Proximity Path Queries on Point Cloud using Rapid Construction Path Oracle

Anonymous
Anonymous
Anonymous

Anonymous
Anonymous
Anonymous

ABSTRACT

The prevalence of computer graphics technology boosts the developments of the point cloud in recent years, and researchers started to utilize its advantages over the terrain surface (represented by *Triangular Irregular Network*, i.e., *TIN*) in the proximity path queries, including the *shortest path query*, the *k-Nearest Neighbor* (*kNN*) *path query*, and the *range path query*. As could be observed from the existing studies, the on-the-fly and oracle-based shortest path algorithms on a *TIN* are very expensive. All existing on-the-fly shortest path algorithms on a point cloud are still not efficient, and there are no oracle-based shortest path algorithms on a point cloud. Motivated by this, we propose an efficient $(1+\epsilon)$-approximate shortest path oracle on the point cloud, which has a good performance (in terms of the oracle construction time, oracle size, and shortest path query time) due to the concise information about the pairwise shortest path between any pair of points-of-interests stored in the oracle. Then, we propose algorithms for answering the *kNN* path query and the range path query with the assistance of our path oracle. Our experimental results show that our oracle is up to 97,500 times, 2 times, and 6 times better than the best-known oracle-based algorithm on a *TIN* in terms of the oracle construction time, oracle size and shortest path query time, respectively. Our algorithms for the other two proximity path queries are both up to 6 times faster than the best-known algorithms.

ACM Reference Format:
Anonymous and Anonymous. 2023. Proximity Path Queries on Point Cloud using Rapid Construction Path Oracle. In *Proceedings of 2024 International Conference on Management of Data (SIGMOD '24).* ACM, New York, NY, USA, 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The point cloud is becoming one of the most important data types for representing a three-dimensional (3D) object nowadays, and conducting proximity path queries, including (1) the *shortest path query*, (2) the *k-Nearest Neighbor* (*kNN*) *path query* [6], and (3) the *range path query* [8], on a point cloud aroused widespread concern in industry and academia [23]. In industry, Metaverse uses point clouds of objects such as mountains and hills to help users reach the destination faster in Virtual Reality [35, 36], Tesla uses point clouds of the driving environment for autonomous driving [17, 37, 41]. In academia, proximity path queries on point clouds also captivate tremendous researchers' attention [23, 43, 51, 59].

For the proximity path queries: (1) In the *shortest path query*, given a source point s and a destination t on a point cloud, it answers the shortest path between s and t on the point cloud. (2) In the *kNN path query*, given a set of objects and a query point q on the point cloud, it answers all the shortest paths from q to the k nearest objects of q using the shortest path query on the point cloud. (3) In the *range path query*, given a range value r, a set of objects and a query point q on the point cloud, it answers all the shortest paths from q to the objects whose distance to q are at most r using the shortest path query on the point cloud. The shortest path query is the most fundamental type of query. For the *kNN* path query and range path query, in the case of bushfires for refusing, their applications include finding the shortest paths from the center of the bushfires to k nearest fire trucks, to all the fire trucks that are not further than r km (the mountain or the forest could be represented in the form of a point cloud). Other applications of them include rover path planning [13] and military tactical analysis [32].

A point cloud is represented by a set of 3D *points* in space. Figure 1 (a) shows a satellite map of Mount Rainier [45] (a renowned national park in the USA) in an area of 20km × 20km, and Figure 1 (b) shows the point cloud with 81 points of the Mount Rainier. Furthermore, a terrain surface could be represented by *Triangular Irregular Network* (*TIN*), which contains a set of *faces* each of which is denoted by a triangle. Each face consists of three line segments called *edges* connected with each other at three *vertices*. The gray surface in Figure 1 (c) is an example of a *TIN*, which consists of vertices, edges and faces. Answering the shortest path query on a *TIN* has been extensively studied in [18, 28, 29, 38, 54, 55, 57, 58].

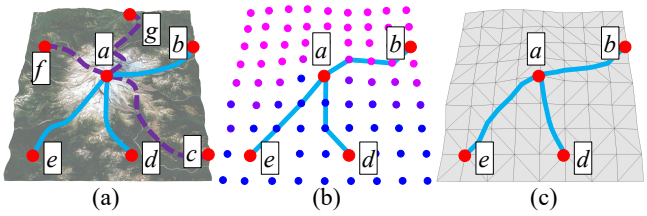

Figure 1: A (a) satellite map, (b) point cloud, and (c) *TIN* of Mount Rainier with shortest paths

1.1 Motivation

1.1.1 ***Advantages of point cloud.*** Answering the proximity path query on a point cloud, has the following three advantages compared with on a *TIN*. (1) We can obtain a point cloud directly and

SIGMOD '24, June 11–16, 2024, Santiago, Chile

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

quickly (e.g., we could use an iPhone 12/13 Pro LiDAR scanner [52] to scan an object and generate the point cloud in 10s, or could use a satellite to obtain the elevation of a region in an area of 1km^2 and generate the point cloud in 144s ≈ 2.4 min [42]), but in order to obtain a *TIN*, typically, researchers need to transform a point cloud to a *TIN* [27] (our experimental result shows that it needs 210s ≈ 3.5 min to transform a point cloud with 25M points to a *TIN*). (2) It needs more memory consumption to store a *TIN* compared with a point cloud, since we only need to store the point information of a point cloud, but need to store the vertex and face information of a *TIN* (our experimental result shows that storing a point cloud with 25M points needs 390MB, but storing a *TIN* generated by this point cloud needs 1.7GB). (3) Answering the proximity path query on a point cloud is faster than on a *TIN* (our experimental result shows that answering one shortest path query needs 3s on a point cloud with 2.5M points, but needs 580s ≈ 10 min on the *TIN* generated by this point cloud).

1.1.2 ***Usages of POIs.*** Given a set of *points-of-interest (POIs)* on a point cloud, conducting proximity path queries between *pairs of POIs* on the point cloud, i.e., *points of interest-to-points of interest (P2P) proximity path query*, is important. For example, in Tesla autonomous driving, POIs could be several destinations that the driver needs to reach. In other applications, POIs could be residential locations used in conducting proximity path queries of the wildness animals when studying their migration patterns [20, 39], and POIs could be reference points used in measuring similarities between two different 3D objects [31, 50].

1.1.3 ***Real-life example.*** Consider one scenario that we need to perform P2P proximity path queries on a point cloud. Suppose that we need to evacuate from Mount Rainier due to the frequent heavy snowfall [46], where Mount Rainier has the highest seasonal total snowfall world record [9]. The blizzard wreaking havoc across the USA in December 2022 killed more than 60 lives [10], and one may be dead due to asphyxiation [33] if s/he gets buried in the snow. Therefore, in the case of snowfall, Mount Rainier National Park, will be closed and staffs will evacuate tourists in the mountain to the closest hotels immediately for ensuring tourists' safety. The evacuation (walking from multiple viewing platforms on the mountain to the hotels, where the viewing platforms and hotels are regarded as POIs) is expected to be finished in 2.4 ($= \frac{10\text{centimeters}\times 24\text{hours}}{1\text{meter}}$) hours, since the maximum snowfall rate (which is defined to be the maximum amount of snow accumulates in depth during a given time [15, 48]) in Mount Rainier is 1 meter per 24 hours [47], and it becomes difficult to walk, easy to lose the trail and get buried in the snow if the snow is deeper than 10 centimeters [25]. In Figure 1 (a), we would like to find the shortest paths (in blue and purple lines) from one of the viewing platforms on the mountain to its k-nearest hotels due to the limited capacity of each hotel (a is the viewing platform, b to g are the hotels). b, e, and f are the k-nearest hotels to this viewing platform where $k = 3$. Figure 1 (b) shows the shortest paths from a to b, e, f pass on a point cloud, Figure 1 (c) shows the same shortest paths pass on a *TIN*. The shortest paths pass on the point cloud and the *TIN* are very similar. Our experimental result shows that for a point cloud of Mount Rainier with 1.5M points, for each viewing platform, calculating the shortest path to its k nearest hotels (there are total 125 viewing platforms and 125 hotels) is 225s ≈ 3.7 min on a point cloud, but is 43,500s ≈ 12.1 hours on a *TIN* (constructed based on the point cloud). The average distance between the viewing platforms and hotels in Mount Rainier National Park is 8.2km [5], and the average human walking speed is 5.1km/h [7], so the evacuation could be finished in 1.6 ($= \frac{8.2\text{km}}{5.1\text{km/h}}$) hours, together with the query time 3.7 min, which shows that the usefulness of performing proximity path queries on point cloud with POIs in real-life application.

1.2 Challenges

Efficiently conducting the proximity path queries between pairs of POIs on a point cloud is very challenging.

1.2.1 ***Inefficiency for on-the-fly algorithm.*** Most (if not all) existing algorithms [43, 51, 59] for conducting the proximity path queries on a point cloud *on-the-fly* are very slow even on a moderate-size point cloud. When conducting the proximity path queries, we first need the shortest path query, and all of them calculate the shortest path based on an implicit structure, i.e., a *TIN*, which means that they (1) first construct an implicit *TIN* in $O(N)$ time, where N is the number of points in the point cloud, and (2) then calculate the shortest path between a source point s and a destination point t on this implicit *TIN* on-the-fly (which is time-consuming).

Exact algorithm: According to [28, 29, 49, 56], [14] is recognized as the best-known on-the-fly *TIN* exact shortest path query algorithm, which needs $O(N^2)$ time. We denote algorithm *On-the-Fly shortest path query on implicit TIN Face Exact*, i.e., *Fly(FaceExact)*, to be the adapted algorithm of [43, 51, 59], which first constructs a *TIN*, and then uses [14] for computing the exact shortest path on a point cloud.

Approximate algorithm: According to [54, 55], [28] is recognized as the best-known on-the-fly *TIN* approximate shortest path query algorithm, which needs $O((N + N')\log(N + N'))$ time, where N' is the number of additional points introduced for bound guarantee. We denote algorithm *On-the-Fly shortest path query on implicit TIN Face Approximate*, i.e., *Fly(FaceAppr)*, to be the adapted algorithm of [43, 51, 59], which first constructs a *TIN*, and then uses [28] for computing the approximate shortest path on a point cloud.

But, our experimental result shows *Fly(FaceExact)* [14] (resp. *Fly(FaceAppr)* [28]) needs 290,000s ≈ 3.4 days (resp. 90,000s ≈ 1 day) to perform the *kNN* path query and the range path query for all 2500 POIs on a point cloud with 0.5M points, which is very slow.

1.2.2 ***No oracle-based algorithm.*** There is no existing work for answering the proximity path queries between pairs of POIs on a point cloud in the form of an index (called *oracle*). When conducting the proximity path queries between pairs of POIs on a point cloud, we first need to calculate the pairwise P2P shortest path on a point cloud, and there no oracle on a point cloud for this. Although the best-known existing work [54, 55] aiming at pre-computing the approximate pairwise P2P shortest path in the form of oracle on a *TIN* (the time taken to pre-compute the oracle is called the *oracle construction time* and the space complexity of the oracle is called the *oracle size*), and returning the shortest path efficiently (the time taken to return the result is called the *shortest path query time*), it is not originally designed for *point cloud*. Even though

we can adapt the on-the-fly *point cloud* shortest path query algorithm in the oracle of [54, 55] for pairwise P2P *point cloud* shortest path oracle construction, its oracle construction time is still very large because (1) it involves *numerous* shortest path queries when constructing the oracle, and (2) in order to build their oracle, two additional data structures, called *compressed partition tree* [54, 55] and *well-separated node pair set* [12], are needed to be constructed beforehand, where the construction of these two data structures is also time-consuming. The oracle construction time and oracle size of the best-known oracle [54, 55] (after the adaptation on the point cloud) are $O(cnN \log N)$ and $O(cn)$, where n is the number of POIs on the point cloud and c is a constant with the value close to n. In our experiment, the oracle construction time and oracle size of it (after adaptation) are 39,000s $\approx$ 10.8 hours and 106MB for a point cloud with 10k points and 250 POIs.

1.3 Our Path Oracle and Proximity Path Queries Processing Algorithms

Motivated by these, we propose an efficient $(1+\epsilon)$-approximate path oracle on a point cloud called *Rapid Construction path Oracle on Point cloud*, i.e., *RC-Oracle(Point)*, which has a good performance in terms of the oracle construction time, oracle size, and shortest path query time compared with the best-known adapted point cloud oracle [54, 55] due to the concise information about the pairwise shortest path between any pair of POIs stored in the oracle, where ϵ is a non-negative real user parameter for controlling the error, called the *error parameter*. Based on *RC-Oracle(Point)*, we develop efficient query processing algorithms for other proximity queries, i.e., the *kNN* path query and range path query. We introduce the key idea of the small oracle construction time of *RC-Oracle(Point)*.

1.3.1 ***Novel point cloud shortest path on-the-fly path query algorithm.*** When constructing *RC-Oracle(Point)*, we do not use any existing on-the-fly path query algorithm [43, 51, 59], that is, we do not (1) construct an *implicit TIN*, and (2) calculate the point cloud shortest path on this *implicit TIN* on-the-fly using [14, 28, 34, 40]. Instead, we propose a new algorithm, called algorithm *On-the-Fly shortest path query on Point cloud*, i.e., *Fly(Point)*, which uses Dijkstra algorithm [21] and the shortest path calculated by it will only pass on the *point* of the point cloud. This could significantly reduce the algorithm's running time, since computing the shortest path on a *TIN* is very expensive. Our experimental result shows that the shortest path query time of algorithm *Fly(Point)* is just 3s on a point cloud with 2.5M points, but the time is 580s $\approx$ 10 min for the best-known algorithm *Fly(FaceExact)* and 330s $\approx$ 5.5 min for algorithm *Fly(FaceAppr)* under the same setting.

1.3.2 ***Novel oracle construction.*** When constructing the oracle part of *RC-Oracle(Point)*, we do not use any existing best-known oracle [54, 55] due to their large oracle construction time. We use an example to illustrate the construction process of *RC-Oracle(Point)*. In Figure 2 (b), we first use algorithm *Fly(Point)* to calculate the shortest path from a to all other POIs. Then, in Figure 2 (c), given another POI b, if b is close to a, and b is far away from d and e, we can use the shortest path between a and d, a and e, to approximate the shortest path between b and d, b and d, respectively. Since algorithm *Fly(Point)* is a Dijkstra-based algorithm, so given a POI p, we can compute the shortest path from p to all other POIs simultaneously, i.e., it is a *Single-Source All-Destination* (*SSAD*) algorithm. When computing the shortest path from b to all other POIs, if we do not need to calculate the shortest path between b and d, b and e, we can terminate algorithm *Fly(Point)* earlier when we have covered the POIs that are close to b, i.e., c. Given n POIs, we just need to run algorithm *Fly(Point)* for at most n times (and in most cases, we can terminate them earlier), and thus, we can finish constructing *RC-Oracle(Point)*. Furthermore, we directly construct *RC-Oracle(Point)* on the point cloud, without any other additional data structures.

Limitation in the best-known oracle: But, in the best-known oracle [14, 54, 55], it first constructs a *compressed partition tree*, then partitions the POIs into several levels of *well-separated pairs* [12] using the compressed partition tree. Since different well-separated pairs are constructed at different stages, when using algorithm *Fly(Point)* to calculate the exact shortest path on a point cloud, it will calculate the exact shortest path between two POIs a and b only, since it does not know whether a and other POIs are well-separated or not, thus, it does not know whether there is a need to also calculate the exact shortest path on the point cloud between a and other POIs. But, at a later stage, they may need to use algorithm *Fly(Point)* to calculate the exact shortest path on the point cloud between a and other POIs (e.g., c). They can terminate algorithm *Fly(Point)* after calculating the exact shortest path between a and b for time-saving. But, it needs to run algorithm *Fly(Point)* more than n times (our experimental result shows that it needs to run the algorithm almost n^2 times when $\epsilon = 0.1$).

Limitation in the best-known oracle after adaption: Even though we can adapt the best-known oracle [14, 54, 55], i.e., pre-computing the shortest path between each pair of POIs that uses algorithm *Fly(Point)* for n times, it is still not efficient. In this case, it cannot terminate algorithm *Fly(Point)* earlier, since it needs to pre-compute the exact shortest path between *each* pair of POIs. In addition, no matter whether we adapt it, it always needs to build the oracle based on the compressed partition tree and the well-separated node pair set, where the construction of these two data structures is also time-consuming. By both using algorithm *Fly(Point)*, our experimental result shows that the oracle construction time of *RC-Oracle(Point)* is 70s, but is 333,600s $\approx$ 3.9 days for the best-known oracle [54, 55] (without adaption), and is 15,700s $\approx$ 4.4 hours for the best-known oracle [54, 55] (with adaption) on a point cloud with 0.5M points and 2500 POIs.

1.4 Contribution & Organization

We summarize our major contributions as follows.

(1) We propose a novel oracle called *RC-Oracle(Point)*, which is the first oracle that efficiently answers the P2P shortest path query on a point cloud to the best of our knowledge. We also propose algorithm *Fly(Point)* for calculating the shortest path on a point cloud on-the-fly, which is used for constructing *RC-Oracle(Point)*. By proposing different on-the-fly shortest path query algorithms, our oracle could be easily extended to different data formats (e.g., a *TIN*, a graph, a road network).

(2) We also develop efficient query processing algorithms for other proximity queries such as the kNN path query and range path query, with the assistance of *RC-Oracle(Point)*.

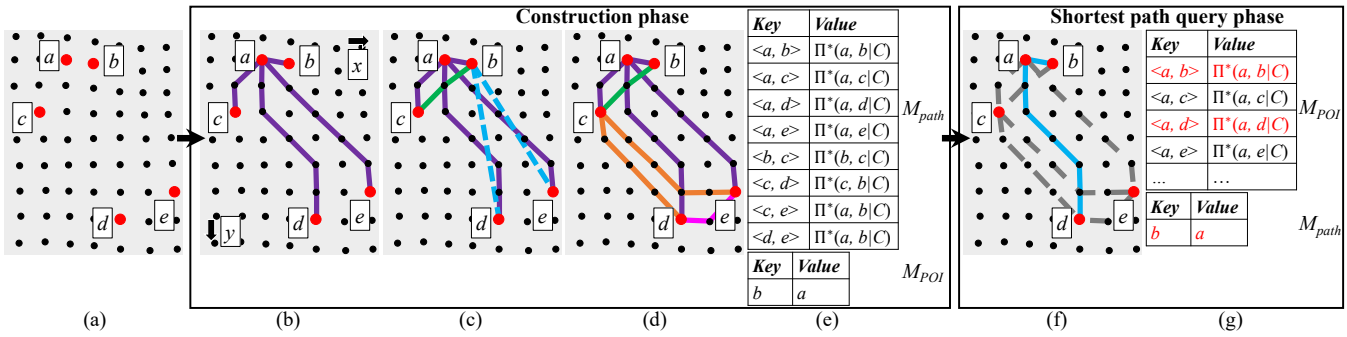

Figure 2: Framework overview

(3) We provide thorough theoretical analysis on the oracle construction time, oracle size, shortest path query time, and error bound of *RC-Oracle(Point)*, and on the *kNN* path query time, range path query time, and error bound for other proximity path queries. It is worth mentioning that our proposed error bound for other proximity path queries is not limited to use in this paper, and could be easily extended to other database queries. We also provide theoretical analysis on the relationships between the shortest path passes on points of a point cloud, and passes on the implicit *TIN* of the point cloud. These relationships fill the gap of answering the proximity path queries on a point cloud and a *TIN*.

(4) *RC-Oracle(Point)* performs much better than the best-known oracle [54, 55] in terms of the oracle construction time, oracle size, and shortest path query time, and *RC-Oracle(Point)* support real-time response. The *kNN* path query time and range path query time with the assistance of *RC-Oracle(Point)* also performs much better than [54, 55]. Our experimental results show that the oracle construction time of *RC-Oracle(Point)* is only 0.4s, but the best-known oracle needs more than 39,000s $\approx$ 10.8 hours for a point cloud with 10k points and 250 POIs. The *kNN* path query time and range path query time of all 500 POIs for *RC-Oracle(Point)* are both 25s, but the best-known exact on-the-fly algorithm [14] needs 58,000s $\approx$ 16.2 hours, and the best-known oracle needs 150s for a point cloud with 0.5M points.

The remainder of the paper is organized as follows. Section 2 provides the problem definition. Section 3 shows the related work. Section 4 presents our shortest path oracle *RC-Oracle(Point)*. Section 5 shows our discussion. Section 6 presents the experimental results and Section 7 concludes the paper.

2 PROBLEM DEFINITION

Given a set of points, we let C be a point cloud of these points, and N be the number of points in C (i.e., $N = |C|$). Let L be the maximum side length of C. Each point $p \in C$ has three coordinate values, denoted by x_p, y_p and z_p. In this paper, the point cloud C that we considered is a grid-based point cloud [11, 22], because a grid-based 3D object, e.g., a grid-based point cloud [11, 22] and a grid-based *TIN* [19, 38, 49, 54, 55], is commonly adopted in many papers. Given a point p in C, we define $N(p)$ to be a set of neighbor points of p, which denotes the closest top, bottom, left, right, top-left, top-right, bottom-left, and bottom-right points of p in the xy coordinate 2D plane. Figure 3 (a) shows an example of a point cloud C. In this

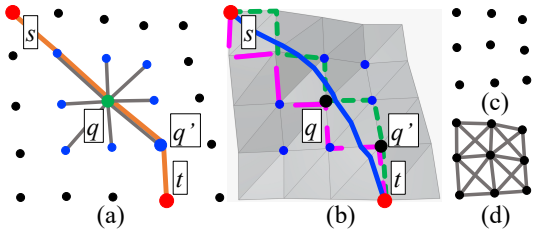

Figure 3: (a) Orange $\Pi^*(s,t|C)$ on a point cloud, (b) blue $\Pi^*(s,t|T)$, green $\Pi_V(s,t|T)$, pink $\Pi_N(s,t|T)$ on a *TIN*, (c) a point cloud, and (d) the conceptual graph

figure, given a green point q, $N(q)$ is denoted as eight blue points. We can easily extend our problem to the non-grid-based point cloud. The only difference is that we need to re-define $N(p)$. Given a point p in a non-grid-based point cloud, we define $N(p)$ to be a set of neighbor points of p such that the Euclidean distance between p and all points in this non-grid-based point cloud is smaller than a user-defined parameter, e.g., r.

Given a pair of points p and p' in C, we define $d_E(p,p')$ to be the Euclidean distance between point p and p'. Given a pair of points s and t in P, we define $\Pi^*(s,t|C) = (s = q_1, q_2, \ldots, q_l = t)$, with $l > 1$, to be the exact shortest path between s and t passes on the *points* of a point cloud C, such that the total Euclidean distance $\sum_{i=1}^{l-1} d_E(q_i, q_{i+1})$ is minimum, where q_i for $i \in \{1, \ldots, l\}$ is a point in C and $q_{i+1} \in N(q_i)$. We further define $|\cdot|$ to be the length of a path on C (e.g., $|\Pi^*(s,t|C)|$ is the length of the exact shortest path $\Pi^*(s,t|C)$ on C). The orange line Figure 3 (a) shows an exact shortest path $|\Pi^*(s,t|C)|$ on a point cloud C. Let $\Pi(s,t|C)$ be the shortest path of returned by *RC-Oracle(Point)*. *RC-Oracle(Point)* guarantees that $|\Pi(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any s and t in P. Performing the shortest path query on C could be regarded as on a conceptual graph G. Let $G.V$ and $G.E$ be the set of vertices and edges of G, where each point in C is denoted by a vertex in $G.V$. For each vertex in $G.V$, there are eight edges connecting this vertex and its closest top, bottom, left, right, top-left, top-right, bottom-left, and bottom-right vertices. Figure 3 (d) is the conceptual graph formed by the point cloud in Figure 3 (c).

Let P be a set of POIs on the point cloud and n be the size of P (i.e., $n = |P|$). Since a POI could only be a point on C, so

$n \leq N$. There are two types of proximity path queries on a point cloud, including (1) *P2P proximity path query*, and (2) *any points-to-any points (A2A) proximity path query*, i.e., given a point cloud, conducting proximity path queries between *pairs of any points* on the point cloud. Conducting the P2P proximity path query is more general than conducting the A2A proximity path query. This is because we can create POIs which has the same coordinate values as all points in the point cloud, and then the A2A proximity path query could be regarded as one form of the P2P proximity path query. Thus, for clarity, in the main body of this paper, we focus on the P2P proximity path query. We study the A2A proximity path query in the appendix. Furthermore, in the P2P proximity path query, there is no need to consider the case when a new POI is added, or a POI is removed. In the case when a POI is added, we can create an oracle to answer the A2A proximity path query, which implies we have considered all possible POIs to be added. In the case when a POI is removed, we can still use the original oracle that answers the P2P proximity path query by just removing a specific POI. A notation table could be found in the appendix of Table 2.

Problem: We would like to design an efficient $(1 + \epsilon)$-approximate shortest path oracle on a point cloud, which has a good performance in terms of the oracle construction time, oracle size, and shortest path query time. We also would like to use this oracle for efficiently answering the other two proximity queries, i.e., the *kNN* path query and the range path query.

3 RELATED WORK

3.1 On-the-fly Algorithm

Most (if not all) existing algorithms [43, 51, 59] for conducting the proximity path queries on a point cloud *on-the-fly* are very slow. This is because when conducting the proximity path queries, we first need the shortest path query, and all of them calculate the shortest path on a point cloud are based on an implicit structure (e.g., a *TIN*). Given a point cloud, they first triangulate it into a *TIN* [44] in $O(N)$ time. Then, there are two types of algorithms for computing the shortest path on the *TIN*, which are (1) *exact* algorithm [14, 40] and (2) *approximate* algorithm [28, 38].

- **Exact algorithm**: [40] uses continuous Dijkstra algorithm to calculate the exact shortest path on a *TIN* on-the-fly in $O(N^2 \log N)$ time, and the best-known exact algorithm *Fly(FaceExact)* [14] unfolds the 3D *TIN* into a 2D *TIN*, and then connects the source and destination using a line segment on this 2D *TIN* to calculate the result in $O(N^2)$ time. But, [14] cannot be adapted for point cloud shortest path query, because there is no face to be unfolded in a point cloud.
- **Approximate algorithm** Both [28, 34] place discrete points (i.e., Steiner points) on edges of a *TIN*, and then construct a graph using these Steiner points together with the original vertices to calculate the $(1+\epsilon)$-approximate shortest path on the *TIN* on-the-fly. The best-known approximate algorithm *Fly(FaceAppr)* [28] runs in $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$, where l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of the *TIN*, and θ is the minimum inner angle of any face in the *TIN*. If we let the path passes on the point of the point cloud, [28] can be adapted for the point cloud shortest path query, and this adaption is the same as algorithm *Fly(Point)*.

Drawbacks of the on-the-fly algorithms: All the on-the-fly algorithms are very slow even on a moderate-size point cloud. Our experimental result show algorithm *Fly(FaceExact)* (resp. *Fly(FaceAppr)*) 290,000s $\approx$ 3.2 days (resp. 90,000s $\approx$ 1 day) to perform the *kNN* path query and the range path query for all 2500 POIs on a point cloud with 0.5M points.

3.2 Oracle

There is no existing work for answering the proximity path queries between pairs of POIs on a point cloud in the form of an oracle (i.e., there is no oracle for calculating the pairwise P2P shortest path on a point cloud). The oracle [54, 55], called *Space Efficient Oracle* (*SE-Oracle*), only pre-compute the approximate pairwise P2P shortest path in the form of oracle on a *TIN*. We can adapt the on-the-fly *point cloud* shortest path query algorithm in the oracle of [54, 55] for pairwise P2P *point cloud* shortest path oracle construction. We denote *Space Efficient Oracle on implicit TIN Face Exact*, i.e., *SE-Oracle(FaceExact)*, as the best-known oracle, to be the adapted oracle of *SE-Oracle* [54, 55] that uses algorithm *Fly(FaceExact)* for the point cloud shortest path query. *SE-Oracle(FaceExact)* mainly uses *well-separated pair decomposition* idea [12]. It uses a *compressed partition tree* and a *well-separated node pair set* to index the $(1 + \epsilon)$-approximation pairwise P2P shortest path on a point cloud. The oracle construction time, oracle size, and shortest path query time of *SE-Oracle(FaceExact)* is $O(\frac{nhN^2}{\epsilon^{2\beta}})$, $O(\frac{nh}{\epsilon^{2\beta}})$, $O(h^2)$, respectively, where h is the height of the compressed partition tree and β is the largest capacity dimension [24, 30] ($\beta \in [1.5, 2]$ in practice according to [54, 55]).

We could further adapt *SE-Oracle(FaceExact)* to be *SE-Oracle-Adapt(FaceExact)*, such that we pre-compute the shortest path between each pair of POIs that uses *SSAD* algorithm for n times, then use the similar idea for oracle construction. The oracle construction time, oracle size, and shortest path query time of *SE-Oracle-Adapt(FaceExact)* is $O(nN^2 + nh \log n + \frac{nh}{\epsilon^{2\beta}})$, $O(\frac{nh}{\epsilon^{2\beta}})$, $O(h^2)$, respectively.

Drawbacks of the best-known oracle: The oracle construction time and oracle size for *SE-Oracle(FaceExact)* is very large, because it needs to use *SSAD* algorithm for almost n^2 times when constructing the well-separated node pair set. The oracle construction time and oracle size for *SE-Oracle-Adapt(FaceExact)* is still large, because the oracle is constructed based on the compressed partition tree and the well-separated node pair set, where the construction of these two data structures is also time-consuming (*SE-Oracle(FaceExact)* also has this issue). Our experimental results show that for a point cloud with 10k points and 250 POIs, the oracle construction time and oracle size of (1) *SE-Oracle(FaceExact)* are 39,000s $\approx$ 10.8 hours and 106MB, (2) *SE-Oracle-Adapt(FaceExact)* are 337s $\approx$ 5.6 min and 106MB, respectively, while our oracle *RC-Oracle(Point)* just needs 0.4s and 2.3MB. This is because *RC-Oracle(Point)* uses *Fly(Point)* at most n times, and could terminate *Fly(Point)* earlier for most of the cases. In addition, *RC-Oracle(Point)* do not need to pre-compute any other additional data structures.

4 METHODOLOGY

4.1 Overview of *RC-Oracle(Point)* and Proximity Path Queries Processing Algorithms

4.1.1 ***Components of RC-Oracle(Point).*** There are two components in *RC-Oracle(Point)*, i.e., the *path mapping table* and the *POI mapping table*.

The path mapping table M_{path} is a *hashing table* [16] stores the selected pairs of POIs u and v in P, i.e., a key $\langle u, v\rangle$, and their corresponding exact shortest path $\Pi^*(u, v|C)$, i.e., a value, on C. M_{path} needs linear space in terms of the number of the exact shortest paths to be stored. Given a pair of POIs u and v, M_{path} can return the associated exact shortest path $\Pi^*(u, v|C)$ in constant time. In Figure 2 (e), M_{path} stores eight exact shortest paths on C. For the exact shortest paths between b and c, M_{path} stores $\langle b, c\rangle$ as key and $\Pi^*(b, c|C)$ as value.

The POI mapping table M_{POI} is a *hashing table* stores the POI u, i.e., a key, that we do not store all the exact shortest paths in M_{path} from u to other non-processed POIs, and the POI v, i.e., a value, that we use the exact shortest path with v as a source to approximate the shortest path with u as a source. The space consumption and query time of M_{POI} is similar to M_{path} (i.e., linear space consumption and constant query time). In Figure 2 (e), we store b as key, and a as value, since we use the exact shortest path with a as a source to approximate the shortest path with b as a source.

4.1.2 ***Phases of RC-Oracle(Point).*** There are two phases of *RC-Oracle(Point)*, i.e., *construction phase* and *shortest path query phase* (see Figure 2). (1) In the construction phase, given a point cloud C and a set of POIs P, we pre-compute the exact shortest paths between some selected pairs of POIs on C, store them in M_{path}, and store the non-selected POIs and their corresponding selected POIs in M_{POI}. (2) In the shortest path query phase, given a pair of query POIs, M_{path}, and M_{POI}, we answer the path results between this pair of POIs efficiently.

4.1.3 ***Overview of small oracle construction time of RC-Oracle(Point).*** The reason why *RC-Oracle(Point)* has a small oracle construction time is due to the (1) novel point point cloud shortest path on-the-fly path query algorithm, and the (2) novel oracle construction, that we discussed in Section 1.3.

4.1.4 ***Overview of small oracle size of RC-Oracle(Point).*** We introduce the reason why *RC-Oracle(Point)* has a small oracle size. When constructing *RC-Oracle(Point)*, we do not calculate the exact shortest path between each pair of POIs on C. Instead, we use some exact shortest paths to approximate pairwise P2P shortest path on C. In Figure 2 (d), for a pair of POIs b and d, we use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ to approximate $\Pi^*(b, d|C)$, i.e., we will not store $\Pi^*(b, d|C)$ in M_{path} for memory saving.

4.1.5 ***Overview of small shortest path query time of RC-Oracle(Point).*** We introduce the reason why *RC-Oracle(Point)* has a small shortest path query time. We use an example to illustrate it. In Figure 2 (f), in the **shortest path query** phase of *RC-Oracle(Point)*, we need to query the shortest path (1) between a and d, (2) between b and d. (1) For a and d, since $\Pi^*(a, d|C)$ is in M_{path}, we can directly return it as output. (2) For b and d, since $\Pi^*(b, d|C)$ is not in M_{path}, and b is a key in M_{POI}, so we retrieve the value a using the key b in M_{POI}, then in M_{path}, we retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, i.e., two purple lines, using $\langle b, a\rangle$ and $\langle a, d\rangle$, and use them to approximate $\Pi^*(b, d|C)$.

4.1.6 ***Overview of proximity path queries processing algorithms.*** Given a query point $q \in P$, we could perform n shortest path queries between q and all other POIs in P with the assistance of the shortest path query phase in *RC-Oracle(Point)*, then we could answer the *kNN* path query and the range path query.

4.2 Algorithm *Fly(Point)*

In algorithm *Fly(Point)*, given a point cloud C and a pair of points s and t in C, it can calculate the *exact* shortest path between s and t passes on the *points* of C, i.e., $\Pi^*(s, t|C)$. As a Dijkstra-based shortest path query algorithm [21], given a current processing point q, algorithm *Fly(Point)* will use one of the eight neighbors of q as the next searching points. In Figure 3 (a), suppose that the current processing point is green point q, algorithm *Fly(Point)* will use eight blue neighbor points as the next searching points, i.e., it performs Dijkstra algorithm on a conceptual graph as shown in Figure 3 (d).

4.3 Construction Phase

In the construction phase of *RC-Oracle(Point)*, given a point cloud C and a set of POIs P, we pre-compute the exact shortest paths between some selected pairs of POIs on C, store them in M_{path}, and store the non-selected POIs and their corresponding selected POIs in M_{POI}. As mentioned in Section 4.1.3, there are two steps involved, i.e., (1) **POIs sorting** step, and (2) **shortest path calculation** step.

Before we discuss these two steps, we need more notations. Let $P_{remain} = \{p_1, p_2, \dots\}$ be a set of remaining POIs of P on C that we have not used algorithm *Fly(Point)* to calculate the exact shortest path on C with $p_i \in P_{remain}$ as a source. P_{remain} is initialized to be P. Let $P_{dest}(q) = \{p_1, p_2, \dots\}$ be a set of POIs of P on C that we need to use algorithm *Fly(Point)* to calculate the exact shortest path on C from q to $p_i \in P_{dest}(q)$ as destinations. $P_{dest}(q)$ is empty at the beginning. In Figure 2 (c), $P_{remain} = \{c, d, e\}$ since we have not used algorithm *Fly(Point)* to calculate the exact shortest path on C with c, d, e as source. $P_{dest}(b) = \{c\}$ since we need to use algorithm *Fly(Point)* to calculate the exact shortest path on C from b to c as destinations. We then discuss the two steps.

(1) **POIs sorting:** We sort POIs based on their x-coordinate (resp. y-coordinate) in ascending order if the side length of C along x-axis is longer (resp. shorter) than that of y-axis, so we can start processing with POI that is far away from other POIs (in Figure 2 (b), since the side length of C along y-axis is longer than that of x-axis, so the sorted POIs are a, b, c, e, d).

(2) **Shortest path calculation:** We do the following two substeps iteratively until P_{remain} is empty.

- **Exact shortest path calculation:** For each POI in P_{remain}, we select the POI $u \in P_{remain}$ with the smallest x-coordinate / y-coordinate based on the sorted POIs, delete u from P_{remain}, calculate the exact shortest path on C from u to each $v \in P_{remain}$ simultaneously using algorithm *Fly(Point)*, and store $\langle u, v\rangle$ as key and $\Pi^*(u, c|C)$ as value in M_{path} (in Figure 2 (b), a has the smallest y-coordinate based on the sorted POIs in P_{remain}, we delete a from P_{remain}, so $P_{remain} = \{b, c, d, e\}$, calculate the exact

shortest paths on C from a to b, e, c, d using algorithm *Fly(Point)*, these paths are in purple lines, and store them in M_{path}).

- **Shortest path approximation:** For each POI in P_{remain}, we select u's closest POI $v \in P_{remain}$ based on the distance of the exact shortest path on C between u and v, i.e., $\Pi^*(u, v|C)$, there are two cases (in Figure 2 (c), b is the POI in P_{remain} closest to a, c is the POI in P_{remain} second closest to a):
  - **Entering approximation looping:** If $\Pi^*(u, v|C) \leq \epsilon L$, it means u and v are not far away. We delete v from P_{remain}. For each POI $w \in P_{remain}$, depending on the Euclidean distance between v and w, i.e., $d_E(v, w)$, there are two more sub-cases (in Figure 2 (c), we first select a's closest POI in P_{remain}, i.e., b, since $d_E(a, b) \leq \epsilon L$, we enter approximation looping, we delete b from P_{remain}, so $P_{remain} = \{c, d, e\}$, then for each POI in P_{remain}, we have two sub-cases):
    - **Far away POIs:** If $d_E(v, w) > \frac{2}{\epsilon} \cdot \Pi^*(u, v|C)$, it means w is far away from v, we can use $\Pi^*(v, u|C)$ and $\Pi^*(u, w|C)$ that we have already calculated before to approximate $\Pi^*(v, w|C)$, so we get the approximate shortest path $\Pi(v, w|C)$ by appending $\Pi^*(v, u|C)$ and $\Pi^*(u, w|C)$, we store v as key and u as value in M_{POI} (in Figure 2 (c), d and e are far away from b, since $d_E(b, d) > \frac{2}{\epsilon} \cdot \Pi^*(a, b|C)$ and $d_E(b, e) > \frac{2}{\epsilon} \cdot \Pi^*(a, b|C)$, so we get the approximate shortest path $\Pi(b, d|C)$ by appending $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, and get $\Pi(b, e|C)$ by appending $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$, we store b as key and a as value in M_{POI}).
    - **Close POIs:** If $d_E(v, w) \leq \frac{2}{\epsilon} \cdot \Pi^*(u, v|C)$, it means w is close to v, so we cannot use any existing exact shortest paths to approximate $\Pi^*(v, w|C)$, then we store w into $P_{dest}(v)$ (in Figure 2 (c), c is close to b, since $d_E(b, c) \leq \frac{2}{\epsilon} \cdot \Pi^*(a, b|C)$, so we cannot use any existing exact shortest path result to approximate $\Pi^*(b, c|C)$, then we store c into $P_{dest}(b)$).

    Until we have processed all POIs $w \in P_{remain}$, we use algorithm *Fly(Point)* to calculate the exact shortest paths from v to each POI in $P_{dest}(v)$, and store each of them with key-value pair in M_{path}. Note that we can terminate algorithm *Fly(Point)* earlier since we just need to cover all POIs that are close to v (in Figure 2 (c), we have processed all POIs in P_{remain} with b as a source, we have $P_{dest}(b) = \{c\}$, and we use algorithm *Fly(Point)* to calculate the exact shortest path on C between b and c, i.e., $\Pi^*(b, c|C)$ in green line, and store it in M_{path}, note that we do not need to cover d and e).
  - **Leaving approximation looping:** If $\Pi^*(u, v|C) > \epsilon L$, it means u and v are far away, and it is unlikely to have a POI m that satisfies $d_E(v, m) > \frac{2}{\epsilon} \cdot \Pi^*(u, v|C)$ (in Figure 2 (c), since we have processed b, and $P_{remain} = \{c, d, e\}$, we select a's closest POI in P_{remain}, i.e., c, since $d_E(a, c) > \epsilon L$, we leave approximation looping and terminate the iteration).

  We repeat the above two steps until P_{remain} is empty (in Figure 2 (d), we repeat the above two steps, and calculate the exact shortest paths with c as a source, and with e as a source using algorithm *Fly(Point)*, these paths are in purple lines).

4.4 Shortest Path Query Phase

In the shortest path query phase of *RC-Oracle(Point)*, a pair of POIs s and t in P, M_{path}, and M_{POI}, *RC-Oracle(Point)* can answer the associated shortest path $\Pi(s, t|C)$, which is a $(1 + \epsilon)$-approximated exact shortest path of $\Pi^*(s, t|C)$ on C in constant time. Given a pair of POIs s and t, there are two cases:

- **Retrieve exact shortest path:** If $\Pi^*(s, t|C)$ exists in M_{path}, we retrieve it using $\langle s, t\rangle$ in constant time (in Figure 2 (g), given a pair of POIs a and d, since $\Pi^*(a, d|C)$ exists in M_{path}, so we can retrieve it in constant time).
- **Retrieve approximate shortest path:** If $\Pi^*(s, t|C)$ does not exists in M_{path}, it means $\Pi^*(s, t|C)$ is approximated by two exact shortest paths in M_{path}, and (1) either s or t is a key in M_{POI}, or (2) both s and t are keys in M_{POI}. Without loss of generality, suppose that (1) s exists in M_{POI} if s or t is a key in M_{POI}, or (2) s is processed before t during construction phase if both s and t are keys in M_{POI}, we retrieve s' from M_{POI} using s in constant time, then retrieve $\Pi^*(s, s'|C)$ and $\Pi^*(s', t|C)$ from M_{path} using $\langle s, s'\rangle$ and $\langle s', t\rangle$ in constant time, and use $\Pi^*(s, s'|C)$ and $\Pi^*(s', t|C)$ to approximate $\Pi^*(s, t|C)$ (in Figure 2 (c), given a pair of POIs b and d, since $\Pi^*(b, d|C)$ does not exist in M_{path}, and b is a key in M_{POI}, so we retrieve the value a using the key b in M_{POI}, then in M_{path}, we retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, i.e., two purple lines, using $\langle b, a\rangle$ and $\langle a, d\rangle$, and use them to approximate $\Pi^*(b, d|C)$, the case for both s and t are keys in M_{POI} is similar, we just use the POI that is processed first during construction phase as the key for retrieving the value in M_{POI}).

4.5 Theoretical Analysis

The shortest path query time, memory usage, and error of algorithm *Fly(Point)* are in Theorem 4.1. The oracle construction time, oracle size, shortest path query time, and error of *RC-Oracle(Point)* are in Theorem 4.2.

Theorem 4.1. *The shortest path query time and memory usage of algorithm Fly(Point) are $O(N \log N)$ and $O(N)$, respectively. Algorithm Fly(Point) returns the exact shortest path on the point cloud.*

Proof. Since algorithm *Fly(Point)* is a Dijkstra algorithm and there are total N points, we obtain the shortest path query time and memory usage. Since algorithm *Fly(Point)* considers all points on the point cloud, and Dijkstra algorithm is guaranteed to return the exact shortest path, so algorithm *Fly(Point)* returns the exact shortest path on the point cloud. □

Theorem 4.2. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle(Point) are $O(\mu N \log N + n \log n)$, $O(\mu n)$, and $O(1)$, respectively, where μ is a data-dependent variable. In average case, μ is $O(\frac{1}{\epsilon^2})$. In worst case, μ is $O(n)$. In our experiment, $\mu \in [1, 10]$. RC-Oracle(Point) always has $|\Pi(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C)|$ for each pair of POIs s and t in P.*

Proof Sketch. The oracle construction time contains the POIs sorting time $O(n \log n)$ and the shortest path calculation time $O(\mu N \log N + n)$. The oracle size contains M_{POI} with size $O(n)$ and M_{path} with size $O(\mu n)$. The shortest path query time is due to hash table constant query time of M_{POI} and M_{path}. The error bound is due to the distance checking when constructing *RC-Oracle(Point)*. For the sake of space, the detailed proof in the remaining of the paper could be found in the appendix. □

4.6 Proximity Path Queries Processing Algorithms

We can answer other proximity path queries, i.e., the *kNN* path query and the range path query, using *RC-Oracle(Point)*.

4.6.1 ***Definitions.*** We first give the formal definitions of these two proximity path queries.

- **The *kNN* path query**: Given a point cloud C, a set of points P on C, and a query point $q \in P$, it returns all the shortest paths on C from q to a set of k POIs, denoted by $X_1 = \{u_1, u_2, \dots u_k\}$, which are k POIs in P nearest to q, in other words, $\max_{\forall u \in X_1} |\Pi^*(q, u|C)| \leq \min_{\forall o \in P \setminus X_1} |\Pi^*(q, o|C)|$.
- **The range path query**: Given a point cloud C, a range value r, a set of points P on C, and a query point $q \in P$, it returns all the shortest paths on C from q to a set of POIs, denoted by $X_2 = \{u_1, u_2, \dots\}$, which are a set of POIs in P with shortest distance to q at most r, in other words, $\max_{\forall u \in X_2} |\Pi^*(q, u|C)| \leq r$.

4.6.2 ***Algorithms.*** We then give the algorithms for processing these two proximity path queries. Given a query point $q \in P$, we first perform n shortest path queries between q and all other POIs in P with the assistance of the shortest path query phase in *RC-Oracle(Point)*. Recall that $\Pi(s, t|C)$ is the shortest path of returned by *RC-Oracle(Point)* between s and t. Then, we process them as follows.

- **The *kNN* path query**: We return the shortest paths on C from q to a set of POIs X_1' containing k POIs in P, such that $\max_{\forall u' \in X_1'} |\Pi(q, u'|C)| \leq \min_{\forall o' \in P \setminus X_1'} |\Pi^*(q, o'|C)|$.
- **The range path query**: We return the shortest paths on C from q to a set of POIs X_2' containing k POIs in P, such that $\max_{\forall u' \in X_2'} |\Pi(q, u'|C)| \leq r$.

4.6.3 ***Theoretical analysis.*** We then give the theoretical analysis of these two proximity path queries. But we need more notations first. For the *kNN* path query and the range path query, since both of them return a set of POIs, for simplicity, given a query point $q \in P$, (1) we let X be a set of POIs containing the *exact* (1a) k nearest POIs of q or (1b) POIs whose distance to q are at most r, calculated using the distance of the exact shortest path on C. Furthermore, given a query point $q \in P$, (2) we let X' be a set of POIs containing (2a) k nearest POIs of q or (2b) POIs whose distance to q are at most r, calculated using the distance of the approximated shortest path on C returned by *RC-Oracle(Point)*. We let v (resp. v') be the furthest POI to q in X (resp. X') based on the distance of the exact shortest path on C, i.e., $|\Pi^*(q, v|C)| \leq \max_{\forall v \in X} |\Pi^*(q, v|C)|$ (resp. $|\Pi^*(q, v'|C)| \leq \max_{\forall v' \in X'} |\Pi^*(q, v'|C)|$). In Figure 1 (a), suppose that the exact k nearest POIs ($k = 3$) of a is b, d, f, i.e., $X = \{b, d, f\}$. And f is the furthest POI to a among these three POIs, i.e., the value for v is f. Suppose that our *kNN* path query algorithm finds the k nearest POIs ($k = 3$) of a is b, d, e, i.e., $X' = \{b, d, e\}$. And e is the furthest POI to a among these three POIs, i.e., the value for v' is e.

We define the approximate ratio of the *kNN* path query and range path query to be $\frac{|\Pi^*(q,v'|C)|}{|\Pi^*(q,v|C)|}$, which is a real number no smaller than 1. In Figure 1 (a), the approximate ratio is $\frac{|\Pi^*(q,e|C)|}{|\Pi^*(q,f|C)|}$. Recall the error parameter of *RC-Oracle(Point)* is ϵ. Then, we show the query time and approximate ratio of *kNN* and range path query by using *RC-Oracle(Point)* in Theorem 4.3.

Theorem 4.3. *The query time and approximate ratio of both the kNN and range path query by using RC-Oracle(Point) are $O(n)$ and $1 + \epsilon$, respectively.*

Proof Sketch. The query time is due to the n time usages of the shortest path query phase of *RC-Oracle(Point)*. The approximate ratio is due to its definition and the error of *RC-Oracle(Point)*. □

4.6.4 ***Extension to other database queries.*** Given an algorithm that could answer the shortest path query on any data format (e.g., a point cloud, a *TIN*, a graph, a road network), if its shortest path query time is t, and it has the error ratio ϵ', then the query time and approximate ratio of *kNN* and range path query by using this algorithm are in Theorem 4.4.

Theorem 4.4. *The query time and approximate ratio of both the kNN and range path query by using the given algorithm are $O(nt)$ and $1 + \epsilon'$, respectively.*

Proof Sketch. The proof is similar to that in Theorem 4.3. □

5 DISCUSSION

In this section, we provide discussions on two different settings of the path, i.e., (1) path on the point cloud and the face of the implicit *TIN* constructed by the point cloud, and (2) path on the point cloud and the vertex of the implicit *TIN* constructed by the point cloud.

We denote different combinations of *on-the-fly algorithms / oracles*, and *path type* as ***A*(*P*)**. ***A*** is a placeholder for the implemented on-the-fly algorithms / oracles, with values {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*}, where (1a) *Fly* denotes the on-the-fly algorithm, (1b) *SE-Oracle* denotes the best-known oracle [54, 55], (1c) *SE-Oracle-Adapt* denotes the adapted version of the best-known oracle [54, 55] that pre-computes the shortest path between each pair of POIs using *SSAD* algorithm, (1d) *RC-Oracle-Naive* denotes the naive version of our oracle *RC-Oracle* without shortest path approximation step, and (1e) *RC-Oracle* denotes our oracle. ***P*** is a placeholder for path type, with values {*FaceExact, FaceAppr, Vertex, Point*}, where (2a) *FaceExact* denotes the path calculated by the best-known on-the-fly exact algorithm [14] that passes on the *TIN* face, (2b) *FaceAppr* denotes the path calculated by the best-known on-the-fly approximate algorithm [28] that passes on the *TIN* face, (2c) *Vertex* denotes the path calculated by Dijkstra algorithm that passes on the *TIN* vertex, and (2d) *Point* denotes the path calculated by Dijkstra algorithm that passes on the point cloud. For example, *RC-Oracle(Point)* means our oracle, and the path passes on the point cloud.

Extension to other data formats: The different combination of path type on *RC-Oracle(Point)* implies that it could be easily extended to other data formats (e.g., a *TIN*, a graph, a road network) by proposing different on-the-fly shortest path query algorithms.

5.1 Path on Point Cloud & Face of *TIN*

In this section, we provide discussions on the path passes on the point cloud, and the path passes on the face of the *TIN*.

5.1.1 Baselines. **(1) Baseline on-the-fly algorithms:** Algorithm *Fly(**P**)*, where ***P*** = {*FaceExact, FaceAppr*}. **(2) Baseline oracles:** (2a) *SE-Oracle(**P**)*, (2b) *SE-Oracle-Adapt(**P**)*, and (2c) *RC-Oracle-Naive(**P**)*,

where $\boldsymbol{P}$ = {*FaceExact, FaceAppr, Point*}, and (2e) *RC-Oracle(**P'**)*, where $\boldsymbol{P'}$ = {*FaceExact, FaceAppr*}.

5.1.2 Comparisons. We compare the distance of the exact shortest path passes on the points of the point cloud (calculated using *Fly(Point)*), and the distance of the exact shortest path passes on the faces of the implicit *TIN* (calculated using *Fly(FaceExact)*). Before we provide the lemma of this comparison, we need more notations.

Let T be an implicit *TIN* triangulated by the points in C. Given a pair of points s and t in P, let $\Pi^*(s,t|T)$ be the exact shortest path between s and t passes on the faces of *TIN* T, let $\Pi_V(s,t|T)$ be the shortest path between s and t passes on the vertices of T where these vertices belongs to the faces that $\Pi^*(s,t|T)$ passes, let $\Pi_N(s,t|T)$ be the shortest network path [49] between s and t passes on T. Note that $\Pi_N(s,t|T)$ only passes on the vertices of T. Let θ be the minimum interior angle of a triangle in T. Figure 3 (a) shows an example of $\Pi^*(s,t|C)$ in orange line, Figure 3 (b) shows an example of $\Pi^*(s,t|T)$ in blue line, $\Pi_V(s,t|T)$ in green line and $\Pi_N(s,t|T)$ in pink line. The distance relationship between $\Pi^*(s,t|C)$ and $\Pi^*(s,t|T)$ is in Lemma 5.1.

Lemma 5.1. *Given a pair of points s and t in P, we have $|\Pi^*(s,t|C)| \leq k \cdot |\Pi^*(s,t|T)|$, where $k = \max\{\frac{2}{\sin\theta}, \frac{1}{\sin\theta\cos\theta}\}$.*

Proof Sketch. According to left hand side equation in Lemma 2 of [29], we have $|\Pi_V(s,t|T)| \leq k \cdot |\Pi^*(s,t|T)|$. Since $\Pi_N(s,t|T)$ considers all the vertices on T, so $|\Pi_N(s,t|T)| \leq |\Pi_V(s,t|T)|$. In Figure 3 (a), given a green point q on C, it can connect with one of its eight blue neighbors. In Figure 3 (b), given a black vertex q on T, it can only connect with one of its six blue neighbors. So $|\Pi^*(s,t|C)| \leq |\Pi_N(s,t|T)|$. Thus, we finish the proof by combining these inequalities. □

We compare the oracle construction time, oracle size, and shortest path query time for $\boldsymbol{A(P)}$, where $\boldsymbol{A}$ = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and $\boldsymbol{P}$ = {*FaceExact, FaceAppr, Point*} in Table 1. The detailed theoretical analysis with proofs of these baselines could be found in the appendix.

5.2 Path on Point Cloud & Vertex of *TIN*

In this section, we provide discussions on the path passes on the point cloud, and the path passes on the vertex of the *TIN*.

5.2.1 Baselines. **(1) Baseline on-the-fly algorithms:** Algorithm *Fly(Vertex)*. **(2) Baseline oracles:** $\boldsymbol{A}$*(Vertex)* where $\boldsymbol{A}$ = {*SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*}.

5.2.2 Comparisons. We compare the distance of the exact shortest path passes on the points of the point cloud (calculated using *Fly(Point)*), and the distance of the shortest path passes on vertices of the implicit *TIN*, i.e., the shortest network distance, (calculated using *Fly(Vertex)*). Given a pair of points s and t in P, the distance relationship between $\Pi^*(s,t|C)$ and $\Pi_N(s,t|T)$ is in Lemma 5.2.

Lemma 5.2. *Given a pair of points s and t in P, we have $|\Pi^*(s,t|C)| \leq |\Pi_N(s,t|T)|$.*

Proof Sketch. The proof is similar to that in Lemma 5.1. □

We compare the oracle construction time, oracle size, and shortest path query time for $\boldsymbol{A(P)}$, where $\boldsymbol{A}$ = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and $\boldsymbol{P}$ = {*Vertex, Point*} in Table 1. The detailed theoretical analysis with proofs of these baselines could be found in the appendix.

6 EMPIRICAL STUDIES

6.1 Experimental Setup

We conducted our experiments on a Linux machine with 2.2 GHz CPU and 512GB memory. All algorithms were implemented in C++. For the following experiment setup, we mainly follow the experiment setup in the work [28, 29, 38, 54, 55].

Datasets: We conducted our experiment based on six real point cloud datasets, which are (1) *BearHead* (*BH*, with 0.5M points) [2, 54, 55], (2) *EaglePeak* (*EP*, with 0.5M points) [2, 54, 55], (3) *Robinson Mountain* [4, 53] (*RM*, with 0.5M points), (4) a small-version of *BH* (*BH-small*, with 10k points), (5) a small-version of *EP* (*EP*, with 10k points), and (6) a small-version of *RM* (*RM-small*, with 10k points), respectively. For *BH* and *EP* datasets, they are originally represented as a *TIN*. We remove the edges and faces to obtain the point cloud. For *RM*, we first obtain the satellite map from [3], and then used Blender [1] to generate the point cloud. According to [19, 38, 49, 54, 55], these three datasets have a resolution of 10m. Following [54, 55], we extracted 500 POIs using OpenStreetMap. For *BH-small*, *EP-small* and *RM-small* datasets, we use the same region of the *BH*, *EP* and *RM* datasets with a resolution of 70m to generate them. We follow the generation procedure of multi-resolution dataset in [38, 54, 55] to generate *BH-small*, *EP-small* and *RM-small* datasets. This procedure could be found in the appendix. The reason for having these three small version datasets is that $\boldsymbol{A(P)}$, where $\boldsymbol{A}$ = {*SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive*} and $\boldsymbol{P}$ = {*FaceExact, FaceAppr, Vertex, Point*} are not feasible on any of the large datasets with 500 POIs due to their expensive oracle construction time. In addition, we have four sets of datasets with different number of points (three sets of large-version datasets and one set of small-version datasets) for testing the scalability of our oracle, which are generated using *BH*, *EP*, *RM* and *EP-small* datasets with the same multi-resolution dataset generation procedure.

Algorithms: We divide the comparison algorithms into two types, i.e., (1) our algorithm / oracle that the calculated path passes on the point cloud and on the faces of the implicit *TIN*, i.e., **path on point cloud & face of *TIN*** (we denote it as *Point-Face*), and (2) our algorithm / oracle that the calculated path passes on the point cloud and on the vertices of the implicit *TIN*, i.e., **path on point cloud & vertex of *TIN*** (we denote it as *Point-Vertex*).

(1) For the *Point-Face* type, $\boldsymbol{A(P)}$, where $\boldsymbol{A}$ = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and $\boldsymbol{P}$ = {*FaceExact, FaceAppr, Point*} are studied in the experiments. We combined different types of *on-the-fly algorithms / oracles* and *path type* for **ablation test**. The path calculated by *Fly(FaceExact)* is regarded as the exact shortest path because the path passes on the *TIN*. Since $\boldsymbol{A(P)}$, where $\boldsymbol{A}$ = {*SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive*} and $\boldsymbol{P}$ = {*FaceExact, FaceAppr, Point*} are not feasible on large datasets due to their expensive oracle construction time, so we (1a) compared $\boldsymbol{A(P)}$, where $\boldsymbol{A}$ = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and $\boldsymbol{P}$ = {*FaceExact, FaceAppr, Point*} on *BH-small*, *EP-small*,

Algorithm	Oracle construction time		Oracle size		Shortest path query time		Error
SE-Oracle(FaceExact) [14, 54, 55]	$O(N + \frac{nhN^2}{\epsilon^{2\beta}})$	Gigantic	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
SE-Oracle(FaceAppr) [28, 54, 55]	$O(N + \frac{nhl_{max}N}{\epsilon^{(2\beta+1)}l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$	Gigantic	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
SE-Oracle(Vertex)	$O(N + \frac{nhN\log N}{\epsilon^{2\beta}})$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Medium
SE-Oracle(Point)	$O(\frac{nhN\log N}{\epsilon^{2\beta}})$	Medium	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Medium
SE-Oracle-Adapt(FaceExact)	$O(N + nN^2 + nh\log n + \frac{nh}{\epsilon^{2\beta}})$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
SE-Oracle-Adapt(FaceAppr)	$O(N + \frac{nl_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + nh\log n + \frac{nh}{\epsilon^{2\beta}})$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
SE-Oracle-Adapt(Vertex)	$O(N + nN\log N + nh\log n + \frac{nh}{\epsilon^{2\beta}})$	Medium	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Medium
SE-Oracle-Adapt(Point)	$O(nN\log N + nh\log n + \frac{nh}{\epsilon^{2\beta}})$	Small	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Medium
RC-Oracle-Naive(FaceExact)	$O(N + nN^2 + n^2)$	Large	$O(n^2)$	Large	$O(1)$	Tiny	No error
RC-Oracle-Naive(FaceAppr)	$O(N + \frac{nl_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + n^2)$	Large	$O(n^2)$	Large	$O(1)$	Tiny	Small
RC-Oracle-Naive(Vertex)	$O(N + nN\log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Tiny	Medium
textitRC-Oracle-Naive(Point)	$O(nN\log N + n^2)$	Small	$O(n^2)$	Large	$O(1)$	Tiny	Medium
RC-Oracle(FaceExact)	$O(N + \mu N^2 + n\log n)$	Medium	$O(\mu n)$	Small	$O(1)$	Tiny	Small
RC-Oracle(FaceAppr)	$O(N + \frac{\mu l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + n\log n)$	Medium	$O(\mu n)$	Small	$O(1)$	Tiny	Small
RC-Oracle(Vertex)	$O(N + \mu N\log N + n\log n)$	Small	$O(\mu n)$	Small	$O(1)$	Tiny	Medium
***RC-Oracle(Point)* (ours)**	$\boldsymbol{O(\mu N\log N + n\log n)}$	**Tiny**	$\boldsymbol{O(\mu n)}$	**Small**	$\boldsymbol{O(1)}$	**Tiny**	**Medium**
Fly(FaceExact) [14]	-	N/A	-	N/A	$O(N + N^2)$	Large	No error
Fly(FaceAppr) [28]	-	N/A	-	N/A	$O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$	Large	Small
Fly(Vertex)	-	N/A	-	N/A	$O(N + N\log N)$	Medium	Medium
***Fly(Point)* (ours)**	-	N/A	-	N/A	$\boldsymbol{O(N\log N)}$	**Medium**	**Medium**

Table 1: Comparison of different algorithms

Remark: $n << N$, h is the height of the compressed partition tree, β is the largest capacity dimension [24, 30], θ is the minimum inner angle of any face in T, l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of T. μ is a data-dependent variable. In the average case, μ is $O(\frac{1}{\epsilon^2})$. In the worst case, μ is $O(n)$. In our experiment, $\mu \in [1, 10]$. We include the implicit *TIN* construction time $O(N)$ explicitly for all implicit *TIN* based algorithms / oracles.

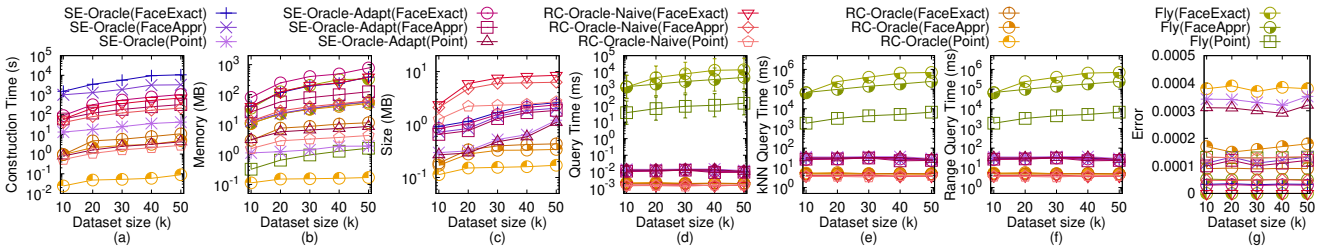

Figure 4: Effect of N on *EP-small* dataset for *Point-Face* type

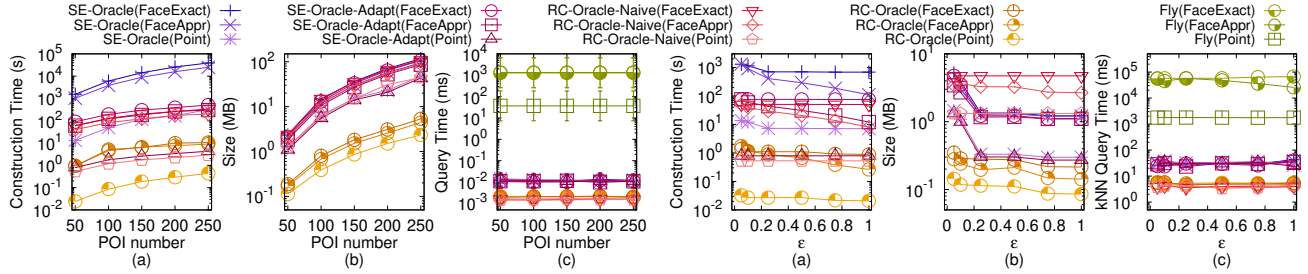

Figure 5: Effect of n on *EP-small* dataset for *Point-Face* type **Figure 6: Effect of ϵ on *RM-small* dataset for *Point-Face* type**

RM-small, and the set of small-version datasets with default 50 POIs, and (1b) compared ***A*(*P*)**, where ***A*** = {*Fly, RC-Oracle*} and ***P*** = {*Face-Exact, FaceAppr, Point*} on *BH*, *EP*, *RM*, and the set of large-version datasets with default 500 POIs.

(2) For the *Point-Vertex* type, ***A*(*P*)**, where ***A*** = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and ***P*** = {*Vertex, Point*} are studied in the experiments. We combined different types of *on-the-fly algorithms / oracles* and *path type* for **ablation test**. The path calculated by *Fly(Point)* is regarded as the exact shortest path because the path we consider here passes on points of the point cloud or the vertex of the *TIN*. Since ***A*(*P*)**, where ***A*** = {*SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive*} and ***P*** = {*Vertex, Point*} are not feasible with 500 POIs due to their expensive oracle construction time, so we (2a) compared ***A*(*P*)**, where ***A*** = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and ***P*** = {*Vertex, Point*}

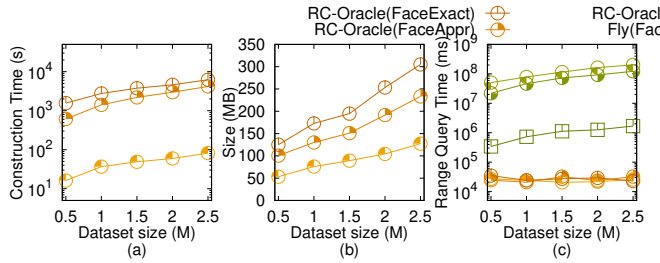

Figure 7: Effect of N on *RM* dataset for *Point-Face* type

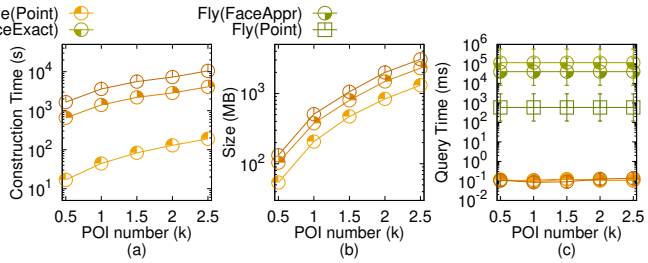

Figure 8: Effect of n on *BH* dataset for *Point-Face* type

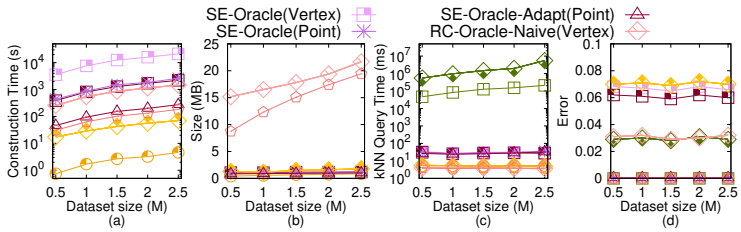

Figure 9: Effect of N on *BH* dataset for *Point-Vertex* type

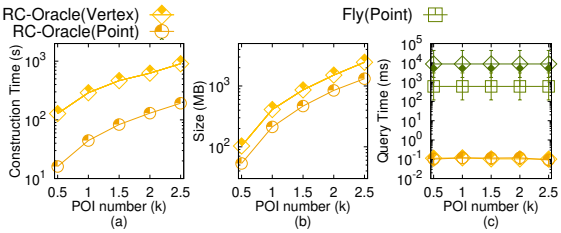

Figure 10: Effect of n on *EP* dataset for *Point-Vertex* type

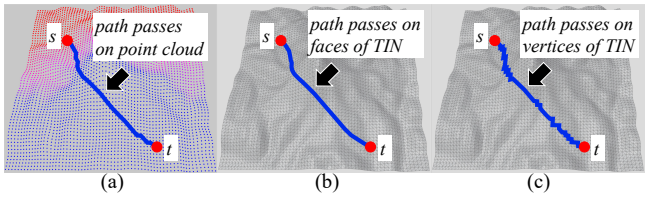

Figure 11: Shortest path result on point cloud and *TIN*

on *BH*, *EP*, *RM*, and the set of large-version datasets with small-version POIs (50 POIs as default), and (2b) compared ***A(P)***, where ***A*** = {*Fly, RC-Oracle*} and ***P*** = {*Vertex, Point*} on same datasets with large-version POIs (500 POIs as default)

Query Generation: We conducted all proximity path queries, i.e., (1) shortest path query, (2) all POIs *kNN* path query, and (3) all POIs range path query. (1) For the shortest path query, we randomly chose two POIs in P on C, one as a source and the other as a destination, then 100 queries were answered and the average, minimum and maximum result was returned (in the experimental result figures, the vertical bar and the points mean the minimum, maximum and average result). (2) For the all POIs *kNN* path query, we first calculate the shortest path from each POI to all other POIs, then select the k nearest POIs as output, and we repeat this for all POIs. (3) For the all POIs range path query, we first calculate the shortest path from each POI to all other POIs, then select the POIs that are no further than a given distance value, i.e., r, of the current POI, and we repeat this for all POIs. For the *kNN* path query and range path query, since we first need to find the shortest path from a given POI to all other POIs, then select k paths with the smallest distance, or the paths smaller than r, the value of k and r will not affect their query time, so we set k to be 3 and r to be 1km.

Factors & Measurements: We studied three factors in the experiments, namely (1) N (i.e., the number of points in a point cloud), (2) n (i.e., the number of POIs), and (3) ϵ (i.e., the error parameter). Since both the on-the-fly shortest path query algorithm and the oracle in ***A(FaceAppr)*** where ***A*** = {*SE-Oracle, SE-Oracle-Adapt, RC-Oracle*} are based on ϵ, so we assign the error in both of their on-the-fly shortest path query algorithms and their oracles to be $\sqrt{(1+\epsilon)}-1$. In addition, we used seven measurements to evaluate the algorithm performance, namely (1) *oracle construction time*, (2) *memory usage* (i.e., the space consumption when running the algorithm), (3) *oracle size*, (4) *shortest path query time*, (5) *all POIs kNN path query time*, (6) *all POIs range path query time*, (7) *distance error*, (8) *kNN path query error*, and (9) *range path query error*. But, the *kNN* path query error and range path query error are all equal to 0 for all the experiments we tested (since the distance error is very small), so their results are omitted.

6.2 Experimental Results

Figure 4 to 10 show the P2P proximity path query result when varying N, n, and ϵ on *EP-small*, *RM-small*, *RM*, *BH* and *EP* datasets. Figure 11 shows the shortest path result passes on point cloud, faces of *TIN* and vertices of *TIN* of Mount Rainier in an area of 20km × 20km. The shortest path passes on the point cloud and the faces of *TIN* are very similar, but calculating the shortest path on the point cloud is much faster than on the faces of *TIN*. The results on other combinations of dataset, the variation of N, n, and ϵ, and the A2A proximity path query could be found in the appendix.

6.2.1 Path on Point Cloud & Face of TIN. We first show the experimental results for ***A(P)***, where ***A*** = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and ***P*** = {*FaceExact, FaceAppr, Point*}, such that the calculated path passes on points of the point cloud and the faces of the implicit *TIN*.

Effect of N (scalability test) for P2P proximity path query. In Figure 4 (resp. Figure 7), we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} (resp. {0.5M, 1M, 1.5M, 2M, 2.5M}) on *EP-small* (resp.

RM) dataset by setting n to be 50 and ϵ to be 0.1 (n to be 500 and ϵ to be 0.1) for scalability test (with total 20 datasets). *RC-Oracle(Point)* superior performance of all the remaining algorithms in terms of all measurements. The distance error of *RC-Oracle(Point)* and *Fly(Point)* are also very small.

Effect of n for P2P proximity path query. In Figure 5 (resp. Figure 8), we tested 5 values of n from {50, 100, 150, 200, 250} (resp. {500, 1000, 1500, 2000, 2500}) on *EP-small* (resp. *BH*) dataset by setting N to be 10k and ϵ to be 0.1 (N to be 0.5M and ϵ to be 0.1). In Figure 5 (a) (resp. Figure 8 (a)), the oracle construction time for *SE-Oracle(Point)* (resp. the shortest path query time for *RC-Oracle(FaceExact)*) is very large compared with *RC-Oracle(Point)*, which shows the superior performance of *RC-Oracle(Point)* in terms of the oracle construction (resp. shortest path querying).

Effect of ϵ for P2P proximity path query. In Figure 6, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *RM-small* dataset by setting N to be 10k and n to be 50. The oracle construction time, oracle size, and all POIs *kNN* path query time of *RC-Oracle(Point)* still perform better than the best-known oracle *SE-Oracle(FaceExact)*, and other adapted algorithms / oracles.

A2A proximity path query. We tested the A2A proximity path query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 5k on a multi-resolution of *EP* dataset. We selected 50 points as reference points for the *kNN* path query and range path query. The result could be found in the appendix. *RC-Oracle(Point)* still performs much better than the best-known oracle *SE-Oracle(FaceExact)*, and other adapted algorithms / oracles in terms of all measurements.

6.2.2 Path on Point Cloud & Vertex of TIN. We then show the experimental results for $\boldsymbol{A(P)}$, where $\boldsymbol{A}$ = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and $\boldsymbol{P}$ = {*Vertex, Point*}, such that the calculated path passes on points of the point cloud and vertices of the implicit *TIN*.

Effect of N (scalability test) for P2P proximity path query. In Figure 9, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on *BH* dataset (with small-version POIs) by setting n to be 50 and ϵ to be 0.1 for scalability test (with total 15 datasets). *RC-Oracle(Point)* still superior performance all baselines. The distance error of $\boldsymbol{A}$*(Vertex)*, where $\boldsymbol{A}$ = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} are large.

Effect of n for P2P proximity path query. In Figure 10, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on *EP* dataset (with large-version POIs) by setting N to be 0.5M and ϵ to be 0.1. Even though *RC-Oracle(Vertex)* and algorithm *Fly(Vertex)* pass on the vertices of the *TIN*, they still perform worse than *RC-Oracle(Point)* and algorithm *Fly(Point)*.

Effect of ϵ for P2P proximity path query. We also tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1}. The result is similar to Figure 6. The experiment figures could be found in the appendix.

A2A proximity path query. We tested the A2A proximity path query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 10k on a multi-resolution of *EP* dataset. We selected 50 points as reference points for the *kNN* path query and range path query. The result could be found in the appendix. *RC-Oracle(Point)* still perform much better than other adapted algorithms / oracles.

6.3 Case Study

We conducted a case study on an evacuation simulation in Mount Rainier [45] due to the frequent heavy snowfall [46] as mentioned in Section 1.1. Recall that the evacuation (walking from multiple viewing platforms on the mountain to the hotels) is expected to be finished in 2.4 hours. Figure 1 (a) shows the satellite map (from [3]) of Mount Rainier. We conduct the *kNN* path query to find the shortest paths (in blue and purple lines) from different viewing platforms on the mountain to k-nearest hotels (a is one of the viewing platforms, b to g are the hotels). b, e, and f are the three nearest hotels to this viewing platform. Figure 1 (b) shows the shortest paths from a to b, e, f pass on the points of the point cloud, Figure 1 (c) shows the same shortest paths pass on the *TIN* constructed by the point cloud. The shortest paths pass on the point cloud and the *TIN* are very similar (the distance error for the shortest paths pass on the point cloud and the faces of the *TIN* is only 3×10^{-4}). Our experimental result shows that for a point cloud of Mount Rainier with 10k points and 250 POIs (125 viewing platforms and 125 hotels), *RC-Oracle(Point)* just needs 0.4s for constructing the oracle and 0.03s for calculating 10 nearest hotels, but the best-known oracle *SE-Oracle(FaceExact)* needs 39,000s $\approx$ 10.8 hours and 0.1s. Furthermore, for a point cloud of Mount Rainier with 0.5M points and 500 POIs (250 viewing platforms and 250 hotels), *RC-Oracle(Point)* needs 12.5s for returning 10 nearest hotels, but the best-knows on-the-fly algorithm *Fly(FaceExact)* needs 29,000s $\approx$ 8.1 hours and the best-knows oracle *SE-Oracle(FaceExact)* needs 75s.

6.4 Experimental Results Summary

RC-Oracle(Point) consistently outperforms the best-known oracle, i.e., *SE-Oracle(FaceExact)*, and all other adapted baselines in terms of all measurements. Specifically, *RC-Oracle(Point)* is up to 97,500 times, 2 times, and 6 times better than *SE-Oracle(FaceExact)*, in terms of the oracle construction time, oracle size and shortest path query time. With the assistance of *RC-Oracle(Point)*, our algorithms for the *kNN* path query and the range path query are both up to 6 times faster than *SE-Oracle(FaceExact)*. For a point cloud with 10k points and 250 POIs, *RC-Oracle(Point)*'s oracle construction time is 0.4s, but the best-known oracle *SE-Oracle(FaceExact)* needs 39,000s $\approx$ 10.8 hours. When the point cloud has 0.5M points and 500 POIs, the all POIs *kNN* path query time and the all POIs range path query time of *RC-Oracle(Point)* are both 25s, but the results are both 58,000s $\approx$ 16.2 hours for the best-known on-the-fly algorithm, i.e., algorithm *Fly(FaceExact)*, and are both 150s for the best-known oracle, i.e., *SE-Oracle(FaceExact)*.

7 CONCLUSION

In our paper, we propose an efficient $(1+\epsilon)$-approximate shortest path oracle on a point cloud called *RC-Oracle(Point)*, which has a good performance (in terms of the oracle construction time, oracle size, and shortest path query time) compared with the best-known oracle. With the assistance of *RC-Oracle(Point)*, we propose algorithms for answering other proximity path queries, i.e., the *kNN* path query and the range path query. The future work could be proposing a new pruning step to further reduce the oracle construction time and oracle size.

REFERENCES

[1] 2022. *Blender.* https://www.blender.org
[2] 2022. *Data Geocomm.* http://data.geocomm.com/
[3] 2022. *Google Earth.* https://earth.google.com/web
[4] 2022. *Robinson Mountain.* https://en.wikipedia.org/wiki/Robinson_Mountain
[5] 2023. *Google Map.* https://www.google.com/maps
[6] 2023. *k-nearest neighbors algorithm.* https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
[7] 2023. *Preferred walking speed.* https://en.wikipedia.org/wiki/Preferred_walking_speed
[8] 2023. *Range query.* https://en.wikipedia.org/wiki/Range_query_(database)
[9] 2023. *Snow.* https://en.wikipedia.org/wiki/Snow
[10] Mithil Aggarwal. 2022. *More than 60 killed in blizzard wreaking havoc across U.S.* https://www.cnbc.com/2022/12/26/death-toll-rises-to-at-least-55-as-freezing-temperatures-and-heavy-snow-wallop-swaths-of-us.html
[11] Gergana Antova. 2019. Application of areal change detection methods using point clouds data. In *IOP Conference Series: Earth and Environmental Science*, Vol. 221. IOP Publishing, 012082.
[12] Paul B Callahan and S Rao Kosaraju. 1995. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)* 42, 1 (1995), 67–90.
[13] Joseph Carsten, Arturo Rankin, Dave Ferguson, and Anthony Stentz. 2007. Global path planning on board the mars exploration rovers. In *2007 IEEE Aerospace Conference*. IEEE, 1–11.
[14] Jindong Chen and Yijie Han. 1990. Shortest Paths on a Polyhedron. In *SOCG*. New York, NY, USA, 360–369.
[15] The Conversation. 2022. *How is snowfall measured? A meteorologist explains how volunteers tally up winter storms.* https://theconversation.com/how-is-snowfall-measured-a-meteorologist-explains-how-volunteers-tally-up-winter-storms-175628
[16] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.
[17] Yaodong Cui, Ren Chen, Wenbo Chu, Long Chen, Daxin Tian, Ying Li, and Dongpu Cao. 2021. Deep learning for image and point cloud fusion in autonomous driving: A review. *IEEE Transactions on Intelligent Transportation Systems* 23, 2 (2021), 722–739.
[18] Ke Deng, Heng Tao Shen, Kai Xu, and Xuemin Lin. 2006. Surface k-NN query processing. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 78–78.
[19] Ke Deng and Xiaofang Zhou. 2004. Expansion-based algorithms for finding single pair shortest path on surface. In *International Workshop on Web and Wireless Geographical Information Systems*. Springer, 151–166.
[20] Brett G Dickson and P Beier. 2007. Quantifying the influence of topographic position on cougar (Puma concolor) movement in southern California, USA. *Journal of Zoology* 271, 3 (2007), 270–277.
[21] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.
[22] David Eriksson and Evan Shellshear. 2014. Approximate distance queries for path-planning in massive point clouds. In *2014 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, Vol. 2. IEEE, 20–28.
[23] David Eriksson and Evan Shellshear. 2016. Fast exact shortest distance queries for massive point clouds. *Graphical Models* 84 (2016), 28–37.
[24] Mingyu Fan, Hong Qiao, and Bo Zhang. 2009. Intrinsic dimension estimation of manifolds by incising balls. *Pattern Recognition* 42, 5 (2009), 780–787.
[25] Fresh Off The Grid. 2022. *Winter Hiking 101: Everything you need to know about hiking in snow.* https://www.freshoffthegrid.com/winter-hiking-101-hiking-in-snow/
[26] Anupam Gupta, Robert Krauthgamer, and James R Lee. 2003. Bounded geometries, fractals, and low-distortion embeddings. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE, 534–543.
[27] GreenValley International. 2023. *3D Point Cloud Data and the Production of Digital Terrain Models.* https://geo-matching.com/content/3d-point-cloud-data-and-the-production-of-digital-terrain-models
[28] Manohar Kaul, Raymond Chi-Wing Wong, and Christian S Jensen. 2015. New lower and upper bounds for shortest distance queries on terrains. *Proceedings of the VLDB Endowment* 9, 3 (2015), 168–179.
[29] Manohar Kaul, Raymond Chi-Wing Wong, Bin Yang, and Christian S Jensen. 2013. Finding shortest paths on terrains by killing two birds with one stone. *Proceedings of the VLDB Endowment* 7, 1 (2013), 73–84.
[30] Balázs Kégl. 2002. Intrinsic dimension estimation using packing numbers. *Advances in neural information processing systems* 15 (2002).
[31] Marcel Körtgen, Gil-Joo Park, Marcin Novotni, and Reinhard Klein. 2003. 3D shape matching with 3D shape contexts. In *The 7th central European seminar on computer graphics*, Vol. 3. Citeseer, 5–17.
[32] Baki Koyuncu and Erkan Bostancı. 2009. 3D battlefield modeling and simulation of war games. *Communications and Information Technology proceedings* (2009).
[33] Russell LaDuca. 2020. *What would happen to me if I was buried under snow?* https://qr.ae/prt6zQ
[34] Mark Lanthier, Anil Maheshwari, and J-R Sack. 2001. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica* 30, 4 (2001), 527–562.
[35] Lik-Hang Lee, Tristan Braud, Pengyuan Zhou, Lin Wang, Dianlei Xu, Zijun Lin, Abhishek Kumar, Carlos Bermejo, and Pan Hui. 2021. All one needs to know about metaverse: A complete survey on technological singularity, virtual ecosystem, and research agenda. *arXiv preprint arXiv:2110.05352* (2021).
[36] Lik-Hang Lee, Zijun Lin, Rui Hu, Zhengya Gong, Abhishek Kumar, Tangyao Li, Sijia Li, and Pan Hui. 2021. When creators meet the metaverse: A survey on computational arts. *arXiv preprint arXiv:2111.13486* (2021).
[37] Ying Li, Lingfei Ma, Zilong Zhong, Fei Liu, Michael A Chapman, Dongpu Cao, and Jonathan Li. 2020. Deep learning for lidar point clouds in autonomous driving: A review. *IEEE Transactions on Neural Networks and Learning Systems* 32, 8 (2020), 3412–3432.
[38] Lian Liu and Raymond Chi-Wing Wong. 2011. Finding shortest path on land surface. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 433–444.
[39] Anders Mårell, John P Ball, and Annika Hofgaard. 2002. Foraging and movement paths of female reindeer: insights from fractal analysis, correlated random walks, and Lévy flights. *Canadian Journal of Zoology* 80, 5 (2002), 854–865.
[40] Joseph SB Mitchell, David M Mount, and Christos H Papadimitriou. 1987. The discrete geodesic problem. *SIAM J. Comput.* 16, 4 (1987), 647–668.
[41] Geo Week News. 2022. *Tesla using radar to generate point clouds for autonomous driving.* https://www.geoweeknews.com/news/tesla-using-radar-generate-point-clouds-autonomous-driving
[42] Janet E Nichol, Ahmed Shaker, and Man-Sing Wong. 2006. Application of high-resolution stereo satellite images to detailed landslide hazard assessment. *Geomorphology* 76, 1-2 (2006), 68–75.
[43] Sebastian Pütz, Thomas Wiemann, Jochen Sprickerhof, and Joachim Hertzberg. 2016. 3d navigation mesh generation for path planning in uneven terrain. *IFAC-PapersOnLine* 49, 15 (2016), 212–217.
[44] Fabio Remondino. 2003. From point cloud to surface: the modeling and visualization problem. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 34 (2003).
[45] National Park Service. 2022. *Mount Rainier.* https://www.nps.gov/mora/index.htm
[46] National Park Service. 2022. *Mount Rainier Annual Snowfall Totals.* https://www.nps.gov/mora/planyourvisit/annual-snowfall-totals.htm
[47] National Park Service. 2022. *Mount Rainier Frequently Asked Questionss.* https://www.nps.gov/mora/faqs.htm
[48] National Weather Service. 2023. *Measuring Snow.* https://www.weather.gov/dvn/snowmeasure
[49] Cyrus Shahabi, Lu-An Tang, and Songhua Xing. 2008. Indexing land surface for efficient knn query. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1020–1031.
[50] Jamie Shotton, John Winn, Carsten Rother, and Antonio Criminisi. 2006. Textonboost: Joint appearance, shape and context modeling for multi-class object recognition and segmentation. In *European conference on computer vision*. Springer, 1–15.
[51] Barak Sober, Robert Ravier, and Ingrid Daubechies. 2020. Approximating the riemannian metric from point clouds via manifold moving least squares. *arXiv preprint arXiv:2007.09885* (2020).
[52] Spatial. 2022. *LiDAR Scanning with Spatial's iOS App.* https://support.spatial.io/hc/en-us/articles/360057387631-LiDAR-Scanning-with-Spatial-s-iOS-App
[53] Open Topography. 2022. *USGS 1/3 arc-second Digital Elevation Model.* https://doi.org/10.5069/G98K778D
[54] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, and David M. Mount. 2017. Distance oracle on terrain surface. In *SIGMOD/PODS'17*. New York, NY, USA, 1211–1226.
[55] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, David M Mount, and Hanan Samet. 2022. Proximity queries on terrain surface. *ACM Transactions on Database Systems (TODS)* (2022).
[56] Songhua Xing, Cyrus Shahabi, and Bei Pan. 2009. Continuous monitoring of nearest neighbors on land surface. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1114–1125.
[57] Da Yan, Zhou Zhao, and Wilfred Ng. 2012. Monochromatic and bichromatic reverse nearest neighbor queries on land surfaces. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 942–951.
[58] Yinzhao Yan and Raymond Chi-Wing Wong. 2021. Path Advisor: a multi-functional campus map tool for shortest path. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2683–2686.
[59] Hongchuan Yu, Jian J Zhang, and Zheng Jiao. 2014. Geodesics on point clouds. *Mathematical Problems in Engineering* 2014 (2014).

A SUMMARY OF FREQUENT USED NOTATIONS

Table 2 shows a summary of frequent used notations.

Notation	Meaning
C	The point cloud with a set of points
N	The number of points of C
L	The maximum side length of C
$N(p)$	A set of neighbor points of p
$d_E(p, p')$	The Euclidean distance between point p and p'
$\Pi^*(s, t\|C)$	The exact shortest path between s and t passes on the points of C
$\|\Pi^*(s, t\|C)\|$	The length of $\Pi^*(s, t\|C)$
$\Pi(s, t\|C)$	The shortest path between s and t returned by *RC-Oracle(Point)*
P	The set of POI
n	The number of vertices of P
ϵ	The error parameter
μ	A data-dependent variable, which is $O(\frac{1}{\epsilon^2})$ in the average case and $O(n)$ the worst case
M_{path}	A hashing table stores the selected pairs of POIs u and v in P, i.e., a key $\langle u, v\rangle$, and their corresponding exact shortest path $\Pi^*(s, t\|C)$, i.e., a value, on C
M_{POI}	A hashing table stores the POI u, i.e., a key, that we do not store all the exact shortest paths in M_{path} from u to other non-processed POIs, and the POI v, i.e., a value, that we use the exact shortest path with v as a source to approximate the shortest path with u as a source
P_{remain}	A set of remaining POIs of P on C that we have not used algorithm *Fly(Point)* to calculate the exact shortest path on C with $p_i \in P_{remain}$ as source
$P_{dest}(q)$	A set of POIs of P on C that we need to use algorithm *Fly(Point)* to calculate the exact shortest path on C from q to $p_i \in P_{dest}(q)$ as destinations
T	The implicit *TIN* constructed by C
h	The height of the compressed partition tree
β	The largest capacity dimension
θ	The minimum inner angle of any face in T
l_{max}/l_{min}	The length of the longest / shortest edge of T
$\Pi^*(s, t\|T)$	The exact shortest path between s and t passes on the *TIN* T
$\Pi_V(s, t\|T)$	The shortest path between s and t passes on the vertices of T where these vertices belongs to the faces that $\Pi^*(s, t\|T)$ passes
$\Pi_N(s, t\|T)$	The shortest network path between s and t passes on T

Table 2: Summary of frequent used notations

B A2A PROXIMITY PATH QUERY

Apart from the P2P proximity path query that we discussed in the main body of this paper, we also present an oracle to answer the *any points-to-any points (A2A) shortest path query* based on our oracle *RC-Oracle(Point)*. This adapted oracle is similar to the one presented in Section 4, the only difference is that we need to create POIs which has the same coordinate values as all points in the point cloud, then *RC-Oracle(Point)* could answer the A2A proximity path query. In this case, the number of POI becomes N. Thus, for the A2A proximity path query, the oracle construction time, oracle size, and shortest path query time of *RC-Oracle(Point)* are $O(\mu' N \log N)$, $O(\mu' N)$, and $O(1)$, respectively, where μ' is a data-dependent variable. In average case, μ' is $O(\frac{1}{\epsilon^2})$. In worst case, μ' is $O(N)$. In our experiment, $\mu' \in [1, 1000]$. For the A2A proximity path query, *RC-Oracle(Point)* always has $|\Pi(s, t|C)| \leq (1+\epsilon)|\Pi^*(s, t|C)|$ for each pair of points s and t in C. The query time and approximate ratio of both the A2A *kNN* and range path query by using *RC-Oracle(Point)* are $O(N)$ and $1+\epsilon$, respectively.

C EMPIRICAL STUDIES

C.1 Experimental Results on the P2P proximity path query

C.1.1 Path on Point Cloud & Face of TIN. We first show the experimental results for ***A(P)***, where ***A*** = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and ***P*** = {*FaceExact, FaceAppr, Point*}, such that the calculated path passes on points of the point cloud and the faces of the implicit *TIN*.

Effect of N (scalability test). In Figure 4, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on *EP-small* dataset by setting n to be 50 and ϵ to be 0.1 for scalability test (with total 5 datasets). In Figure 18, Figure 21 and Figure 24, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on *BH*, *EP* and *RM* dataset by setting n to be 500 and ϵ to be 0.1 for scalability test (with total 15 datasets). *RC-Oracle(Point)* superior performance of all the remaining algorithms in terms of the oracle construction time, memory usage, oracle size, shortest path query time, all POIs *kNN* path query time, and all POIs range path query time. Even though the distance error of *RC-Oracle(Point)* and *Fly(Point)* are larger than other algorithms, the values are no more than 0.0005, which is very small.

Effect of n. In Figure 12, Figure 14 and Figure 16, we tested 5 values of n from {50, 100, 150, 200, 250} on *BH-small*, *EP-small* and *RM-small* dataset by setting N to be 10k and ϵ to be 0.1. In Figure 19, Figure 22 and Figure 25, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on *BH*, *EP* and *RM* dataset by setting N to be 0.5M and ϵ to be 0.1. For all the oracles, *RC-Oracle(Point)* has the smallest oracle construction time, memory usage and size. For all the on-the-fly algorithms, algorithm *Fly(Point)* has the smallest memory usage and shortest path query time. When we need to conduct all POIs *kNN* path query and all POIs range path query, all the on-the-fly algorithms will have a very large running time, so we need to use *RC-Oracle(Point)* for time-saving.

Effect of ϵ. In Figure 13, Figure 15 and Figure 17, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *BH*, *EP* and *RM* dataset by setting N to be 10k and n to be 50. In Figure 20, Figure 23 and Figure 26, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5,

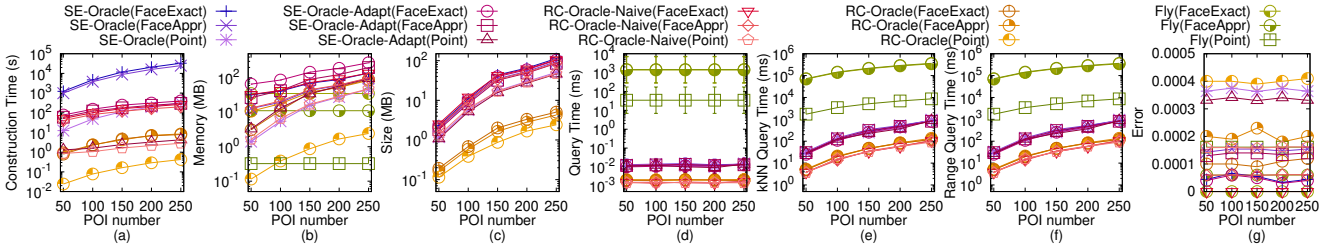

Figure 12: Effect of n on *BH-small* dataset for *Point-Face* type (P2P proximity path query)

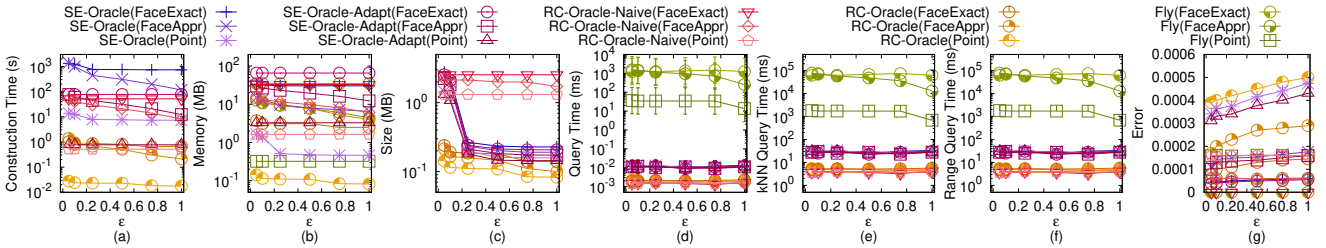

Figure 13: Effect of ϵ on *BH-small* dataset for *Point-Face* type (P2P proximity path query)

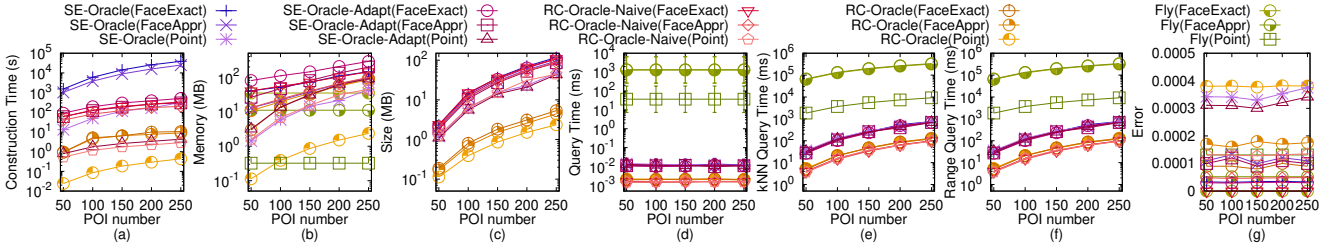

Figure 14: Effect of n on *EP-small* dataset for *Point-Face* type (P2P proximity path query)

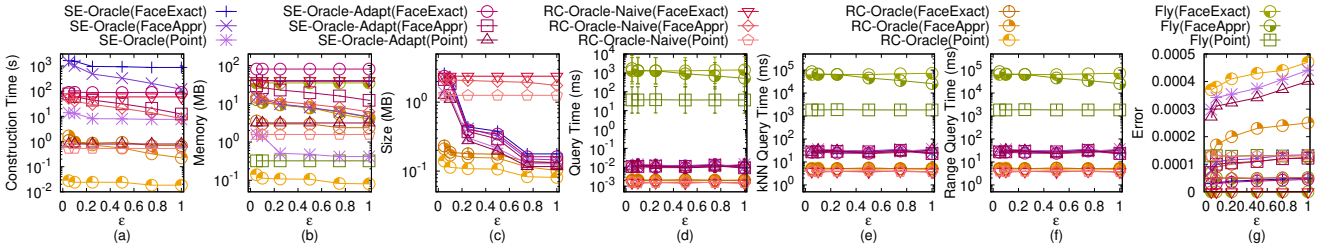

Figure 15: Effect of ϵ on *EP-small* dataset for *Point-Face* type (P2P proximity path query)

0.75, 1} on *BH*, *EP* and *RM* dataset by setting N to be 0.5M and n to be 500. Even though varying ϵ will not affect *RC-Oracle(Point)* a lot, the oracle construction time, memory usage, oracle size, shortest path query time, all POIs *kNN* path query time, and all POIs range path query time of *RC-Oracle(Point)* still perform much better than the best-known oracle *SE-Oracle(FaceExact)*, and other adapted algorithms / oracles.

C.1.2 Path on Point Cloud & Vertex of TIN. We then show the experimental results for $\boldsymbol{A(P)}$, where $\boldsymbol{A}$ = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and $\boldsymbol{P}$ = {*Vertex, Point*}, such that the calculated path passes on points of the point cloud and vertices of the implicit *TIN*.

Effect of N (scalability test). In Figure 27, Figure 30 and Figure 33, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on *BH*, *EP* and *RM* dataset (with small-version POIs) by setting n to be 50 and ϵ to be 0.1 for scalability test (with total 15 datasets). In Figure 36, Figure 39 and Figure 42, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on *BH*, *EP* and *RM* dataset (with large-version POIs) by setting n to be 500 and ϵ to be 0.1 for scalability test (with total 15 datasets). Even though we adapt the shortest path query algorithm of $\boldsymbol{A(Vertex)}$, where $\boldsymbol{A}$ = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} to make them pass on the vertex of the *TIN*, they are still not efficient enough.

Effect of n. In Figure 28, Figure 31 and Figure 34, we tested 5 values of n from {50, 100, 150, 200, 250} on *BH*, *EP* and *RM* dataset

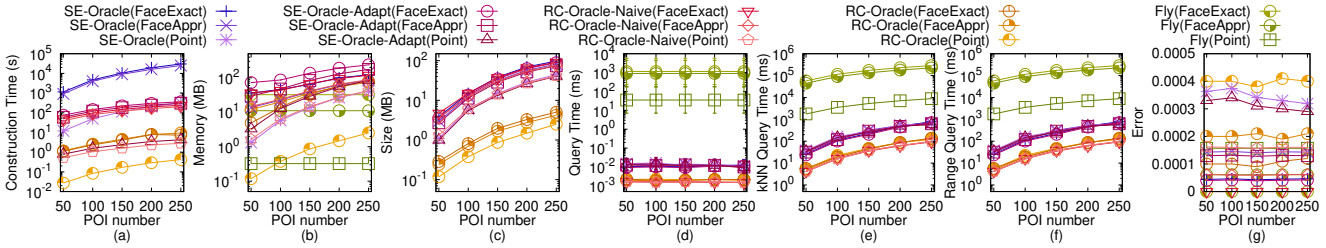

Figure 16: Effect of n on *RM-small* dataset for *Point-Face* type (P2P proximity path query)

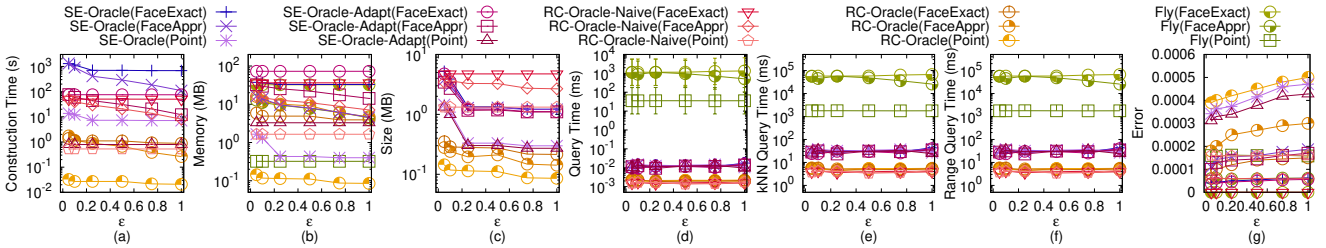

Figure 17: Effect of ϵ on *RM-small* dataset for *Point-Face* type (P2P proximity path query)

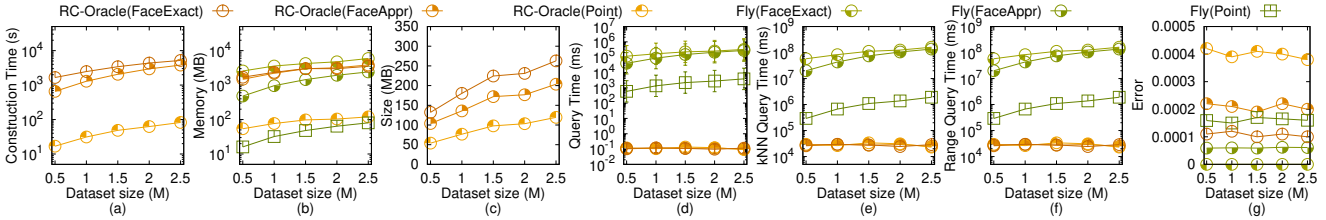

Figure 18: Effect of N on *BH* dataset for *Point-Face* type (P2P proximity path query)

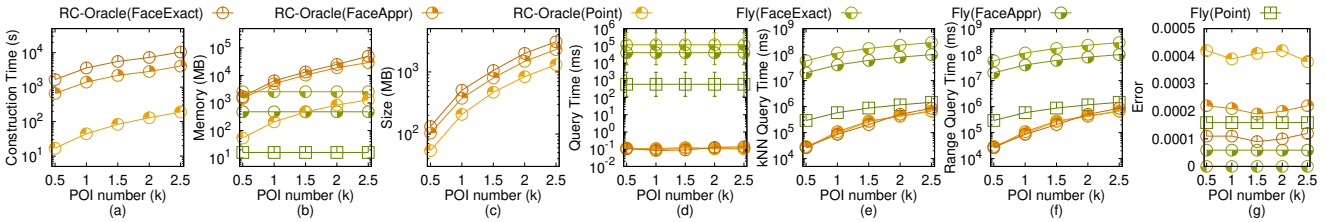

Figure 19: Effect of n on *BH* dataset for *Point-Face* type (P2P proximity path query)

(with small-version POIs) by setting N to be 0.5M and ϵ to be 0.1. In Figure 37, Figure 40 and Figure 43, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on *BH*, *EP* and *RM* dataset (with large-version POIs) by setting N to be 0.5M and ϵ to be 0.1. Even though *RC-Oracle(Vertex)* and algorithm *Fly(Vertex)* pass on the vertex of the *TIN*, they still perform worse than *RC-Oracle(Point)* and algorithm *Fly(Point)*.

Effect of ϵ. In Figure 29, Figure 32 and Figure 35, we tested 5 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *BH*, *EP* and *RM* dataset (with small-version POIs) by setting N to be 0.5M and n to be 50. In Figure 38, Figure 41 and Figure 44, we tested 5 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *BH*, *EP* and *RM* dataset (with large-version POIs) by setting N to be 0.5M and n to be 500. *RC-Oracle(Point)* could still perform much better than other adapted algorithms / oracles.

C.2 Experimental Results on the A2A proximity path query

C.2.1 Path on Point Cloud & Face of TIN. We first show the experimental results for $\boldsymbol{A(P)}$, where $\boldsymbol{A}$ = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and $\boldsymbol{P}$ = {*FaceExact, FaceAppr, Point*}, such that the calculated path passes on points of the point cloud and the faces of the implicit *TIN*.

In Figure 45, we tested the A2A proximity path query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 5k on a multi-resolution of *EP* dataset. We selected 50 points as reference points for the *kNN* path query and range path query. Even though varying ϵ will not affect *RC-Oracle(Point)* a lot, the oracle construction time, memory usage, oracle size, shortest path query time, all POIs

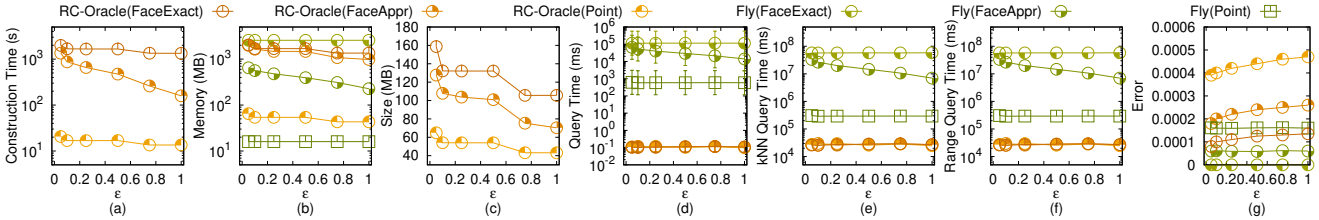

Figure 20: Effect of ϵ on *BH* dataset for *Point-Face* type (P2P proximity path query)

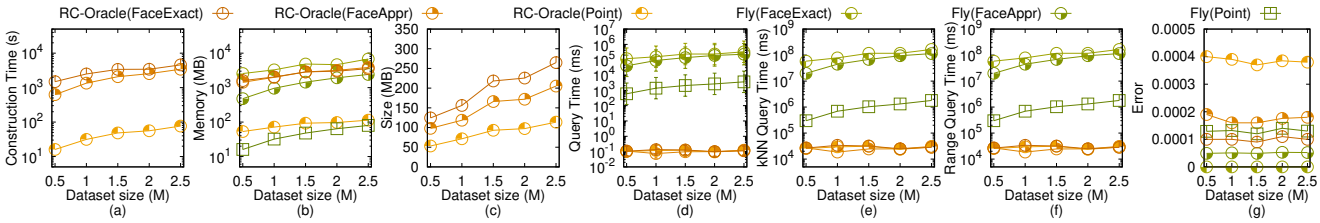

Figure 21: Effect of N on *EP* dataset for *Point-Face* type (P2P proximity path query)

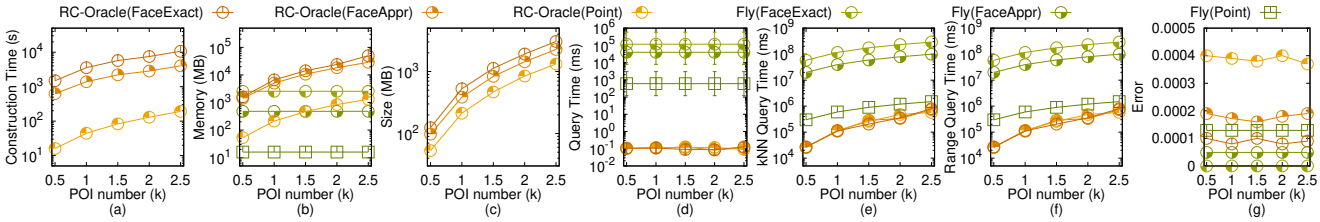

Figure 22: Effect of n on *EP* dataset for *Point-Face* type (P2P proximity path query)

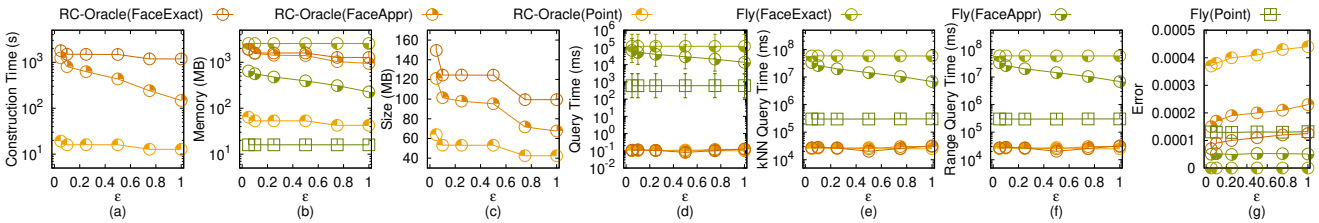

Figure 23: Effect of ϵ on *EP* dataset for *Point-Face* type (P2P proximity path query)

kNN path query time, and all POIs range path query time of *RC-Oracle(Point)* still perform much better than the best-known oracle *SE-Oracle(FaceExact)*, and other adapted algorithms / oracles.

C.2.2 Path on Point Cloud & Vertex of TIN. We then show the experimental results for ***A(P)***, where ***A*** = {*Fly, SE-Oracle, SE-Oracle-Adapt, RC-Oracle-Naive, RC-Oracle*} and ***P*** = {*Vertex, Point*}, such that the calculated path passes on points of the point cloud and vertices of the implicit *TIN*.

In Figure 46, we tested the A2A proximity path query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 10k on a multi-resolution of *EP* dataset. We selected 50 points as reference points for the *kNN* path query and range path query. *RC-Oracle(Point)* could still perform much better than other adapted algorithms / oracles.

C.3 Generating datasets with different dataset sizes

The procedure for generating the point cloud datasets with different dataset sizes is as follows. We mainly follow the procedure for generating datasets with different dataset sizes in the work [38, 54, 55]. Let C_t be our target point cloud that we want to generate with qx_t points along x-coordinate, qy_t points along y-coordinate and N_t points, where $N_t = qx_t \cdot qy_t$. Let C_o be the original point cloud that we currently have with qx_o edges along x-coordinate, qy_o edges along y-coordinate and N_o points, where $N_o = qx_o \cdot qy_o$. We then generate $qx_t \cdot qy_t$ 2D points (x, y) based on a Normal distribution $N(\mu_N, \sigma_N^2)$, where $\mu_N = (\overline{x} = \frac{\sum_{qo \in C_o} x_{qo}}{qx_o \cdot qy_o}, \overline{y} = \frac{\sum_{qo \in C_o} y_{qo}}{qx_o \cdot qy_o})$ and $\sigma_N^2 = (\frac{\sum_{qo \in C_o} (x_{qo} - \overline{x})^2}{qx_o \cdot qy_o}, \frac{\sum_{qo \in C_o} (y_{qo} - \overline{y})^2}{qx_o \cdot qy_o})$. In the end, we project each generated point (x, y) to the implicit surface of C_o and take the projected point as the newly generate C_t.

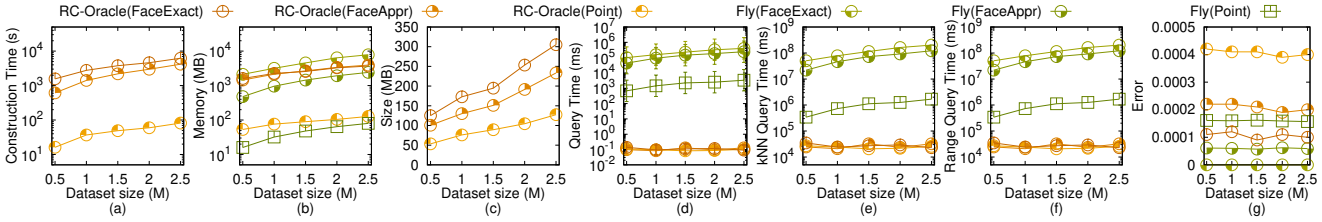

Figure 24: Effect of N on *RM* dataset for *Point-Face* type (P2P proximity path query)

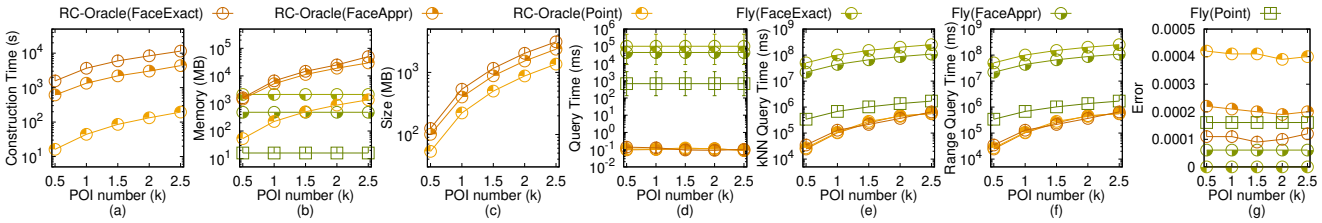

Figure 25: Effect of n on *RM* dataset for *Point-Face* type (P2P proximity path query)

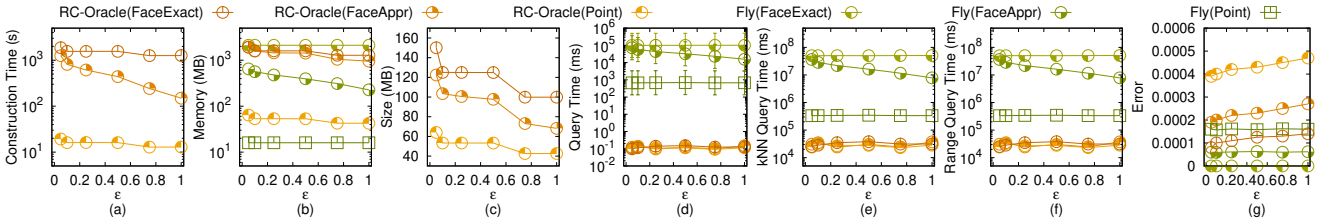

Figure 26: Effect of ϵ on *RM* dataset for *Point-Face* type (P2P proximity path query)

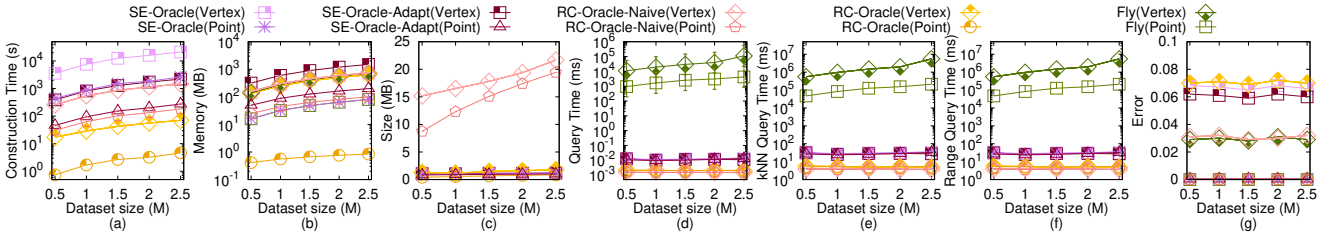

Figure 27: Effect of N on *BH* dataset for *Point-Vertex* type (less POIs and P2P proximity path query)

D PROOF

Proof of Theorem 4.2. Firstly, we prove the oracle construction time of *RC-Oracle(Point)*.

- In **POIs sorting** step, it needs $O(n \log)$ time. By using quick sort, we can sort n POIs in $O(n \log)$ time.
- In **shortest path calculation** step, it needs $O(\mu N \log N + n)$ time. In the average case, it needs to run algorithm *Fly(Point)* for $O(\frac{1}{\epsilon^2})$ times since according to standard packing property [26], we just need to use $O(\frac{1}{\epsilon^2})$ POIs as a source to use algorithm *Fly(Point)* for exact shortest path calculation (which is also shown by our experiment), and it needs $O(N \log N)$ time. For other $O(n)$ POIs that there is no need to use them as a source to run algorithm *Fly(Point)* for exact shortest path calculation, we just need to calculate the Euclidean distance from these POIs to other POIs in $O(1)$ time for shortest path approximation, and it needs $O(n)$ time. So the total running time is $O(\frac{N \log N}{\epsilon^2} + n)$. In the worst case, it needs to run algorithm *Fly(Point)* for $O(n)$ times, so the total running time is $O(nN \log N)$. Since μ is $O(\frac{1}{\epsilon^2})$ in the average case and is $O(n)$ in the worst case, we obtain the total running time is $O(\mu N \log N + n)$.

So the oracle construction time of *RC-Oracle(Point)* is $O(\mu N \log N + n \log n)$.

Secondly, we prove the oracle size of *RC-Oracle(Point)*. There are two components in *RC-Oracle(Point)*, i.e., M_{path} and M_{POI}. For M_{POI}, its size is at most $O(n)$. Thus, we mainly focus on the size of M_{path}. In the average case, we just need to use $O(\frac{1}{\epsilon^2})$ POIs as a source to use algorithm *Fly(Point)* for the exact shortest path calculation. Given that there are total n POIs, so there are $O(n)$ shortest paths for these $O(\frac{1}{\epsilon^2})$ POIs. For other $O(n)$ POIs that there is no need to use them as a source to run algorithm *Fly(Point)* to all POIs for

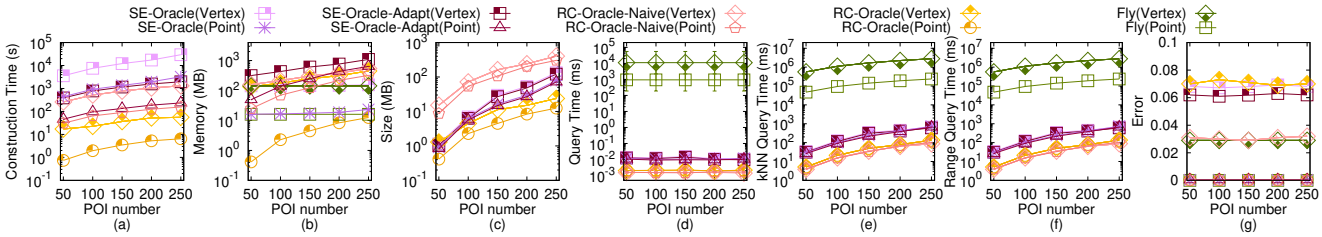

Figure 28: Effect of n on *BH* dataset for *Point-Vertex* type (less POIs and P2P proximity path query)

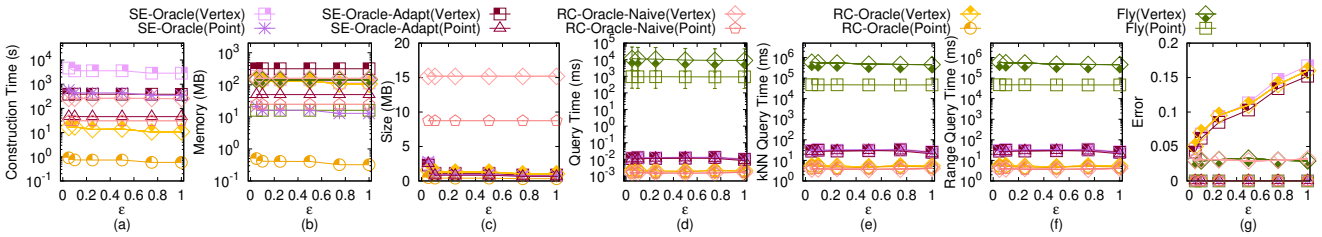

Figure 29: Effect of ϵ on *BH* dataset for *Point-Vertex* type (less POIs and P2P proximity path query)

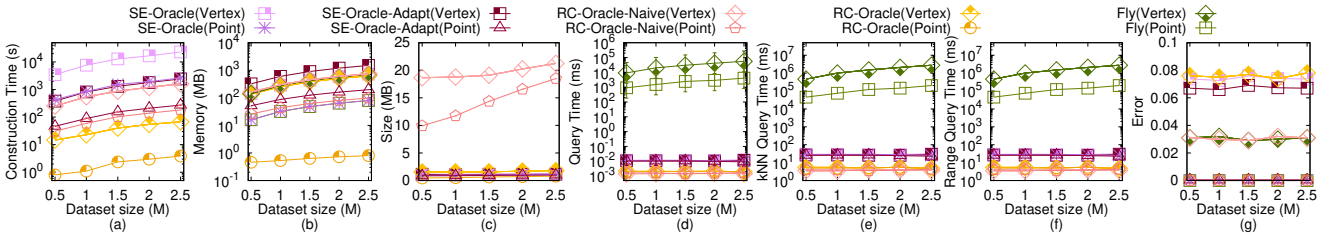

Figure 30: Effect of N on *EP* dataset for *Point-Vertex* type (less POIs and P2P proximity path query)

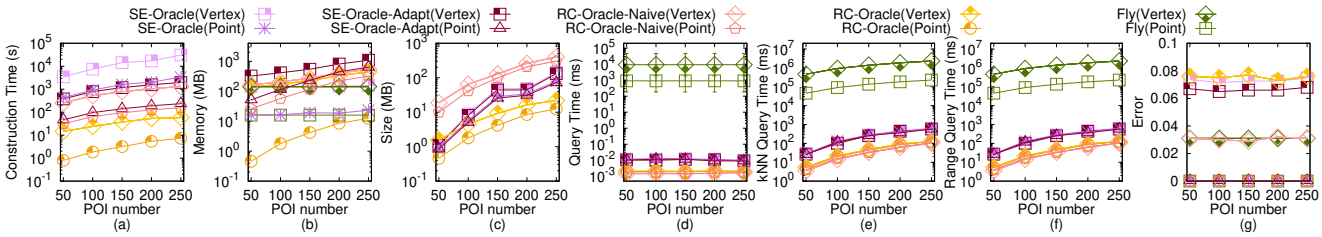

Figure 31: Effect of n on *EP* dataset for *Point-Vertex* type (less POIs and P2P proximity path query)

exact shortest path calculation, we will not store the exact shortest paths between these POIs to other POIs in M_{path}, since they are approximated by the other exact shortest paths stored in M_{path}. For these POIs, we just need to calculate the exact shortest paths between themselves with size $O(1)$, so there are $O(n)$ shortest paths for these POIs. In total, there are $O(n)$ exact shortest paths stored in M_{path}. In the worst case, M_{path} needs to store $O(n^2)$ pairwise P2P exact shortest paths. Since μ is $O(\frac{1}{\epsilon^2})$ in average case and is $O(n)$ in worst case, we obtain the oracle size is $O(\mu n)$.

Thirdly, we prove the shortest path query time of *RC-Oracle(Point)*. If $\Pi^*(s,t|C)$ exists in M_{path}, the shortest path query time is $O(1)$. If $\Pi^*(s,t|C)$ does not exist in M_{path}, we need to retrieve s' from M_{POI} using s in constant time, and retrieve $\Pi^*(s,s'|C)$ and $\Pi^*(s',t|C)$ from M_{path} using $\langle s,s'\rangle$ and $\langle s',t\rangle$ in constant time, so the shortest path query time is still $O(1)$. Thus, the shortest path query time of *RC-Oracle(Point)* is $O(1)$.

Fourthly, we prove the error bound of *RC-Oracle(Point)*. Given a pair of POIs s and t, if $\Pi^*(s,t|C)$ exists in M_{path}, then there is no error. Thus, we only consider the case that $\Pi^*(s,t|C)$ does not exist in M_{path}. Without loss of generality, in Figure 2 (c), suppose that $\Pi^*(b,d|C)$ does not exists in M_{path}, we let a be the POI close to b, such that approximate shortest path $\Pi(b,d|C)$ is calculated by appending $\Pi^*(b,a|C)$ and $\Pi^*(a,d|C)$. This means that $d_E(b,d) > \frac{2}{\epsilon}\cdot\Pi^*(a,b|C)$, since we will only use $\Pi^*(b,a|C)$ and $\Pi^*(a,d|C)$ to approximate $\Pi(b,d|C)$ when this condition satisfies. According to [54, 55], the distance of the path on a *TIN* is a metric, and it satisfies the triangle inequality. Since the path on the point cloud will only pass on the points, which is a sub-type of the path on a *TIN*, so the

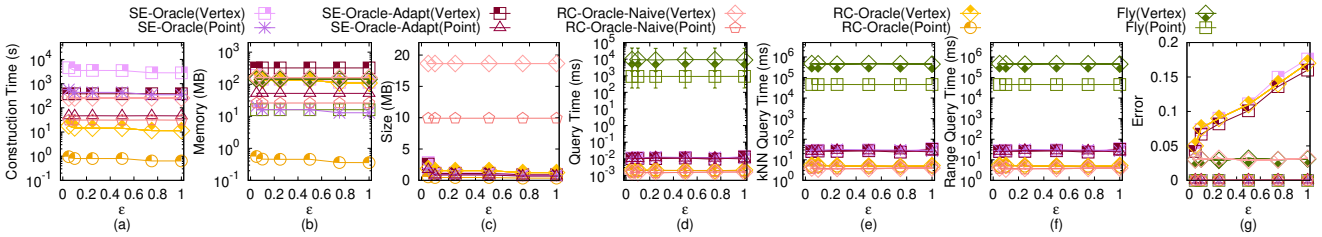

Figure 32: Effect of ϵ on *EP* dataset for *Point-Vertex* type (less POIs and P2P proximity path query)

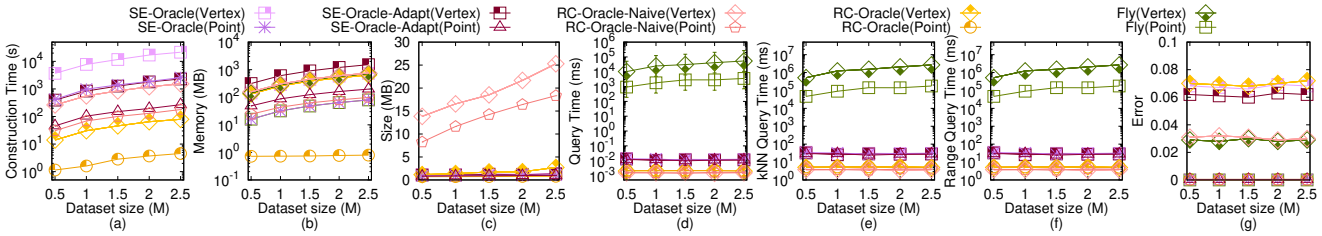

Figure 33: Effect of N on *RM* dataset for *Point-Vertex* type (less POIs and P2P proximity path query)

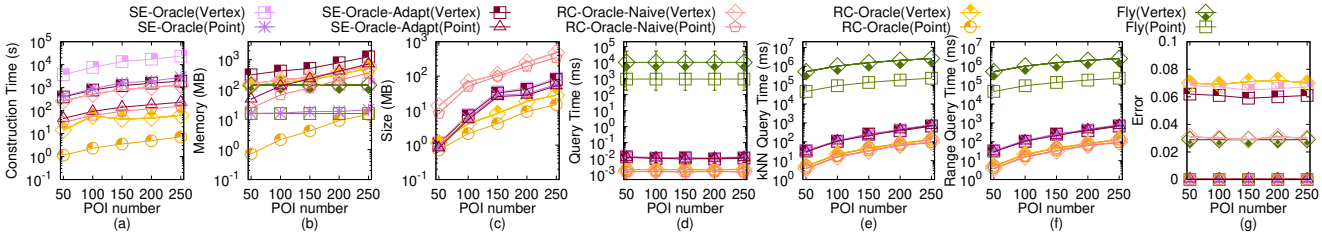

Figure 34: Effect of n on *RM* dataset for *Point-Vertex* type (less POIs and P2P proximity path query)

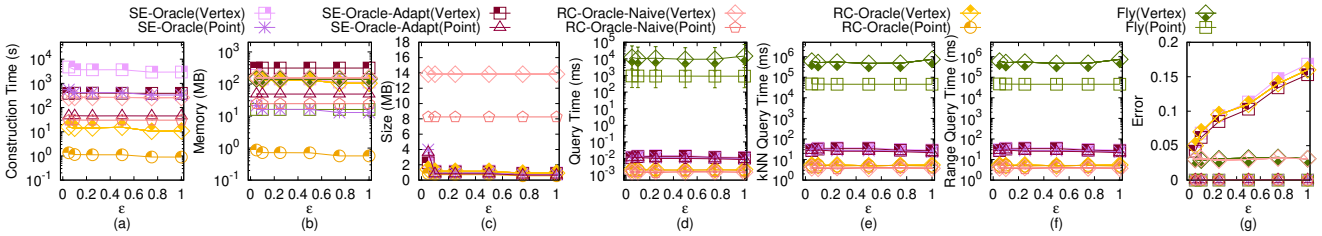

Figure 35: Effect of ϵ on *RM* dataset for *Point-Vertex* type (less POIs and P2P proximity path query)

distance of the path on point also satisfies the triangle inequality. So we have $|\Pi^*(b,a|C)| + |\Pi^*(a,d|C)| < |\Pi^*(b,a|C)| + |\Pi^*(a,b|C)| + |\Pi^*(b,d|C)| = |\Pi^*(b,d|C)| + 2 \cdot |\Pi^*(a,b|C)| < |\Pi^*(b,d|C)| + \epsilon \cdot d_E(b,d) \leq |\Pi^*(b,d|C)| + \epsilon \cdot |\Pi^*(b,d|C)| = (1+\epsilon)|\Pi^*(b,d|C)|$. The first inequality is due to triangle inequality. The second equation is because $|\Pi^*(a,b|C)| = |\Pi^*(b,a|C)|$. The third inequality is because we have $d_E(b,d) > \frac{2}{\epsilon} \cdot \Pi^*(a,b|C)$. The fourth inequality is because Euclidean distance between two points is no larger than the distance of the shortest path on the point cloud between the same two points. The final equation is due to the distributive law of multiplication. Since we can substitute b to s, d to t, and a to a POI near b, so *RC-Oracle(Point)* always has $|\Pi(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for each pair of POIs s and t in P. □

Proof of Lemma 4.3. Firstly, we prove the query time of both the *kNN* and range path query algorithm. Given a query POI, when we need to perform the *kNN* path query or the range path query, we need to check the distance between this query POI to all other POIs using the shortest path query phase of *RC-Oracle(Point)* in $O(1)$ time. Since there are total n POIs, the query time is $O(n)$.

Secondly, we prove the approximate ratio of both the *kNN* and range path query algorithm. Recall that we let v (resp. v') be the furthest POI to q in X (resp. X'), i.e., $|\Pi^*(q,v|C)| \leq \max_{\forall v \in X} |\Pi^*(q,v|C)|$ (resp. $|\Pi^*(q,v'|C)| \leq \max_{\forall v' \in X'} |\Pi^*(q,v'|C)|$). We further let w (resp. w') be the furthest POI to q in X (resp. X') based on the distance of the approximated shortest path on C returned by *RC-Oracle(Point)*, i.e., $|\Pi(q,w|C)| \leq \max_{\forall w \in X} |\Pi(q,w|C)|$ (resp. $|\Pi(q,w'|C)| \leq$

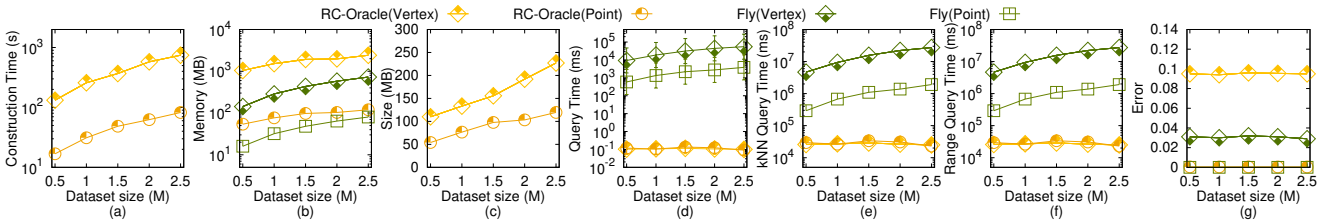

Figure 36: Effect of N on *BH* dataset for *Point-Vertex* type (more POIs and P2P proximity path query)

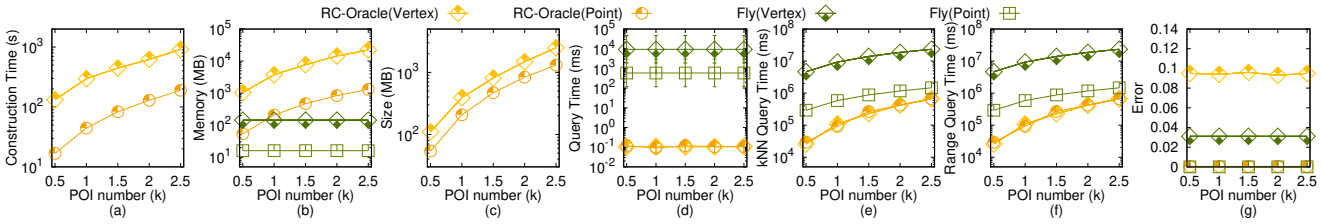

Figure 37: Effect of n on *BH* dataset for *Point-Vertex* type (more POIs and P2P proximity path query)

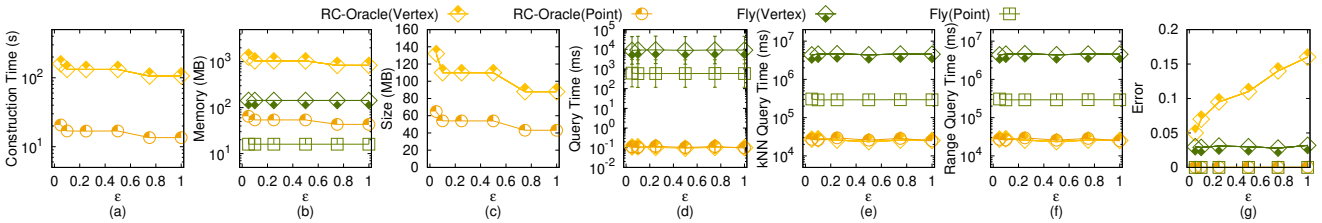

Figure 38: Effect of ϵ on *BH* dataset for *Point-Vertex* type (more POIs and P2P proximity path query)

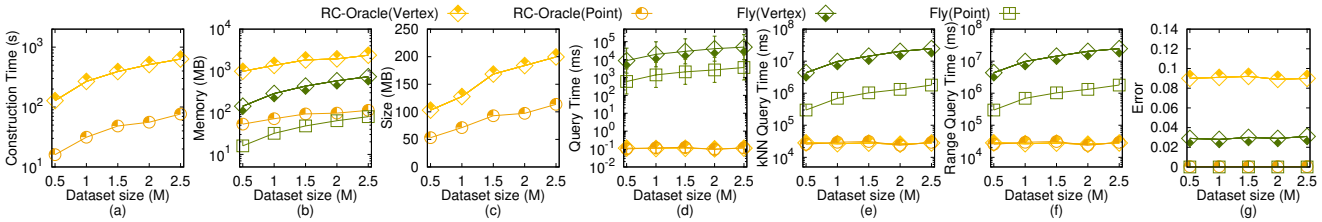

Figure 39: Effect of N on *EP* dataset for *Point-Vertex* type (more POIs and P2P proximity path query)

$\max_{\forall w' \in X'} |\Pi(q, w'|C)|$). Recall the approximate ratio of the *kNN* path query and range path query is $\alpha = \frac{|\Pi^*(q,v'|C)|}{|\Pi^*(q,v|C)|}$. Since the distance of the approximated shortest path on C returned by *RC-Oracle(Point)* is always longer than the distance of the exact shortest path on C, we have $|\Pi(q, v'|C)| \geq |\Pi^*(q, v'|C)|$. Thus, we have $\alpha \leq \frac{|\Pi(q,v'|C)|}{|\Pi^*(q,v|C)|}$. By the definition of v and w, we have $|\Pi^*(q, v|C)| \geq |\Pi^*(q, w|C)|$. Thus, we have $\alpha \leq \frac{|\Pi(q,v'|C)|}{|\Pi^*(q,w|C)|}$. By the definition of v' and w', we have $|\Pi(q, v'|C)| \leq |\Pi(q, w'|C)|$. Thus, we have $\alpha \leq \frac{|\Pi(q,w'|C)|}{|\Pi^*(q,w|C)|}$. Since the error ratio of the distance of the approximated shortest path on C returned by *RC-Oracle(Point)* is $1 + \epsilon$, we have $|\Pi(q, w|C)| \leq (1 + \epsilon)|\Pi^*(q, w|C)|$. Then, we have $\alpha \leq \frac{|\Pi(q,w'|C)|(1+\epsilon)}{|\Pi(q,w|C)|}$. By our *kNN* and range path query algorithm, we have $|\Pi(q, w'|C)| \leq |\Pi(q, w|C)|$. Thus, we have $\alpha \leq 1 + \epsilon$. □

Proof of Lemma 4.4 . Firstly, we prove the query time of both the *kNN* and range path query algorithm using any given shortest path query algorithm. Given a query POI, when we need to perform the *kNN* path query or the range path query, we need to check the distance between this query POI to all other POIs using the shortest path query algorithm in t time. Since there are total n POIs, the query time is $O(nt)$.

Secondly, we prove the approximate ratio of both the *kNN* and range path query algorithm using any given shortest path query algorithm. Given a pair of POI s and t on any data format (e.g., a point cloud, a *TIN*, a graph, a road network), we define $\Pi^*(s, t)$ to be the exact shortest path between s and t on the data format, and define $\Pi(s, t)$ to be the approximated shortest path between s and t calculated by the given shortest path query algorithm on the data format.

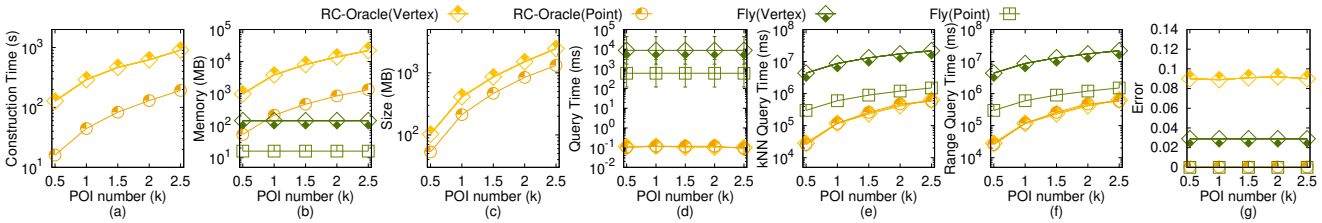

Figure 40: Effect of n on *EP* dataset for *Point-Vertex* type (more POIs and P2P proximity path query)

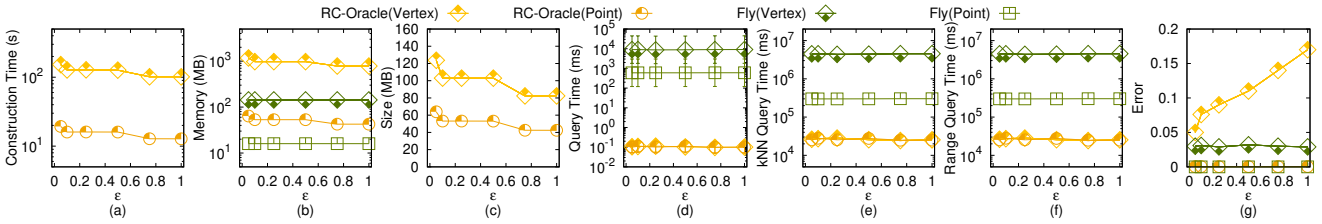

Figure 41: Effect of ϵ on *EP* dataset for *Point-Vertex* type (more POIs and P2P proximity path query)

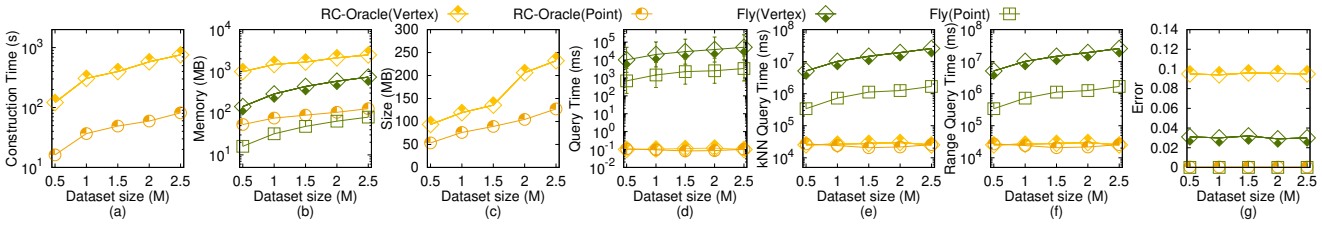

Figure 42: Effect of N on *RM* dataset for *Point-Vertex* type (more POIs and P2P proximity path query)

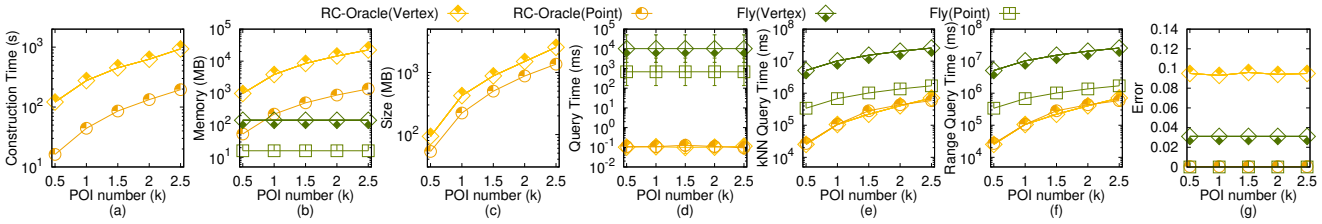

Figure 43: Effect of n on *RM* dataset for *Point-Vertex* type (more POIs and P2P proximity path query)

Given a query point $q \in P$, we let X be a set of POIs containing the *exact* (1) k nearest POIs of q or (2) POIs whose distance to q are at most r, calculated using the distance of the exact shortest path on any data format. Furthermore, given a query point $q \in P$, we let X' be a set of POIs containing (1) k nearest POIs of q or (2) POIs whose distance to q are at most r, calculated using the distance of the approximated shortest path on any data format calculated using any given shortest path query algorithm.

We let v (resp. v') be the furthest POI to q in X (resp. X') based on the distance of the exact shortest path, i.e., $|\Pi^*(q,v)| \leq \max_{\forall v \in X} |\Pi^*(q,v)|$ (resp. $|\Pi^*(q,v')| \leq \max_{\forall v' \in X'} |\Pi^*(q,v')|$). We further let w (resp. w') be the furthest POI to q in X (resp. X') based on the distance of the approximated shortest path returned using any given shortest path query algorithm, i.e., $|\Pi(q,w)| \leq \max_{\forall w \in X} |\Pi(q,w)|$ (resp. $|\Pi(q,w')| \leq \max_{\forall w' \in X'} |\Pi(q,w')|$). We define the approximate ratio of the *kNN* path query and range path query to be $\alpha' = \frac{|\Pi^*(q,v')|}{|\Pi^*(q,v)|}$, which is a real number no smaller than 1. Since the distance of the approximated shortest path returned using any given shortest path query algorithm is always longer than the distance of the exact shortest path, we have $|\Pi(q,v')| \geq |\Pi^*(q,v')|$. Thus, we have $\alpha \leq \frac{|\Pi(q,v')|}{|\Pi^*(q,v)|}$. By the definition of v and w, we have $|\Pi^*(q,v)| \geq |\Pi^*(q,w)|$. Thus, we have $\alpha \leq \frac{|\Pi(q,v')|}{|\Pi^*(q,w)|}$. By the definition of v' and w', we have $|\Pi(q,v')| \leq |\Pi(q,w')|$. Thus, we have $\alpha \leq \frac{|\Pi(q,w')|}{|\Pi^*(q,w)|}$. Since the error ratio of the distance of the approximated shortest path returned using any given shortest path query algorithm is $1+\epsilon'$, we have $|\Pi(q,w)| \leq (1+\epsilon')|\Pi^*(q,w)|$. Then, we have $\alpha \leq \frac{|\Pi(q,w')|(1+\epsilon')}{|\Pi(q,w)|}$. By the *kNN* and range path query algorithm, we have $|\Pi(q,w')| \leq |\Pi(q,w)|$. Thus, we have $\alpha \leq 1+\epsilon'$. □

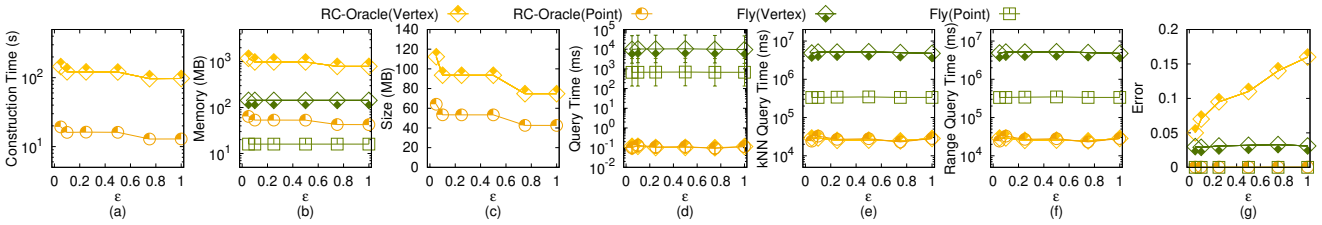

Figure 44: Effect of ϵ on *RM* dataset for *Point-Vertex* type (more POIs and P2P proximity path query)

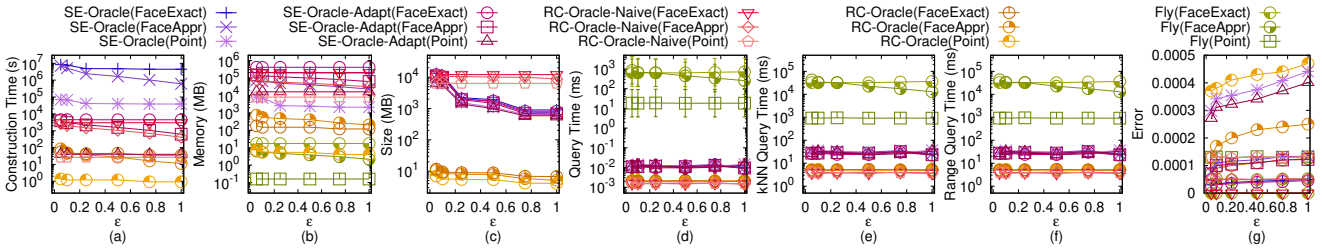

Figure 45: A2A proximity path query for *Point-Face* type

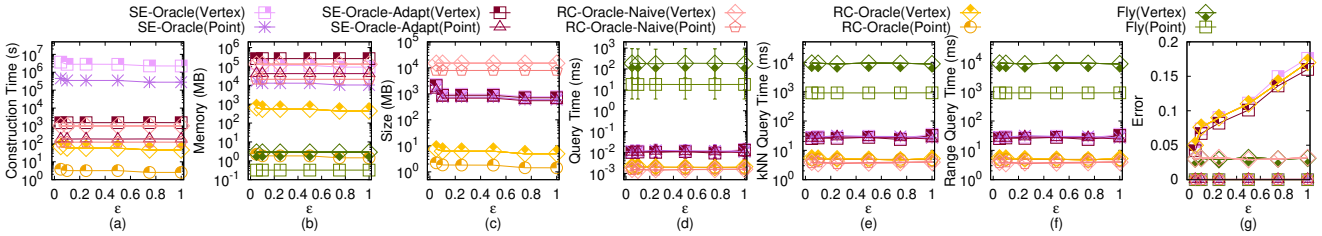

Figure 46: A2A proximity path query for *Point-Vertex* type

Proof of Lemma 5.1. According to left hand side equation in Lemma 2 of [29], we have $\lambda \cdot |\Pi_V(s,t|T)| \leq |\Pi^*(s,t|T)|$, where $\lambda = \min\{\frac{\sin\theta}{2}, \sin\theta\cos\theta\}$. Even though both $\Pi_V(s,t|T)$ and $\Pi_N(s,t|T)$ passes on the vertices of T, according to [49] in Section 3.1, $\Pi_N(s,t|T)$ may pass different face of $\Pi_V(s,t|T)$. Since $\Pi_V(s,t|T)$ passes on the vertices on T where these vertices belong to the faces that $\Pi^*(s,t|T)$ passes, it may not be the shortest path that passes on the vertices considering all the vertices on T. But, $\Pi_N(s,t|T)$ is the shortest network path that passes on the vertices considering all the vertices on T, so $|\Pi_N(s,t|T)| \leq |\Pi_V(s,t|T)|$. In Figure 3 (a), given a green point q on C, it can connect with one of its eight blue neighbor points. In Figure 3 (b), given a black vertex q on T, it can only connect with one of its six blue neighbor vertices. Since $\Pi^*(s,t|C)$ passes on points of C, and $\Pi_N(s,t|T)$ passes on vertices of T, so for a point / vertex u, if its next searching point / vertex v is its diagonal neighbor point /vertex, $\Pi^*(s,t|C)$ can directly connect u and v ($\Pi^*(s,t|C)$ can directly connect q and q' in Figure 3 (a)), but $\Pi_N(s,t|T)$ needs one more vertex to connect u and v ($\Pi_N(s,t|T)$ needs one more vertex to connect q and q' in Figure 3 (b)), so $|\Pi^*(s,t|C)| \leq |\Pi_N(s,t|T)|$. Thus, by combining $|\Pi^*(s,t|C)| \leq |\Pi_N(s,t|T)|$, $|\Pi_N(s,t|T)| \leq |\Pi_V(s,t|T)|$, and $|\Pi_V(s,t|C)| \leq \frac{1}{\lambda} \cdot |\Pi^*(s,t|T)|$, we have $|\Pi^*(s,t|C)| \leq k \cdot |\Pi^*(s,t|T)|$, where $k = \max\{\frac{2}{\sin\theta}, \frac{1}{\sin\theta\cos\theta}\}$. □

Proof of Lemma 5.2. In Figure 3 (a), given a green point q on C, it can connect with one of its eight blue neighbor points. In Figure 3 (b), given a black vertex q on T, it can only connect with one of its six blue neighbor vertices. Since $\Pi^*(s,t|C)$ passes on points of C, and $\Pi_N(s,t|T)$ passes on vertices of T, so for a point / vertex u, if its next searching point / vertex v is its diagonal neighbor point /vertex, $\Pi^*(s,t|C)$ can directly connect u and v ($\Pi^*(s,t|C)$ can directly connect q and q' in Figure 3 (a)), but $\Pi_N(s,t|T)$ needs one more vertex to connect u and v ($\Pi_N(s,t|T)$ needs one more vertex to connect q and q' in Figure 3 (b)), so $|\Pi^*(s,t|C)| \leq |\Pi_N(s,t|T)|$. □

Theorem D.1. *The shortest path query time and memory usage of algorithm Fly(FaceExact) are $O(N + N^2)$ and $O(N)$, respectively. Algorithm Fly(FaceExact) returns the exact shortest path that passes on the implicit TIN constructed by the point cloud.*

Proof. Firstly, we prove the shortest path query time of algorithm *Fly(FaceExact)*. The proof of the shortest path query time of algorithm *Fly(FaceExact)* is in [14]. But since algorithm *Fly(FaceExact)* first needs to construct the implicit *TIN* using the point cloud, it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *Fly(FaceExact)* is $O(N + N^2)$.

Secondly, we prove the memory usage of algorithm *Fly(FaceExact)*. The proof of the memory usage of algorithm *Fly(FaceExact)* is in [14]. Thus, the memory usage of algorithm *Fly(FaceExact)* is $O(N)$.

Thirdly, we prove the error bound of algorithm *Fly(FaceExact)*. The proof that algorithm *Fly(FaceExact)* returns the exact shortest path on the *TIN* is in [14]. Since the *TIN* is constructed by the point cloud, so algorithm *Fly(FaceExact)* returns the exact shortest path that passes on the implicit *TIN* constructed by the point cloud. □

Theorem D.2. *The shortest path query time and memory usage of algorithm Fly(FaceAppr) are* $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ *and* $O(N)$*, respectively. Algorithm Fly(FaceAppr) always has* $|\Pi_{Fly(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for each pair of POIs s and t in P, where* $\Pi_{Fly(FaceAppr)}(s,t|T)$ *is the shortest path of algorithm Fly(FaceAppr) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the shortest path query time of algorithm *Fly(FaceAppr)*. The proof of the shortest path query time of algorithm *Fly(FaceAppr)* is in [28]. Note that in Section 4.2 of [28], the shortest path query time of algorithm *Fly(FaceAppr)* is $O((N+N')(\log(N+N') + (\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}})^2))$, where $N' = O(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}}N)$ and K is a parameter which is a positive number at least 1. By Theorem 1 of [28], we obtain that its error bound ϵ is equal to $\frac{1}{K-1}$. Thus, we can derive that the shortest path query time of algorithm *Fly(FaceAppr)* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + \frac{l_{max}^2}{(\epsilon l_{min}\sqrt{1-\cos\theta})^2})$. Since for N, the first term is larger than the second term, so we obtain the shortest path query time of algorithm *Fly(FaceAppr)* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$. But since algorithm *Fly(FaceExact)* first needs to construct the implicit *TIN* using the point cloud, so it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *Fly(FaceAppr)* is $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$.

Secondly, we prove the memory usage of algorithm *Fly(FaceAppr)*. Since algorithm *Fly(FaceExact)* is a Dijkstra algorithm and there are total N vertices on the implicit *TIN*, the memory usage is $O(N)$. Thus, the memory usage of algorithm *Fly(FaceAppr)* is $O(N)$.

Thirdly, we prove the error bound of algorithm *Fly(FaceAppr)*. The proof of the error bound of algorithm *Fly(FaceAppr)* is in [28]. Since the *TIN* is constructed by the point cloud, so algorithm *Fly(FaceAppr)* always has $|\Pi_{Fly(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem D.3. *The shortest path query time and memory usage of algorithm Fly(Vertex) are* $O(N + N\log N)$ *and* $O(N)$*, respectively. Algorithm Fly(Vertex) always has* $|\Pi_{Fly(Vertex)}(s,t|T)| \geq |\Pi^*(s,t|C)|$ *for each pair of POIs s and t in P, where* $\Pi_{Fly(Vertex)}(s,t|T)$ *is the shortest path of algorithm Fly(Vertex) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the shortest path query time of algorithm *Fly(Vertex)*. Since algorithm *Fly(Vertex)* only passes on the vertex of the implicit *TIN T* constructed by the point cloud, it is a Dijkstra algorithm and there are total N points, so the shortest path query time is $O(N\log N)$. But since algorithm *Fly(Vertex)* first needs to construct the implicit *TIN* using the point cloud, so it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *Fly(Vertex)* is $O(N + N\log N)$.

Secondly, we prove the memory usage of algorithm *Fly(Vertex)*. Since algorithm *Fly(Vertex)* is a Dijkstra algorithm and there are total N vertices on the implicit *TIN*, the memory usage is $O(N)$. Thus, the memory usage of algorithm *Fly(FaceAppr)* is $O(N)$.

Thirdly, we prove the error bound of algorithm *Fly(Vertex)*. Recall that $\Pi_N(s,t|T)$ is the shortest network path between s and t passes on the implicit *TIN T* constructed by the point cloud, which only passes on the vertices of T, so actually $\Pi_N(s,t|T)$ is the same as $\Pi_{Fly(Vertex)}(s,t|T)$. In Lemma 5.2, we have $|\Pi^*(s,t|C)| \leq |\Pi_N(s,t|T)|$, so we obtain that algorithm *Fly(Vertex)* always has $|\Pi_{Fly(Vertex)}(s,t|T)| \geq |\Pi^*(s,t|C)|$ for each pair of POIs s and t in P. □

Theorem D.4. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle(FaceExact) are* $O(N+\frac{nhN^2}{\epsilon^{2\beta}})$, $O(\frac{nh}{\epsilon^{2\beta}})$, *and* $O(h^2)$*, respectively. SE-Oracle(FaceExact) always has* $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle(FaceExact)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for each pair of POIs s and t in P, where* $\Pi_{SE\text{-}Oracle(FaceExact)}(s,t|T)$ *is the shortest path of SE-Oracle(FaceExact) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *SE-Oracle(FaceExact)*. The oracle construction time of the original oracle in [54, 55] is $O(\frac{nhm}{\epsilon^{2\beta}})$, where m is the on-the-fly shortest path query time. In *SE-Oracle(FaceExact)*, we use algorithm *Fly(FaceExact)* for the point cloud shortest path query, which has shortest path query time $O(N+N^2)$ according to Theorem D.1. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with N^2, and *SE-Oracle(FaceExact)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle(FaceExact)* is $O(N+\frac{nhN^2}{\epsilon^{2\beta}})$.

Secondly, we prove the oracle size of *SE-Oracle(FaceExact)*. The proof of the oracle size of *SE-Oracle(FaceExact)* is in [54, 55]. Thus, the oracle size of *SE-Oracle(FaceExact)* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the shortest path query time of *SE-Oracle(FaceExact)*. The proof of the shortest path query time of *SE-Oracle(FaceExact)* is in [54, 55]. Thus, the shortest path query time of *SE-Oracle(FaceExact)* is $O(h^2)$.

Fourthly, we prove the error bound of *SE-Oracle(FaceExact)*. Since the on-the-fly shortest path query algorithm in *SE-Oracle(FaceExact)* is algorithm *Fly(FaceExact)*, which returns the exact shortest path that passes on the implicit *TIN* constructed by the point cloud according to Theorem D.1, so the error of *SE-Oracle(FaceExact)* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle(FaceExact)* is in [54, 55]. Since the *TIN* is constructed by the point cloud, we obtain that *SE-Oracle(FaceExact)* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle(FaceExact)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem D.5. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle(FaceAppr) are* $O(N + \frac{nhl_{max}N}{\epsilon^{(2\beta+1)}l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$, $O(\frac{nh}{\epsilon^{2\beta}})$, *and* $O(h^2)$*, respectively. SE-Oracle(FaceAppr) always has* $|\Pi_{SE\text{-}Oracle(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for each pair of POIs s and t in P, where* $\Pi_{SE\text{-}Oracle(FaceAppr)}(s,t|T)$ *is the shortest path of SE-Oracle(FaceAppr)*

between s and t that passes on the implicit TIN T constructed by the point cloud.

Proof. Firstly, we prove the oracle construction time of *SE-Oracle(FaceAppr)*. The oracle construction time of the original oracle in [54, 55] is $O(\frac{nhm}{\epsilon^{2\beta}})$, where m is the on-the-fly shortest path query time. In *SE-Oracle(FaceAppr)*, we use algorithm *Fly(FaceAppr)* for the point cloud shortest path query, which has shortest path query time $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ according to Theorem D.2. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$, and *SE-Oracle(FaceAppr)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle(FaceAppr)* is $O(N + \frac{nhl_{max}N}{\epsilon^{(2\beta+1)}l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$.

Secondly, we prove the oracle size of *SE-Oracle(FaceAppr)*. The proof of the oracle size of *SE-Oracle(FaceAppr)* is in [54, 55]. Thus, the oracle size of *SE-Oracle(FaceAppr)* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the shortest path query time of *SE-Oracle(FaceAppr)*. The proof of the shortest path query time of *SE-Oracle(FaceAppr)* is in [54, 55]. Thus, the shortest path query time of *SE-Oracle(FaceAppr)* is $O(h^2)$.

Fourthly, we prove the error bound of *SE-Oracle(FaceAppr)*. Since the on-the-fly shortest path query algorithm in *SE-Oracle(FaceAppr)* is algorithm *Fly(FaceAppr)*, which always has $|\Pi_{Fly(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P according to Theorem D.2, so the error of *SE-Oracle(FaceAppr)* is due to the algorithm *Fly(FaceAppr)* and oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle(FaceAppr)* is in [54, 55]. Since we assign the error in both algorithm *Fly(FaceAppr)* and the oracle itself to be $\sqrt{(1+\epsilon)}-1$, and the *TIN* is constructed by the point cloud, so we obtain that *SE-Oracle(FaceAppr)* always has $|\Pi_{SE\text{-}Oracle(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P (since algorithm *Fly(FaceAppr)* is an approximated algorithm, we can only obtain the upper bound of error ratio for *SE-Oracle(FaceAppr)*). □

Theorem D.6. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle(Point) are $O(\frac{nhN\log N}{\epsilon^{2\beta}})$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$, respectively. SE-Oracle(Point) always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle(Point)}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P, where $\Pi_{SE\text{-}Oracle(Vertex)}(s,t|C)$ is the shortest path of SE-Oracle(Vertex) between s and t that passes on points of the point cloud C.*

Proof. Firstly, we prove the oracle construction time of *SE-Oracle(Point)*. The oracle construction time of the original oracle in [54, 55] is $O(\frac{nhm}{\epsilon^{2\beta}})$, where m is the on-the-fly shortest path query time. In *SE-Oracle(Point)*, we use algorithm *Fly(Point)* for the point cloud shortest path query, which has shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute m with $N\log N$. Thus, the oracle construction time of *SE-Oracle(Point)* is $O(\frac{nhN\log N}{\epsilon^{2\beta}})$.

Secondly, we prove the oracle size of *SE-Oracle(Point)*. The proof of the oracle size of *SE-Oracle(Point)* is in [54, 55]. Thus, the oracle size of *SE-Oracle(Point)* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the shortest path query time of *SE-Oracle(Point)*. The proof of the shortest path query time of *SE-Oracle(Point)* is in [54, 55]. Thus, the shortest path query time of *SE-Oracle(Point)* is $O(h^2)$.

Fourthly, we prove the error bound of *SE-Oracle(Point)*. Since the on-the-fly shortest path query algorithm in *SE-Oracle(Point)* is algorithm *Fly(Point)*, which returns the exact shortest path that passes points on the point cloud according to Theorem 4.1, the error of *SE-Oracle(Point)* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle(Point)* is in [54, 55]. So we obtain that *SE-Oracle(Point)* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle(Point)}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for each pair of POIs s and t in P □

Theorem D.7. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle(Vertex) are $O(N + \frac{nhN\log N}{\epsilon^{2\beta}})$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$, respectively. SE-Oracle(Vertex) always has $|\Pi_{SE\text{-}Oracle(Vertex)}(s,t|T)| \geq |\Pi_{SE\text{-}Oracle(Point)}(s,t|T)|$ for each pair of POIs s and t in P, where $\Pi_{SE\text{-}Oracle(Vertex)}(s,t|T)$ is the shortest path of SE-Oracle(Vertex) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *SE-Oracle(Vertex)*. The oracle construction time of the original oracle in [54, 55] is $O(\frac{nhm}{\epsilon^{2\beta}})$, where m is the on-the-fly shortest path query time. In *SE-Oracle(Vertex)*, we use algorithm *Fly(Vertex)* for the point cloud shortest path query, which has shortest path query time $O(N+N\log N)$ according to Theorem D.3. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with $N\log N$, and *SE-Oracle(Vertex)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle(Vertex)* is $O(N + \frac{nhN\log N}{\epsilon^{2\beta}})$.

Secondly, we prove the oracle size of *SE-Oracle(Vertex)*. The proof of the oracle size of *SE-Oracle(Vertex)* is in [54, 55]. Thus, the oracle size of *SE-Oracle(Vertex)* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the shortest path query time of *SE-Oracle(Vertex)*. The proof of the shortest path query time of *SE-Oracle(Vertex)* is in [54, 55]. Thus, the shortest path query time of *SE-Oracle(Vertex)* is $O(h^2)$.

Fourthly, we prove the error bound of *SE-Oracle(Vertex)*. Since the on-the-fly shortest path query algorithm in *SE-Oracle(Vertex)* is algorithm *Fly(Vertex)*, which always has $|\Pi_{Fly(Vertex)}(s,t|T)| \geq |\Pi^*(s,t|C)|$ for each pair of POIs s and t in P according to Theorem D.3. Since the error bound of the oracle itself regarding *SE-Oracle(FaceExact)* is the same as *SE-Oracle(Point)*, and since the *TIN* is constructed by the point cloud, we obtain that *SE-Oracle(Vertex)* always has $|\Pi_{SE\text{-}Oracle(Vertex)}(s,t|T)| \geq |\Pi_{SE\text{-}Oracle(Point)}(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem D.8. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle-Adapt(FaceExact) are $O(N + nN^2 + nh\log n + \frac{nh}{\epsilon^{2\beta}})$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$, respectively. SE-Oracle-Adapt(FaceExact) always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq$*

$|\Pi_{SE\text{-}Oracle\text{-}Adapt(FaceExact)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for each pair of POIs* s *and* t *in* P*, where* $\Pi_{SE\text{-}Oracle\text{-}Adapt(FaceExact)}(s,t|T)$ *is the shortest path of SE-Oracle-Adapt(FaceExact) between* s *and* t *that passes on the implicit TIN* T *constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *SE-Oracle-Adapt(FaceExact)*. The oracle construction time of the original oracle in [54, 55] (after pre-computing the shortest path between each pair of POIs) is $O(nm + nh\log n + \frac{nh}{\epsilon^{2\beta}})$, where m is the on-the-fly shortest path query time. In *SE-Oracle-Adapt(FaceExact)*, we use algorithm *Fly(FaceExact)* for the point cloud shortest path query, which has shortest path query time $O(N + N^2)$ according to Theorem D.1. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with N^2, and *SE-Oracle-Adapt(FaceExact)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle-Adapt(FaceExact)* is $O(N + nN^2 + nh\log n + \frac{nh}{\epsilon^{2\beta}})$.

Secondly, we prove the oracle size of *SE-Oracle-Adapt(FaceExact)*. The proof of the oracle size of *SE-Oracle-Adapt(FaceExact)* is in [54, 55]. Thus, the oracle size of *SE-Oracle-Adapt(FaceExact)* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the shortest path query time of *SE-Oracle-Adapt(FaceExact)*. The proof of the shortest path query time of *SE-Oracle-Adapt(FaceExact)* is in [54, 55]. Thus, the shortest path query time of *SE-Oracle-Adapt(FaceExact)* is $O(h^2)$.

Fourthly, we prove the error bound of *SE-Oracle-Adapt(FaceExact)*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-Adapt(FaceExact)* is algorithm *Fly(FaceExact)*, which returns the exact shortest path that passes on the implicit *TIN* constructed by the point cloud according to Theorem D.1, so the error of *SE-Oracle-Adapt(FaceExact)* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle-Adapt(FaceExact)* is in [54, 55]. Since the *TIN* is constructed by the point cloud, we obtain that *SE-Oracle-Adapt(FaceExact)* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt(FaceExact)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem D.9. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle-Adapt(FaceAppr) are* $O(N + \frac{nl_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + nh\log n + \frac{nh}{\epsilon^{2\beta}})$*,* $O(\frac{nh}{\epsilon^{2\beta}})$*, and* $O(h^2)$*, respectively. SE-Oracle-Adapt(FaceAppr) always has* $|\Pi_{SE\text{-}Oracle\text{-}Adapt(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for each pair of POIs* s *and* t *in* P*, where* $\Pi_{SE\text{-}Oracle\text{-}Adapt(FaceAppr)}(s,t|T)$ *is the shortest path of SE-Oracle-Adapt(FaceAppr) between* s *and* t *that passes on the implicit TIN* T *constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *SE-Oracle-Adapt(FaceAppr)*. The oracle construction time of the original oracle in [54, 55] (after pre-computing the shortest path between each pair of POIs) is $O(nm + nh\log n + \frac{nh}{\epsilon^{2\beta}})$, where m is the on-the-fly shortest path query time. In *SE-Oracle-Adapt(FaceAppr)*, we use algorithm *Fly(FaceAppr)* for the point cloud shortest path query, which has shortest path query time $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ according to Theorem D.2. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$, and *SE-Oracle-Adapt(FaceAppr)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle-Adapt(FaceAppr)* is $O(N + \frac{nl_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + nh\log n + \frac{nh}{\epsilon^{2\beta}})$.

Secondly, we prove the oracle size of *SE-Oracle-Adapt(FaceAppr)*. The proof of the oracle size of *SE-Oracle-Adapt(FaceAppr)* is in [54, 55]. Thus, the oracle size of *SE-Oracle-Adapt(FaceAppr)* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the shortest path query time of *SE-Oracle-Adapt(FaceAppr)*. The proof of the shortest path query time of *SE-Oracle-Adapt(FaceAppr)* is in [54, 55]. Thus, the shortest path query time of *SE-Oracle-Adapt(FaceAppr)* is $O(h^2)$.

Fourthly, we prove the error bound of *SE-Oracle-Adapt(FaceAppr)*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-Adapt(FaceAppr)* is algorithm *Fly(FaceAppr)*, which always has $|\Pi_{Fly(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P according to Theorem D.2, so the error of *SE-Oracle-Adapt(FaceAppr)* is due to the algorithm *Fly(FaceAppr)* and oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle-Adapt(FaceAppr)* is in [54, 55]. Since we assign the error in both algorithm *Fly(FaceAppr)* and the oracle itself to be $\sqrt{(1+\epsilon)}-1$, and the *TIN* is constructed by the point cloud, so we obtain that *SE-Oracle-Adapt(FaceAppr)* always has $|\Pi_{SE\text{-}Oracle\text{-}Adapt(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P (since algorithm *Fly(FaceAppr)* is an approximated algorithm, we can only obtain the upper bound of error ratio for *SE-Oracle-Adapt(FaceAppr)*). □

Theorem D.10. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle-Adapt(Point) are* $O(N + nN\log N + nh\log n + \frac{nh}{\epsilon^{2\beta}})$*,* $O(\frac{nh}{\epsilon^{2\beta}})$*, and* $O(h^2)$*, respectively. SE-Oracle-Adapt(Point) always has* $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt(Point)}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ *for each pair of POIs* s *and* t *in* P*, where* $\Pi_{SE\text{-}Oracle\text{-}Adapt(Vertex)}(s,t|C)$ *is the shortest path of SE-Oracle-Adapt(Vertex) between* s *and* t *that passes on points of the point cloud* C*.*

Proof. Firstly, we prove the oracle construction time of *SE-Oracle-Adapt(Point)*. The oracle construction time of the original oracle in [54, 55] (after pre-computing the shortest path between each pair of POIs) is $O(nm + nh\log n + \frac{nh}{\epsilon^{2\beta}})$, where m is the on-the-fly shortest path query time. In *SE-Oracle-Adapt(Point)*, we use algorithm *Fly(Point)* for the point cloud shortest path query, which has shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute m with $N\log N$. Thus, the oracle construction time of *SE-Oracle-Adapt(Point)* is $O(N + nN\log N + nh\log n + \frac{nh}{\epsilon^{2\beta}})$.

Secondly, we prove the oracle size of *SE-Oracle-Adapt(Point)*. The proof of the oracle size of *SE-Oracle-Adapt(Point)* is in [54, 55]. Thus, the oracle size of *SE-Oracle-Adapt(Point)* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the shortest path query time of *SE-Oracle-Adapt(Point)*. The proof of the shortest path query time of *SE-Oracle-Adapt(Point)* is in [54, 55]. Thus, the shortest path query time of *SE-Oracle-Adapt(Point)* is $O(h^2)$.

Fourthly, we prove the error bound of *SE-Oracle-Adapt(Point)*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-Adapt(Point)* is algorithm *Fly(Point)*, which returns the exact shortest path that passes points on the point cloud according to Theorem 4.1, the error of *SE-Oracle-Adapt(Point)* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *SE-Oracle-Adapt(Point)* is in [54, 55]. So we obtain that *SE-Oracle-Adapt(Point)* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt(Point)}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for each pair of POIs s and t in P □

Theorem D.11. *The oracle construction time, oracle size, and shortest path query time of SE-Oracle-Adapt(Vertex) are* $O(nN\log N + nh\log n + \frac{nh}{\epsilon^{2\beta}})$, $O(\frac{nh}{\epsilon^{2\beta}})$, *and* $O(h^2)$, *respectively. SE-Oracle-Adapt(Vertex) always has* $|\Pi_{SE\text{-}Oracle\text{-}Adapt(Vertex)}(s,t|T)| \geq |\Pi_{SE\text{-}Oracle\text{-}Adapt(Point)}(s,t|T)|$ *for each pair of POIs s and t in P, where* $\Pi_{SE\text{-}Oracle\text{-}Adapt(Vertex)}(s,t|T)$ *is the shortest path of SE-Oracle-Adapt(Vertex) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *SE-Oracle-Adapt(Vertex)*. The oracle construction time of the original oracle in [54, 55] (after pre-computing the shortest path between each pair of POIs) is $O(nm + nh\log n + \frac{nh}{\epsilon^{2\beta}})$, where m is the on-the-fly shortest path query time. In *SE-Oracle-Adapt(Vertex)*, we use algorithm *Fly(Vertex)* for the point cloud shortest path query, which has shortest path query time $O(N + N\log N)$ according to Theorem D.3. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with $N\log N$, and *SE-Oracle-Adapt(Vertex)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *SE-Oracle-Adapt(Vertex)* is $O(nN\log N + nh\log n + \frac{nh}{\epsilon^{2\beta}})$.

Secondly, we prove the oracle size of *SE-Oracle-Adapt(Vertex)*. The proof of the oracle size of *SE-Oracle-Adapt(Vertex)* is in [54, 55]. Thus, the oracle size of *SE-Oracle-Adapt(Vertex)* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we prove the shortest path query time of *SE-Oracle-Adapt(Vertex)*. The proof of the shortest path query time of *SE-Oracle-Adapt(Vertex)* is in [54, 55]. Thus, the shortest path query time of *SE-Oracle-Adapt(Vertex)* is $O(h^2)$.

Fourthly, we prove the error bound of *SE-Oracle-Adapt(Vertex)*. Since the on-the-fly shortest path query algorithm in *SE-Oracle-Adapt(Vertex)* is algorithm *Fly(Vertex)*, which always has $|\Pi_{Fly(Vertex)}(s,t|T)| \geq |\Pi^*(s,t|C)|$ for each pair of POIs s and t in P according to Theorem D.3. Since the error bound of the oracle itself regarding *SE-Oracle-Adapt(FaceExact)* is the same as *SE-Oracle-Adapt(Point)*, and since the *TIN* is constructed by the point cloud, we obtain that *SE-Oracle-Adapt(Vertex)* always has $|\Pi_{SE\text{-}Oracle\text{-}Adapt(Vertex)}(s,t|T)| \geq |\Pi_{SE\text{-}Oracle\text{-}Adapt(Point)}(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem D.12. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle-Naive(FaceExact) are* $O(N + nN^2 + n^2)$, $O(n^2)$, *and* $O(1)$, *respectively. RC-Oracle-Naive(FaceExact) returns the exact shortest path that passes on the implicit TIN constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *RC-Oracle-Naive(FaceExact)*. Since there are total n POIs, *RC-Oracle-Naive(FaceExact)* first needs $O(nm)$ time to calculate the shortest path from each POI to all other remaining POIs using on-the-fly shortest path query algorithm (which is a *SSAD* algorithm), where m is the on-the-fly shortest path query time. It then needs $O(n^2)$ time to store pairwise P2P shortest paths into a hash table. In *RC-Oracle-Naive(FaceExact)*, we use algorithm *Fly(FaceExact)* for the point cloud shortest path query, which has shortest path query time $O(N+N^2)$ according to Theorem D.1. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with N^2, and *RC-Oracle-Naive(FaceExact)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *RC-Oracle-Naive(FaceExact)* is $O(N + nN^2 + n^2)$.

Secondly, we prove the oracle size of *RC-Oracle-Naive(FaceExact)*. *RC-Oracle-Naive(FaceExact)* stores $O(n^2)$ pairwise P2P shortest paths. Thus, the oracle size of *RC-Oracle-Naive(FaceExact)* is $O(n^2)$.

Thirdly, we prove the shortest path query time of *RC-Oracle-Naive(FaceExact)*. *RC-Oracle-Naive(FaceExact)* has a hash table to store the pairwise P2P shortest path. Thus, the shortest path query time of *RC-Oracle-Naive(FaceExact)* is $O(1)$.

Fourthly, we prove the error bound of *RC-Oracle-Naive(FaceExact)*. Since the on-the-fly shortest path query algorithm in *RC-Oracle-Naive(FaceAppr)* is algorithm *Fly(FaceExact)*, which returns the exact shortest path that passes on the implicit *TIN* constructed by the point cloud according to Theorem D.1, and the oracle itself regarding *RC-Oracle-Naive(FaceExact)* also computes the pairwise P2P exact shortest paths, so *RC-Oracle-Naive(FaceExact)* returns the exact shortest path that passes on the implicit *TIN* constructed by the point cloud. □

Theorem D.13. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle-Naive(FaceAppr) are* $O(N + \frac{nl_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + n^2)$, $O(n^2)$, *and* $O(1)$, *respectively. RC-Oracle-Naive(FaceAppr) always has* $|\Pi_{RC\text{-}Oracle\text{-}Naive(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for each pair of POIs s and t in P, where* $\Pi_{RC\text{-}Oracle\text{-}Naive(FaceAppr)}(s,t|T)$ *is the shortest path of RC-Oracle-Naive(FaceAppr) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *RC-Oracle-Naive(FaceAppr)*. Since there are total n POIs, *RC-Oracle-Naive(FaceAppr)* first needs $O(nm)$ time to calculate the shortest path from each POI to all other remaining POIs using on-the-fly shortest path query algorithm (which is a *SSAD* algorithm), where m is the on-the-fly shortest path query time. It then needs $O(n^2)$ time to store pairwise P2P shortest paths into a hash table. In *RC-Oracle-Naive(FaceAppr)*, we use algorithm *Fly(FaceAppr)* for the point cloud shortest path query, which has shortest path query time $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ according to Theorem D.2. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with N^2, and *RC-Oracle-Naive(FaceAppr)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud.

Thus, the oracle construction time of *RC-Oracle-Naive(FaceAppr)* is $O(N + \frac{nl_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + n^2)$.

Secondly, we prove the oracle size of *RC-Oracle-Naive(FaceAppr)*. *RC-Oracle-Naive(FaceAppr)* stores $O(n^2)$ pairwise P2P shortest paths. Thus, the oracle size of *RC-Oracle-Naive(FaceAppr)* is $O(n^2)$.

Thirdly, we prove the shortest path query time of *RC-Oracle-Naive(FaceAppr)*. *RC-Oracle-Naive(FaceAppr)* has a hash table to store the pairwise P2P shortest path. Thus, the shortest path query time of *RC-Oracle-Naive(FaceAppr)* is $O(1)$.

Fourthly, we prove the error bound of *RC-Oracle-Naive(FaceAppr)*. Since the on-the-fly shortest path query algorithm in *RC-Oracle-Naive(FaceAppr)* is algorithm *Fly(FaceAppr)*, which always has $|\Pi_{Fly(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P according to Theorem D.2, and the oracle itself regarding *RC-Oracle-Naive(FaceAppr)* also computes the pairwise P2P exact shortest paths, so the error of *SE-Oracle(FaceAppr)* is due to the algorithm *Fly(FaceAppr)*, and *RC-Oracle-Naive(FaceAppr)* always has $|\Pi_{RC\text{-}Oracle\text{-}Naive(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem D.14. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle-Naive(Point) are $O(nN\log N + n^2)$, $O(n^2)$, and $O(1)$, respectively. RC-Oracle-Naive(Point) returns the exact shortest path that passes points on the point cloud.*

Proof. Firstly, we prove the oracle construction time of *RC-Oracle-Naive(Point)*. Since there are total n POIs, *RC-Oracle-Naive(Point)* first needs $O(nm)$ time to calculate the shortest path from each POI to all other remaining POIs using on-the-fly shortest path query algorithm (which is a *SSAD* algorithm), where m is the on-the-fly shortest path query time. It then needs $O(n^2)$ time to store pairwise P2P shortest paths into a hash table. In *RC-Oracle-Naive(Point)*, we use algorithm *Fly(Point)* for the point cloud shortest path query, which has shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute m with $N\log N$. Thus, the oracle construction time of *RC-Oracle-Naive(Point)* is $O(nN\log N + n^2)$.

Secondly, we prove the oracle size of *RC-Oracle-Naive(Point)*. *RC-Oracle-Naive(Point)* stores $O(n^2)$ pairwise P2P shortest paths. Thus, the oracle size of *RC-Oracle-Naive(Point)* is $O(n^2)$.

Thirdly, we prove the shortest path query time of *RC-Oracle-Naive(Point)*. *RC-Oracle-Naive(Point)* has a hash table to store the pairwise P2P shortest path. Thus, the shortest path query time of *RC-Oracle-Naive(Point)* is $O(1)$.

Fourthly, we prove the error bound of *RC-Oracle-Naive(Point)*. Since the on-the-fly shortest path query algorithm in *RC-Oracle-Naive(Point)* is algorithm *Fly(Point)*, which returns the exact shortest path that passes points on the point cloud according to Theorem 4.1, and the oracle itself regarding *RC-Oracle-Naive(Point)* also computes the pairwise P2P exact shortest paths, so *RC-Oracle-Naive(Point)* returns the exact shortest path that passes points on the point cloud. □

Theorem D.15. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle-Naive(Vertex) are $O(N+nN\log N + n^2)$, $O(n^2)$, and $O(1)$, respectively. RC-Oracle-Naive(Vertex) always has $|\Pi_{RC\text{-}Oracle\text{-}Naive(Vertex)}(s,t|T)| \geq |\Pi_{RC\text{-}Oracle\text{-}Naive(Point)}(s,t|C)|$ for each pair of POIs s and t in P, where $\Pi_{RC\text{-}Oracle\text{-}Naive(Vertex)}(s,t|T)$ is the shortest path of RC-Oracle-Naive(Vertex) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *RC-Oracle-Naive(Vertex)*. Since there are total n POIs, *RC-Oracle-Naive(Vertex)* first needs $O(nm)$ time to calculate the shortest path from each POI to all other remaining POIs using on-the-fly shortest path query algorithm (which is a *SSAD* algorithm), where m is the on-the-fly shortest path query time. It then needs $O(n^2)$ time to store pairwise P2P shortest paths into a hash table. In *RC-Oracle-Naive(Vertex)*, we use algorithm *Fly(Vertex)* for the point cloud shortest path query, which has shortest path query time $O(N + N\log N)$ according to Theorem D.3. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with N^2, and *RC-Oracle-Naive(Vertex)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *RC-Oracle-Naive(Vertex)* is $O(N + nN\log N + n^2)$.

Secondly, we prove the oracle size of *RC-Oracle-Naive(Vertex)*. *RC-Oracle-Naive(Vertex)* stores $O(n^2)$ pairwise P2P shortest paths. Thus, the oracle size of *RC-Oracle-Naive(Vertex)* is $O(n^2)$.

Thirdly, we prove the shortest path query time of *RC-Oracle-Naive(Vertex)*. *RC-Oracle-Naive(Vertex)* has a hash table to store the pairwise P2P shortest path. Thus, the shortest path query time of *RC-Oracle-Naive(Vertex)* is $O(1)$.

Fourthly, we prove the error bound of *RC-Oracle-Naive(Vertex)*. Since the on-the-fly shortest path query algorithm in *RC-Oracle-Naive(Vertex)* is algorithm *Fly(Vertex)*, which always has $|\Pi_{Fly(Vertex)}(s,t|T)| \geq |\Pi^*(s,t|C)|$ for each pair of POIs s and t in P according to Theorem D.3. Since the error bound of the oracle itself regarding *RC-Oracle-Naive(Vertex)* is the same as *RC-Oracle-Naive(Point)*, and since the *TIN* is constructed by the point cloud, so we obtain that *RC-Oracle-Naive(Vertex)* always has $|\Pi_{RC\text{-}Oracle\text{-}Naive(Vertex)}(s,t|T)| \geq |\Pi_{RC\text{-}Oracle\text{-}Naive(Point)}(s,t|C)|$ for each pair of POIs s and t in P. □

Theorem D.16. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle(FaceExact) are $O(N+\mu N^2+n\log n)$, $O(\mu n)$, and $O(1)$, respectively. RC-Oracle(FaceExact) always has $|\Pi_{RC\text{-}Oracle(FaceExact)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P, where $\Pi_{RC\text{-}Oracle(FaceExact)}(s,t|T)$ is the shortest path of RC-Oracle(FaceExact) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *RC-Oracle(FaceExact)*. The oracle construction time of oracle itself regarding *RC-Oracle(FaceExact)* in Theorem 4.2 is $O(\mu m + \log n)$, where m is the on-the-fly shortest path query time. In *RC-Oracle(FaceExact)*, we use algorithm *Fly(FaceExact)* for the point cloud shortest path query, which has shortest path query time $O(N+N^2)$ according to Theorem D.1. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with N^2, and *RC-Oracle(FaceExact)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *RC-Oracle(FaceExact)* is $O(N + \mu N^2 + n^2)$.

Secondly, we prove the oracle size of *RC-Oracle(FaceExact)*. The oracle size of *RC-Oracle(FaceExact)* is the same as the oracle size of *RC-Oracle(Point)* in Theorem 4.2. Thus, the oracle size of *RC-Oracle(FaceExact)* is $O(\mu n)$.

Thirdly, we prove the shortest path query time of *RC-Oracle(FaceExact)*. The shortest path query time of *RC-Oracle(FaceExact)* is the same as the shortest path query time of *RC-Oracle(Point)* in Theorem 4.2. Thus, the shortest path query time of *RC-Oracle(FaceExact)* is $O(1)$.

Fourthly, we prove the error bound of *RC-Oracle(FaceExact)*. Since the on-the-fly shortest path query algorithm in *RC-Oracle(FaceAppr)* is algorithm *Fly(FaceExact)*, which returns the exact shortest path that passes on the implicit *TIN* constructed by the point cloud according to Theorem D.1, so the error of *RC-Oracle(FaceExact)* is due to the oracle itself. The proof of the error bound of the oracle itself regarding *RC-Oracle(FaceExact)* is the same as *RC-Oracle(Point)* in Theorem 4.2. Since the *TIN* is constructed by the point cloud, so we obtain that *RC-Oracle(FaceExact)* always has $|\Pi_{RC\text{-}Oracle(FaceExact)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem D.17. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle(FaceAppr) are* $O(N + \frac{\mu l_{max} N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max} N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + n^2)$, $O(\mu n)$, *and* $O(1)$, *respectively. RC-Oracle(FaceAppr) always has* $|\Pi_{RC\text{-}Oracle(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ *for each pair of POIs s and t in P, where* $\Pi_{RC\text{-}Oracle(FaceAppr)}(s,t|T)$ *is the shortest path of RC-Oracle(FaceAppr) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *RC-Oracle(FaceAppr)*. The oracle construction time of oracle itself regarding *RC-Oracle(FaceAppr)* in Theorem 4.2 is $O(\mu m + n^2)$, where m is the on-the-fly shortest path query time. In *RC-Oracle(FaceAppr)*, we use algorithm *Fly(FaceAppr)* for the point cloud shortest path query, which has shortest path query time $O(N + \frac{l_{max} N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max} N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ according to Theorem D.2. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with N^2, and *RC-Oracle(FaceAppr)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *RC-Oracle(FaceAppr)* is $O(N + \frac{\mu l_{max} N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max} N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + n^2)$.

Secondly, we prove the oracle size of *RC-Oracle(FaceAppr)*. The oracle size of *RC-Oracle(FaceAppr)* is the same as the oracle size of *RC-Oracle(Point)* in Theorem 4.2. Thus, the oracle size of *RC-Oracle(FaceAppr)* is $O(\mu n)$.

Thirdly, we prove the shortest path query time of *RC-Oracle(FaceAppr)*. The shortest path query time of *RC-Oracle(FaceAppr)* is the same as the shortest path query time of *RC-Oracle(Point)* in Theorem 4.2. Thus, the shortest path query time of *RC-Oracle(FaceAppr)* is $O(1)$.

Fourthly, we prove the error bound of *RC-Oracle(FaceAppr)*. Since the on-the-fly shortest path query algorithm in *RC-Oracle(FaceAppr)* is algorithm *Fly(FaceAppr)*, which always has $|\Pi_{Fly(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P according to Theorem D.2, so the error of *RC-Oracle(FaceAppr)* is due to the algorithm *Fly(FaceAppr)* and oracle itself. The proof of the error bound of the oracle itself regarding *RC-Oracle(FaceAppr)* is the same as *RC-Oracle(Point)* in Theorem 4.2. Since we assign the error in both algorithm *Fly(FaceAppr)* and the oracle itself to be $\sqrt{(1+\epsilon)}-1$, and the *TIN* is constructed by the point cloud, so we obtain that *RC-Oracle(FaceAppr)* always has $|\Pi_{RC\text{-}Oracle(FaceAppr)}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for each pair of POIs s and t in P. □

Theorem D.18. *The oracle construction time, oracle size, and shortest path query time of RC-Oracle(Vertex) are* $O(N + \mu N \log N + n^2)$, $O(\mu n)$, *and* $O(1)$, *respectively. RC-Oracle(Vertex) always has* $|\Pi_{RC\text{-}Oracle(Vertex)}(s,t|T)| \geq |\Pi(s,t|C)|$ *for each pair of POIs s and t in P, where* $\Pi_{RC\text{-}Oracle(Vertex)}(s,t|T)$ *is the shortest path of RC-Oracle(Vertex) between s and t that passes on the implicit TIN T constructed by the point cloud.*

Proof. Firstly, we prove the oracle construction time of *RC-Oracle(Vertex)*. The oracle construction time of oracle itself regarding *RC-Oracle(Vertex)* in Theorem 4.2 is $O(\mu m + n^2)$, where m is the on-the-fly shortest path query time. In *RC-Oracle(Vertex)*, we use algorithm *Fly(Vertex)* for the point cloud shortest path query, which has shortest path query time $O(N + N \log N)$ according to Theorem D.3. But, we just need to construct the implicit *TIN* using the point cloud once at the beginning, so we substitute m with N^2, and *RC-Oracle(Vertex)* only needs an additional $O(N)$ time for constructing the implicit *TIN* using the point cloud. Thus, the oracle construction time of *RC-Oracle(Vertex)* is $O(N + \mu N \log N + n^2)$.

Secondly, we prove the oracle size of *RC-Oracle(Vertex)*. The oracle size of *RC-Oracle(Vertex)* is the same as the oracle size of *RC-Oracle(Point)* in Theorem 4.2. Thus, the oracle size of *RC-Oracle(Vertex)* is $O(\mu n)$.

Thirdly, we prove the shortest path query time of *RC-Oracle(Vertex)*. The shortest path query time of *RC-Oracle(Vertex)* is the same as the shortest path query time of *RC-Oracle(Point)* in Theorem 4.2. Thus, the shortest path query time of *RC-Oracle(Vertex)* is $O(1)$.

Fourthly, we prove the error bound of *RC-Oracle(Vertex)*. Since the on-the-fly shortest path query algorithm in *RC-Oracle(Vertex)* is algorithm *Fly(Vertex)*, which always has $|\Pi_{Fly(Vertex)}(s,t|T)| \geq |\Pi^*(s,t|C)|$ for each pair of POIs s and t in P according to Theorem D.3. Since the error bound of the oracle itself regarding *RC-Oracle(Vertex)* is the same as *RC-Oracle(Point)*, and since the *TIN* is constructed by the point cloud, we obtain that *RC-Oracle(Vertex)* always has $|\Pi_{RC\text{-}Oracle(Vertex)}(s,t|T)| \geq |\Pi(s,t|C)|$ for each pair of POIs s and t in P. □