Distance Oracle on Terrain Surface


Published in:
Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
© 2017 ACM
This work is made available online in accordance with the publisher’s policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.
ABSTRACT

Due to the advance of the geo-spatial positioning and the computer graphics technology, digital terrain data become more and more popular nowadays. Query processing on terrain data has attracted considerable attention from both the academic community and the industry community. One fundamental and important query is the shortest distance query and many other applications such as proximity queries (including nearest neighbor queries and range queries), 3D object feature vector construction and 3D object data mining are built based on the result of the shortest distance query. In this paper, we study the shortest distance query which is to find the shortest distance between a point-of-interest and another point-of-interest on the surface of the terrain due to a variety of applications. As observed by existing studies, computing the exact shortest distance is very expensive. Some existing studies proposed $\epsilon$-approximate distance oracles where $\epsilon$ is a non-negative real number and is an error parameter. However, the best-known algorithm has a large oracle construction time, a large oracle size and a large distance query time. Motivated by this, we propose a novel $\epsilon$-approximate distance oracle called the Space Efficient distance oracle (SE) which has a small oracle construction time, a small oracle size and a small distance query time due to its compactness storing concise information about pairwise distances between any two points-of-interest. Our experimental results show that the oracle construction time, the oracle size and the distance query time of SE are up to two orders of magnitude, up to 3 orders of magnitude and up to 5 orders of magnitude faster than the best-known algorithm.

1. INTRODUCTION

With the advance of geo-spatial positioning and computer graphics technology, digital terrain data has become increasingly popular nowadays, and it has been used in many applications such as Microsoft’s Bing Maps and Google Earth in the industry community. The terrain data has also attracted considerable attention from the academic community [8, 10, 29, 35, 24, 36, 20, 19].

Terrain data is usually represented by a set of faces each of which corresponds to a triangle. Each face (or triangle) has three line segments called edges connected with each other at three vertices. An example of a piece of terrain data is shown in Figure 1, where we have 24 faces, 40 edges and 17 vertices.

The geodesic distance between two given locations (or points) on the surface of the terrain is the length of the shortest path/route from one point to the other on the surface. For example, in Figure 1, $s$ and $t$ are two POIs on the terrain surface and the shortest path from point $s$ to point $t$ is shown and is denoted by $GP$, which corresponds to a sequence of line segments on the faces of the terrain. Note that the geodesic distance is usually quite different from the Euclidean distance, and according to [8], the ratio of the geodesic shortest distance and the Euclidean distance is up to 300%. In Figure 1, the Euclidean distance between $s$ and $t$ is the length of the line segment $EP$.

1.1 Application

In many applications, a set of points-of-interest (POIs) is given on the surface of the terrain, and it is required to compute the geodesic distances between pairs of POIs. Some examples are introduced as follows.

(1) Geographic Information System (GIS). In GIS, it is important to compute the geodesic distance between two POIs. For example, hikers need the geodesic distance to measure the travel time between a source and a destination which are landmarks (which are POIs) in practice [28]. Besides, the vehicles (e.g., Google Map camera car and military vehicles) estimate the geodesic distance to measure the travel cost [23, 32]. In life sciences, scientists conduct distance queries on residential locations (which are POIs) of the animals in the wildness to study their migration patterns [11, 25].

(2) Computer Graphics and Vision. In computer graphics and vision [22, 30], measuring similarities between two different 3D objects is very important. In order to measure similarities between objects, a number of reference points (which are POIs) [22, 30] are
selected on the surface of each object. These reference points play an important role in similarity measurement since they are invariant to transformations such as rotation and translation. For each object, geodesic distances between all pairs of reference points are computed and stored as a feature vector for similarity measurement. In this application, multiple geodesic distance computations are involved.

(3) Scientific Data 3D Modeling. There is a need to model scientific data in 3D models in areas like biology, chemistry, anthropology and archeology [1, 17]. In neuroimaging, similar to computer graphics and vision, a 3D model of an organ is associated with a set of reference points [1, 17] (which are POIs) and these reference points correspond to functional units on the organ and the scientists use the geodesic distance between reference points to analyze tumor development with magnetic resonance imaging (MRI) images. In neuroscience, scientists conducted spatial queries on a 3D brain model to study the neuron density and the number of branches in a region of the brain [31]. Similarly, multiple geodesic distance computations are involved in this application.

(4) Online 3D Virtual Game. In some online 3D virtual games like INGRESS, a city (e.g., San Francisco in game INGRESS) has a terrain surface which consists of a number of portals (which are POIs). For each portal, it is important to calculate the geodesic distance from this portal to each of the other portals so that the influence of this portal is estimated. Here, multiple computations for geodesic distances are involved.

(5) Spatial Data Mining. There are many data mining techniques used in the spatial databases. For example, in the clustering technique, the inner-cluster distance and the inter-cluster distance are needed. In the co-location pattern mining, shortest distance queries, namely vertex-based oracles (edges) and the wildness, radio-telemetry receivers set up for collecting animal movement data could be POIs. In the context of spatial data mining, the number of geodesic distance computations is very large.

1.2 Motivation

Consider a terrain \( T \) with \( N \) vertices. Let \( P \) be a set of \( n \) POIs on the surface of the terrain.

Due to a variety of applications in different domains as described in Section 1.1, computing geodesic distances [26, 6, 34, 24, 20, 19, 2, 3, 12] is very important and is very fundamental to other proximity queries such as nearest neighbor queries [9, 10, 29, 35, 20, 19], range queries [20, 19] and reverse nearest neighbor queries [36, 20].

Motivated by this, we aim to study three kinds of queries, namely vertex-to-vertex (V2V) distance queries, POI-to-POI (P2P) distance queries and arbitrary point-to-arbitrary point (A2A) distance queries. Consider the first two types of queries. Each V2V distance query returns the geodesic distance between a starting point \( s \) and a destination point \( t \), where both \( s \) and \( t \) are vertices (from \( V \)). Each P2P distance query returns the geodesic distance between a starting point \( s \) and a destination point \( t \), where both \( s \) and \( t \) are POIs (from \( P \)). Since P2P distance queries, considering both the concept of vertices and the concept of POIs, is more general than V2V distance queries, considering only the concept of vertices without the concept of POIs, P2P distance queries could be regarded as a generalization of V2V distance queries. Specifically, under the problem setting for P2P distance queries, if for each vertex in the problem setting for V2V distance queries, we create a POI which has the same coordinate values as this vertex, then the P2P distance queries will become the V2V distance queries. Thus, for clarity, in this paper, we focus on P2P distance queries. Consider the third type of queries. Each A2A distance query returns the geodesic distance between a starting point \( s \) and a destination point \( t \), where both \( s \) and \( t \) are two arbitrary points on the surface of the terrain. Since A2A distance queries allow all possible points on the surface of the terrain, A2A distance queries generalize both P2P and V2V distance queries. For the ease of illustration, in the main body of this paper, we first study P2P distance queries. Later, in Appendix C, we study A2A distance queries.

Our natural goal of answering each P2P distance query is to return the corresponding distance in a short time. However, none of the existing studies [26, 6, 34, 24, 20, 19, 2, 3, 12] could achieve this goal satisfactorily.

Firstly, all existing algorithms [26, 6, 34] computing exact geodesic distances on-the-fly are still slow even in the moderate-sized terrain data. The time complexities of the algorithms for computing exact geodesic distances proposed by [26, 6, 34], are \( O(N^2 \log N) \), \( O(N^2) \), \( O(N \log^2 N) \) and \( O(N^2 \log N) \), respectively, which is still very large when \( N \) is large. In the literature [29, 35, 20, 19], the algorithm proposed in [6] is recognized as a state-of-the-art fastest algorithm. Many existing papers [29, 35, 20, 19] adopt this for finding the geodesic distance. According to [19], the algorithm proposed in [6] took more than 300 seconds on a terrain with 200K vertices, which is very slow.

Secondly, although some existing algorithms [24, 20, 19] were proposed to compute approximate geodesic distances on-the-fly for reducing the computation time, all of these algorithms are still not efficient enough for proximity queries and applications involving many distance queries. The algorithm in [24] computes the approximate geodesic distance/path satisfying a slope condition, the algorithm in [20] computes the lower and upper bounds of the geodesic distances between two points, and it provides no guarantees on the qualities of the bounds found, and the algorithm in [19], which is an improved version of that in [20], runs in \( O((N+N') \log(N+N')) \) time where \( N' \) is the number of additional vertices introduced for the sake of the guarantee on the qualities of the lower and upper bounds found.

1.3 Distance Oracle

Motivated by these, to efficiently process the geodesic distance queries, especially for those cases where queries for many different pairs of points are issued, some existing studies [2, 3, 12] aim at designing geodesic distance (and/or the corresponding shortest path) oracles. To the best of our knowledge, all existing studies focused on building oracles for returning approximate geodesic distances only but no existing studies focused on building oracles for returning exact geodesic distances (which could be explained by the high computation cost of computing the exact geodesic distances). All of these studies [2, 3, 12] are based on auxiliary point-based oracles. Specifically, they first introduce a large number of auxiliary points (edges), namely Steiner points (edges), on the surface of the terrain where each Steiner edge connects two Steiner points. Then, they construct a graph \( G_t \), whose vertices (edges) are either original vertices (edges) or the Steiner points (edges). The exact distance between any two vertices/points on \( G_t \) is an \( \epsilon \)-approximate geodesic distance between these two vertices/points. The \( \epsilon \)-approximate geodesic distance oracles proposed in [2, 3, 12] indexes the exact distances on \( G_t \). Among these studies, the oracle in [12] is the best, where the space complexity of the oracle (called the oracle size) is \( O(N \sin(\theta) + \frac{1}{\epsilon^2} \log^2 \theta) \cdot \frac{\log^2 \theta}{\epsilon^2} \) where \( \theta \) is the minimum inner angle of any face of the terrain surface. It can answer \( \epsilon \)-approximate P2P distance queries in \( O(\frac{1}{\epsilon^2} \log \frac{1}{\epsilon} + \log \log N) \) time.

Unfortunately, these auxiliary point-based oracles have two
drawbacks. The first drawback is that each of these oracles has a large oracle building time and a large oracle size. This is because a large number of Steiner points (edges) are introduced during the oracle construction and the number of Steiner points could be several orders of magnitude larger than the number of vertices on the surface of the terrain. Thus, each of these oracles has a poor empirical performance in terms of both the oracle building time and the oracle size. The second drawback is that each of these oracles is constructed based on the structure of the terrain without considering the information about POIs. In other words, it is constructed based on the set of vertices regardless of the set of POIs. For example, consider the case where there are only two POIs, a naive oracle storing the geodesic distance for one pair (of POIs) occupies a $O(1)$ space only but the oracle in [12] could introduce millions of Steiner points, resulting in a large oracle size and a large building time.

Motivated by the drawbacks of the existing methods, we propose a distance oracle called the **Space-Efficient Distance Oracle (SE)** such that for any point $s$ and any point $t$ in $P$, the oracle returns an $\epsilon$-approximation of the geodesic distance between $s$ and $t$ efficiently, where $\epsilon$ is a non-negative real user parameter, called the error parameter. Our SE has three good features: (1) low construction time, (2) small size and (3) low query time (compared with the best-known oracle [12]). This is because SE is space-efficient in the sense that its size is linear to $n$ (i.e., no of POIs). Due to this space-efficient property, it is much easier for us to design an efficient algorithm for constructing the SE and an efficient algorithm for answering distance queries.

### 1.4 Contribution & Organization

We summarize our major contributions as follows. Firstly, we propose a novel distance oracle called SE, which can be computed efficiently, has small size and can answer $\epsilon$-approximate geodesic distance queries efficiently. Secondly, our SE answers not only P2P distance queries but also V2V distance queries. Thirdly, in V2V distance queries, our experimental results show that the building time, oracle size and query time of SE are, respectively, 5-100 times, 10-100 times and more than 1000 times smaller than those of the best-known distance oracle [12] on benchmark real datasets. In P2P distance queries, the building time, oracle size and query time of SE are 10-100 times, 10-1000 times and 100-10000 times smaller than those of the best-known distance oracle [12] on benchmark real datasets, respectively.

The remainder of the paper is organized as follows. Section 2 provides the problem definition. Section 3 presents our distance oracle, namely SE. Section 4 reviews the related work and introduces some baseline methods. Section 5 presents the experimental results and Section 6 concludes the paper.

### 2. PROBLEM DEFINITION

Consider a terrain $T$. Let $V$ be the set of all vertices on the surface of the terrain $T$, and $E$ be the set of all edges on the surface of the terrain $T$. Let $N$ be the size of $T$ (i.e., $N = |V|$). Each vertex $v \in V$ has three coordinate values, denoted by $x_v$, $y_v$ and $z_v$.

Let $P$ be a set of POIs on the surface of the terrain $T$ and $n$ be the size of $P$ (i.e., $n = |P|$). In the following discussion, we focus on the case when $n \leq N$. This is because in real-life applications, $n \leq N$. For example, in the BearHead dataset, one benchmark dataset used in the literature, $n = 4k$ and $N = 1.4M$. In the EaglePeak dataset, the other benchmark dataset, $n = 4k$ and $N = 1.5M$. The discussion about how we handle the case when $n > N$ can be found in Appendix D. Each POI $p \in P$ also has three coordinate values, denoted by $x_p$, $y_p$, and $z_p$. In this paper, we assume that $P$ contains no duplicate points since any two co-located POIs can be regarded as one POI in practice, and we can merge any two co-located POIs into one POI by a simple preprocessing step.

Given two points, $s$ and $t$, on the surface of $T$, the geodesic shortest path between $s$ and $t$, denoted by $\Pi(s, t)$, is defined to be the shortest path between the two points on the surface of $T$. Note that the geodesic shortest path corresponds to a sequence of line segments on the surface of the terrain. Consider the example in Figure 1 where the geodesic shortest path between two points $s$ and $t$ is denoted by $GP$. Given two points, $s$ and $t$, on the surface of $T$, the geodesic distance between $s$ and $t$, denoted by $d_{SE}(s, t)$, is defined to be the length of the geodesic shortest path between the two points, i.e., $\Pi(s, t)$, where the length of a path is defined to be the sum of the lengths of all line segments of the path. The geodesic distance $d_{SE}(s, t)$ is a metric, and therefore it satisfies the triangle inequality.

Note that a full materialization of geodesic distances for all possible pairs of points in $P$ is not feasible since the complexity of the oracle size and the complexity of the oracle building time are $O(n^2)$ and $O(nN \log^2 N)$, respectively, which are prohibitively large.

### 3. DISTANCE ORACLE

We first present the overview of our distance oracle called SE in Section 3.1. Then, we present the first component of SE, called the *compressed partition tree*, in Section 3.2, the second component of SE, called the *node pair set*, in Section 3.3, the query processing algorithm based on SE in Section 3.4, the construction algorithm of SE in Section 3.5, and some theoretical results of SE in Section 3.6.

#### 3.1 Overview

Before giving an overview, we first give the concept of a disk. Given a point $p \in P$ and a non-negative real number $r$, a disk centered at $p$ with radius equal to $r$ on the terrain surface, denoted by $D(p, r)$, is defined to be a set of all possible points on the terrain surface whose geodesic shortest distance to $p$ is at most $r$. That is, $D(p, r) = \{p' | d_{SE}(p, p') \leq r \text{ and } p' \text{ is an arbitrary point on the terrain surface.}\}$

With this concept, we are ready to describe our distance oracle SE which includes two major components, namely the *compressed partition tree* and the *node pair set*.

The first component is the compressed partition tree in which each node corresponds to a disk containing a set of POIs. In the leaf level of the tree, there are $n$ nodes each of which corresponds to a disk containing only one POI. Each node in this level has the smallest radius (since each node contains only one POI). In the level just above the leaf level of the tree, there are fewer nodes each of which corresponds to a disk containing one or more POIs. Each node in this level has a larger radius (since each node contains one or more POIs). Similarly, each node in a higher level has a larger radius. At the root level of the tree, the (root) node has the largest radius since it contains all $n$ POIs. Note that for different levels, the tree has different number of nodes (with different radius).

The second component is the node pair set which is a set of the pairs of nodes from the compressed partition tree. In this node pair set, each node pair in the form of $(O, O')$ is associated with the distance between the centers of the corresponding disks of $O$ and $O'$ where $O$ and $O'$ are two nodes in the compressed partition tree. Besides, the node pair set satisfies one interesting property called the unique node pair match property which is the key to the query efficiency of our SE. The unique node pair match property states that for any two points, namely $p$ and $q$, in $P$, there exists exactly
one node pair \( (O, O') \) in the node pair set such that \( O \) contains \( p \) and \( O' \) contains \( q \).

Consider a distance query with a source point \( s \in P \) and a destination point \( t \in P \). Let \( h \) be the height of the tree. In all of our experimental results on benchmark real terrain datasets, \( h \) is smaller than 30. We could answer this distance query in \( O(h) \) time using \( SE \). The major idea is to find a node pair \( (O, O') \) in the node pair set efficiently such that \( O \) contains \( s \) and \( O' \) contains \( t \), and return the distance associated with this node pair. Interestingly, even though the distance returned is associated to this node pair, it will be shown later that the distance returned is an \( \epsilon \)-approximation of the geodesic distance between \( s \) and \( t \).

The major challenge here is how to design \( SE \) which achieves the space-efficient property (mentioned in Section 1). We will describe the details of how we address this challenge.

### 3.2 Oracle Component 1: Compressed Partition Tree

In this section, we present a hierarchical structure called a partition tree to index all POIs in \( P \), which is used for constructing the first component (i.e., the compressed partition tree) of our distance oracle \( SE \).

A partition tree is defined to be a tree with the following components:

- Each node \( O \) in the tree has two attributes, namely its center, denoted by \( c_O \), and its radius, denoted by \( r_O \), where \( c_O \) is a point in \( P \) and \( r_O \) is a non-negative real number.
- For each leaf node \( O \), \( D(c_O, r_O) \) contains only one point in \( \mathbb{P} \) (which is \( c_O \)) and thus contains no objects in \( P \) other than \( c_O \). Note that there are \( n \) leaf nodes.
- For each internal node \( O \), the center of each child of node \( O \) is in \( D(c_O, r_O) \) and the radius of each child of node \( O \) is equal to \( r_O \).
- Each node \( O \) in the tree is associated with its representative set, denoted by \( RS(O) \), which is defined to be a set containing the centers of all the leaf nodes in the subtree rooted at \( O \).

Given two nodes, namely \( O \) and \( O' \), the (geodesic) distance between \( O \) and \( O' \), denoted by \( d_g(O, O') \), is defined to be \( d_g(c_O, c_{O'}) \).

Let \( h \) be the height of the partition tree. The partition tree has \( h + 1 \) layers, namely Layer 0, Layer 1, ..., Layer \( h \). Layer 0 is the layer containing the root node only. For each \( i \in [1, h] \), Layer \( i \) is the layer containing all child nodes of each node in Layer \( (i - 1) \). Finally, Layer 0 is the layer containing all leaf nodes. If a node is in Layer \( i \) where \( i \in [0, h] \), we also say that the depth of this node is \( i \). Note that all nodes in the same layer have the same radii. The radius of Layer \( i \), denoted by \( r_i \), is defined to be the radius of one of the nodes in Layer \( i \). For any \( i, j \in [0, h] \), we say that Layer \( i \) is higher than Layer \( j \) (or Layer \( j \) is lower than Layer \( i \)) if and only if \( i < j \).

Next, we give the three properties of this partition tree to be satisfied. We will describe how to construct a partition tree satisfying these three properties later.

- **Separation Property**: For each \( i \in [0, h] \), the radius of each node in Layer \( i \) is \( \frac{r_i}{2} \) and the geodesic distance between any two nodes in this layer is at least \( \frac{r_i}{2} \).

- **Covering Property**: For each layer where \( X \) denotes a set of all nodes in this layer, the region represented by \( \bigcup_{O \in X} D(c_O, r_O) \) covers all points in \( P \).

- **Distance Property**: For each node \( O \) in the tree, if \( O' \) is one of the descendant nodes of \( O \), then \( d_g(c_O, c_{O'}) \) is at most \( 2 \cdot r_O \), i.e., \( c_{O'} \) is in the disk \( D(c_O, 2 \cdot r_O) \).

Given a node \( O \) in the partition tree, the enlarged disk of node \( O \) is defined to be \( D(c_O, 2 \cdot r_O) \). From the Distance Property, we deduce that for each node \( O \) in the partition tree, all points in \( RS(O) \) (which are points in \( P \)) are in the enlarged disk of node \( O \).

**Example 1 (Partition Tree).** Consider the points on a terrain surface as shown in Figure 2. There are 12 points \( p_1, p_2, p_3, ..., p_{12} \) in \( P \).

Figure 3 shows three small disks, namely \( D(p_1, r_3), D(p_2, r_3) \) and \( D(p_3, r_3) \), one medium-small disk, namely \( D(p_2, r_2) \), one medium-large disk, namely \( D(p_2, r_1) \), and one large disk, namely \( D(p_7, r_0) \), where \( r_0, r_1, r_2, r_3 \) are four non-negative real numbers. Note that \( r_3 \) is the radius of the large disk, \( r_1 \) is the radius of the medium-large disk, \( r_2 \) is the radius of the medium-small disk and \( r_3 \) is the radius of one of the small disks. We also show all disks to be used in this example in Figure 4.

There are 21 disks in the figure, each of which centers at a point. For example, the disk \( D(p_7, r_0) \) is a disk with its center equal to \( p_7 \) and its radius equal to \( r_0 = d_g(p_7, p_{11}) \).

Figure 5 shows a partition tree of height equal to 3 which is built based on the 12 points shown in Figure 2. In this figure, each black dot corresponds to a node in the tree. By definition, any two nodes in the same layer have the same radii. In Layer 0, there is only one node \( O_0 \) (i.e., the root node) with its radius \( r_0 \) equal to \( d_g(p_7, p_{11}) \). In Layer 1, there are three nodes, namely \( O_1, O_{10}, O_{19} \), each with its radius \( r_1 \) equal to \( 0.25r_0 \). In Layer 2, there are 5 nodes, namely \( O_3, O_{14}, O_{15}, O_{16} \), each with its radius \( r_2 \) equal to \( 0.125r_0 \). In the figure, we list the center of each node below the label of the node. For example, there is a label \( p_2 \) below the label \( O_{13} \), which means that the center of \( O_{13} \) is \( p_2 \).

Consider the leaf node \( O_{13} \) with its center equal to \( p_1 \) and its radius equal to \( r_3 \). It is easy to see that disk \( D(p_1, r_3) \) contains only one point in \( P \) (i.e., \( p_1 \)) as shown in Figure 3. The representative set of this node is a set containing only the center of this node (i.e., \( p_1 \)). This holds as well for each of the other leaf nodes (e.g., node \( O_{2} \) and node \( O_{3} \)).

Consider the internal node \( O_{13} \) with its center equal to \( p_2 \) and its radius equal to \( r_2 \). The center of each child of node \( O_{13} \) (i.e., node \( O_1, node O_2 \) and node \( O_3 \)) is in disk \( D(p_2, r_2) \) as shown in Figure 3. Besides, the radius of each child of node \( O_{13} \) is equal to \( 0.5 \cdot r_2 \) (since the radius of each child is equal to \( 0.125r_0 \) and \( r_2 = 0.25r_0 \)). The representative set of this node is a set containing the centers of all the leaf nodes in the subtree root at \( O_{13} \) (i.e., the center of node \( O_1 \) which is \( p_1 \)), the center of node \( O_2 \) (which is \( p_2 \)) and the center of node \( O_3 \) (which is \( p_3 \)). This holds as well for each of the other internal nodes.

It is easy to verify that the partition tree shown in this figure satisfies the three properties described above.

Next, we present our top-down method for building the partition tree.

- **Step 1 (Root Node Construction):** We create the root node as follows.
  - **Step (a) (Initialization):** We assign a variable \( i \), denoting the layer number, with 0.
  - **Step (b) (Point Selection):** We randomly select a point \( p \) in \( P \).
  - **Step (c) (Radius Computation):** We create the root node as \( \bigcup_{O \in X} D(c_O, r_O) \) covers all points in \( P \).
algorithm, we obtain the maximum distance \( d \) between \( p \) and a point in \( P \).

- **Step (d) (Node Construction):** We create a root node \( O \) where its center is set to \( p \) and its radius is set to \( d \). Note that in this layer, \( r_0 = d \).

- **Step 2 (Non-Root Node Construction):** We perform the following operations.
  - **Step (a) (Initialization):** We increment variable \( i \) by 1. We assign a variable \( P' \), denoting a set of remaining points in \( P \) to be “covered” by a node in Layer \( i \), with \( P \).
  - **Step (b) (Iterative Step):** We perform the following iterative steps.
    - **Step (i) (Point Selection):** Let \( C \) be a set containing the centers of all nodes in Layer \( i - 1 \) and let \( P_C \) be the set of remaining points in \( P' \) each of which is one of the centers of all nodes in Layer \( i - 1 \) (i.e., \( P_C = P' \setminus C \)). We randomly select a point \( p \) from \( P_C \) if \( P_C \neq \emptyset \), and select a point \( p' \) from \( P' \) based on a point selection strategy (to be described later) otherwise.
    - **Step (ii) (Point Covering):** We find a set \( S \) of all points in \( P' \) that are in \( D(p, \frac{r}{2}) \) by performing a single-source all-destination (SSAD) exact shortest path algorithm which takes \( p \) as an input of the source point and executes the algorithm until the distance between the boundary of the search region and \( p \) is greater than \( \frac{r}{2} \). We remove all points in \( S \) from \( P' \).
    - **Step (iii) (Node Creation):** We create a node \( O \) where its center is set to \( p \) and its radius is set to \( \frac{r}{2} \). Then, we find the node \( O_{\text{parent}} \) in Layer \( (i - 1) \) whose distance to \( O \) is the minimum. We set the parent of \( O \) to \( O_{\text{parent}} \).
    - **Step (iv) (Additional Node Creation):** We repeat the above steps (i.e., Steps (i)-(iii)) until \( P' \) is empty.

- **Step (e) (Next Layer Processing):** We repeat the above steps (i.e., Step (a) and Step (b)) until the number of nodes in Layer \( i \) is equal to \( n \).

**Lemma 1.** The partition tree generated by the above procedure satisfies the Separation, Covering and Distance Properties.

**Proof.** For the sake of space, all the proofs in the paper can be found in Appendix B.

Some implementation details of this algorithm are given as follows.

**Implementation Detail 1 (Point Selection Strategy in Step 2(b)(i)):** We propose two heuristic-based point selection strategies as follows. The first one is called the random selection strategy. It randomly selects a point \( p \) from \( P' \). The second one is called the greedy selection strategy which is to select a point from \( P' \) in the “densest” region (or formally cell) on the surface of the terrain. The major idea of this strategy is to select a point from \( P' \) in the densest region (because if this point is selected as the “center” of the disk, then this disk can cover many points (which could come from the densest region)). Specifically, this strategy requires some additional operations included in other steps, and we describe them as follows. (A) Between Step 2(a) and Step 2(b), we construct a grid on the \( x-y \) plane with the cell width equal to \( O(\frac{r}{2}) \). Then, we insert all points from \( P' \) in corresponding cells, and all point IDs in each cell are indexed in a B+-tree. We also build a max-heap containing all non-empty cells whose keys are the sizes of their B+-trees. (B) In Step 2(b)(i), in the case that \( P_C = \emptyset \), we select a point \( p \) from \( P' \) by finding the cell with the greatest number of points in \( P' \) and randomly selecting a point \( p \) from \( P' \) in the cell. (C) In Step 2(b)(ii), for each point \( p' \) in \( S \), we remove \( p' \) from the B+-tree of its corresponding cell and decrease the key of the cell in the max-heap by 1.

**Implementation Detail 2 (SSAD algorithm):** Note that in Step 1(c) and Step 2(b)(ii), we need to perform the SSAD algorithm [6,
which is a best-first search algorithm. There are two versions of
this algorithm here, but the major principle is the same for each but
with different stopping criteria. The major principle is described
as follows. The algorithm performs a search that starts from s and
expands its search with the vertex in V which has not been pro-
cessed and has its minimum geodesic distance \(d_{min}\) to s. For each
vertex expansion, all points in \(P\) on each face expanded together
with the vertex are computed with their geodesic distances. Note
that we know that for each vertex expansion, all vertices in \(V\) with
their geodesic distances smaller than \(d_{min}\) have been processed.
The first version of this algorithm (in Step 1(c)) has an input of a
source point \(s\) only. For each vertex expansion, the first version of
the algorithm checks whether all points in \(P\) have been visited.
If yes, this algorithm terminates. The second version of this al-
gorithm (in Step 2(b)(ii)) takes as its inputs a source point \(s\) and a
distance threshold \(\delta\) (denoting the boundary of the search region
starting from \(s\)). For each vertex expansion, the second version of
the algorithm checks whether \(d_{min}\) is larger than \(\delta\). If yes, this
algorithm terminates. The time complexity of each of these two
versions is \(O(N \log N + k)\), where \(N\) is the number of vertices in \(V\) pro-
cessed and \(k\) is the number of points in \(P\) processed.

Finally, we analyze the depth \(h\) of the partition tree. The follow-
ing lemma presents the depth of the partition tree.

**Lemma 2.** \(h \leq \log \left( \frac{\max_{p \in P} d_p(p,q)}{\min_{p \in P} d_p(p,q)} \right) + 1\)

By our assumption of Section 2 that there are no duplicate POIs,
it follows that \(\min_{p \in P} d_p(p,q)\) is strictly positive. We want to
emphasize that the upper bound of \(h\) (i.e., \(\log \left( \frac{\max_{p \in P} d_p(p,q)}{\min_{p \in P} d_p(p,q)} \right) + 1\)) is a small value in practice. Firstly, in all of our experimental
results, \(h\) is at most 30. Secondly, even in the extreme case where
the minimum distance is one nanometer (\(= 10^{-9}\) m) and the
maximum distance is the length of the Earth’s equator (\(\approx 4 \times 10^{7}\) m),
Lemma 2 yields an upper bound of only 56.

Consider the first component called the **compressed partition tree**
which is a variation of the partition tree.

We construct the compressed partition tree \(T_{\text{compress}}\) based on
the original partition tree \(T_{\text{org}}\) as follows. Firstly, we generate
\(T_{\text{compress}}\) with the same process as \(T_{\text{org}}\). Secondly, whenever there is a node \(O\) in \(T_{\text{compress}}\) containing only one child node \(O_{\text{child}}\), if there is a
parent node \(O_{\text{parent}}\) of \(O\), then we remove the parent-and-child re-
lationship between \(O_{\text{parent}}\) and \(O_{\text{child}}\) and then the parent of \(O_{\text{child}}\) is set to \(O_{\text{parent}}\). Then, we delete \(O\). We repeat this step iteratively until there is no node in \(T_{\text{compress}}\) containing only one child. Thirdly, for
each leaf node in \(T_{\text{compress}}\), we set its radius to 0.

Note that each leaf node (containing no child node) is still kept
after the above operation since each node removal operation in-
volves a node containing only one child node. Note that for each
point \(p\) in \(P\), there exists exactly one leaf node whose center is \(p\).
Given a point \(p\) in \(P\), its **corresponding leaf node**, denoted by \(O_p\), is
defined to be the leaf node in the compressed partition tree whose
center is \(p\). Besides, given a node \(O\) in the compressed partition
tree, the layer number of the layer containing \(O\) in the compressed
partition tree is defined to be the layer number of the layer contain-
ning \(O\) in the (original) partition tree.

**Example 2. (Compressed Partition Tree).** Consider the partition tree (Figure 5) in Example 1. According to the above
procedure, since node \(O_{17}\) has only one child node (i.e., node
\(O_{22}\)), we remove the parent-and-child relationship between \(O_{17}\)
and \(O_{22}\) and then set the parent of \(O_{22}\) to node \(O_{17}\) (which is
the parent of \(O_{17}\) in the original partition tree). Then, we remove
node \(O_{17}\). After this operation, we do a similar operation for node
\(O_{18}\) containing only one child node \(O_{13}\). After that, no node in the
resulting tree contains only one child. Finally, for each leaf node in
the resulting tree, we set its radius (i.e., \(r_3\)) to 0. The resulting tree
is the compressed partition tree as shown in Figure 6. Note that the
layer number of the layer containing node \(O_{20}\) is 1 and the layer
number of the layer containing node \(O_{12}\) is 3 (although the node
\(O_{20}\) in Layer 2 of the (original) partition tree (which connects \(O_{12}\)
and \(O_{20}\)) is removed).

As will be shown later, the space complexity of the compressed
partition tree is \(O(n)\) (which is linear to \(n\)).

### 3.3 Oracle Component 2: Node Pair Set

Consider the second component of \(SE\) called the **node pair set**.
Before we define this, we give some definitions based on the
compressed partition tree which will be used in the node pair set.

Given two nodes \(O\) and \(O'\) in the compressed partition tree, \(O\) and \(O'\) are **well-separated** if and only if \(d_{p,q}(O,O') \geq (\frac{\delta}{2} + 2) \cdot \max\{r,r'\} \) where \(r\) is the radius of the enlarged disk of \(O\)
and \(r'\) is the radius of the enlarged disk of \(O'\). Given two nodes \(O\) and
\(O'\) which are well-separated in the compressed partition tree, we
say that \((O,O')\) is a well-separated (node) pair.

Given a node pair \((O,O')\) and two nodes \(O\) and \(O'\) in a tree
where (1) \(O\) is either a descendant node of \(O\) or (2) \(O'\) is
either \(O\) or a descendant node of \(O'\), we say that \((O,O')\) contains
\((O,O')\). Note that in our context, a node pair \((O,O')\) has an order.
Specifically, if \((O,O')\) contains \((O,O')\), it is possible that
\((O,O')\) does not contain \((O,O')\).

In practice, given two points \(p\) and \(q\) in \(P\) with their correspond-
ing leaf nodes \(O_p\) and \(O_q\) in the compressed partition tree, we say that
\((O,O')\) contains \((p,q)\) if \((O,O')\) contains \((O_q,O_p)\).

Next, we give a method of generating the node pair set given a
compressed partition tree. We maintain a variable \(S\) storing a set of
node pairs, initialized as \([(O_{\text{root}},O_{\text{root}})]\) where \(O_{\text{root}}\) is the root
node of the compressed partition tree. At each iteration, we extract
a pair \((O_i,O_j)\) from \(S\) which is not well-separated. Then, we select
the node in the pair \((O_i,O_j)\) whose radius is larger. Without loss of
generality, we assume that \(O_i\) is selected and let \(C_1,C_2,\ldots,C_m\)
denote its children. Next, we insert \((C_1,O_i),(C_2,O_i),\ldots,(C_m,O_i)\)
into \(S\). For each \(x \in [1,m]\), \((C_x,O_i)\) is said to be a
pair generated by \((O_i,O_j)\) and \(O_i\) is said to be split from \((O_i,O_j)\).
Note that if \(O_i\) and \(O_j\) have the same radius, then we select the
node with a smaller node ID in the pair \((O_i,O_j)\) for processing. We
repeat the above procedure until each pair in \(S\) is well-separated.

Let \(S\) be the set of node pairs returned by the above procedure.
\(S\) is called the **node pair set** of \(SE\).

In the above procedure, note that whenever we check whether a
node pair \((O_i,O_j)\) is well-separated, we have to compute the
distance between \(O_i\) and \(O_j\). Later in Section 3.5 as a part of
the oracle construction, we will explain how we compute this distance
efficiently.

The following theorem shows a key property of the node pair set
generated, namely the **unique node pair match property**.

**Theorem 1.** Let \(S\) be the node pair set of \(SE\). Each node pair
in \(S\) is a well-separated pair and for any two points \(p\) and \(q\) in \(P\),
there exists exactly one node pair \((O,O')\) in \(S\) containing \((p,q)\)
and the distance associated with this node pair is an \(\epsilon\)-approximate
distance of \(d_{p,q}\).

Next, we present the following theorem showing that there are
\(O(\frac{m^2}{\beta^2})\) node pairs considered in the procedure of generating
the node pair set (which is linear to \(n\)) where \(\beta\) is a real number and is
in the range from 1.5 and 2.3
Theorem 2. There are only $O(h^2)$ node pairs considered in the procedure of generating the node pair set and thus there are $O(h^2)$ in the node pair set of SE.

Finally, we adopt a standard hashing technique, namely the perfect hashing scheme [7], to index all node pairs in the node pair set of SE. The hashing technique takes a linear space and requires a linear preprocessing time in expectation in terms of the number of the node pairs in the node pair set of SE. Given two nodes $O$ and $O'$ in the compressed partition tree, we could check whether there exists a node pair $(O, O')$ in the node pair set of SE in constant time and if so, it could also return the associated geodesic distance $d_g(O, O')$ in constant time.

3.4 Query Processing

Next, we present how we use our distance oracle SE for a distance query with a source point $s \in P$ and a destination point $t \in P$.

We first present one naive method, whose time complexity is $O(h^2)$ for this distance query. Next, we present an efficient algorithm whose time complexity is $O(h)$.

Naive Method: Before we introduce the naive method, we give some notations first. Let $O_{root}$ be the root node of the compressed partition tree. By our notation convention, we know that $O_i$ denotes the corresponding leaf node of point $s \in P$ in the compressed partition tree. Let $A_i$ be the array of size $h + 1$ where $A_i[i]$ is equal to the node in Layer $i$ along the path from $O_i$ to $O_{root}$ in the compressed partition tree if there exists a node in Layer $i$ and is equal to 0 otherwise for each $i \in [0, h]$. We have another notation $A_s$, which has a definition similar to $A_i$, and involves the path starting from $O_i$ instead of $O_s$. We denote the Cartesian product between the set of all nodes in $A_i$ and the set of all nodes in $A_s$ by $A_i \times A_s$. It is easy to have the following observation from Theorem 1: there exists exactly one pair $(O, O') \in A_i \times A_s$ such that $(O, O')$ contains $(s, t)$ and $(O, O')$ is in the node pair set of our SE.

Based on this observation, we have the following naive method for a distance query. Firstly, we find a leaf node $O_i$ and a leaf node $O_s$. Then, we construct the node pairs $(O_i, O_s)$ to $O_{root}$. Secondly, for each node $O \in A_i$ and each node $O' \in A_s$, we check whether node pair $(O, O')$ is in the node pair set of SE. If so, we return the distance associated with $(O, O')$. Otherwise, we continue to check the next node pair.

Note that by this observation, the above naive method must return one distance value (associated with one node pair) at the end.

The correctness of the naive method (i.e., the $c$-approximation) comes naturally from Theorem 1. It is easy to verify that the time complexity of the naive method is $O(h^2)$ since the first step takes $O(h)$ time and the second step takes $O(h^2)$ time (because the second step involves $O(h^2)$ node pairs and each node pair requires to be checked with its existence in the node pair set of our SE in $O(1)$ time using the perfect hashing scheme).

Efficient Method: Next, we will present our efficient algorithm for the distance query which takes $O(h)$ time. Before we present the algorithm, we give some concepts first.

Let Layer($O$) be the layer number of the layer containing node $O$.

We categorize node pairs $(O, O')$ into one of three types. A node pair $(O, O')$ is said to be a same-layer node pair if $O$ has the same layer as $O'$ in the compressed partition tree (i.e., Layer($O$) = Layer($O'$)). A node pair $(O, O')$ is said to be a first-higher-layer node pair if $O$ has a higher layer than $O'$ in the compressed partition tree (i.e., Layer($O$) < Layer($O'$)). A node pair $(O, O')$ is said to be a first-lower-layer node pair if $O$ has a lower layer than $O'$ in the compressed partition tree (i.e., Layer($O$) > Layer($O'$)).

Consider the compressed partition tree as shown in Figure 6. The node pair $(O_{14}, O_{15})$ is a same-layer node pair. The node pair $(O_{14}, O_{20})$ is a first-higher-layer node pair and the node pair $(O_{6}, O_{15})$ is a first-lower-layer node pair.

By definition, in a same-layer node pair $(O, O')$, both node $O$ and node $O'$ are in the same layer. We know that in a first-higher-layer node pair $(O, O')$, since node $O$ has a higher layer than node $O'$, we know that there exists a layer higher than the layer containing node $O'$, and thus we deduce that $O'$ has a parent node in the compressed partition tree. With the following lemma, interestingly, we know that the layer containing the parent of node $O'$ is equal to or higher than the layer containing node $O$. We could have a similar conclusion for a first-lower-layer node pair.

Lemma 3. Consider a node pair $(O, O')$ in the node pair set of our SE. If $(O, O')$ is a first-higher-layer node pair, then the layer containing the parent of node $O'$ is equal to or higher than the layer containing node $O$. If $(O, O')$ is a first-lower-layer node pair, then the layer containing the parent of node $O$ is equal to or higher than the layer containing node $O$.

Consider the compressed partition tree as shown in Figure 6. The error parameter $\epsilon$ is set to 2. Note that for illustration purpose, this error parameter is set to 2 but in practice, it should be set to a smaller value (e.g., 0.1) as what we did in our experimental studies. The node pairs $(O_{14}, O_{7})$ and $(O_{16}, O_{12})$ are both first-higher-layer node pairs in the node pair set of our SE. The parent of $O_{2}$ ($O_{7}$) is $O_{15}$ ($O_{20}$). The layer containing $O_{15}$ is the same as that containing $O_{14}$ and the layer containing $O_{20}$ is higher than that containing $O_{16}$. Similar illustrations could be made to the two first-lower-layer node pairs in the node pair set of our SE, namely $(O_{6}, O_{15})$ and $(O_{13}, O_{20})$, in a symmetric way.

Let parent($O$) be the parent of node $O$ in the compressed partition tree.

With Lemma 3, we have the following observation.

Observation 1. Consider a node pair $(O, O')$ in the node pair set of our SE. If $(O, O')$ is a first-higher-layer node pair, then Layer(parent($O'$)) $\leq$ Layer($O$) $<$ Layer($O'$). If $(O, O')$ is a first-lower-layer node pair, then Layer(parent($O$)) $\leq$ Layer($O'$) $<$ Layer($O$).

Consider the compressed partition tree as shown in Figure 6. The node pairs $(O_{14}, O_{7})$ and $(O_{16}, O_{12})$ are both first-higher-layer node pairs in the node pair set of our SE. parent($O_{2}$) (parent($O_{7}$)) is $O_{15}$ ($O_{20}$). It is clear that Layer(parent($O_{2}$)) $\leq$ Layer($O_{14}$) $<$ Layer($O_{2}$) and Layer(parent($O_{7}$)) $\leq$ Layer($O_{16}$) $<$ Layer($O_{12}$). Similar illustrations could be made to the two first-layer-lower node pairs in the node pair set of our SE, namely $(O_{6}, O_{15})$ and $(O_{13}, O_{20})$, in a symmetric way.

Based on Observation 1, we give the major idea why we could have an efficient algorithm. Note that the naive method requires that $O(h^2)$ node pairs should be enumerated. However, our efficient method just needs to enumerate $O(h)$ node pairs. Specifically, our efficient method involves three steps. Roughly speaking, the first step handles same-layer node pairs in $A_i \times A_s$, the second step handles first-higher-layer node pairs in $A_i \times A_s$, and the third step handles first-lower-layer node pairs in $A_i \times A_s$. 
Specifically, the first step checks whether there exists a node \( O \) in \( A_i \) and a node \( O' \) in \( A_i \) such that \( (O, O') \) is a same-layer node pair and \( (O, O') \) is in the node pair set of \( SE \). If there exists such a node pair \( (O, O') \), we return the distance associated with \( (O, O') \). This can be done in \( O(h) \) time by linearly scanning both arrays \( A_i \) and \( A_i \) from index 0 through \( h \) and checking whether \( (A_i[j], A_i[j]) \) is in the node pair set of \( SE \) where \( i \in [0, h] \) (note that \( (A_i[j], A_i[j]) \) is a same-layer node pair). The second step is to check whether there exists a node \( N \) in \( A_i \) and a node \( N' \) in \( A_i \) such that \( (O, O') \) is a first-higher-layer node pair and \( (O, O') \) is in the node pair set of \( SE \). If there exists such a node pair \( (O, O') \), we return the distance associated with \( (O, O') \). This can be done in \( O(h) \) time by following the above sub-steps. For each \( i \in [1, h] \), if \( A_i[j] \neq \emptyset \), then we obtain the layer number \( j \) of the layer containing the parent of \( A_i[j] \) in \( (O(1) \text{ time}) \), and, for each \( k \in [j, i] \), check whether \( (A_i[k], A_i[k]) \) is in the node pair set of \( SE \) (in \( O(j - i) \) time) (note that it is sufficient to scan to check \( (A_i[j], A_i[j]), (A_i[j + 1], A_i[j]), \ldots, (A_i[i - 1], A_i[i]) \) for one particular node \( A_i[j] \) in \( A_i \) based on Observation 1). It is easy to verify that the second step takes \( O(h) \) time since we can scan \( O(h) \) elements in \( A_i \) and \( O(h) \) elements in \( A_i \). The third step is similar to the second step, but this step focuses on the first-lowest-layer node pairs instead of the first-higher-layer node pairs. Details are skipped here since similar descriptions are applied. Thus, the overall time complexity of the efficient method is \( O(h) \).

**Example 3 (Query Processing).** The error parameter \( \epsilon \) is set to 2. Consider the example as shown at the left hand side in Figure 7, where \( O_i \) is \( O_1 \) and \( O_i \) is \( O_{10} \). It shows all edges and all nodes along the path from the leaf node \( O_i \) with its center \( p_i \) to the root node and the path from the leaf node \( O_i \) with its center \( p_i \) to the root node. The pair \( (O_1, O_{10}) \) containing \( (O_1, O_{10}) \) is the pair in the node pair set of \( SE \). In this example, \( A_i = [O_{21}, O_{13}, O_{1}, O_{10}] \) and \( A_i = [O_{21}, O_{13}, O_{1}, O_{10}] \). Consider the figure at the right hand side in Figure 7. All node pairs processed in the query processing algorithm are shown in the form of node pairs connected by lines (which are solid lines, thin dashed lines and thick dashed lines). Specifically, each node pair connected by a solid line is a same-layer node pair processed. Each node pair connected by a thin dashed line is a first-higher-layer node pair processed. Each node pair connected by a thick dashed line is a first-lowest-layer node pair processed. Our query algorithm checks all the three types of node pairs. When one of the node pairs processed in the node pair set of \( SE \), we return the distance associated with this node pair. It is worth mentioning that the total number of lines in this figure corresponds to the greatest number of node pairs processed, which is equal to \( O(h) \) instead of \( O(h^2) \) (denoting the total number of lines in a complete bipartite graph between \( A_i \) and \( A_i \)). Thus, the query step is very efficient.

It is easy to verify that the distance returned by the efficient method is \( \epsilon \)-approximate based on Theorem 1.

### 3.5 Oracle Construction

In this section, we first present a naive method of constructing \( SE \) and then present an efficient method of constructing \( SE \).

**Naive Method:** We first present a naive method of constructing \( SE \). First, we build a partition tree \( T_{org} \). Then, we build a compressed partition tree \( T_{compress} \) based on \( T_{org} \) and delete \( T_{org} \). Next, we follow the procedure described in Section 3.3 to generate all node pairs for the node pair set. Note that for each node pair considered, we have to compute the distance between the two nodes in the node pair. In the naive method, for each node pair considered, we perform the SSAD algorithm, which takes the center of one node in the node pair as an input of the starting point and performs the search until it reaches the center of the other node in the node pair.

We proceed to analyze the running time of the naive method. It takes \( O(hn^2) \) time to build \( T_{org} \) since there are \( O(nh) \) nodes in \( T_{org} \) and each node has to perform the SSAD algorithm which takes the center of this node as an input of the starting point and performs the search until it reaches a certain radius in \( O(Nn^2) \) time. It takes \( O(nh) \) time to construct \( T_{compress} \), since \( T_{compress} \) could be constructed with a postorder traversal of \( T_{org} \) and there are \( O(nh) \) nodes in \( T_{org} \). For each node pair \( (O, O') \) generated, we need to perform the SSAD algorithm which takes the center of one node in the node pair as an input of the starting point and performs the search until it reaches the center of the other node in the node pair to compute \( d_p(O, O') \). Thus, the total running time of generating the node pair set is \( O(nhNn^2) \). In conclusion, the total running time of the naive method of constructing \( SE \) is \( O(nhNn^2) \).

**Efficient Method:** Since the naive method takes \( O(nhNn^2) \) time to construct the \( SE \) distance oracle, which is very costly, we propose an efficient algorithm of constructing \( SE \) next. The major reason why the naive method is slow is that in the naive method, for each node pair considered in the procedure described in Section 3.3, the naive method has to perform an expensive SSAD algorithm, and thus the number of times that the SSAD algorithm is called is equal to the number of node pairs considered. However, we will present an efficient algorithm which could reduce the number of times that the SSAD algorithm is called from the total number of node pairs considered to the total number of nodes in the (original) partition tree by using a new concept called an enhanced node pair (which is a node pair involving two nodes in the same layer of the (original) partition tree and satisfying a condition) (to be introduced later). Specifically, the efficient method has two major differences from the naive method. The first difference is that the efficient method includes an additional (pre-computation) step of computing the distance between the two nodes involved in each possible enhanced node pair. Although there are \( O(hn) \) possible enhanced node pairs and we have to compute the distances of these pairs, the total number of times that the SSAD algorithm is called in this additional step is just equal to the total number of nodes in the (original) partition tree (which is \( O(hn) \)). The second difference is that the efficient method finds the distance of each node pair \( (O, O') \) considered in the procedure described in Section 3.3 by searching one of the “pre-computed” distances of the enhanced node pairs containing the node pair \( (O, O') \) and assigning this distance (of the enhanced node pair found) to the distance of the node pair \( (O, O') \) (instead of performing the expensive SSAD algorithm). Note that the time complexities of both the search operation and the assignment operation are \( O(h) \) (to be shown later), which is much lower than the time complexity of the SSAD algorithm (i.e., \( O(nhNn^2) \)). Later, we will show that for each node pair \( (O, O') \) considered in the procedure described in Section 3.3, there exists one enhanced node pair containing the node pair \( (O, O') \), which is a key to the efficiency of the efficient method.

Before we present the efficient method, we define the concept of the enhanced node pair. Given two nodes \( O \) and \( O' \) in the (original) partition tree, \( (O, O') \) is said to be an enhanced node pair if \( O \) and \( O' \) are in the same layer of the (original) partition tree and \( d_p(O, O') < l \cdot T_Q \) where \( l = \frac{\epsilon}{2} + 10.0 \). Note that \( l \) is about 4 times the well-separated factor (i.e., \( \frac{1}{2} + 2 \)). The ratio of \( 4 (= 2 \cdot 2) \)
is split two parts. The first part (i.e., a ratio of 2) comes from the radius of the enlarged disk of a node O (defined in the definition of the well-separated pair) which is two times the radius of node O. The second part (i.e., another ratio of 2) comes from our design.

With the definition of the enhanced node pair, we give the following lemma which is used in our efficient method.

**Lemma 4.** Consider a node pair \(\langle O, O'\rangle\) considered in the procedure described in Section 3.3. There exists an enhanced node pair \(\langle \overline{O}, \overline{O'}\rangle\) such that (1) \(\langle \overline{O}, \overline{O'}\rangle\) contains \(\langle O, O'\rangle\), (2) \(c_{\overline{O}} = c_O\) and (3) \(c_{\overline{O'}} = c_{O'}\).

The major idea why we can design an efficient method compared with the naive method is that the efficient method is designed based on Lemma 4 using the concept of the enhanced node pair.

We present the efficient algorithm of constructing \(SE\) as follows.

- **Step 1 (Tree Construction):** We build the partition tree \(T_{\text{org}}\) and a compressed partition tree \(T_{\text{compress}}\) based on \(T_{\text{org}}\). \(T_{\text{compress}}\) just constructed becomes the first component of \(SE\).

- **Step 2 (Enhanced Edge Creation):** We insert all possible enhanced edges into \(T_{\text{org}}\). Specifically, for any two nodes \(O\) and \(O'\) in the same layer of the (original) partition tree \(T_{\text{org}}\), if \(\langle O, O'\rangle\) is an enhanced node pair, then we add an edge connecting them. We call an edge added in this step an enhanced edge. We associate a distance to each enhanced edge added. Specifically, for each enhanced edge connecting \(O\) and \(O'\), we associate the distance between these two nodes (i.e., \(d_{\langle CO, CO'\rangle}\)) with this edge. To construct all the enhanced edges together, for each node \(O\) in the partition tree, we perform the SSAD algorithm which takes \(c_O\) as an input of the source point and performs the search until the disk \(D(c_O, \cdot, r_O)\) is totally expanded.

- **Step 3 (Perfect Hash Construction):** We insert all enhanced edges into the perfect hash [7] (with an oracle building time and a space cost which are linear to the total number of edges in expectation).

- **Step 4 (Node Pair Set Generation):** We generate the node pair set, the second component of \(SE\), using \(T_{\text{org}}\) added with enhanced edges. Specifically, we follow the procedure described in Section 3.3 to generate all node pairs for the node pair set. However, we present a detailed implementation of how to compute \(d_{\langle CO, CO'\rangle}\) for each node pair \(\langle O, O'\rangle\) generated in the procedure. For each node pair \(\langle O, O'\rangle\) generated, we find an enhanced edge connecting a node \(O\) and a node \(O'\) in \(T_{\text{org}}\) such that (1) \(\langle O, O'\rangle\) is an enhanced node pair, (2) \(\langle \overline{O}, \overline{O'}\rangle\) contains \(\langle O, O'\rangle\), (3) \(c_{\overline{O}} = c_O\) and (4) \(c_{\overline{O'}} = c_{O'}\). (Note that by Lemma 4, there exists such an enhanced edge.) This step of finding an enhanced edge can be done in \(O(h)\) time by

  - (1) first obtaining \(c_O\) from \(O\) and \(c_{O'}\) from \(O'\) (in \(O(1)\) time)
  - (2) then accessing the corresponding leaf node \(\overline{O}\) of \(c_O\) and the corresponding leaf node \(\overline{O'}\) of \(c_{O'}\) (in \(O(1)\) time)
  - (3) traversing both the path \(P\) from \(O\) to the root node and the path \(P'\) from \(O'\) to the root node together starting from Layer \(h\) to Layer 0 to check whether the node \(\overline{O}\) being traversed along \(P\) and the node \(\overline{O'}\) being traversed along \(P'\) (in the same layer) have their node pair \(\langle \overline{O}, \overline{O'}\rangle\) found in the perfect hash (in \(O(h)\) time), and
  - (4) returning the enhanced node edge connecting \(\overline{O}\) and \(\overline{O'}\) (if these two nodes have their node pair \(\langle \overline{O}, \overline{O'}\rangle\)

found in the perfect hash) (in \(O(1)\) time).

Then, the distance associated with this enhanced edge corresponds to the distance we want (i.e., \(d_{\langle CO, CO'\rangle}\)).

**3.6 Theoretical Analysis**

Before analyzing \(SE\), we introduce a well-known concept called the largest capacity dimension originally defined on a metric space [21, 13]. For the sake of space, the definition and the discussion of the largest capacity dimension could be found in the appendix. In the appendix, we show that in an extreme case where the terrain surface is a 2D plane, the largest capacity dimension \(\beta\) is at most 1.3. In a general case, \(\beta\) is a little bit larger than 1.3 (since the terrain surface could be regarded as a 2D surface with some fluctuations in terms of height).

Our experimental results show that the largest capacity dimension \(\beta\) of the terrain surface that we considered is between 1.3 and 1.5.

Then, we present the oracle building time, oracle size, query time and distance error bound of our \(SE\) in the following theorem.

**Theorem 3.** The oracle building time, oracle size, query time and distance error bound of \(SE\) are \(O\left(\frac{N^2 \log^2 N}{\epsilon^2 25} + nh \log n + \frac{nh^2}{25}\right), O\left(\frac{N}{\epsilon^2 25}\right), O(h)\) and \(\epsilon\), respectively.

**4. RELATED WORK AND BASINES**

In this section, we present the related work and baseline methods in Section 4.1 and Section 4.2, respectively.

**4.1 Related Work**

The existing studies of finding the exact geodesic distance between two vertices are [26, 6] and [34]. Their time complexities are \(O(N^2 \log N), O(N^2), O(N \log^2 N)\) and \(O(N^2 \log N)\), respectively, which are impractical even on moderate terrain data.

Motivated by the intrinsic expensive cost of computing exact geodesic distances, many existing studies focus on computing approximate geodesic distances [24, 20, 19]. In [24], the authors studied the problem of finding an approximate geodesic shortest path which satisfies a slope constraint. In [20], the authors proposed an algorithm for finding a geodesic path between two points satisfying a condition on the terrain surface and computing the lower and upper bounds of the geodesic shortest distance based on the length of the path found, but the gap between the bounds depends on the structure of the terrain surface, and thus it could be very large implying that there exists no guarantee on the qualities of the bounds. In [19], the authors proposed a Steiner point-based algorithm introducing additional points called Steiner points on the surface of the terrain for finding an \(\epsilon\)-approximate geodesic shortest path between two points, where \(\epsilon\) is a user-specified parameter. The algorithm computes tighter lower and upper bounds of the geodesic distance than those of [20], which do not depend on the underlying terrain. According to the experimental results in [19], the algorithm ran more than 300 seconds even for a setting with a very loose error parameter \(\epsilon = 0.25\). All of these algorithms compute the approximate geodesic distances on-the-fly, which is not efficient enough in (real-time) applications involving many distance queries.

In order to answer the geodesic shortest path/distance queries more efficiently, some existing studies aim at designing oracles [18, 2, 3, 12]. [18] proposed a data structure for the Single-Source All-Destination (SSAD) approximate geodesic shortest path queries, where the source point of each shortest path query is already given before the data structure is built. This data structure could answer
any shortest path query from this fixed source point to any destination. However, this data structure is limited to a fixed source point. Even though different data structures from all possible source points could be built, the total space occupied by all these data structures is prohibitively large, which is not feasible in practice. [2, 3] designed an oracle for approximate geodesic shortest path queries and [12] designed an oracle for approximate geodesic shortest distance queries. These two oracles share similar ideas, and the one in [12] is better in terms of oracle size and query time mainly because geodesic distance queries are intrinsically easier than geodesic path queries. Specifically, the oracle in [12] has its space complexity of \(O(\epsilon^2 \log^2 N + n \log n)\) and its query time complexity of \(O(\epsilon^2 \log^2 N + n \log n)\) and its query time complexity of \(O(\epsilon^2 \log^2 N + n \log n)\) and its query time complexity of \(O(\epsilon^2 \log^2 N + n \log n)\), where \(\theta\) is the minimum inner angle of any face on the terrain surface.

As will be introduced later, we use this oracle as a baseline oracle for comparison, and our experimental results show that this oracle has a scalability issue due to its large oracle size, and its corresponding query time is significantly larger than that of our oracle.

Some other related studies include those proximity queries relying on the geodesic shortest distance queries [9, 10, 29, 35, 36]. Specifically, [9, 10, 29] studied \(k\)-NN queries, [35] studied dynamic \(k\)-NN queries and [36] studied reverse nearest neighbor queries.

Besides, some studies [5, 14, 27] focused on studying well-separated pairs. [5] studied it in the Euclidean space, [14] studied its dynamic case (e.g., insertion and deletion) and [27] studied it on road networks. However, they are different from ours because we studied it in the terrain context and different contexts give different challenges (e.g., in the terrain context, how to build a distance oracle involving many expensive geodesic distance computations is very challenging).

### 4.2 Baseline Methods

In this section, we first present two baseline oracles (Section 4.2.1), then give one baseline on-the-fly algorithm (Section 4.2.2) and finally compare them with SE (Section 4.2.3).

#### 4.2.1 Baseline Oracles

In this part, we first introduce two baseline oracles, namely the Steinor point-based oracle (in short, SP-Oracle) and the naive implementation of SE (in short, SE(Naive)).

**Steiner Point-Based Oracle:** The first baseline oracle is called the Steinor point-based oracle (in short, SP-Oracle) proposed in [12] which were originally proposed for vertex-to-vertex distance queries and could also be adapted for both POI-to-POI (P2P) distance queries and arbitrary point-to-arbitrary point (A2A) distance queries. Next, we describe how this adapted distance oracle [12] could handle A2A distance queries only (since A2A distance queries could be regarded as a general setting compared with P2P distance queries). Its major idea is as follows. It first introduces \(O(\epsilon^2 \log^2 N)\) additional points called Steinor points on each face of the terrain surface and \(O(\max(\sqrt{\frac{N}{\epsilon}}, \sqrt{\frac{n}{\epsilon}}))\).

**SE(Naive):** The second baseline is called the naive method of SE (in short, SE(Naive)) which is exactly our SE with the naive method for the both the oracle construction and the query processing. We present the oracle building time, oracle size, query time, and distance error bound of SE(Naive) in Table 1.

### 4.2.2 Baseline On-the-fly Algorithm

The Kaul’s algorithm (in short, K-Algo) recently proposed in [19] could be used as the baseline algorithm which computes the approximate geodesic distance on-the-fly (since it is the best-known algorithm in the literature). Although K-Algo is a non-distance oracle algorithm, it is interesting to compare it with our SE. The time complexity of K-Algo is \(O\left(\frac{\epsilon^2}{(\epsilon\log N + \sqrt{\epsilon \cdot \log N})} \right)\) where \(\epsilon\) is the length of the shortest (resp., longest) edge and \(\theta\) is the minimum inner angle of any face on the terrain surface.

\[\text{Building Time (s)} = O\left(\frac{\epsilon^2}{(\epsilon\log N + \sqrt{\epsilon \cdot \log N})} \right) + \frac{\epsilon^2}{(\epsilon\log N + \sqrt{\epsilon \cdot \log N})} \]

\[\text{Oracle Size} = O\left(\frac{\epsilon^2}{(\epsilon\log N + \sqrt{\epsilon \cdot \log N})} \right) + \frac{\epsilon^2}{(\epsilon\log N + \sqrt{\epsilon \cdot \log N})} \]

\[\text{Query Time} = O\left(\frac{\epsilon^2}{(\epsilon\log N + \sqrt{\epsilon \cdot \log N})} \right) + \frac{\epsilon^2}{(\epsilon\log N + \sqrt{\epsilon \cdot \log N})} \]

### Table 1: Comparison of Different Methods with Error Bound \(\epsilon\) (where \(\beta \in [1.3, 1.5]\) and \(h < 30\) in practice)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. of Vertices</th>
<th>Resolution</th>
<th>Region Covered</th>
<th>No. of POIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td>1.4M</td>
<td>10 meters</td>
<td>14.4km x 10km</td>
<td>4k</td>
</tr>
<tr>
<td>EP</td>
<td>1.5M</td>
<td>10 meters</td>
<td>10.7km x 14km</td>
<td>4k</td>
</tr>
<tr>
<td>SF</td>
<td>175k</td>
<td>30 meters</td>
<td>14km x 11.1km</td>
<td>51k</td>
</tr>
</tbody>
</table>

### Table 2: Dataset Statistics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. of Vertices</th>
<th>Resolution</th>
<th>Region Covered</th>
<th>No. of POIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td>1.4M</td>
<td>10 meters</td>
<td>14.4km x 10km</td>
<td>4k</td>
</tr>
<tr>
<td>EP</td>
<td>1.5M</td>
<td>10 meters</td>
<td>10.7km x 14km</td>
<td>4k</td>
</tr>
<tr>
<td>SF</td>
<td>175k</td>
<td>30 meters</td>
<td>14km x 11.1km</td>
<td>51k</td>
</tr>
</tbody>
</table>

### Figure 8: Effect of \(\epsilon\) on SF dataset (Smaller Version) (P2P Distance Queries)
inner angle of any face.

4.2.3 Comparison

We compare the oracle proposed in this paper, i.e., SE, and the three baselines, i.e., SP-Oracle, SE(Naive) and K-Algo, in terms of error bound, oracle building time, oracle size and query time, and the results are shown in Table 1. We highlight some of the comparison results as follows. Consider the error bound. Our SE and all baseline methods, namely SP-Oracle, SE(Naive) and K-Algo, have the same error bound equal to \( \epsilon \). Consider the oracle building time. As described before, we know that SE has a lower oracle building (or oracle construction) time complexity than SE(Naive). Besides, in our experimental results, the empirical oracle building time of SE is smaller than that of SP-Oracle. Consider the oracle size. The oracle size of SE is the same as that of SE(Naive). Besides, in our experimental results, the empirical oracle size of SE is smaller than that of SP-Oracle. Consider the query time. Since \( h \) is very small (at most 30 in our experimental results), SE has the lowest query time complexity compared with SE(Naive) and SP-Oracle. K-algo has the largest query time which is significantly larger than others.

5. EMPIRICAL STUDIES

5.1 Experimental Setup

We conducted our experiments on a Linux machine with 2.67 GHz CPU and 48GB memory. All algorithms were implemented in C++.

Datasets. Following some existing studies on terrain data [29, 9, 24], we used three real terrain datasets, namely BearHead (in short, BH), EaglePeak (in short, EP) and San Francisco South (in short, SF) and these datasets can be downloaded from http://data.geocomm.com/. For each of these terrain datasets, we extracted a set of POIs from the corresponding region in OpenStreetMap. Table 2 shows the dataset statistics. Besides, a smaller version of SF dataset which corresponds to a small sub-region of the SF dataset and contains 1k vertices and 60 POIs was also used since one of the baselines, SE-Naive, is not feasible on any of the full datasets due to its expensive cost of building an oracle.

Algorithms. Our new oracle SE and three baselines, SP-Oracle [12], K-Algo [19] and SE-Naive, are studied in the experiments. For SE, we study two variations: one is SE(Greedy) which is based on the greedy point selection strategy and the other is SE(Random) which is based on the random point selection strategy.

Query Generation. Each P2P (V2V) query was generated by randomly sampling two POIs (vertices) on the surface of a terrain, one as a source and the other as a destination. Each A2A query was generated by randomly selecting two POIs (vertices) on the terrain, were studied. Four measurements, namely \( SE(Naive) \) and \( SE(Greedy) \), were used for evaluating the oracles. For the query time, 100 queries were answered and the average running time was returned.

5.2 Experimental Results

In this section, we present the results of P2P distance queries in Section 5.2.1, other experiments (e.g., V2V distance queries and A2A distance queries) in Section 5.2.2, and a summary of the results in Section 5.2.3.

5.2.1 P2P Distance Queries

Effect of \( \epsilon \). We tested 5 different values of \( \epsilon \) from \{0.05, 0.1, 0.15, 0.2, 0.25\}. Figure 8(a)-(d) show the results on the smaller version of the SF dataset. According to the results, (1) the building times of SE(Greedy) and SE(Random) are almost the same and are both smaller than those of SP-Oracle and SE-Naive, e.g., when \( \epsilon = 0.05 \), SE(Greedy) and SE(Random) have their building times 1 order (resp., at least 2 orders) smaller than that of SP-Oracle (resp., SE-Naive), (2) the sizes of SE(Greedy), SE(Random) and SE-Naive are 2-3 orders of magnitude smaller than that of SP-Oracle, (3) the query time of SE(Greedy) is the smallest and about half of that of SE(Random), and the query times of both SE(Greedy) and SE(Random) are orders of magnitude smaller than those of others, and (4) the errors of all oracles are very small and much smaller than the theoretical bound (which is \( \epsilon \)).

Based on the results shown above, we adopt the following for the simplicity of presentation: (1) the results of error for the rest of experiments are omitted since the errors of all oracles are similar and very small (smaller than \( \epsilon/10 \) compared with the error bound, (2) the results of SE-Naive on any full datasets are not shown simply because it cannot be built within a reasonable amount of time, e.g., within a month, and (3) the results of SE(Greedy) are omitted for the rest of experiments since SE(Random) and SE(Greedy) have similar performance and we omit SE(Greedy) for the clarity and by SE, it means SE(Random) for the remaining presentation.

The results on the other two datasets, namely BH and EP, are shown in Figure 13 and Figure 14, respectively, where the results of SP-Oracle for all settings of \( \epsilon \) are not shown since the size of SP-Oracle exceeds our memory budget (i.e., 48GB). Since the results on the BH and EP datasets are similar to those on the SF datasets, for the sake of space, they could be found in [33].

Effect of \( n \). We tested 5 different values of \( n \) from \{60k, 90k, 120k, 150k, 180k\} and used the SF dataset for this ex-

<table>
<thead>
<tr>
<th>Dataset</th>
<th>max</th>
<th>min</th>
<th>avg</th>
<th>std</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td>128.7</td>
<td>95.3</td>
<td>102.5</td>
<td>3.15</td>
</tr>
<tr>
<td>EP</td>
<td>14.15</td>
<td>0.33</td>
<td>6.25</td>
<td>3.15</td>
</tr>
<tr>
<td>SF</td>
<td>10.92</td>
<td>0.48</td>
<td>7.09</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Table 3: Statistics of Query Distances (km)

Figure 11: Effect of \( n \) on SF dataset (V2V Distance Queries)
As mentioned in Section 5.1, we have 51k POIs in the SF South dataset (170k vertices), and in order to obtain a set of the targeted number of POIs, we do as follows. Let \( n \) denote the targeted number of POIs we want to generate. Let \( P \) be the set of POIs that we have and \( n' \) be the number of POIs in \( P \). We generate \((n - n')\) 2-dimensional points \((x, y)\) based on a Normal distribution \(N(\mu, \sigma^2)\), where \( \mu = (\bar{x} = \frac{\sum_{p' \in P} x_{p'}}{n}, \bar{y} = \frac{\sum_{p' \in P} y_{p'}}{n'}) \) and \( \sigma^2 = \left( \frac{1}{n} \sum_{p' \in P} (x_{p'} - \bar{x})^2, \frac{1}{n} \sum_{p' \in P} (y_{p'} - \bar{y})^2 \right) \). If a generated point \((x, y)\) is outside the range of the terrain, we simply discard it and re-do the process until a point within the range is generated.

At the end, we project each generated point \((x, y)\) to the surface of the terrain and take the projected point as a newly generated POI. The results are shown in Figure 9. According to these results, our oracle \( SE \) outperforms \( SP-Oracle \) in terms of oracle building time, oracle size and query time and significantly outperforms \( K-Algo \) in terms of query time.

### Effect of \( N \)

We tested 5 values of \( N \) from \( \{0.5M, 1M, 1.5M, 2M, 2.5M\} \) on synthetic datasets. Each synthetic dataset with \( N \) vertices is a terrain surface from an enlarged BH dataset (4.2M vertices) simplified by a surface simplification algorithm [24]. Note that each simplified terrain surface covers the same region as the original BH dataset with a different simplification ratio and still has 4k POIs. The enlarged BH dataset was generated from the BH dataset as follows. On each face of BH, we add a new vertex on its geometric center and add a new edge between the new vertex and each of the three vertices on the face. The results are shown in Figure 10, where the results of \( SP-Oracle \) are not shown since the size of \( SP-Oracle \) exceeds our memory budget (i.e., 48GB).

### 5.2.2 Other Experiments

#### V2V Distance Queries

In V2V queries, the original POIs are discarded, and we treat all vertices as POIs. We varied \( n \) and \( \epsilon \) for the experiments. Consider the experiment studying the effect of \( n \). Note that \( N = n \) in this experiment. We tested 5 values of \( n \) (i.e., \( N \)) from \( \{60k, 90k, 120k, 150k, 180k\} \) on synthetic datasets, and each synthetic dataset with \( N \) vertices corresponds to a sub-region of a SF dataset with a higher resolution (10m x 10m resolution, 1M vertices). The results are shown in Figure 11, and according to the results, \( SE \) has its building time and size both at least \( 1 \) order of magnitude smaller than \( SP-Oracle \) and its query time \( 2-3 \) (resp., \( 5-6 \)) orders of magnitude smaller than that of \( SP-Oracle \) (resp., \( K-Algo \)).

We also conducted the experiment studying the effect of \( \epsilon \) with values in \( \{0.05, 0.1, 0.15, 0.2, 0.25\} \) on the smaller version of the SF dataset. The results are also similar. In particular, the query time of \( SE \) is \( 5-6 \) orders (resp., \( 6-8 \) orders) of magnitude smaller than that of \( SP-Oracle \) (resp., \( K-Algo \)).

#### Arbitrary Point to Arbitrary Point (A2A) Queries

We tested the A2A distance queries where the query point is not a POI but an arbitrary point on the terrain surface. We used the low resolution BH (resolution: 30 meter, 150k vertices) dataset by varying \( \epsilon \) from \( \{0.05, 0.1, 0.15, 0.2, 0.25\} \). Figure 12(a), (b), and (c) shows the building time, oracle size and query time, respectively. According to the results, \( SE \) consistently outperforms \( SP-Oracle \) by several times in terms of building time and size. The query time of \( SE \) is \( 2-3 \) (resp., \( 5-6 \) orders of magnitude smaller than that of \( SP-Oracle \) (resp., \( K-Algo \)).

**5.2.3 Experimental Result Summary**

Our \( SE \) consistently outperforms the state-of-the-art oracle, i.e., \( SP-Oracle \), in terms of all measurements (i.e., building time, oracle size, and query time) and for any types of distance queries (i.e., P2P queries, V2V queries and A2A queries).

### 6. CONCLUSION

In this paper, we studied an important spatial query, the shortest distance query, which is fundamental to many other spatial queries and many data mining applications. We proposed a distance oracle called \( SE \) which have three good features: (1) low construction time, (2) small size and (3) low query time (compared with the best-known oracle [12]). Our experimental studies show that \( SE \) consistently outperforms than the best-known algorithm, \( SP-Oracle \), in terms of all measurements for P2P queries, V2V queries and also A2A queries. There are several interesting research directions. One of them is to study how to efficiently update the distance oracle when there is an update on some POIs.

**Acknowledgements:** We are grateful to the anonymous reviewers for their constructive comments on this paper. The research of Victor Junqiu Wei and Raymond Chi-Wing Wong is supported by HKRGC GRF 16219816. The research of David M. Mount is supported by NSF CCF-1618866.


B. Callahan, R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. JACM, 1995.


S.-Q. Xin and G.-J. Wang. Improving Chen and Han’s algorithm on shortest paths on terrains by killing two birds with one stone. VLDB, 2013.

A. LARGEST CAPACITY DIMENSION

Consider a metric space X with a distance metric d(·, ·). Given a positive real number r, a set Y ⊆ X is said to be r-separated if for any two distinct points x, y ∈ Y, d(x, y) ≥ r. Given a positive real number r and a set S ⊆ X, the r-packing number of S, denoted by M(r, S), is defined to be the maximum cardinality of an r-separated subset of S. Given a metric space X and a distance measure d, a set S ⊆ X and two positive real numbers r1 and r2, the scale-dependent capacity dimension of S w.r.t. r1 and r2, denoted by D(S, r1, r2), is defined to be $\frac{\log M(r_1, S)}{\log r_2 - \log r_1}$. This dimension is used to measure the ‘intrinsic dimension’ of a metric space. Many high-dimensional data points are believed to be distributed in a manifold with a low ‘intrinsic dimension’. Intuitively, the ‘intrinsic dimension’ is the number of independent variables needed to represent the whole dataset. A good estimate of the ‘intrinsic dimension’ could be used to set the input parameters of the dimension reduction algorithms (e.g., Principle Component Analysis).

Thus, we obtain that given a disk $B(p, r)$ on a terrain surface. Given a point $p$ in $P$ and a non-negative real number $r$, a ball centered at $p$ with radius equal to $r$, denoted by $B(p, r)$, is defined to be a set of all POIs in the disk $D(p, r)$. Then, we give the definition of the capacity dimension of a ball $B(p, r)$ on the terrain surface which only measures the growth rate of $M(r, B(p, r))$ when $r$ reduces from $2r$ to $\frac{r}{2}$.

**Definition 1.** Given a ball $B(p, r)$ on a terrain surface, where $p$ is a point in $P$ and $r$ is a positive real number, the capacity dimension of $B(p, r)$ is defined to be $D(B(p, r), 2r, \frac{r}{2}) = \log \frac{M(2r, B(p, r))}{M(r, B(p, r))} = 0.5 \log \frac{M(2r, B(p, r))}{M(r, B(p, r))}$.

Consider a set of points whose pairwise geodesic distances are at least $2r$. Since the disk $D(p, r)$ could overlap with at most 2 of them, we obtain that $M(2r, B(p, r)) = 2$. Thus, for a given ball $B(p, r)$, the capacity dimension is equal to $0.5 \log \frac{M(2r, B(p, r))}{M(r, B(p, r))}$. Thus, we obtain that given a disk $D(p, r)$, $2 \cdot 2^{\frac{r}{2}D(B(p, r), 2r, \frac{r}{2})}$ is the maximum number of POIs whose pairwise geodesic distances are at least $\frac{r}{2}$ such that the disk $D(p, r)$ contains them.
Next, we define the largest capacity dimension of a set $P$ of POIs on a terrain surface to be $\max_{p \in P, r \in (0, \infty)} D(B(p, r), 2r, T)$, denoted by $\beta$. By the definition of the largest capacity dimension, any disk $D(p, r)$, where $p$ is a point in $P$ and $r$ is a positive real number, could contain at most $2 \cdot 2^{2d}$ (i.e., $O(2^{2d})$) POIs whose pairwise geodesic distance is at least $2r$.

The definition of largest capacity dimension has an equivalent presentation as follows. Given a set $P$ of POIs on a terrain surface, its largest capacity dimension $\beta$ is the largest positive real number such that for any point $p$ in $P$ and any non-negative real number $r$, the disk $D(p, r)$ overlaps with at most $2 \cdot 2^{2d}$ disjoint disks each with radius at least $\frac{r}{2}$. Based on this, we obtain that in an extreme case where the terrain surface is a 2D plane, the largest capacity dimension $\beta$ is at most 1.3. In general cases, intuitively, $\beta$ is a little bit larger than 1.3, since the terrain surface could be regarded as a 2D surface with some fluctuations in terms of height.

**B. PROOF**

**Lemma 5.** Given disks $D_1(c_1, r_1)$ and $D_2(c_2, r_2)$ where (1) $c_1$ and $c_2$ are two points in $P$ and (2) $r_1$ and $r_2$ are two non-negative real numbers, for any point $p_1$ in $D_1$ and any point $p_2$ in $D_2$, $d_{\alpha}(c_1, c_2)$ is an $\alpha$-approximate distance of $d_{\alpha}(p_1, p_2)$ if $d_{\alpha}(c_1, c_2) \geq \left(\frac{\alpha}{\alpha - 1}\right) \cdot \max\{r_1, r_2\}$. 

**Proof.** By Triangle Inequality, we obtain that $d_{\alpha}(p_1, p_2) - r_1 - r_2 \leq d_{\alpha}(p_1, c_1) + d_{\alpha}(c_1, c_2) + d_{\alpha}(c_2, p_2) \leq d_{\alpha}(p_1, p_2) + \epsilon \cdot d_{\alpha}(p_1, c_1) + d_{\alpha}(c_1, c_2) + d_{\alpha}(c_2, p_2)$. Furthermore, we obtain that $d_{\alpha}(p_1, p_2) - r_1 - r_2 \leq d_{\alpha}(p_1, p_2) + \epsilon \cdot d_{\alpha}(p_1, c_1) + d_{\alpha}(c_1, c_2) + d_{\alpha}(c_2, p_2)$.

**Proof of Lemma 1.** By Step (b) in the partition tree construction algorithm, the tree built satisfies Separation Property and Covering Property. Then, we prove that it also satisfies the distance property: Consider a node $O$ and any of its descendents $O'$. Let $(O, O_1, O_2, O_3, ..., O_i, O')$ denote the path from $O$ to $O'$ in the partition tree. By Separation Property, we obtain that $r_{O_i} = \frac{T}{2^i} \cdot \frac{T}{2^{i - 1}} \leq t$, $r_{O_i} = 2 \cdot r_{O_i}^o$. By our building method, we obtain that $d_{\alpha}(c_{O_i}, c_{O_{i + 1}}) \leq r_{O_i} \leq t - 1$. Thus, for all $i = 1, ..., \alpha - 1$, $d_{\alpha}(c_{O_i}, c_{O_{i + 1}}) \leq r_{O_i}$. By Triangle Inequality, we obtain that $d_{\alpha}(c_{O_i}, c_{O_{i + 1}}) \leq d_{\alpha}(c_{O_i}, c_{O_{i + 1}}) + d_{\alpha}(c_{O_{i + 1}}, c_{O_{i + 2}})$ and $d_{\alpha}(c_{O_i}, c_{O_{i + 1}}) \leq d_{\alpha}(c_{O_i}, c_{O_{i + 1}}) + d_{\alpha}(c_{O_{i + 1}}, c_{O_{i + 2}})$. Integrating all the inequalities above, we obtain that $d_{\alpha}(c_{O_i}, c_{O_{N + 1}}) \leq \frac{\alpha}{\alpha - 1} \cdot \frac{T}{2^{N + 1}} \leq \frac{T}{2} - 2$.

**Lemma 6.** Consider a chain of node pairs $(O_1, O'_1), (O_2, O'_2), ..., (O_i, O'_i), ..., (O_m, O'_m)$, where $(O_i, O'_i)$ is generated by $(O_{i - 1}, O'_{i - 1})$ for each integer $i \in [2, m]$. Let $r_{O_i}^t$ denote $\max\{r_{O_i}, r_{O'_i}\}$ for each integer $i \in [1, m]$. Then, we obtain that $r_{O_i}^t \geq r_{O'_j}^t$, since otherwise, we would not be able to prove the existence of a first-higher-layer node pair. We denote the parent of $O$ in $T_{\text{compres}}$ by $\text{parent}(O)$, then there exists an integer $k$ such that $k \in [1, m - 1]$ and parent($O'$) is split from $(O_k, O'_k)$. This is because otherwise, $(O, O')$ would not be generated. Consider the pair $(O_k, O'_k)$ from which parent($O'$) is split and thus, $r_{\text{parent}(O')} \geq \max\{r_{O_k}, r_{O'_k}\}$. By Lemma 6, we obtain that $r_{\text{parent}(O')} \geq \max\{r_{O_k}, r_{O'_k}\}$ and thus the layer containing parent($O'$) is higher than or equal to that containing $O$. For the case where $(O, O')$ is a first-lower-layer node pair, the proof is symmetric and we omit the details.

**Proof of Lemma 4.** Consider a node pair $(O, O')$ considered in the procedure described in Section 3.3, where $(O, O') \neq \langle O_{\text{root}}, O_{\text{root}} \rangle$. Let parent($O$) (resp., parent($O'$)) denote the parent of $O$ (resp., $O'$) in $T_{\text{compres}}$ if $O$ (resp., $O'$) is not the root node. It is obvious that $(O, O')$ is generated by $(\text{parent}(O), \text{parent}(O'))$ or $(\text{parent}(O), O')$.

W.l.o.g., we assume that $(O, O')$ is generated by $(\text{parent}(O), O')$. By Lemma 6, we obtain that $r_{\text{parent}(O') \leq r_{\text{parent}(O')}}$ since parent($O'$) is split before $(\text{parent}(O), O')$ is generated. By Lemma 3, we obtain that $r_{O} \leq r_{\text{parent}(O')}$ and $r_{O} \leq r_{\text{parent}(O') \leq r_{\text{parent}(O')}}$. Let $O_k$ denote the child of parent($O$) in the original partition tree $T_{\text{compres}}$, which is on the path from parent($O$) to $O$. There are two cases of $O_k$ and we present that in both cases, it is true that $r_{O_k} = c_{2O_k}$. If $O = O_k$, it is obvious that $r_{O_k} = c_{2O_k}$.
Then, consider the case where \( O \neq O \). Since any node on the path from \( C \) to \( O \) excluding \( O \) must have only one child, we obtain that \( c_{\beta} = c_{\alpha} \) by Step (i) of the building algorithm of \( T_{\text{org}} \). Similarly, the child \( O_{1} \) of \( \text{parent}(O) \) in the original partition tree \( T_{\text{org}} \) which is on the path from \( \text{parent}(O') \) to \( O_{1} \) has the same center with \( O \). Since \( r_{\text{parent}(O)} \leq r_{\text{parent}(O')} \), we obtain that \( r_{\alpha} \leq r_{\beta} \).

Then, consider the node \( C \) on the path from \( O_{1} \) to \( O \) in \( T_{\text{org}} \) which is on the same layer as \( O \) and has the same center as \( O_{1} \). By Step (ii), we could always find such a node \( O \). Since \( c_{\beta} = c_{\alpha} \), we obtain that \( c_{\text{parent}(O)} = d_{\text{parent}(O)} \). Since \( (\text{parent}(O'),O) \) is not a well-separated pair, we obtain that \( d_{\text{parent}(O')}(O) \leq 2(\delta + 2) \max\{r_{\text{parent}(O')}, r_{\beta}\} \leq 2(\delta + 2)r_{\text{parent}(O')} \). Thus, we obtain \( d_{\text{parent}(O')}(O) \leq 2(\delta + 2)r_{\text{parent}(O')} \).

Since \( (O',O) \) is a child of \( \text{parent}(O) \) in the original partition tree \( T_{\text{org}} \), we obtain that \( d_{\text{parent}(O)}(O') \leq r_{\text{parent}(O)} \). By triangle inequality, we obtain that \( d_{\text{parent}(O)}(O') \leq d_{\text{parent}(O)}(O) + d_{\text{parent}(O)}(O') \leq 2(\delta + 2)r_{\text{parent}(O')} + 2r_{\text{parent}(O)} = 4(\delta + 2)r_{\text{parent}(O')} + 2r_{\text{parent}(O)} = (\delta + 10)r_{\text{parent}(O')} \).

Thus, \( (O',O) \) must be an enhanced node pair, where \( c_{\alpha} = c_{\beta} \) and \( c_{\text{parent}(O)} = c_{\beta} \).

**Lemma 7.** The maximum number of child nodes of each node \( O \) in a partition tree or a compressed partition tree is \( O(2^{2\beta}) \).

**Proof.** By the definition of the partition tree, the center of each children of \( O \) must lie in the disk \( D(c_{\beta}, r_{\beta}) \). Besides, by the Separation Property, the minimum pairwise distance of all its children must be at most \( \frac{\delta}{2} \). Thus, by the definition of the largest capacity dimension \( \beta \), we obtain that the maximum number of children of each node \( O \) in a partition tree is \( O(2^{2\beta}) \). It is easy to see that the converting from a partition tree to a compressed partition tree does not change the number of children of any undeleted node. Thus, we obtain that the maximum number of children of each node \( O \) in a compressed partition tree is \( O(2^{2\beta}) \).

**Lemma 8.** Any disk \( D(c, r) \), where \( c \in P \) and \( r \) is a non-negative real number, can hold at most \( O(2^{2\beta}) \) points, the minimum pairwise geodesic distance of which is at least \( \frac{\delta}{2} \).

**Proof.** Consider a set \( PSE \) of points in \( D(c, r) \) such that their minimum pairwise geodesic distance is at least \( \frac{\delta}{2} \). We first build a partition tree upon \( PSE \) as follows: first, we create the root to be \( O(c, c, r, r) \) instead of following Step(a) and the building procedure of other nodes is the same as Step(b). Since the radius of each non-root node is half of its parent’s radius, we obtain that there are totally \( i \) layers in the tree. By Lemma 7, we obtain that the number of children of any node in the compressed partition tree is at most \( O(2^{2\beta}) \). Thus, the number of nodes in Layer \( i \) is at most \( O(2^{2\beta}) \) times that in Layer \( i - 1 \), where \( 0 < i < h \). Thus, we obtain that there are at most \( O(2^{2\beta}) \) nodes in Layer \( i \). In other words, \( PSE \) has at most \( O(2^{2\beta}) \) points.

**Lemma 9.** The compressed partition tree \( T_{\text{compressed}} \) has \( O(n) \) nodes.

**Proof.** Let \( m, k \) denote the number of nodes and edges in \( T_{\text{compressed}} \), respectively. By the construction of \( T_{\text{compressed}} \), \( T_{\text{compressed}} \) has \( n \) leaf nodes and every inner node in \( T_{\text{compressed}} \) has at least \( 2 \) children. Thus, \( T_{\text{compressed}} \) has \( n - m \) inner nodes and at least \( 2 \cdot (m - n) \) edges. Since \( T_{\text{compressed}} \) is a tree, we obtain that \( k = m + 1 \). Thus, we obtain that \( \frac{k}{m} \geq 2(m - n) \). Finally, we obtain that \( m \leq 2n - 1 \).

**Proof of Theorem 2.**

**Proof Sketch.** To give the intuition of the proof, we present an intermediate node pair set \( S' \) which is conceptual. Let \( T' \) denote the tree which is the same as the original partition tree except that the radius of each leaf node is 0. \( S' \) denote a node pair set built by the node pair generating algorithm presented in Section 3.3 which takes \( T' \) as input instead of the compressed partition tree. It is clear from a high-level intuition that the node pair set \( S \) of \( T \) is not larger than \( S' \) and the number of the node pairs considered in the process of generating \( S \) is \( O\left(|S|\right) \) (see the full proof for the details). In the following, we denote \( r_{\alpha} \) as the radius of a node \( O_{1} \) in the original partition tree. Consider a node \( O \) in \( T \) and a set \( S(O) \) which is \( \{O' | (O', O) \text{ or } (O', O') \text{ is in } S \text{ and } r_{\beta} \geq r_{\alpha} \} \). Note that \( \cup_{O \in T} S(O) = S \).

By the node pair generating algorithm, we obtain that for each node \( O \) in \( S(O) \), \( r_{\beta} \) is in the same layer as \( O \) or one layer lower than \( O \) (see Lemma 10). There is a upper bound on \( d_{\text{parent}(O)} \), i.e., \( O' \) lies in a disk \( D_{C_{\beta}} \) with a \( r_{\alpha} \)- and \( r_{\beta} \)-related radius (see Lemma 11). Then, together with a property (see Lemma 8) derived from \( \beta \), we obtain that \( S(O) = O(\frac{\beta}{2})^{2} \) and thus \( |S| = O(\frac{\beta}{2})^{2} 2^{\beta} \).

**Detailed Proof.** Now, we delve into the detailed proof and adopt the same notations as shown in the proof sketch.

**Lemma 10.** \( \forall O' \in S(O), r_{\alpha} \leq r_{\beta} \leq 2 \cdot r_{\beta} \).

**Proof.** Since \( \text{parent}(O') \) is split before the node pair \( (O', O') \) is generated, by Lemma 6, we obtain that \( r_{\text{parent}(O')} \geq r_{\beta} \). Since \( r_{\text{parent}(O')} = 2 \cdot r_{\beta} \), we obtain that \( r_{\beta} \leq r_{\alpha} \leq 2 \cdot r_{\beta} \).

**Lemma 11.** \( \forall O' \in S(O), d_{\text{parent}(O)}(O') \leq (4 \cdot \frac{\beta}{2} + 10) \cdot r_{\beta} \).

**Proof.** By our node pair set generating algorithm presented in Section 3.3, \( (O, O') \) is generated by \( \langle \text{parent}(O), O' \rangle \) or \( \langle O, \text{parent}(O') \rangle \) and the node pair which generated \( (O, O') \) is not well separated. Consider the case where \( (O, O') \) is generated by \( \langle O, \text{parent}(O') \rangle \) (the analysis of the case where \( (O, O') \) is generated by \( \langle \text{parent}(O), O' \rangle \) is symmetric, i.e., just with \( O \) and \( O' \) swapped, and has the same result and thus, we do not present this case for the sake of space). We obtain that \( d_{\text{parent}(O)}(O') \leq 2 \cdot (\frac{\delta}{2} + 2) \cdot \max\{r_{\alpha}, r_{\beta}\} \). By the definition of the partition tree. By Lemma 10, we obtain that \( d_{\text{parent}(O)}(O') \leq 2 \cdot (\frac{\delta}{2} + 2) \cdot \max\{r_{\alpha}, r_{\beta}\} \). By Triangle Inequality, we obtain that \( d_{\text{parent}(O)}(O') \leq d_{\text{parent}(O)}(O) + d_{\text{parent}(O)}(O') = (\frac{\delta}{2} + 10) \cdot r_{\beta} \).

Let \( S'' \) be a point set containing the centers of all nodes in \( S(O) \). By Lemma 10, we obtain that \( O \) is either in the same layer as \( O \) in the partition tree or one layer lower than \( O \) in the partition tree. Let \( S(O') \) (resp., \( S(O) \)) denote \( \{O' | (O', O) \in S, O' \text{ is in the same } \}

**Next, we prove that \( |S| \) is at most \( |S'| \) (i.e., \( O(\frac{\beta}{2})^{2} 2^{\beta} \)), where \( S \) is the node pair set of \( SE \). Consider a node pair \( (O, O') \) in \( S \). We
The oracle building time of SE consists of the time $t_{org}$ of building the original partition tree $T_{org}$, the time $t_{trans}$ of transforming the partition tree to the compressed partition tree $T_{compressed}$, the time $t_{en}$ of creating all the enhanced edges and the time $t_{np}$ of generating the node pair set of SE. $t_{org}$ consists of the running time of B-tree operations, the point selection algorithm and all SSAD algorithms invoked. Note that we index a set of points/nodes with a B-tree and we find the parent of O in Step (iii) with a SSAD algorithm and $d_{g}(O, O_{parent})$ is at most $2r_{0}$ due to the covering property. The running time of all the B-tree operations in $t_{org}$ is $O(nh \log n)$ since there are $h$ layers and each layer needs $O(n)$ insertion/search operation in B-tree. For the point selection algorithm, the random selection algorithm takes $O(nh)$ time since there are at most $nh$ nodes in $T_{org}$ and each takes at most $O(1)$ time. The greedy selection algorithm takes $O(nh \log n)$ time since in each layer since it takes $O(n \log n)$ time to create the grid and corresponding B-tree in each cell and there are at most $O(n)$ heap operations and B-tree operations. It is obvious that the overall running time of all SSAD algorithms performed in $t_{org}$ is smaller than $t_{en}$; $t_{en}$ is $O(nh)$ since the partition tree has $O(nh)$ nodes. By Theorem 2, there are at most $O(\frac{N}{\sqrt{n}})$ node pairs considered in the procedure described in Section 3.3 and thus, $t_{np}$ is $O(\frac{N}{\sqrt{n}} \cdot \log n)$. Next, we will analyze $t_{en}$.

We introduce a new parameter of the terrain surface, denoted by $\theta$. $\theta$ is defined as the largest positive real number such that the number of vertices on the terrain surface in a disk $D(c, r)$ is at least $2^\theta$ times the number of vertices on the terrain surface contained in the disk $D(c, \frac{r}{2})$, where $c$ is a POI on the terrain surface and $r$ is a positive real number at most $\max_{c \in \mathcal{V}} d_{g}(c, p)$. Thus, the number of vertices contained in a disk centered at any POI is at most $\frac{1}{2^\theta}$ times that in a disk with double radius and the same center. For the root node $O_{root}$, we expand the disk $D(c, r_{0})$. For any node $O$ in other layers, we expand the disk $D(c, l \cdot r_{0})$, where $l = \frac{1}{2} + 10$. Note that $\forall i \in [\log l], h_{i} \cdot r_{i} > r_{0}$. Since $D(c, r_{0})$ is a sub-region on the terrain surface, where $c$ is any point on the surface, we obtain that the vertices of terrain visited by SSAD algorithm invoked for each node in layer $i$ is at most $N$, if $i \in [0, \lceil \log l \rceil - 1]$; otherwise, it is $\frac{N}{\sqrt{n} \cdot \log n}$.

By Lemma 7, there are at most $\left(\frac{2^{2\theta}}{2^{2\theta} - 1}\right)$ nodes in Layer $i$. Thus, we obtain that $t_{en}$ is at most $O(\sum_{i=\log l}^{\lceil \log l \rceil - 1} \left(\frac{2^{2\theta}}{2^{2\theta} - 1}\right) N \log^2 N + \sum_{i=\log l}^{\lceil \log l \rceil - 1} \left(\frac{2^{2\theta}}{2^{2\theta} - 1}\right) \sum_{h=0}^{\lceil \log l \rceil - 1} (\frac{2^{2\theta}}{2^{2\theta} - 1}) N \log^2 N)$

$= O(\frac{N}{\sqrt{n} \cdot \log n} \cdot \log^2 N)$

Our empirical study verified that $\theta \geq \beta$. Thus, we obtain that $\frac{2^{2\theta}}{2^{2\theta} - 1} < 1$ and thus $t_{en}$ is $O((\frac{2^{2\theta}}{2^{2\theta} - 1}) N \log^2 N) = O(\frac{N}{\sqrt{n} \cdot \log n} \cdot \log^2 N)$. Thus, we obtain that the oracle building time is $O(\frac{N}{\sqrt{n} \cdot \log n} \cdot \log^2 N + \log n \cdot \log \log n + \frac{\log n}{\sqrt{n}})$.

PROOF OF THEOREM 3. By Lemma 12, Theorem 2, and the analysis in Section 3.4, we obtain the result.

C. A2A DISTANCE QUERY PROCESSING

We present an oracle to answer the arbitrary point-to-arbitrary point (A2A) distance query based on our proposed distance oracle SE. This oracle is the same as that presented in Section 3 except that it takes some Steiner points introduced as input instead of all POIs, where Steiner points are introduced by the method proposed in [12] (there are $O(\frac{\log n}{\sqrt{n} \cdot \log n} \cdot \log \log n)$ Steiner points, where $\theta$ is the minimum inner angle of any face). Then, we present the query processing. Given two arbitrary points $s$ and $t$, we first find the neighborhood of $s$ (resp., $t$), denoted by $N(s)$ (resp., $N(t)$) (It is a set of Steiner points on the same face containing $s$ (resp., $t$) and its adjacent face(s) [12]. Finally, we return $d_{g}(s, t) = \min_{p \in N(s), q \in N(t)} d_{g}(s, p) + d_{g}(p, q) + d_{g}(q, t)$, where $d_{g}(s, p)$ and $d_{g}(q, t)$ could be computed in constant time by SSAD algorithm and $d_{g}(p, q)$ is the distance between $p$ and $q$ estimated by the oracle constructed. By [12], $|N(s) \cdot N(t)| = \frac{1}{\sin(h)} \cdot \frac{1}{\sin(\theta)}$ and if $d_{g}(p, q)$ is an $\epsilon$-approximate distance of $d_{g}(p, q)$, then $d_{g}(s, t)$ is also an $\epsilon$-approximate distance of $d_{g}(s, t)$. By Theorem 3, we obtain that for any two Steiner points $p$ and $q$, $d_{g}(p, q)$ is an $\epsilon$-approximate distance of $d_{g}(p, q)$ and it takes $O(h)$ time to compute $d_{g}(p, q)$ and the building time (resp., oracle size) is $O(\frac{\log n}{\sqrt{n} \cdot \log n} \cdot \log \log n + \frac{\log n}{\sqrt{n} \cdot \log n} \cdot \log \frac{\log n}{\sqrt{n} \cdot \log n} \cdot \log \frac{\log n}{\sqrt{n} \cdot \log n} + \frac{\log n}{\sqrt{n} \cdot \log n} \cdot \log \frac{\log n}{\sqrt{n} \cdot \log n} \cdot \log \frac{\log n}{\sqrt{n} \cdot \log n} \cdot \log \frac{\log n}{\sqrt{n} \cdot \log n})$. Thus, we obtain that for any two arbitrary points $s$ and $t$, the oracle gives an $\epsilon$-approximate distance of $d_{g}(s, t)$ and the query time is $O(\frac{1}{\sin(h)} \cdot \frac{1}{\sin(\theta)})$.

D. DISCUSSION FOR CASE WHEN $n > N$

When $n > N$, we adopt the same distance oracle described in Appendix C, which is POI-independent. This distance oracle could answer not only A2A distance queries but also V2V distance queries and P2P distance queries (because A2A distance queries could be regarded as a general setting compared with V2V distance queries and P2P distance queries).