the set of vertices and edges, respectively, and the weight of each edge is the Euclidean distance between the two end-points of the edge. In the following, for brevity, we simply write “graph” for “Delaunay graph.” We also use the terms “surface”, “terrain” and “triangulation” interchangeably.

A TIN is typically represented as a triangulation \( T \). \( T \) must satisfy the following conditions. 1) Two triangles in \( T \) cannot in-
tersect, except at a vertex or a common edge shared by the two triangles, 2) A triangle in \( T \) cannot share an edge with more than one other triangle, and 3) a vertex on the boundary must be connected (via an edge) to exactly two other boundary vertices.

The direct line segment connecting any two points \( x \) and \( y \) on \( T \) is denoted by \([x, y]\). We define the shortest network path \( \Pi_G(s, t) \) from vertex \( s \) to \( t \) as the path consisting of only vertices and edges belonging in \( V \) and \( E \), respectively, and whose total length \(|\Pi_G(s, t)|\) is the shortest. Note that \( \Pi_G(s, t) \) cannot cut across the face of any triangle and is always restricted to the edges of the triangles.

Consider a sequence of points \( PS = \langle s = p_1, p_2, ..., p_n = t \rangle \) on the triangulation \( T \), such that every point \( p_i \in PS \) is located on an edge in \( E \). We can thus define a surface path between \( s \) and \( t \) to be composed of line segments \((p_1, p_2), (p_2, p_3), ..., (p_n−1, p_n)\). The total length of this surface path is then calculated by summing up the length of each individual line segment that composes the surface path. Then, the shortest surface path \( \Pi(s, t) \) from \( s \) to \( t \) is the surface path from \( s \) to \( t \) with the smallest length and the length of this path is denoted by \(|\Pi(s, t)|\). Figure 2 shows three triangles composing the terrain \( T \), namely \( \Delta v_1 v_2 v_3 \), \( \Delta v_1 v_2 v_3 \), and \( \Delta v_1 p_2 p_3 \). In this figure, the shortest surface path \( \Pi(s, t) \) from \( s \) to \( t \) is path \( \langle s = p_1, p_2, p_3, p_4 = t \rangle \). This path cuts across the faces of the triangles. The corresponding shortest network path \( \Pi_G(s, t) \) is path \( \langle s = p_1, v_1, v_3, p_4 = t \rangle \).

![Figure 3: Additional possible paths for \( \Pi_{FC}(s, t) \).](image1)

![Figure 4: \(|\{(a, o_{i+1})\}_n|_{min}\) near corner vertex \( A \).](image2)

From the literature, we have the following lemma.

**Lemma 1** ([16, 24, 30]). For any two vertices \( v \) and \( v' \) in \( V \), \(|\Pi(v, v')| \leq |\Pi_G(v, v')|\).

Further, we define \( V' \) to be the set of newly introduced vertices located between the end-points of each edge in \( E \). We refer to these new vertices as cut-vertices. We introduce an edge \((v_a, v_b)\) between cut-vertices \( v_a \) and \( v_b \) if they are located on adjacent edges of a face. This new set of edges is denoted as \( E' \). Given the definitions of \( V, V', E \) and \( E' \), we proceed to define a surface face-crossing path, which is represented by a sequence \( \langle s = v_1, v_2, ..., v_n = t \rangle \) where each \( v_i \) is a vertex in \( V \cup V' \), and each \((v_i, v_{i+1})\), except when \( v_1 \) and \( v_{n-1} \) are located on the same edge, is a newly introduced edge in \( E \cup E' \). Then, the shortest surface face-crossing path \( \Pi_{FC}(s, t) \) from \( s \) to \( t \) is the surface face-crossing path with the smallest length and the length of this path is denoted by \(|\Pi_{FC}(s, t)|\). The path \( \Pi_{FC}(s, t) \), unlike the network path \( \Pi_G(s, t) \), is not restricted to the edges of the face, but can also cut across the face of a triangle.

For better illustration, Figure 3 shows \( \Delta ABC \) with corner vertices \( A, B, C \in V \) and cut vertices \( \{A_1, A_2, B_1, B_2, C_1, C_2\} \in V' \). The dotted lines in \( \Delta ABC \) denote the newly introduced edges in \( E' \) (e.g. \( C_1, B_1 \)). A complete graph is a simple, undirected graph in which every pair of distinct vertices is connected by an edge. \( \Delta ABC \) can be viewed as a complete graph with the exception no new edges are introduced between any pair of vertices placed on the same edge, except the original edge between the corner vertices of that edge. For example, edges \((C_1, C_2)\) or \((B_1, C)\) are nonexistent. Thus, we can say that the additional edges in \( E' \) form a nearly-complete graph and allow additional possibilities for path \( \Pi_{FC}(s, t) \) to cut across the face of \( \Delta ABC \) via the cut-vertices introduced on the edges.

3. **Upper and Lower Bounds**

In this section, we propose an upper and lower bound for the shortest surface distance \(|\Pi(s, t)|\) between a start vertex \( s \) and destination vertex \( t \) on a triangulation \( T \).

Suppose that \( \Pi(s, t) \) is a sequence \(<p_1, p_2, ..., p_n>\), where \( p_i \) is a point along an edge in \( E \). Note that \( p_1 = s \) and \( p_n = t \). Each line connecting \( p_i \) and \( p_{i+1} \) is on face \( f_i \). Additionally, we define that each point \( p_i \) has its owner, denoted by \( o_i \), which is one of the closest corners in \( V \) or cut-vertices in \( V' \), on the face containing \( p_i \). Thus, line segment \((o_i, o_{i+1})\) can be viewed as the closest approximation to line segment \((p_i, p_{i+1})\) and \((o_i, o_{i+1})\) is termed as the owner segment of line segment \((p_i, p_{i+1})\). Figure 4(a) shows the closest vertices to \( p_i \) and \( p_{i+1} \) chosen as their owners, \( o_i \) and \( o_{i+1} \), respectively. Additionally, Figure 4(c) also illustrates how segment \((p_i, p_{i+1})\) is approximated by segment \((o_i, o_{i+1})\). Furthermore, we set \( o_1 = s \) and \( o_n = t \).

In the close vicinity of corner vertices, line segment \((p_i, p_{i+1})\) can have an infinitesimal length and hence cannot be approximated by a corresponding \((o_i, o_{i+1})\) line segment. Thus, we introduce a minimum length threshold \(|(o_i, o_{i+1})|_{min}\), below which any \((p_i, p_{i+1})\) has its corresponding \( o_i = o_{i+1} \), i.e., they are approximated by the corner vertex itself and \(|(o_i, o_{i+1})| = 0 \).

For illustration, in Figure 4, the corner vertex \( A \) and the length of line segment \((X, Y)\) denotes our \(|(o_i, o_{i+1})|_{min}\). Since \(|(p_i, p_{i+1})| \leq |(o_i, o_{i+1})|_{min}\), both \( p_i \) and \( p_{i+1} \) have the same owner, i.e., corner vertex \( A \) and \((o_i, o_{i+1}) = 0 \).

Furthermore, our cut-vertex placement strategy must ensure that every possible owner segment \((o_i, o_{i+1})\) satisfies our constant bound property. In order to do this, we must first ascertain the minimum possible length \(|(o_i, o_{i+1})|_{min}\) of any owner segment \((o_i, o_{i+1})\) in face \( f_i \).

Let \( \theta_m \) denote the minimum interior angle of a single face, not to be confused with the earlier \( \theta_{min} \), which is the minimum interior angle amongst all the faces belonging to a triangulation \( T \). Depending on the shape of the face, \(|(o_i, o_{i+1})|_{min}\) varies accordingly. For example, \(|(o_i, o_{i+1})|_{min}\) is shorter for faces with a smaller \( \theta_m \), than faces with larger \( \theta_m \).

In order to compute \(|(o_i, o_{i+1})|_{min}\) for a face \( f_i \), we focus on the corner vertex \( v_{cor} \) to which \( \theta_m \) belongs. Let \( l_{min} \) denote the length of the shortest edge in all the faces of \( T \). Then, an initial distance of \( l_{min}/2 \) from corner vertex \( v_{cor} \) is chosen on both the edges that are adjacent to \( v_{cor} \). Figure 4 shows an example where \( v_{cor} = A \) and \( |AX| ≥ |AY| = l_{min}/2 \), forming an isosceles triangle \( \Delta AXY \), where line segments \( AX \) and \( AY \) subtend the angle \( \theta_m \) of the triangle.

Applying the law of cosines, we have \(|(o_i, o_{i+1})|_{min} = l_{min}/2 \cdot \sqrt{2(1−\cos \theta_m)} \). We chose \( l_{min}/2 \cdot \sqrt{2(1−\cos \theta_m)} \) as an initial start value of \(|(o_i, o_{i+1})|_{min}\), to improve the time complexity of our algorithm.

After having computed the \(|(o_i, o_{i+1})|_{min}\) for a face, cut-vertices are placed on the edge with a gap \( \Delta \) that is at most \(|(o_i, o_{i+1})|_{min}\), where \( K \) is a constant subject to the constraint \( K ≥ 10 \). Setting \( K ≥ 10 \), we have \( ΔI ≤ |(o_i, o_{i+1})|_{min}\), which also ensures that our lower bound is always tighter than the bound proposed in [12] for all possible values of \( \theta_m \). Note that \( K \) can be replaced by a user defined variable, but this results in added
time-complexity, since many more cut-vertices get introduced on the edges.

More intuitively, for lower values of \( \theta_m \) (skinny faces), \(|(a_i, a_{i+1})|_{min}\) is automatically lowered and so is \( \Delta I \), thus many more cut-vertices and new edges connecting these cut-vertices are introduced on this face, improving how closely segment \((a_i, a_{i+1})\) approximates segment \((p_i, p_{i+1})\). This property allows our lower bound to achieve a constant-factor bound even for degenerate skinny faces.

Recall the definition of our shortest surface face-crossing path \( \Pi_F(s, t) \): \((a_1, a_2, ..., a_n)\). Each \( a_i \) is a vertex in \( V \cup V' \), on the terrain. Each line segment connecting \( a_i \) and \( a_{i+1} \) is on face \( f_i \). Additionally, for each \((a_i, a_{i+1})\) segment, we have a minimum length \(|(a_i, a_{i+1})|_{min}\), as shown in Figure 4. Thus, \( \Pi_F(s, t) \) is made up of line segments that either have (i) \( a_i = a_{i+1} \) and zero-length or (ii) \((a_i, a_{i+1})\) is an edge with non-zero length in \( E \cup E' \).

With these definitions, we derive the lower and upper distance bounds of the shortest surface distance. We begin by breaking down the \((s, t)\)-paths into line segments that must traverse over faces and provide distance bounds over these individual line segments. Later, we accumulate the results to provide the final bounds for the entire shortest surface distance \(|\Pi(s, t)|\).

**Lemma 2.** Let \((p_i, p_{i+1})\) denote the segment portion of the shortest surface path \( \Pi(s, t) \) on face \( f_i \) and \((a_i, a_{i+1})\) denote the segment portion of the shortest surface face-crossing path \( \Pi_F(s, t) \) on face \( f_i \). Then,

\[
\lambda \cdot |(a_i, a_{i+1})| \leq |(p_i, p_{i+1})|
\]

where \( \lambda \cdot |(a_i, a_{i+1})| \) is our lower bound of \(|(p_i, p_{i+1})|\), \( \lambda = \left(1 - \frac{1}{k}\right) \) and \( \Delta I \leq \frac{|(a_i, a_{i+1})|_{min}}{\lambda} \).

**Proof:** This proof can be broken down into several cases.

**Case 1:** \( a_i = a_{i+1} \) This case is illustrated in Figure 6(a). Here, \(|(a_i, a_{i+1})| = 0\). The inequality holds.

**Case 2:** Both \( p_i \) and \( p_{i+1} \) lie along the same edge \((v_1, v_2)\) and \( a_i \neq a_{i+1} \). This case is illustrated in Figure 6(b), where bold lines indicate the scenario depicted in this case, and the dotted lines indicate the actual path that should have been followed. This case never arises because it violates a fundamental observation in [25] which states that a surface shortest path must become a straight line segment when the faces crossed by the path are unfolded onto a plane. It is easy to note that when unfolding the faces over which \( P \) crosses, path \((p_{i-1}, p_i, p_{i+2})\) unfolds into a straight line, which is a shorter path than path \((p_{i-1}, p_i, p_{i+1}, p_{i+2})\).

**Case 3:** \( p_i \) and \( p_{i+1} \) do not lie on the same edge and \( a_i \neq a_{i+1} \). We begin proving this sub-case by making a claim as follows.

**Claim 1.** Let \((a_i, a_{i+1})\) denote the segment of path \( \Pi_F(s, t) \) that crosses face \( f_i \) and \((p_i, p_{i+1})\) denote the segment of path \( \Pi(s, t) \) that crosses face \( f_i \). Then,

\[
|\langle(0_i, 0_{i+1})⟩ - \Delta I| \leq |\langle(p_i, p_{i+1})⟩|
\]

**Proof:** Each edge of the face \( f_i \), having edge length \( l \), has \( k = \left\lfloor \frac{1}{2\pi} \right\rfloor \) cut-vertices the edge, dividing the edge into intervals of size \( \leq \Delta I \). Recall that the gap between a pair of vertices is denoted by the range of interval \( \Delta I \). We provide two figures for each sub-case, to illustrate how faces with smaller \( \theta_m \) have more finer intervals, i.e. lower \( \Delta I \), and hence result in more cut-vertices being placed on each edge. Also note that there are edges joining every pair of vertices placed on different edges of the face, thus forming a nearly complete graph (corner vertices placed on the same edge, do not have an edge between them). These new edges have been omitted from Figure 5 for clarity of images. Two sub-cases arise.

**Case A:** \( p_i \) is located on a corner vertex and \( p_{i+1} \) is located on a cut-vertex of the opposite edge. This case is illustrated in Figures 5(a) and (b). In this case, since \( p_i \) is on a corner-vertex, we choose the closest vertex \( a_i \) to be the same as \( p_i \), \( p_{i+1} \) is closest to \( a_{i+1} \) and the length of segment \((a_i+1, p_{i+1})\), is at most \( \frac{1}{2} \Delta I \). By the triangle inequality, we have that \(|\langle(0_i, 0_{i+1})⟩| \leq |\langle(p_i, p_{i+1})⟩| + \frac{1}{2} \Delta I \). Re-arranging this we get, \(|\langle(0_i, 0_{i+1})⟩ - \frac{1}{2} \Delta I| \leq |\langle(p_i, p_{i+1})⟩|\).

**Case B:** \( p_i \) and \( p_{i+1} \) are located on cut-vertices of adjacent edges. This case is illustrated in Figures 5(c) and (d).

When \( p_i \) and \( p_{i+1} \) are located on cut-vertices of adjacent edges, then their corresponding owners, i.e., \( a_i \) and \( a_{i+1} \), are cut-vertices that are closest to \( p_i \), and \( p_{i+1} \), respectively. In order to prove the inequality, we study the relation between edge \((p_i, p_{i+1})\) and edge \((a_i, a_{i+1})\). In Case B, we notice 3 more sub-cases as follows.

**Case (i) Edges** \((p_i, p_{i+1})\) and \((a_i, a_{i+1})\) cut across each other

This case is illustrated in Figure 7(a). \((p_i, p_{i+1})\) is indicated with a solid line, while \((a_i, a_{i+1})\) is shown with a dashed line. Let \( X \) denote the point at which both edges cross each other.
In $\Delta p_i o X$, using the triangle-inequality we have $|o_i, X| \leq |p_i, X| + |p_i, o_i|$. Applying the same in $\Delta X p_i o^{+1} p_i$, we arrive at the inequality: $|X, o_i| \leq |X, p_i| + |p_i, o_i|$. Adding, both inequalities, we have $|o_i, X| + |o_i, o_i| \leq |p_i, X| + |p_i, o_i|$. We observe that $|p_i, X| + |X, p_i| = |p_i, p_i|$. Also, we observe that $|p_i, o_i| \leq \Delta I/2$ and $|p_i, o_i| + \Delta I/2 \\]. These two observations into the previous inequality, we have $|o_i, o_i| \leq |p_i, p_i| + \Delta I$, which then rearranges to $|o_i, o_i| - \Delta I \leq |p_i, p_i|$. We also have a dotted line joining $o_i$ and $p_i$. The case of edges $(o_i, o_i)$ and $(p_i, p_i)$ are as follows: (i) Edges $(o_i, o_i)$ and $(p_i, p_i)$ do not cross over. 

This case is illustrated in Figure 5(b). $(p_i, p_i)$ is indicated with a solid line, while $(o_i, o_i)$ is shown with a dashed line. We also have a dotted line joining $o_i$ and $p_i$.

For the case of edges $(o_i, o_i)$ and $(p_i, p_i)$ do not cross over, we apply our inequality: $|o_i, o_i| \leq |p_i, p_i| + \Delta I$. This holds even when some edges are approximated as a single edge, i.e., Case 1 in Lemma 2 (Figure 6(a)).

Applying Lemma 2, we know that $|o_i, o_i| \leq |p_i, p_i|$, for each $i \in [1, k + 1]$. Combining the inequalities for each segment constituting paths $\Pi_{FC}(s, t)$ and $\Pi(s, t)$ we get, $\lambda \cdot \sum_{i=1}^{k} |p_i, o_i| \leq \sum_{i=1}^{k} |p_i, p_i|$. This further simplifies gives, $\lambda \cdot |\Pi_{FC}(s, t)| \leq |\Pi(s, t)|$, which proves the first inequality and completes our proof.

$\lambda \cdot |\Pi_{FC}(s, t)|$ corresponds to our new tighter constant-factor lower bound and $|\Pi_{FC}(s, t)|$ corresponds to our new tighter upper bound.

### Algorithm 1: Compute Distance Bound

**get_bounds**($G, s, t$)

1: for each $v \in G$ do
2: $D(v) \leftarrow \infty$
3: $D(s) \leftarrow 0$
4: Initialize event priority queue $Q$ with $(s, D(s))$
5: while $Q$ is not empty do
6: $(u, D(u)) \leftarrow$ Extract vertex of event with least $D(u)$
7: if $u == t$ then
8: Compute $(l, ub) = (\lambda \times D(u), D(u))$
9: output $(l, ub)$
10: $S \leftarrow S \cup \{u\}$
11: $d(u, t) \leftarrow$ Compute vertex $u$’s Euclidean distance to $t$
12: for each adjacent face $f$ of vertex $u$ do
13: Compute for edge $f$: $\theta_{min}(o_i, o_i), o_i, o_i)$ and interval $\Delta I$
14: if $u$ is a corner-vertex then
15: $e \leftarrow$ get edge opposite $u$ in face $f$
16: $L \leftarrow L \cup \{e\}$
17: else if $u$ is a cut-vertex then
18: for each edge $(v_a, v_b)$ in list $L$ do
19: $d(v_a, v_b) \leftarrow$ Get $(v_a, v_b)$’s Euclidean distance to $t$
20: if $d(v_a, v_b) \geq d(u, t)$ then
21: $j \leftarrow 1$
22: while $j \leq \frac{d(v_a, v_b)}{\Delta I} - 1$ do
23: Place cut-vertex $v_c$ at $\Delta I \cdot j$ from $v_a$ on $(v_a, v_b)$
24: if $|v_c, v_b| \geq \Delta I$ then
25: Add edge $(u, v_c)$
26: $D(v_c) \leftarrow D(u) + |v_c, v_b|$
27: Insert $(v_c, D(v_c))$ into $Q$
28: $j \leftarrow j + 1$
29: for each adjacent vertex $v$ of vertex $u$ do
30: $D(v) \leftarrow \min \{D(v), D(u) + |u, v|\}$
31: Update $(v, D(v))$ in $Q$
32: end for
33: end if
34: end for
35: end if
36: end if
37: end for
38: end while
39: end if
40: end for

### 4. Bound Computation

Our bound computation algorithm computes the lower and upper bound distances for the shortest surface distance $|\Pi(s, t)|$ between a source vertex $s$ and target vertex $t$ on the graph $G$. While Lemma 2 assumed knowledge of path $\Pi(s, t)$, our algorithm to compute the bounds cannot make such an assumption.

Some noteworthy properties of the shortest surface path $\Pi(s, t)$ [25] are as follows: 1) $\Pi(s, t)$ is composed of a series of straight line segments, where each line segment, touching the face, connects the points on adjacent edges of a single face. 2) $\Pi(s, t)$ traverses any given face at most once and does not bend on the interior of a face. 3) Two shortest surface paths $\Pi^{(1)}(s, t)$ and $\Pi^{(2)}(s, t)$, originating from a common source $s$, never cross each other.

Based on these aforementioned properties, we observe that surface paths originating from the source vertex $s$, start by cutting across faces adjacent to vertex $s$ and propagate outwards like a wavefront from the edges of the adjacent faces that are opposite to vertex $s$. Further, when a surface path emanates from an edge, it
Lemma 3. Given $G = (V, E)$, the source vertex $s$ and target vertex $t$, the lower bound proposed $\mathcal{L}_s$ in the state-of-the-art [12] is $\lambda' \cdot |\mathcal{P}_C(s, t)|$, where $|\mathcal{P}_C(s, t)|$ is the shortest surface network distance between $s$ and $t$, and $\lambda' = \min \left( \frac{\sin \theta_{\min}}{2}, \sin \theta_{\min} \cdot \cos \theta_{\min} \right)$, where $\theta_{\min}$ is the minimum interior angle of any face/triangle in $G$. We define our tighter lower bound $\mathcal{L}_s$ as $\lambda' \cdot |\mathcal{P}_C(s, t)|$, where $\lambda = |\mathcal{P}_F(s, t)|$ and $|\mathcal{P}_F(s, t)|$ is the shortest surface face-crossing path’s length as defined previously. We then have $\mathcal{L}_s \geq \mathcal{L}_o$.


From the definition of our upper bound $|\mathcal{P}_F(s, t)|$, it can be trivially shown that the inequality $|\mathcal{P}_F(s, t)| \leq |\mathcal{P}_C(s, t)|$ always holds and this shows that our new upper bound is always tighter than the previous upper bound of $|\mathcal{P}_C(s, t)|$ used in [12]. It is also important to note that our new bound $\mathcal{L}_s$ has no dependence on the interior angle of the triangles in $G$ and hence is a constant bound ratio. For $\theta_{\min} \leq 45^\circ$ it is easy to note that $\mathcal{L}_o$ can deteriorate quickly. E.g., for $\theta_{\min} = 20^\circ$, $\lambda' = 0.17$ and for $\theta_{\min} = 10^\circ$, $\lambda' = 0.085$.

4.2 Complexity Analysis

Before analyzing the complexity of the algorithm we first analyze the upper bound on the number of cut-vertices introduced per edge, the new edges connecting them and then the total number of cut-vertices added to the triangulation $T$. 

7 along with new edges $(5, 6)$, $(5, 4)$, and $(5, 7)$. The vertices are then picked from $Q$ in terms of their distances till we finally converge to the target vertex $4$. We illustrate the step-by-step working of the bound computation algorithm outlined in Algorithm 1 in the table shown in Figure 8(b). The columns in gray show: 1) The vertex $u$ extracted for processing by the algorithm, 2) the distance of the vertex from source vertex $s$, 3) the faces adjacent to $u$, and 4) the elements in $\mathcal{Q}$ prior to extraction of $u$. The boxes in white following the gray boxes, indicate the actions taken by the algorithm to add new cut-vertices and edges.

In our example, $\mathcal{P}_F(1, 4) = \{1, 5, 4\}$ and $|\mathcal{P}_F(1, 4)| = 0.85 + 1.3 = 2.15$. Thus, giving a lower bound of $0.9 \times 2.15 = 1.935$ and an upper bound of $2.15$. 

4.1 Correctness Proof

Theorem 2. The bound computation algorithm 1 takes as input the Delaunay Graph $G = (V, E)$, the source vertex $s$ and target vertex $t$. Computing the shortest $\mathcal{P}_F(s, t)$ after placement of cut-vertices and new edges, produces the lower and upper bounds of the shortest surface distance $\mathcal{P}(s, t)$ between $s$ and $t$.


We add another speedup optimization that checks whether an edge has already had cut-vertices placed on them from previous rounds, in which case no new cut-vertices are introduced to this edge.

Example 1. Consider the example terrain with two adjacent faces $f_1$ and $f_2$ illustrated in Figure 8. In this example, we set $\lambda = 0.9$, the source $s = 1$, and the target vertex as $t = 4$. Here, face $f_1$ has $\theta_{\min} = 45^\circ$ and $|\{o_i, o_{i+1}\}|_{\min} = 0.8$, while face $f_2$ has $\theta_{\min} = 60^\circ$ and $|\{o_i, o_{i+1}\}|_{\min} = 1$. For the purposes of this example only, we chose higher interval $\Delta$ values, in order to reduce the number of intermediate cut-vertices that our algorithm must place and consider when constructing the shortest path in this toy example. $\{1, 2, 3, 4\}$ is the set of corner vertices and $\{5, 6, 7\}$ are cut-vertices introduced by the algorithm. Also, note that there are no edges connecting cut-vertices that lie on the same edge or between cut-vertices and corner vertices of the same edge, e.g., no edge connects vertices 2 and 5, or vertices 3 and 5. Thus, the length of edges $(2, 3) = (2, 4) = (3, 4) = 1.5$. The grayed dashed line joining source and target vertices 1 and 4, is the shortest surface path $\mathcal{P}(s, t)$ and in our example, $|\mathcal{P}(s, t)| = 2.125$. The algorithm starts with the source vertex, i.e., $u = 1$. Since it is a corner vertex, it places a new cut-vertex 5 on edge $(2, 3)$. Next, cut-vertex 5 is picked from $Q$, which in turn places vertices 6 and 7 along with new edges $(5, 6), (5, 4)$, and $(5, 7)$. The vertices are then picked from $Q$ in terms of their distances till we finally converge to the target vertex 4. We illustrate the step-by-step working of the bound computation algorithm outlined in Algorithm 1 in the table shown in Figure 8(b). The columns in gray show: 1) The vertex $u$ extracted for processing by the algorithm, 2) the distance of the vertex from source vertex $s$, 3) the faces adjacent to $u$, and 4) the elements in $\mathcal{Q}$ prior to extraction of $u$. The boxes in white following the gray boxes, indicate the actions taken by the algorithm to add new cut-vertices and edges.

In our example, $\mathcal{P}_F(1, 4) = \{1, 5, 4\}$ and $|\mathcal{P}_F(1, 4)| = 0.85 + 1.3 = 2.15$. Thus, giving a lower bound of $0.9 \times 2.15 = 1.935$ and an upper bound of $2.15$.
Let $V'$ denote the cut-vertices introduced on the edges by Algorithm 1 and thus $|V'|$ is the total number of cut-vertices. The total number of vertices in the graph is thus $|V + V'|$.

**Analysis of cut-vertices:** Let $l_{min}$ denote the length of the shortest edge in all of $T$. Then $\theta_{min}$ is the minimum interior angle of $T$, opposite the length of $l_{min}$.

Recall the formula for $\Delta I$. We know that $\Delta I \geq \frac{1}{3} |(\alpha_i, \alpha_i+1)|$ for any given face $f$. Therefore, the lowest possible $\Delta I$ for any face $f \in T$ is $\Delta I_{min} = \frac{l_{min}}{\sqrt{2}} (1 - \cos \theta_{min})$. Note the use of $l_{min}$ and $\theta_{min}$ in the previous formula which are the global minimums in the triangulation $T$.

For a single face $f$. Then $\frac{\theta f}{\min}$ denote the length of the longest edge in face $f$. Then $v_c^\theta(f)$ represents the set of cut-vertices placed on edge $e$ of face $f$ and $|v_c^\theta(f)|$ is the total number of cut-vertices on edge $e$. The inequality below gives an upper bound on the number of cut-vertices on a single edge $e$, as $|v_c^\theta(f)| \leq \left(\frac{\theta f}{\Delta I_{min}}\right)$.

Then for a face $f$, we can say that the total cut-vertices are at most $3|v_c^\theta(f)|$.

According to Euler’s formula [3] on a Delaunay triangulation $T$, which contains a set of vertices $V$ and has $k$ points in $CH(V)$ ($CH$ is the convex hull of the vertices in $T$). Then, the number of faces $|F|$ in $T$ is given by the equation: $|F| = 2|V| - 2 - K$. Now, if we assume that our path will cover all faces in $T$, we get the upper bound: $|V'| \leq 3|F| - |v_c^\theta(f)|$.

In practice, since the $A^u$-like approach is followed, cut-vertices cannot be introduced on edges that have a Euclidean distance to $(\Delta I_{2})$. However, since the number of faces adjacent to a vertex $v$ is finite, we can assume that our path will cover all faces in $T$, we get the upper bound: $|V'| \leq 3|F| - |v_c^\theta(f)|$.

Additionally, we repeated all the experiments from EP on another popular dataset called *Bearhead (BH)* in Washington state, USA, which covers nearly the same region as EP but is more hilly. Furthermore, we studied the effects of the surface $kNN$ query on three additional terrain datasets from USA: 1) Blanca Peak (BP), Colorado, 2) Capitol Peak (CP), Colorado, and 3) Ko‘olau Forest Reserve (KF), Hawaii. These results can be found in our technical report [11].

In order to better evaluate the tradeoff between the tightness of our lower bound and the bound computation time, we introduce a new error parameter $\epsilon$. The relation between $\epsilon$ and $\lambda$ is: $\lambda = \left(1 - \frac{\epsilon}{K+\epsilon}\right)$ where $K \geq 10$ and $c \geq 0$. We set $K = 10$ and $c = 0$ to achieve $\lambda = 0.9$ at $\epsilon = 1$. For all the other $\lambda$ values we set $K = 10$ and $c = 5$ to achieve non-zero values of $\lambda$ that will always be better than the previous bounds.

### 5. EXPERIMENTS

In this section, we empirically study the performance of our proposed bounds, comparing it with the state-of-the-art bounds proposed by Kaul et al. [12].

#### 5.1 Experimental Setup

Experiments were conducted on the Eagle Peak (EP) dataset (http://data.geocomm.com/). This widely used dataset is from Wyoming, USA, covers an area of 10.7 x 14 km2, and has 1.3 million data points [6, 19, 24, 29, 30]. We generated sub-regions of sizes similar to [12] to compare our results. The experiments were conducted by varying several parameters to study the effect of the trade-offs among accuracy, efficiency, and memory usage. Table 1 shows the parameters with their default values shown in bold. Experiments were conducted with default parameter values unless explicitly stated.

Additionally, we repeated all the experiments from EP on another popular dataset called *Bearhead (BH)* in Washington state, USA, which covers nearly the same region as EP but is more hilly. Furthermore, we studied the effects of the surface $kNN$ query on three additional terrain datasets from USA: 1) Blanca Peak (BP), Colorado, 2) Capitol Peak (CP), Colorado, and 3) Ko‘olau Forest Reserve (KF), Hawaii. These results can be found in our technical report [11].

In order to better evaluate the tradeoff between the tightness of our lower bound and the bound computation time, we introduce a new error parameter $\epsilon$. The relation between $\epsilon$ and $\lambda$ is: $\lambda = \left(1 - \frac{\epsilon}{K+\epsilon}\right)$ where $K \geq 10$ and $c \geq 0$. We set $K = 10$ and $c = 0$ to achieve $\lambda = 0.9$ at $\epsilon = 1$. For all the other $\lambda$ values we set $K = 10$ and $c = 5$ to achieve non-zero values of $\lambda$ that will always be better than the previous bounds.

### Table 1: Parameter Settings

<table>
<thead>
<tr>
<th>Datasets</th>
<th>EP, BH, BP, CF, KF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precise Size (points)</td>
<td>20K, 200K, 400K, 800K, 1000K</td>
</tr>
<tr>
<td>User Error Parameter $\epsilon$</td>
<td>0.1, 0.5, 1, 5, 10</td>
</tr>
<tr>
<td>$\theta_{min}$</td>
<td>45°, 30°</td>
</tr>
<tr>
<td>$\Delta I_{2}$</td>
<td>2, 5, 10, 15, 20</td>
</tr>
</tbody>
</table>

In practice, as is evident from our experimental results, our lower bound computation algorithm performs much faster because it employs an $A^u$-like heuristic and also includes other speedup optimizations.

### 5.2 Results

In this section, we empirically study the performance of our proposed bounds, comparing it with the state-of-the-art bounds proposed by Kaul et al. [12].

#### 5.2.1 Accuracy

Experiments were conducted on the Eagle Peak (EP) dataset (http://data.geocomm.com/). This widely used dataset is from Wyoming, USA, covers an area of 10.7 x 14 km2, and has 1.3 million data points [6, 19, 24, 29, 30]. We generated sub-regions of sizes similar to [12] to compare our results. The experiments were conducted by varying several parameters to study the effect of the trade-offs among accuracy, efficiency, and memory usage. Table 1 shows the parameters with their default values shown in bold. Experiments were conducted with default parameter values unless explicitly stated.

Additionally, we repeated all the experiments from EP on another popular dataset called *Bearhead (BH)* in Washington state, USA, which covers nearly the same region as EP but is more hilly. Furthermore, we studied the effects of the surface $kNN$ query on three additional terrain datasets from USA: 1) Blanca Peak (BP), Colorado, 2) Capitol Peak (CP), Colorado, and 3) Ko‘olau Forest Reserve (KF), Hawaii. These results can be found in our technical report [11].

In order to better evaluate the tradeoff between the tightness of our lower bound and the bound computation time, we introduce a new error parameter $\epsilon$. The relation between $\epsilon$ and $\lambda$ is: $\lambda = \left(1 - \frac{\epsilon}{K+\epsilon}\right)$ where $K \geq 10$ and $c \geq 0$. We set $K = 10$ and $c = 0$ to achieve $\lambda = 0.9$ at $\epsilon = 1$. For all the other $\lambda$ values we set $K = 10$ and $c = 5$ to achieve non-zero values of $\lambda$ that will always be better than the previous bounds.

At $\epsilon = 1$, our bound with constant-factor $\lambda = 0.9$ is achieved. For higher values of $\epsilon$, a larger $|(\alpha_i, \alpha_i+1)|_{min}$ is chosen, which in turn increases the gap between cut-vertices, i.e. $\Delta I$. This increase results in fewer cut-vertices and edges being introduced on each face, so that our shortest surface-face-crossing path $\Pi FC(s, t)$ gets longer, but the constant-factor $\lambda$ drops, so that we get an overall looser lower bound. The opposite effect is achieved for values of $\epsilon < 1$.

The core algorithms were implemented in C and C++, and some auxiliary tasks were implemented in Perl. A terrain tool, developed by CMU, called Triangle (http://www.cs.cmu.edu/~quake/triangle.html), was employed for generating the TIN model with a minimum interior angle quality. In addition to generating the constrained Delaunay triangulation of the terrain with $\theta_{min} = 45°$, to

compare with [12], we also generated a synthetic dataset, with this tool, setting \( \theta_{\text{min}} \) to at least 30°. The Chen-and-Han implementation [10] was used to compute shortest surface paths. All experiments were carried out on a Fedora 18 Linux machine with an Intel Xeon E5 CPU (20MB cache, hyper-threading, 8 cores) and 32 GB internal memory. All experiments were conducted 100 times. Average values were reported in our final results. Following [12, 24], for each spatial query that required an initial query point, we generate a query location randomly and select 10% of the vertices in the TIN model randomly as objects.

In each of the experimental sections, we study the effect of introducing:

1. our tighter upper bound only,
2. our tighter lower bound only, and
3. both our tighter upper and lower bounds.

In Section 5.2, we study the effect of our tighter bounds for the shortest surface distance by varying the error, i.e., \( \epsilon \). In Section 5.3, we study the effect of our new bounds on the state-of-the-art surface \( k \)NN query algorithm [24]. In Sections 5.4 and 5.5, we show how the use of our newly proposed bounds improve the state-of-the-art algorithms for reverse surface NN query [30] and surface range query [12], respectively. Section 5.6 depicts the scalability of our bound computation and algorithms using our bounds. Finally, Section 5.7 summarizes our experimental findings.

### 5.2 Effect of our Bounds on Shortest Surface Path Query

In Figures 10(a) and 10(b), we denote \( D_s \) and \( D_e \) to be the surface shortest path distance and Euclidean distance, respectively. We denote the state-of-the-art upper bound, i.e., the network shortest path as \( \text{PrevUB} (=D_n) \). We denote our new tighter upper bound, i.e. the shortest surface face-crossing path as \( \text{OurUB} \). Additionally, \( \text{PrevLB} \) and \( \text{OurLB} \) denote the state-of-the-art and our new tighter lower bounds, respectively. Furthermore, we define the improvement ratio (which is always \( > 1 \)) for the lower bound as \( \frac{\text{OurLB}}{\text{PrevLB}} \) and for the upper bound as \( \frac{\text{PrevUB}}{\text{OurUB}} \).

**Using Our Tighter Upper Bound Only:** In Figure 10(a), we notice that for the least error setting, i.e., \( \epsilon = 0.25 \), our upper bound \( \text{OurUB} \) (9, 200 meters) is much closer to the surface shortest path \( D_s \) (9, 075 meters) than the state-of-the-art upper bound \( \text{PrevUB} (=D_n) \) (10, 150 meters). At the lowest error, we have the highest number of cut-vertices and edges introduced on the faces, so that the network distance gets much shorter than \( D_n \) (which is computed on the original edges of the faces) and is closer to \( D_s \). As \( \epsilon \rightarrow 0 \), \( \text{OurUB} \) approaches \( D_s \). When \( \epsilon \) is increased, we notice that \( \text{OurUB} \) loosens and eventually becomes the same as \( D_n \), due to fewer and fewer cut-vertices being introduced that finally lead to no cut-vertices being added to the edges.

Figure 10(b) displays a reductions in bound computation time as error increases. Increasing the error results in fewer cut-vertices and new edges being introduced by our algorithm, which in turn, results in faster network path computation as the graph has significantly fewer edges to process.

The objective of introducing tighter bounds is to compute a query, such as a range or a \( k \)NN query, as fast as possible.

In Figure 10(b), we notice that \( \text{PrevLB} \) beats \( \text{OurLB} \) in bound computation time, but results in much looser lower and upper bounds, which results in a substantial increase in query time (to be shown in Sections 5.3, 5.4 and 5.5) because the looser bounds result in too many unnecessary candidates being refined for which expensive shortest surface path computations must be carried out.

**Using Our Tighter Lower Bound Only:** Figure 10(a) shows that for the least error setting, i.e., \( \epsilon = 0.25 \), our lower bound \( \text{OurLB} \) is much larger than \( \text{PrevLB} \) and much closer to \( D_s \) (9, 075 meters). Based on this error setting, on average \( \text{PrevLB} \) for \( \epsilon \) dataset is 5,075 meters, while \( \text{OurLB} \) for \( \epsilon \) is 8,802 meters, which gives an improvement ratio for the lower bound as nearly 1.75. As expected, when error is increased, fewer cut-vertices are placed on the edges, thus loosening our lower bound distance that is based on the shortest network path computation after new cut-vertices and the corresponding edges are added to the graph representation of the surface triangulation, i.e., computation of our shortest surface face-crossing path. The runtime behaviour is exactly the same as described for the use of our upper bound only because the lower bound is a factor of the upper bound.

**Using Both Our Tighter Upper and Lower Bounds:** This case has already been covered in the previous two cases. We also repeated the same experiments for \( \theta_{\text{min}} = 30° \). In our technical report [11], i.e., Section 9.3, Figure 31 shows the results. Also in [11], in Section 9.3, we also show the average number of cut-vertices, edges introduced per face (Figure 29(a)) and the average number of faces accessed by the face crossing path (Figure 29(b)).

### 5.3 Effect of our Bounds on Surface \( k \)NN Query

Here, we study the impact of our new bound on the existing state-of-the-art surface \( k \)NN algorithm (VOR), that provides a Voronoi Diagram-based approach [24]. We briefly describe their technique...
and the concept of tight and loose cells as they are also used by the reverse surface NN state-of-the-art [30].

The VOR algorithm proposes a pre-processing task to generate tight and loose cells around each object \( o \), based on similar techniques used to generate Voronoi cells. A tight cell (TC) is enclosed by a loose cell and both these cells are created by using the Euclidean distance as the lower bound and the shortest network distance as the upper bound. Their motivation being that a query point \( q \) found in a tight cell of an object \( o \) can immediately report \( o \) as the 1NN of \( q \), thus avoiding an expensive surface shortest path computation. Additionally, if \( q \) is outside the loose cell (LC) of an object \( o \), then it is clearly not the nearest neighbor of \( o \). Furthermore, they propose an algorithm, that begins by finding the tight cell and loose cells (loose cells can overlap each other, but tight cells cannot), and then they incrementally expand their neighboring loose cells to search for the other neighbors and report till they exhaust \( k \) neighbors. In this approach, the bound computation time includes time to make TCs/LCs and insert them into an R-tree like structure. Query time is the time to search for TCs/LCs in the index.

Kaul et al [12] showed that using their lower and upper bounds, the VOR algorithm’s tight cell area increased, while their loose cell area decreased. This resulted in faster computation of the surface \( \delta \)NN because it avoided many expensive exact surface shortest path computations.

For the rest of the paper, we denote the combinations of algorithms and bound types as \( A-LB(\alpha)-UB(\beta) \). \( A \) is a placeholder for the implemented algorithms, with possible values \{VOR, MSRNN, SF\}, where MSRNN and SF are the reverse NN and range query algorithms, respectively, shown later in our experiments. \( LB(\alpha) \) and \( UB(\beta) \) denote the lower (upper) bounds. \( X \) is a placeholder for the various lower bound types, i.e., Euc, Prev, and Ours which represent the Euclidean lower bound, the state-of-the-art lower bound and our tighter lower bound, respectively. Similarly, \( Y \) is a placeholder for the various upper bound types, i.e., Prev, and Ours which represent the state-of-the-art upper bound and our tighter upper bound, respectively. For example, \( SF-LB(Ours)-UB(Prev) \) denotes the \( SF \) algorithm using our tighter lower bounds and the state-of-the-art upper bounds.

Using Our Tighter Lower Bound Only: In Figure 11(b), the implementation using our tighter upper bounds, i.e., \( VOR-LB(Prev)-UB(Ours) \) has fewer candidates to process as compared to the implementations using the state-of-the-art upper bound (\( UB(Prev) \)). Recall that a tighter upper bound results in the shrinking of a loose cell in VOR. This shrinkage provides efficient pruning which results in fewer candidates for which the expensive exact surface shortest path must be computed. As \( \epsilon \) goes up, the upper bound loosens, causing the loose cells to grow to the same level as when using the state-of-the-art upper bound. Consequently, processing fewer candidates displays a substantial reduction in query time, as shown in Figure 11(a). Specifically, for our tightest upper bound achieved at \( \epsilon = 0.25 \), we note a speedup of nearly 3 times compared to the state-of-the-art upper bound (VOR-LB(Prev)-UB(Prev)).

Using Our Tighter Lower Bound Only: Figure 12(b), shows a marked decrease in the number of candidates to refine in comparison to using our upper bound only (shown in Figure 11(b)). Our improvement in the tightness of the lower bound is larger than in the tightness of the upper bound, as shown in Figure 10(a). This results in the tight cells expanding much more which again results in more efficient pruning of candidates. Figure 12(a) shows the query times when \( \epsilon \) is varied. For the lowest error setting, \( \epsilon = 0.25 \), a speedup of nearly 23 times is achieved, in comparison to the state-of-the-art lower bound. Increasing the error results in a reduction in the number of cut-vertices and edges introduced, which loosens our lower bound, causing the VOR tight-cells to shrink, which results in more candidates needing to be refined. For the default setting of \( k = 2 \), Figure 12(b) shows an increase in the number of candidates that need refinement as error goes up.

Using Both Our Tighter Upper and Lower Bounds: Figures 13(a) and (b) show the results of using both our tighter upper and lower bounds. The tighter lower bound results in increasing the tight cell area and the tighter upper bound reduces the loose cell area, which results in less overlapping between loose cells and causes the VOR algorithm to explore fewer candidates. This also results in much faster query times.

Although there is a relation between the number of candidates to refine and the query time, the relation is non-linear. The distribution of objects is uniform, and an increase in the number of candidates results in an increase in the radius of the circle around the source vertex, which in turn means more triangles to process for the Chen and Han (CH) algorithm that has time complexity \( O(n^2) \). Also, although we have the same number of candidates to refine, we find that the “shape” of the loose cells (shrunk) and tight cells (expanded) causes us to overall process a much smaller
region, i.e., CH has to unfold fewer triangles in this case. VOR uses an incremental NN strategy, finding the first NN, then finding the second NN, etc. If our tight cells are larger, we find that these neighbors are reported quickly, but if they are found in the region LC - TC (Inside Loose Cell but outside the Tight cell) then we must unfold the triangles covered by LC, which takes longer. With the previous bound [12], chances of unfolding are higher since the LCs are bigger and also overlap considerably and the TCs are smaller in size, when compared to our bounds. So query time also depends on the shape of the LCs and the TCs within each LC.

The result shows that the use of both the bounds outperforms the use of any bound individually, as would be expected. For our tightest possible lower bound, we note a speedup in query time of nearly 30 times in comparison to the state-of-the-art.

We also conducted additional experiments by varying the value of $k$ for $\theta_{\min} = 45^\circ$ and $\theta_{\min} = 30^\circ$. The results can be found in our technical report [11], i.e., Section 9, in Figures 23–28.

Cut-vertex Statistics: We compute the following measurements and report their values for the default settings. Average Number of Faces Accessed: is computed by counting the faces on whose edge’s cut-vertices were introduced and averaging over total queries. In the default setting, nearly 285 faces were accessed (refer to Figure 30(b) in [11]). Average Number of Cut-vertices added per face: is measured by counting the total number of cut-vertices placed in each face divided by the number of affected faces. In the default setting, 6.4 cut-vertices were introduced per face (refer to Figure 30(a) in [11]). Total % of Cut-vertices introduced: is calculated as ratio of newly introduced vertices versus the original number of vertices in the terrain. In the default setting, this is nearly 1.2 %. We also show the behavior of this measurement in regards to changes in $\epsilon$ in our technical report (Figure 31(a) in [11]). TotalAVG./Max./Min. % of “useful” cut-vertices introduced: is computed as the number of cut-vertices that were actually dequeued from the priority queue $Q$ versus the total number of introduced cut-vertices. In the default setting, we found the total, average, maximum and minimum ratio to be 82.4 %, 78.2 %, 88.4 %, and 0 %, respectively. Refer to Figure 31(b–d) in [11] for more details. The minimum ratio is achieved when there is a direct edge connecting $s$ and $t$. Also, we noticed that there is a high utilization of cut-vertices from $Q$ as only cut-vertices placed on edges directly connected to the target $t$ are not needed during the relaxation step of our Algorithm because the final path cannot bend on the interior of a face [26] (For instance referring to Example 1, no action is taken on cut-vertices 6 and 7 since they are connected directly to target 4).

5.4 Effect of our Bounds on Reverse Surface NN Query

We implemented the algorithm for monochromatic reverse surface NN queries in [30], namely MSRRN. In the following, we focus on reverse surface 1NN queries only. This algorithm also uses the tight/loose cells proposed in VOR and hence is also affected in a similar fashion. As in VOR, the bound computation time includes time to make TCs/LCs and insert them into an R-tree like structure. Query time is the time to search for TCs/LCs in the index.

Using Our Tighter Upper Bound Only: Figures 14(a) and (b) show a similar trend to what we observed earlier in Section 5.3 (Figures 11(a) and (b)), for the same reasons as explained earlier.

Using Our Tighter Lower Bound Only: Figures 15(a) and (b) show a similar trend to what we observed earlier in Section 5.3 (Figures 12(a) and (b)), for similar reasons.

Using Both Our Tighter Upper and Lower Bounds: Figure 16(a) shows that at $\epsilon = 0.25$, where the tightest lower bound occurs, the query time for MSRRN-LB(Our)-UB(Our) is 2.2 milliseconds, in comparison to 0.1 seconds for MSRRN-LB(Prev)-UB(Prev), which results in an overall speedup of nearly 50 times on average. Additionally, Figure 16(b) shows far fewer candidates explored by MSRRN-LB(Our)-UB(Our), when compared to MSRRN-LB(Prev)-UB(Prev).

5.5 Effect of our Bounds on Range Query

The surface range query is a straightforward (SF) algorithm that was implemented in [12]. This algorithm finds all objects whose shortest surface distances to a given query point $q$ are at most a given range value $r$. SF is an algorithm with two steps. In the filtering step, the algorithm applies both the upper and lower bounds to prune objects. Let $N^\epsilon(q)$ denote the total number of objects, except $q$. Let $U$ denote the set of objects whose upper bounds are within the range $r$. Any object $o \in U$ is guaranteed to be in the final solution and hence needs no surface path computation. Let $L$ denote the set of objects whose lower bounds are outside the range $r$. Any object $o \in L$ is guaranteed not to be in the final solution and hence can be pruned immediately. Therefore, for every object $o \in (L \cup U)$ there is no more refinement (i.e., surface shortest path computation) required. The improvement ratio of our proposed lower bound to that of the state-of-the-art lower bound is much higher than the corresponding improvement ratio between our upper bound and the state-of-the-art upper bound. This results in skipping expensive shortest surface path computations for more objects due to pruning by lower bound than objects being included in the solution set due to the upper bound. Thus, $|L| \gg |U|$.

Using Our Tighter Upper Bound Only: Figure 17(b) shows a slight decrease in the number of candidates to refine because fewer objects belong in $U$. As $\epsilon$ increases and exceeds $\epsilon = 2$, there are no more cut-vertices added to the edges which means that our upper bound loosens and deteriorates to the same distance as that of the state-of-the-art upper bound. When our upper bound and the state-of-the-art upper bound match, then the same number of candidates are processed. Figure 17(a) shows the corresponding behavior in query time, which as expected, gets faster with increasing error since there are no additional vertices and edges introduced to the triangulation network.

Using Our Tighter Lower Bound Only: A better improvement ratio of our proposed lower bound to that of the state-of-the-art
lower bound results in many more objects getting pruned due to our tighter lower bound. A tighter lower bound results in many more objects being pruned, as is evident in Figure 18(b). This also results in a marked lowering of the query runtime. Specifically, for our tightest possible lower bound, i.e., at $\theta = 0.25\pi$, we note a speedup in query time of nearly 30 times in comparison to the state-of-the-art. Comparing the corresponding sub-figures in Figures 17 and 18, we notice that the difference in the number of candidates pruned and query times is due to the difference in the improvement ratio attained by our lower bound as compared to the improvement ratio attained by our upper bound.

Using Both Our Tighter Upper and Lower Bounds: Figures 19(a) and (b) show the results of using both our tighter upper and lower bounds. The result shows that the use of both the bounds outperforms the use of any bound individually, as would be expected. For our tightest possible lower bound, we note a speedup in query time of 42 times in comparison to the state-of-the-art.

5.6 Scalability
The scalability of the existing algorithms using the state-of-the-art and our new bounds are studied by varying the dataset size.

For the scalability study, we set $k = 5$, similar to [12]. Figure 20(a) shows the preprocessing times involved in generation of the VOR cells. In the scalability experiments, we employ both our tighter upper and lower bounds. We notice that $VOR-LB(\text{Our})-UB(\text{Our})$ is nearly 2 orders of magnitude slower than $VOR-LB(\text{Prev})-UB(\text{Prev})$, due to its expensive bound computation that arises due to the introduction of new cut-vertices and edges that increase the complexity of the network. Figure 20(b) shows that $VOR$ has a much lower query time when using our new bounds. In particular, for the largest dataset, for the default settings, a speedup of nearly 31 times is achieved. Thus, as explained earlier in Section 5.3, in spite of a larger bound computation time ($BT$), our tight lower bounds result in much more pruning of candidates to refine, which results in a substantial reduction in the spatial query time $SQT$ (i.e., $VOR$ in this case) which in turn results in an overall reduction in Query time $QT$. The preprocessing time of the surface range query, as shown in Figure 21(a), follows a similar trend to the surface $k$NN query shown in Figure 20(a) for the same reasons. Figure 21(b), shows that for the largest dataset, a speedup of nearly 43 times is achieved.

5.7 Experimental Summary
Recall the definition of the improvement ratio. We define the improvement ratio (which is always $> 1$) for the lower bound as $\frac{P_{\text{Prev}}}{P_{\text{Our}}}$ and for the upper bound as $\frac{P_{\text{Our}}}{P_{\text{Prev}}}$. Our experimental studies show that for $\theta_{\min} = 45^\circ$ (default) and $\theta_{\min} = 30^\circ$, our lower bound achieves an improvement ratio over the previous bound of 1.84 and 3.1, respectively. For $\theta_{\min} = 45^\circ$ (default) and $\theta_{\min} = 30^\circ$, our upper bound achieves an improvement ratio over the previous bound of 1.1 and 1.13, respectively. In spite of a slower bound computation when compared to the state-of-the-art, our bounds are much tighter, which in turn results in far fewer expensive surface distance computations. More importantly, we observe a significant speedup in all the surface spatial queries tested. More specifically, the state-of-the-art surface $k$NN query, i.e., $VOR$, experiences a significant speedup up to 82.2 times on the largest dataset, which has 1 million vertices, $k = 5$ and $\theta_{\min} = 30^\circ$.

Table 2 outlines the speedups that were experienced for various surface spatial queries for different settings of $\theta_{\min}$.

6. RELATED WORK
In this section, we review some of the existing works that explore spatial queries on terrains, that make use of distance bounds. These works propose models to combat the major underlying challenge, which is the massive computational cost associated with the exact shortest surface path calculation on a terrain. Further, we discuss the state-of-the-art lower and upper bounds computation that enables speedups in the spatial queries on terrains by eliminating expensive surface shortest distance computations, by proposing much tighter lower bounds.

Surface Spatial Queries Deng et al. [6] reduce the cost by simplifying the surface data to various resolutions and storing them in a hierarchical structure. Based on this multi-resolution model, they begin by computing $k$NN results on simpler surfaces, and move to higher resolutions when the $k$NN results are ambiguous and require more refinement. Although, later works [24, 30] find that the multi-resolution model does not give accurate $k$NN results, especially when $k$ gets larger. Additionally, this method incurs a large storage overhead. Shahabi et al. [24] propose an extension to the Voronoi-diagram to surfaces to compute the surface $k$NN query in an incremental fashion. This method is the state-of-the-art method for this query. This scheme compiles so-called tight and loose cells and stores them together in an R-tree like structure, they term as the SIR-tree. They incrementally expand the query search region using the SIR-tree and report $k$NN results as they come available. Xing et al. [29] extend the work by Shahabi et al. [24] by allowing continuous surface $k$NN queries in a highly dynamic environment which allows for arbitrary movements of data objects. Yan et al. [30] propose an algorithm to find the bichromatic and monochromatic reverse nearest neighbors on terrains. Their algorithm makes use of the tight and loose cells proposed in [24].

State-of-the-art Distance Bounds for Surface Queries All the aforementioned works used the Euclidean distance between two points on the surface as the lower bound for the shortest surface path between the two points. Kaul et al. [12] proposed a tighter lower bound than the Euclidean distance, which when applied to the existing works on surface spatial queries showed a marked improvement in query execution times. More specifically, in [24] when the lower bounds proposed in [12] are used, the tight cells increase, while their surrounding loose cells shrink, causing a speedup due to being able to prune candidates more effectively. To the best of our knowledge, the lower bound proposed by Kaul et al. [12] is the first study that captures information about the surface of the terrain by computing both the lower and upper bounds from the shortest network path computed on the Delaunay Graph representation of the surface. They further propose a graph compression technique to speedup the bound computation. This method proposed a lower bound that depended on $\theta_{\min}$, i.e., the minimum interior angle of any triangle in the triangulation. They proposed their distance bounds on realistic terrain models [5], because the tightness of their lower bound depends on $\theta_{\min}$. The Computational Geometry community avoids very complicated, hypothetical inputs (such as degenerate skinny triangles, as shown in Figure 1) to create algorithms that are provably efficient in realistic situations and hence uses constrained Delaunay triangulations, where the minimum interior angle can be forced to exceed a specified minimum threshold.

<table>
<thead>
<tr>
<th>Query Type</th>
<th>$\theta_{\text{Prev}}$</th>
<th>$\theta_{\text{Our}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface $k$NN (VOR)</td>
<td>30°</td>
<td>90°</td>
</tr>
<tr>
<td>Surface Range</td>
<td>42°</td>
<td>82.2°</td>
</tr>
<tr>
<td>Surface Reverse $k$NN</td>
<td>30°</td>
<td>85°</td>
</tr>
</tbody>
</table>

Table 2: Speedup Comparison (With Default Parameters)
Surface Shortest Path Approximation Algorithms

In the field of Computational Geometry, the problem of computing an approximate shortest path on a non-convex polyhedral surface is still considered a challenging open problem. Varadarajan et al. [28], partition the faces of the polyhedron using the well-known planar separator theorem [18] and propose an algorithm with quadratic time complexity which produces an approximate shortest path that is at most 7 times the optimal shortest surface path. [17, 21] consider the problem where each face of the given polyhedron has a weight associated with it and the cost of traversing a face is the distance traveled on the face times the weight of the face. They employ a strategy of introducing new points on the edges of the polyhedron and connecting these points with new edges to be treated as a graph. This work is closest to our work as we too employ a similar point placement strategy. The runtime complexity of their approach is \( O(n^2) \), where \( n \) is the total number of vertices in the graph. While, the goal of their work is to compute an upper bound on the surface shortest path length, our work focuses on computing a lower bound, which presents us with a different set of challenges. [9] propose another method that also places points on edges, but uses a selective-refinement strategy to iteratively use Dijkstra’s algorithm on the discrete graph of the polyhedron to reduce the region in which the shortest path can exist. Unlike our method, their method does not provide any theoretical bounds on the approximate shortest surface paths they compute.

7. CONCLUSION

In this paper, we propose tighter upper and lower bounds than the previous state-of-the-art bounds proposed in [12]. We achieve this by approximating the shortest surface path with a shortest surface face-crossing path that closely mimics the shortest surface path, by also being able to cut across the faces of the surface triangulation, and hence produce a much closer approximation to the shortest surface path. Additionally, we ensure that the quality of our bound has no dependence on the minimum interior angle \( \theta_{min} \), thus our lower bound is unaffected by the introduction of degenerate skinny triangles in the triangulation, while the quality of the state-of-the-art lower bound algorithm deteriorates when \( \theta_{min} \) is lowered. Furthermore, we provide theoretical proofs for the tightness of our bounds and why our bounds always outperform the state-of-the-art bounds. Our experiments provide further insight into the tightness of our new bounds and also provide evidence of substantial speedups. Future research directions include studying the effect of our new bounds on continuous surface spatial queries and how our bounds can be modified to improve constrained surface shortest path queries, with static and dynamic obstacles in the path between a source and destination.

8. REFERENCES


---

**Table 1: Scalability: Surface kNN Queries**

<table>
<thead>
<tr>
<th>Dataset size</th>
<th>Preprocessing Time (sec)</th>
<th>Query Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOR-LB(Euc)-UB(Prev)</td>
<td>SF-LB(Euc)-UB(Prev)</td>
<td>SF-LB(Our)-UB(Our)</td>
</tr>
<tr>
<td>SF-LB(Euc)-UB(Prev)</td>
<td>VOR-LB(Euc)-UB(Prev)</td>
<td>SF-LB(Our)-UB(Our)</td>
</tr>
<tr>
<td>SF-LB(Prev)-UB(Prev)</td>
<td>VOR-LB(Prev)-UB(Prev)</td>
<td>SF-LB(Our)-UB(Our)</td>
</tr>
</tbody>
</table>

**Figure 21: Scalability: Surface Range Queries**