Proximity Queries on Point Clouds using Rapid Construction Path Oracle

YINZHAO YAN, The Hong Kong University of Science and Technology, Hong Kong
RAYMOND CHI-WING WONG, The Hong Kong University of Science and Technology, Hong Kong

The prevalence of computer graphics technology boosts the developments of point clouds in recent years, which offer advantages over terrain surfaces (represented by *Triangular Irregular Networks*, i.e., *TIN*s) in proximity queries, including the *shortest path query*, the *k-Nearest Neighbor* (*kNN*) *query* and the *range query*. Since (1) all existing on-the-fly and oracle-based shortest path query algorithms on a *TIN* are very expensive, (2) all existing on-the-fly shortest path query algorithms on a point cloud are still not efficient, and (3) there are no oracle-based shortest path query algorithms on a point cloud, we propose an efficient $(1+\epsilon)$-approximate shortest path oracle that answers the shortest path query for a set of *Points-Of-Interests* (*POIs*) 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 paths between any pair of POIs stored in the oracle. Our oracle can be easily adapted to answering the shortest path query for any points on the point cloud if POIs are not given as input, and also achieve a good performance. Then, we propose efficient algorithms for answering the $(1+\epsilon)$-approximate *kNN* and range query with the assistance of our oracle. Our experimental results show that when POIs are given (resp. not given) as input, our oracle is up to 390 times, 30 times and 6 times (resp. 500 times, 140 times and 50 times) better than the best-known oracle 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 queries are both up to 100 times faster than the best-known algorithms.

CCS Concepts: • **Information systems** → **Proximity search**.

Additional Key Words and Phrases: proximity queries; spatial database; point clouds

ACM Reference Format:
Yinzhao Yan and Raymond Chi-Wing Wong. 2024. Proximity Queries on Point Clouds using Rapid Construction Path Oracle. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 6 (February 2024), 26 pages. https://doi.org/10.1145/3639261

1 INTRODUCTION

Conducting proximity queries, including (1) the *shortest path query*, i.e., given a source s and a destination t, which answers the shortest path between s and t, (2) the *k-Nearest Neighbor* (*kNN*) *query* [51], i.e., given a query object q and a user parameter k, which answers all the shortest paths from q to the k nearest objects of q, and (3) the *range query* [43], i.e., given a query object q and a range value r, which answers all the shortest paths from q to the objects whose distance to q are at most r, on a 3D surface is a topic of widespread interest in both industry and academia [25, 58]. The shortest path query is the most fundamental type of the proximity query. In industry, numerous companies and applications, such as Google Earth [2] and Cyberpunk 2077 [4], utilize the shortest path passing on a 3D surface (such as Earth) for route planning. In academia, the shortest path query

Authors' addresses: Yinzhao Yan, The Hong Kong University of Science and Technology, Hong Kong, yyanas@cse.ust.hk; Raymond Chi-Wing Wong, The Hong Kong University of Science and Technology, Hong Kong, raywong@cse.ust.hk.

ACM 2836-6573/2024/2-ART6
https://doi.org/10.1145/3639261

on a 3D model is a prevalent research topic in the field of databases [19, 30, 31, 39, 55, 56, 59, 60]. There are different representations of a 3D surface, including a terrain surface represented by a *Triangular Irregular Network* (*TIN*) and a point cloud. While performing the shortest path query on a *TIN* has been extensively studied, answering the shortest path query on a point cloud is an emerging topic. For example, Tesla uses the shortest path passing on point clouds of the driving environment for autonomous driving [12, 18, 38, 42], and Metaverse uses the shortest path passing on point clouds of objects such as mountains for efficient navigation in Virtual Reality [36, 37]. Applications of the other two proximity queries include rover path planning [14] and military tactical analysis [33].

Point cloud and *TIN*: (1) A point cloud is represented by a set of 3D *points* in space. Figure 1 (a) shows a satellite map of Mount Rainier [47] (a national park in the USA) in an area of 20km × 20km, and Figure 1 (b) shows the point cloud with 63 points of Mount Rainier. Given a point cloud, we create a *conceptual graph* of the point cloud, such that its *vertices* consist of the points in the point cloud, and its *edges* consist of a set of edges between each vertex and its 8 neighbor vertices in the 2D plane (where this graph is stored in the memory and used for the shortest path query). Figure 1 (c) shows a conceptual graph of a point cloud. (2) A *TIN* 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 (d) is a *TIN* of Mount Rainier, which consists of vertices, edges and faces. We focus on three paths: (1) the path passing on (a conceptual graph of) a point cloud in Figures 1 (b) and (c), (2) the *surface path* [31] passing on (the faces of) a *TIN* in Figure 1 (d), and (3) the *network path* [31] passing on (the edges of a) a *TIN* in Figure 1 (e).

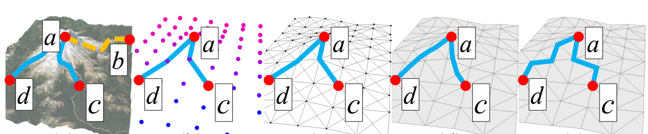

Fig. 1. (a) A satellite map, (b) paths passing on a point cloud, (c) a conceptual graph of a point cloud, (d) surface and (e) network paths passing on a *TIN*

1.1 Motivation

1.1.1 ***Advantages of point cloud.*** (1) Points clouds have four advantages compared with *TIN*s.

(i) *More direct access to point cloud data.* We can use an iPhone 12/13 Pro LiDAR scanner to scan an object and generate a point cloud in 10s [54], or can use a satellite to obtain the elevation of a region in an area of 1km^2 and generate a point cloud in 144s ≈ 2.4 min [44]. But, in order to obtain a *TIN* of an object, typically, researchers need to transform a point cloud to a *TIN* [29]. Our experimental results show that it needs 210s ≈ 3.5 min to transform a point cloud with 25M points to a *TIN*.

(ii) *Lower hard disk usage of a point cloud.* We only store the point information of a point cloud in hard disks, but a *TIN* model needs to store the vertex, edge and face information. Our experimental results show that storing a point cloud with 25M points needs 390MB in the hard disk, but storing a *TIN* generated by this point cloud needs 1.7GB in the hard disk.

(iii) *Faster shortest path query time on a point cloud.* After we transfer a point cloud to a *TIN*, calculating the shortest path passing on the point cloud is faster than calculating the shortest surface or network path passing on this *TIN*, since a *TIN* is more complicated than a point cloud. In addition, calculating the *shortest surface path* passing on a *TIN* is even slower since the search space is larger. Our experimental results show that calculating the shortest path passing on a point cloud with 2.5M points takes 3s, but calculating the shortest surface (resp. network) path passing on a *TIN* constructed by the point cloud takes 580s ≈ 10 min (resp. 17s).

(iv) *Small distance error of the shortest path passing on a point cloud.* In Figures 1 (b) and (d), the shortest path passing on a point cloud is similar to the shortest surface path passing on a *TIN* (since for the former path, each point can connect with 8 neighbor points). But, in Figures 1 (d) and (e), the shortest surface path and the shortest network path passing on a *TIN* are very different (since for the latter path, each vertex can only connected with only 6 neighbor vertices). Our experimental results show that the distance of the shortest path passing on a point cloud (resp. the shortest network path passing on a *TIN*) is 1.008 (resp. 1.1) times larger than that of the shortest surface path passing on a *TIN*.

(2) Although calculating the shortest path passing on a point cloud can be regarded as on a conceptual graph of the point cloud, point clouds have two advantages compared with graphs, i.e., (i) there is no method to directly obtain a graph of an object, and (ii) we need to store the vertex and edge information of a graph in hard disks. They are similar to (i) and (ii) in point (1). Our experimental results show that storing a point cloud with 25M points needs 390MB in the hard disk, but storing a graph generated by this point cloud needs 980MB in the hard disk.

1.1.2 ***P2P and A2A query.*** (1) Given a set of *Points-Of-Interests* (*POIs*) on a point cloud or a *TIN*, conducting (i) the shortest path query between pairs of POIs, or (ii) the *kNN* and range query such that the query object and other objects are all POIs, on the point cloud or the *TIN*, i.e., *POIs-to-POIs* (*P2P*) *query*, is important. For example, we can select POIs as reference points when measuring similarities between two different 3D objects [32, 52], and we can select POIs as residential locations when studying migration patterns of the wildness animals [22, 40]. (2) If POIs are not given as input, we need to conduct (i) the shortest path query between pairs of any points, or (ii) the *kNN* and range query such that the query object and other objects are any points, on the point cloud, i.e., *Any points-to-Any points* (*A2A*) *query*, or (iii) the shortest path query between pairs of arbitrary points, or (iv) the *kNN* and range query such that the query object and other objects are arbitrary points, on the *TIN*, i.e., *ARbitrary points-to-ARbitrary points* (*AR2AR*) *query*. Note that the AR2AR query on a *TIN* is more general than the A2A query on a point cloud since a point may lie on the face of a *TIN*.

1.1.3 ***Usage of oracles.*** Although answering the proximity query on a point cloud *on-the-fly* is fast, if we can pre-compute the pairwise P2P or A2A shortest paths by means of indexing (called an *oracle*) on a point cloud, then we can use the oracle to answer the proximity query more efficiently, where the time taken to pre-compute the oracle is called the *oracle construction time*, the space complexity of the oracle is called the *oracle size*, and the time taken to return the shortest path is called the *shortest path query time*.

1.1.4 ***Example.*** We conducted a case study on an evacuation simulation in Mount Rainier due to snowfall [48]. In Figure 1 (a), we need to find the shortest paths (in blue and yellow lines) from one of the viewing platforms (e.g., POI a) on the mountain to its k-nearest hotels (e.g., POIs b to d) due to the limited capacity of each hotel. In Figures 1 (b) - (e), c and d are the k-nearest hotels to a where $k = 2$. Our experimental results show that we can construct an oracle on a point cloud with 5M points and 500 POIs (250 viewing platforms and 250 hotels) in 400s ≈ 6.6 min, but it needs 77,200s ≈ 21.4 hours on a *TIN* (constructed based on the same point cloud) to construct the same oracle. In addition, we can return the shortest paths from each viewing platform to its k-nearest hotels in 6s with the oracle, but it needs 4,400s ≈ 1.2 hours on a point cloud without the oracle. These show the usefulness of performing proximity queries on point clouds using oracles in real-life applications.

1.2 Challenges

1.2.1 ***Inefficiency of on-the-fly algorithms.*** All existing algorithms [45, 53, 62] for conducting proximity queries on a point cloud *on-the-fly* are very slow, since they (1) first construct a *TIN* using the given point cloud in $O(N)$ time, where N is the number of points in the point cloud, and (2) then calculate the shortest path passing on this *TIN*. For calculating the shortest surface path passing on a *TIN*, the best-known on-the-fly *exact* [15] and *approximate* [30] algorithm run in $O(N^2)$ and $O((N + N') \log(N + N'))$ time, respectively, where N' is the number of additional points introduced for bound guarantee. For calculating the shortest network path passing on a *TIN*, the best-known on-the-fly *approximate* algorithm [31] runs in $O(N \log N)$ time. Our experimental results show (1) algorithm [15] needs 290,000s ≈ 3.4 days, (2) algorithm [30] needs 161,000s ≈ 1.9 days, and (3) algorithm [31] needs 15,000s ≈ 4.2 hours to perform the *kNN* query for all 500 objects on a *TIN* (constructed by the given point cloud) with 0.5M vertices.

1.2.2 ***Non-existence of oracles.*** No existing oracle can answer proximity queries on a point cloud. The best-known oracle [55, 56] for the P2P query and the best-known oracle [28] for the AR2AR query only pre-compute shortest surface paths passing on a *TIN*. Although we can first construct a *TIN* using the point cloud, then use [28, 55, 56] for point cloud oracle construction, their oracle construction time is very large due to the *bad criterion for algorithm earlier termination*. This is because although they use the *Single-Source All-Destination* (*SSAD*) algorithm [15, 30, 31], i.e., a Dijkstra-based algorithm [23], to pre-compute the shortest surface path passing on the *TIN* from each POI (or point) to other POIs (or points), and provide a criterion to *terminate it earlier*, its criterion is very loose, and different POIs (or points) have the *same* earlier termination criterion. In our experiment, even after the *SSAD* algorithm has visited most of the POIs (or points), their earlier termination criterion are still not reached. After constructing a *TIN* using the given point cloud, the oracle construction time is $O(nN^2 + c_1 n)$ for the oracle [55, 56] and is $O(c_2 N^2)$ for the oracle [28], respectively, where n is the number of POIs on the point cloud and c_1, c_2 are constants depending on the point cloud ($c_1 \in [35, 80]$ on a point cloud with 2.5M points, $c_2 \in [75, 154]$ on a point cloud with 100k points). In our experiment, the oracle construction time for the oracle [55, 56] is 78,000s ≈ 21.7 hours on a point cloud with 2.5M points and 500 POIs and for the oracle [28] is 50,000s ≈ 13.9 hours on a point cloud with 100k points.

1.3 Our Oracle and Proximity Query Algorithms

We propose an efficient $(1 + \epsilon)$-approximate shortest path oracle that answers the P2P shortest path query on a point cloud called *Rapid Construction path Oracle*, i.e., *RC-Oracle*, 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 [55, 56] for the P2P query on a point cloud due to the concise information about the pairwise shortest paths between any pair of POIs stored in the oracle, where ϵ is a non-negative real user parameter called an *error parameter*. *RC-Oracle* can be easily adapted to answer the A2A shortest path query on the point cloud if POIs are not given as input (we denote it as *RC-Oracle-A2A*), and also achieve a good performance compared with the best-known oracle [28] for the A2A query on a point cloud. Based on *RC-Oracle* and *RC-Oracle-A2A*, we develop efficient $(1 + \epsilon)$-approximate proximity query algorithms. We introduce the key idea of the small oracle construction time of *RC-Oracle*.

(1) **Rapid point cloud on-the-fly shortest path query algorithm**: When constructing *RC-Oracle*, we propose algorithm *Fast on-the-Fly shortest path query*, i.e., *FastFly*, which is a Dijkstra-based algorithm [23] returning its calculated shortest path passing on a point cloud. It can significantly reduce the algorithm's running time, since computing the shortest path passing on a *TIN* is expensive.

(2) **Rapid oracle construction**: When constructing *RC-Oracle*, we use algorithm *FastFly*, i.e., a *SSAD* algorithm, to calculate the shortest path passing on the point cloud from for each POI to other POIs *simultaneously*, and set *different* earlier termination criterion for different POIs, i.e., this criterion is tight.

1.4 Contributions and Organization

We summarize our major contributions as follows.

(1) We propose *RC-Oracle*, which is the first oracle that efficiently answers the shortest path queries on a point cloud. We also propose algorithm *FastFly* used for constructing *RC-Oracle*, and develop efficient proximity query algorithms using *RC-Oracle*.

(2) We provide theoretical analysis on (i) the oracle construction time, oracle size, shortest path query time and error bound of *RC-Oracle*, (ii) the shortest path query time and error bound of algorithm *FastFly*, (iii) the *kNN* query time, range query time and error bound for proximity queries, and (iv) the distance relationships of the shortest path passing on a point cloud or a *TIN*.

(3) *RC-Oracle* performs much better than the best-known oracle [55, 56] for the P2P query and *RC-Oracle-A2A* performs much better than the best-known oracle [28] for the A2A query on a point cloud in terms of the oracle construction time, oracle size and shortest path query time. The *kNN* and range query time with the assistance of *RC-Oracle* and *RC-Oracle-A2A* also perform much better than the best-known oracles [28, 55, 56]. Our experimental results show that (i) for the P2P query on a point cloud with 2.5M points and 500 POIs, the oracle construction time and oracle size for *RC-Oracle* is 200s $\approx$ 3.2 min and 50MB, but is 78,000s $\approx$ 21.7 hours and 1.5GB for the best-known oracle [55, 56], (ii) the *kNN* and range query time of all 500 POIs for *RC-Oracle* are both 12.5s, but the best-known oracle [55, 56] needs 150s, and the best-known on-the-fly approximate shortest surface path query algorithm [30] on the *TIN* (constructed by the given cloud) needs 161,000s $\approx$ 1.9 days, and (iii) for the A2A query on a point cloud with 100k points and 5000 objects, the oracle construction time, oracle size and *kNN* query time for *RC-Oracle-A2A* is 100s $\approx$ 1.6 min, 150M and 0.25s, but is 50,000s $\approx$ 13.9 hours, 21GB and 12.5s for the best-known oracle [28]. *RC-Oracle* also supports real-time responses, i.e., it can construct the oracle in 0.4s and answer the *kNN* query and range query in both 7ms on a point cloud with 10k points and 250 POIs.

The remainder of the paper is organized as follows. Section 2 provides the problem definition. Section 3 covers the related work. Section 4 presents the methodology. Section 5 covers the empirical studies and Section 6 concludes the paper.

2 PROBLEM DEFINITION

2.1 Notations and Definitions

2.1.1 ***Point cloud and TIN.*** Given a set of points, we let C be a point cloud of these points, and N be the number of points in C. Each point $p \in C$ has three coordinate values, denoted by x_p, y_p and z_p. We let x_{max} and x_{min} (resp. y_{max} and y_{min}) be the maximum and minimum x (resp. y) coordinate value for all points in C. We let $L_x = x_{max} - x_{min}$ (resp. $L_y = y_{max} - y_{min}$) be the side length of C along x-axis (resp. y-axis), and $L = \max(L_x, L_y)$. Figure 2 (a) shows a point cloud C with $L_x = L_y = 4$. In this paper, the point cloud C that we considered is a grid-based point cloud [11, 24], because a grid-based 3D object, e.g., a grid-based point cloud [11, 24] and a grid-based *TIN* [20, 39, 51, 55, 56], 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. In Figure 2 (a), given a green point q, $N(q)$ is denoted as 7 blue points and 1 red point s. We can easily extend our problem to the non-grid-based point cloud. Given a point p in a non-grid-based point cloud, we just change $N(p)$ 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. Let P be a set of POIs each of which is a point on the point cloud and n be the size of P. Since a POI can only be a point on C, $n \leq N$, i.e., POIs are a subset of points in a point cloud. Let T be a *TIN* triangulated [46] by the points in C. Figure 2 (b) shows an example of a *TIN* T. In this figure, given a green vertex q, the neighbor vertices of q are 6 blue vertices.

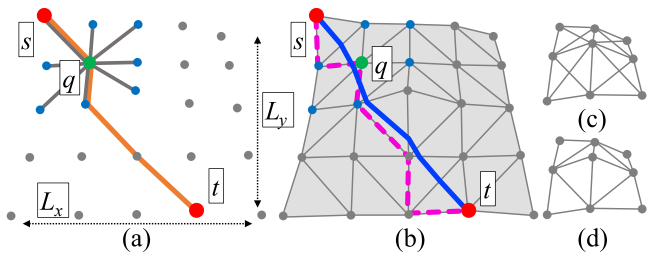

Fig. 2. (a) A point cloud with orange $\Pi^*(s,t|C)$, (b) a *TIN* with blue $\Pi^*(s,t|T)$ and pink $\Pi_N(s,t|T)$, (c) a conceptual graph of a point cloud, and (d) a conceptual graph of a *TIN*

2.1.2 ***Conceptual graph.*** We define G to be a conceptual graph of C. Let $G.V$ and $G.E$ be the set of vertices and edges of G. Each point in C is denoted by a vertex in $G.V$. For each point $q \in C$, $G.E$ consists of a set of edges between q and $q' \in N(q)$. Figure 2 (c) shows a conceptual graph of a point cloud. Given a pair of points p and p' in 3D space, we define $d_E(p,p')$ to be the Euclidean distance between p and p'. Given a pair of POIs s and t in P, (1) let $\Pi^*(s,t|C) = (s = q_1, q_2, \ldots, q_l = t)$, with $l \geq 2$, be the exact shortest path passing on (G of) C between s and t, such that (i) each q_i is a vertex in $G.V$, (ii) each (q_i, q_{i+1}) is an edge in $G.E$, and (iii) $\sum_{i=1}^{l-1} d_E(q_i, q_{i+1})$ is the minimum, and (2) let $\Pi(s,t|C)$ be the shortest path returned by *RC-Oracle*. The shortest path passing on C from a source (POI) to a destination (POI) can contain different sub-paths where a sub-path starts from a point on C to another point on C, i.e., the differences between the points and POIs are that (1) we use points (from C) to construct G, and then calculate the shortest path passing on G, but (2) we use POIs as sources and destinations to calculate the shortest path. G is stored as a data structure in the memory for internal processing and C can be cleared from the memory, so we do not need to construct G every time when we need to calculate the shortest path passing on C. Our experimental results show that it just needs 0.01s to construct G of C with 2.5M points. Figure 2 (a) shows an example of $\Pi^*(s,t|C)$ in orange line. We define $|\cdot|$ to be the distance ofa path (e.g., $|\Pi^*(s,t|C)|$ is the distance of$\Pi^*(s,t|C)$). *RC-Oracle* guarantees that $|\Pi(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any s and t in P.

Similar to G, we define G' to be a conceptual graph of T. Let $G'.V$ and $G'.E$ be the set of vertices and edges of G', where each vertex in T is denoted by a vertex in $G'.V$, and each edge in T is denoted by an edge in $G'.E$. Figure 2 (d) shows a conceptual graph of a *TIN*. Given a pair of POIs s and t in P, (1) let $\Pi^*(s,t|T) = (s = m_1, m_2, \ldots, m_l = t)$ be the exact shortest surface path passing on T between s and t, such that (i) each m_i is a point along an edge of T, and (ii) $\sum_{i=1}^{l-1} d_E(m_i, m_{i+1})$ is the minimum, (2) let $\Pi_N(s,t|T) = (s = n_1, n_2, \ldots, n_l = t)$ be the shortest network path passing on (G' of) T between s and t, such that (i) each n_i is a vertex in $G'.V$, (ii) each (n_i, n_{i+1}) is an edge in $G'.E$, and (iii) $\sum_{i=1}^{l-1} d_E(n_i, n_{i+1})$ is the minimum. G' is also stored as a data structure in the memory for internal processing and T can be cleared from the memory. Figure 2 (b) shows an example of $\Pi^*(s,t|T)$ in blue line and $\Pi_N(s,t|T)$ in pink line. Table 1 shows a notation table.

Table 1. 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
$d_E(p, p')$	The Euclidean distance between point p and p'
P	The set of POI
n	The number of vertices of P
ϵ	The error parameter
T	The *TIN* constructed by C
$\Pi^*(s, t\|C)$	The exact shortest path passing on C between s and t
$\|\Pi^*(s, t\|C)\|$	The distance of$\Pi^*(s, t\|C)$
$\Pi(s, t\|C)$	The shortest path passing on C between s and t returned by *RC-Oracle*
$\Pi^*(s, t\|T)$	The exact shortest surface path passing on T between s and t
$\Pi_N(s, t\|T)$	The shortest network path passing on T between s and t

2.1.3 ***P2P and A2A query.*** By creating POIs that have the same coordinate values as all points in the point cloud, the A2A query can be regarded as one form of the P2P query. Furthermore, in the P2P query, there is no need to consider the case when a new POI is added or removed. In the case when a POI is added, we can create an oracle to answer the A2A 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.

2.2 Problem

The problem is to (1) design an efficient $(1 + \epsilon)$-approximate shortest path oracle on a point cloud with the state-of-the-art performance in terms of the oracle construction time, oracle size and shortest path query time, and (2) use this oracle for efficiently answering the $(1 + \epsilon)$-approximate *kNN* and range query.

3 RELATED WORK

3.1 On-the-fly Algorithms

All existing *on-the-fly* proximity query algorithms [45, 53, 62] on a point cloud are very slow. Given a point cloud, they first triangulate it into a *TIN* [46] in $O(N)$ time, then they calculate the shortest path passing on this *TIN*. To the best of our knowledge, no algorithm can answer proximity queries on a point cloud directly without converting it to a *TIN*. There are two types of *TIN* shortest path query algorithms, i.e., (1) the *shortest surface path* [15, 30, 35, 41, 57] and (2) the *shortest network path* [31] query algorithms.

3.1.1 ***Shortest surface path query algorithms.*** There are two more sub-types. (1) *Exact algorithms*: Algorithm [41] (resp. algorithm [57]) uses continuous Dijkstra (resp. checking window) algorithm to calculate the result in $O(N^2 \log N)$ (resp. $O(N^2 \log N)$) time, and the best-known exact shortest surface path query algorithm *Chen and Han*, i.e., algorithm *CH* [15] (as recognized by [30, 31, 51, 58]) 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, algorithm *CH* (without constructing a *TIN* first) cannot be directly adapted on the point cloud, because there is no face to be unfolded in a point cloud. (2) *Approximate algorithms*: All algorithms [30, 35] 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 result. The best-known $(1 + \epsilon)$-approximate shortest surface path query algorithm, i.e., algorithm *Kaul* [30] (as recognized by [55, 56]) runs in $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ time, 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 pass on the conceptual graph of the point cloud, algorithm *Kaul* (without constructing a *TIN* first) can be adapted on the point cloud, and it becomes algorithm *FastFly*.

3.1.2 ***Shortest network path query algorithm***. Since the shortest network path does not cross the faces of a *TIN*, it is an approximate path. The best-known approximate shortest network path query algorithm *Dijkstra*, i.e., algorithm *Dijk* [31] runs in $O(N \log N)$ time. If we let the path pass on the conceptual graph of the point cloud, algorithm *Dijk* (without constructing a *TIN* first) can be adapted on the point cloud, and it becomes algorithm *FastFly*.

Drawbacks of the on-the-fly algorithms: Although we can pre-process the point cloud and store the generated *TIN* as a data structure in the memory, all these algorithms are still time-consuming. Since the time for calculating the shortest path passing on a *TIN* is 10^2 to 10^5 times larger than the time for converting a point cloud to a *TIN*. Thus, the latter time can be neglected. We denote algorithm (1) *CH-Adapt*, (2) *Kaul-Adapt* and (3) *Dijk-Adapt*, to be the adapted algorithm [45, 53, 62], which first constructs a *TIN* using the given point cloud (i.e., we store the *TIN* as a data structure in the memory and clear the given point cloud from the memory), and then use algorithm (1) *CH* [15], (2) *Kaul* [30] and (3) *Dijk* [31] to compute the corresponding shortest path passing on the *TIN*. Since we regard the shortest path passing on a point cloud as the exact shortest path, algorithm *CH-Adapt*, *Kaul-Adapt* and *Dijk-Adapt* return the approximate shortest path passing on a point cloud. Our experimental results show algorithm *CH-Adapt*, *Kaul-Adapt* and *Dijk-Adapt* first needs to convert a point cloud with 0.5M points to a *TIN* in 4.2s, then perform the *kNN* query for all 2500 objects on this *TIN* in 290,000s $\approx$ 3.2 days, 90,000s $\approx$ 1 day and 15,000s $\approx$ 4.2 hours, respectively.

3.2 Oracles for the shortest path query

No existing oracle can answer the shortest path query between pairs of POIs (or any points) on a point cloud. But, *Space Efficient Oracle* (*SE-Oracle*) [55, 56] (resp. *Efficiently ARbitrary pints-to-arbitrary points Oracle* (*EAR-Oracle*) [28]) can answer the P2P (resp. AR2AR) by using an oracle to index shortest surface paths passing on a *TIN*. We denote (1) *SE-Oracle-Adapt* to be the adapted oracle of *SE-Oracle* [55, 56] that first constructs a *TIN* from a point cloud (i.e., we store the *TIN* as a data structure in the memory and clear the given point cloud from the memory), then uses *SE-Oracle* on this *TIN*. Similarly, we denote (2) *EAR-Oracle-Adapt* as the adapted oracle of *EAR-Oracle* [28]. By performing a linear scan using the shortest path query results, they can answer other proximity queries.

3.2.1 ***SE-Oracle-Adapt***. It uses a *compressed partition tree* [55, 56] and *well-separated node pair sets* [13] to index the $(1+\epsilon)$-approximate pairwise P2P shortest surface paths passing on a *TIN* (constructed by the given point cloud). Its oracle construction time, oracle size and shortest path query time are $O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ and $O(h^2)$, respectively, where h is the height of the compressed partition tree and $\beta \in [1.5, 2]$ is the largest capacity dimension [55, 56]. It is regarded as the best-known oracle for the P2P query on a point cloud.

Drawbacks of *SE-Oracle-Adapt*: Its oracle construction time is large due to the *bad criterion for algorithm earlier termination*. For POIs in the same level of the compressed partition tree, they have the *same* earlier termination criteria. But, in *RC-Oracle*, we have *different* earlier termination criteria for each different POI, to minimize the running time of the *SSAD* algorithm. In the P2P query on a point cloud, for a point cloud with 2.5M points and 500 POIs, the oracle construction time of *SE-Oracle-Adapt* is 78,000s $\approx$ 21.7 hours, while *RC-Oracle* just needs 200s $\approx$ 3.2 min.

3.2.2 ***EAR-Oracle-Adapt***. It also uses well-separated node pair sets, which is similar to *SE-Oracle-Adapt*. But, *EAR-Oracle-Adapt* adapts *SE-Oracle-Adapt* from the P2P query on a point cloud to the A2A query on a point cloud by using Steiner points on the faces of the *TIN* (constructed by the given point cloud) and *highway nodes* as POIs in well-separated node pair sets construction. Its oracle construction time, oracle size and shortest path query time are $O(\lambda\xi mN^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ and $O(\lambda\xi\log(\lambda\xi))$, respectively, where λ is the number of highway nodes covered by a minimum square, ξ is the square root of the number of boxes, and m is the number of Steiner points per face. It is regarded as the best-known oracle for the A2A query on a point cloud.

Drawbacks of *EAR-Oracle-Adapt*: It also has the *bad criterion for algorithm earlier termination* drawback. In the A2A query on a point cloud, for a point cloud with 100k points, the oracle construction time of *EAR-Oracle-Adapt* is 50,0000s $\approx$ 13.9 hours, while *RC-Oracle-A2A* just needs 100s $\approx$ 1.6 min.

3.3 Oracles for other proximity queries

No existing oracle can answer proximity queries on a point cloud. But, studies [20, 21, 51] build an oracle to answer proximity queries on a *TIN*. Specifically, studies [20, 21] use a multi-resolution terrain model (resp. *SUrface Oracle* (*SU-Oracle*) [51] uses a surface index) to answer the *kNN* query on a *TIN* in $O(N^2)$ (resp. $O(N\log^2 N)$) time. We adapt *SU-Oracle* to be *SU-Oracle-Adapt* in a similar way of *SE-Oracle-Adapt*. Although *SU-Oracle-Adapt* is regarded as the best-known oracle to directly answer the *kNN* query, studies [55, 56] show the *kNN* query time of *SU-Oracle-Adapt* is up to 5 times larger than that of using *SE-Oracle-Adapt* with a linear scan of the shortest path query result.

3.4 Comparisons

We compare *RC-Oracle*, algorithm *FastFly* and other algorithms that support the shortest path query on a point cloud in Table 2. Recall that when constructing *RC-Oracle*, we have different earlier termination criteria for different POIs when using algorithm *FastFly*. We denote the naive version of our oracle as *RC-Oracle-Naive* if no earlier termination criterion is used. From the table, *RC-Oracle* is the best oracle and algorithm *FastFly* is the best on-the-fly algorithm.

Table 2. Comparison of algorithms (support the shortest path query) on a point cloud

Algorithm	Oracle construction time		Oracle size		Shortest path query time		Error
Oracle-based algorithm							
SE-Oracle-Adapt [55, 56]	$O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
EAR-Oracle-Adapt [28]	$O(\lambda\xi mN^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi\log(\lambda\xi))$	Medium	Small
RC-Oracle-Naive	$O(nN\log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Tiny	Small
***RC-Oracle* (ours)**	$\boldsymbol{O(\frac{N\log N}{\epsilon} + n\log n)}$	**Small**	$\boldsymbol{O(\frac{n}{\epsilon})}$	**Small**	$\boldsymbol{O(1)}$	**Tiny**	**Small**
On-the-fly algorithm							
CH-Adapt [15]	-	N/A	-	N/A	$O(N^2)$	Large	Small
Kaul-Adapt [30]	-	N/A	-	N/A	$O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$	Large	Small
Dijk-Adapt [31]	-	N/A	-	N/A	$O(N\log N)$	Medium	Medium
***FastFly* (ours)**	-	N/A	-	N/A	$\boldsymbol{O(N\log N)}$	**Medium**	**No error**

Remark: $n << N$, h is the height of the compressed partition tree, β is the largest capacity dimension [55, 56], λ is the number of highway nodes covered by a minimum square, ξ is the square root of the number of boxes, m is the number of Steiner points per face, 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.

4 METHODOLOGY

4.1 Overview of *RC-Oracle*

We first use an example to illustrate *RC-Oracle*. In Figure 3 (a), we have a point cloud and a set of POIs. In Figures 3 (b) - (e), we construct *RC-Oracle* by calculating the shortest paths among these POIs. In Figure 3 (f), we answer the shortest path query between two POIs using *RC-Oracle*. Next, we introduce the two components and two phases of *RC-Oracle*.

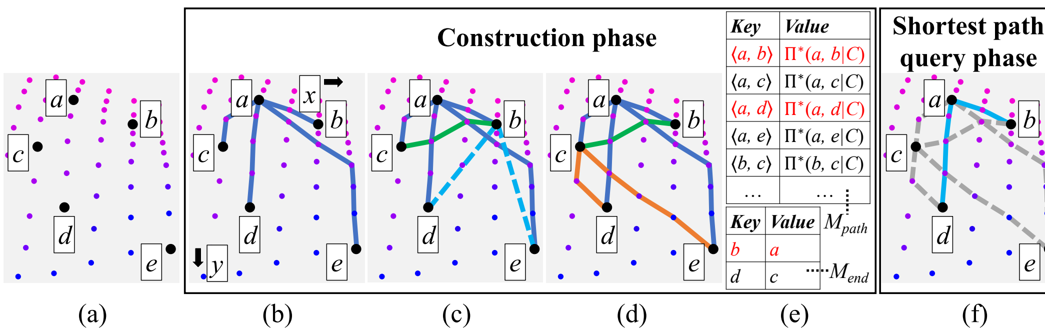

Fig. 3. *RC-Oracle* framework overview

4.1.1 *Components of RC-Oracle.*

There are two components, i.e., the *path map table* and the *endpoint map table*.

(1) **The path map table** M_{path} is a *hash table* [17] that stores a set of key-value pairs. For each key-value pair, it stores a pair of endpoints (i.e., POIs) u and v, as a key $\langle u, v\rangle$, and the corresponding exact shortest path $\Pi^*(u, v|C)$ passing on C, as a value. M_{path} needs linear space in terms of the number of paths to be stored. Given a pair of endpoints (i.e., POIs) u and v, M_{path} can return the associated exact shortest path $\Pi^*(u, v|C)$ passing on C in $O(1)$ time. In Figure 3 (d), there are 7 exact shortest paths passing on C, and they are stored in M_{path} in Figure 3 (e). For the exact shortest paths passing on C between b and c, M_{path} stores $\langle b, c\rangle$ as a key and $\Pi^*(b, c|C)$ as a value.

(2) **The endpoint map table** M_{end} is a *hash table* that stores a set of key-value pairs. For each key-value pair, it stores an endpoint (i.e., a POI) u as a key (such that we do not store all the exact shortest paths passing on C in M_{path} from u to other non-processed endpoints), and another endpoint (i.e., a POI) v as a value (such that v is close to u, and we concatenate $\Pi^*(u, v|C)$ and the exact shortest paths passing on C with v as a source, to approximate the shortest paths passing on C with u as a source). The space consumption and query time of M_{end} is similar to M_{path}. In Figure 3 (d), a is close to b, we concatenate $\Pi^*(b, a|C)$ and the exact shortest paths passing on C with a as a source, to approximate the shortest paths passing on C with b as a source, so we store b as a key and a as a value in M_{end} in Figure 3 (e).

4.1.2 *Phases of RC-Oracle.*

There are two phases, i.e., *construction phase* and *shortest path query phase* (see Figure 3). (1) In the construction phase, given a point cloud C and a set of POIs P, we pre-compute the exact shortest paths passing on C between some selected pairs of POIs, store them in M_{path}, and store the non-selected POIs and their corresponding selected POIs in M_{end}. (2) In the shortest path query phase, given a pair of POIs, M_{path} and M_{end}, we answer the path results between this pair of POIs efficiently.

4.2 Key Idea of *RC-Oracle*

4.2.1 ***Small oracle construction time.*** We give the reason why *RC-Oracle* has a small oracle construction time.

(1) **Rapid point cloud on-the-fly shortest path querying by algorithm *FastFly***: When constructing *RC-Oracle*, given a point cloud C and a pair of POIs s and t on C, we use algorithm *FastFly* (a Dijkstra's algorithm [23]) to directly calculate the *exact* shortest path passing on the conceptual graph of C between s and t. Figure 4 (a) shows the shortest path passing on a point cloud calculated by algorithm *FastFly*, and Figure 4 (b) (resp. Figure 4 (c)) shows the shortest surface (resp. network) path passing on a *TIN* calculated by algorithm *CH-Adapt* (resp. *Dijk-Adapt*) of Mount Rainier in an area of 20km × 20km. The path in Figures 4 (a) and (b) are similar, but calculating the former path is much faster than the latter path, since the query region of the former path is smaller than the latter path. The path in Figure 4 (c) has a larger error than the path in Figure 4 (a). Thus, we use algorithm *FastFly* as the on-the-fly algorithm for constructing *RC-Oracle*.

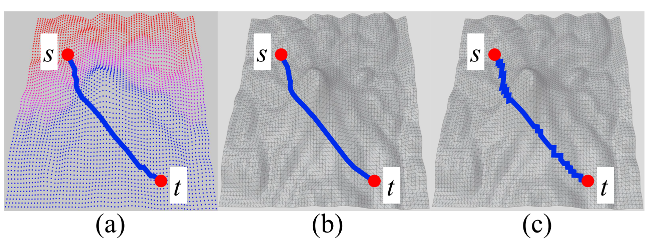

Fig. 4. (a) The shortest path passing on a point cloud, the shortest (b) surface and (c) network path passing on a *TIN*

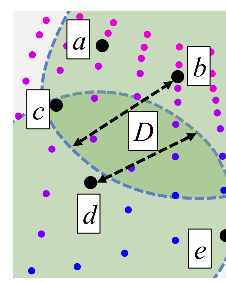

Fig. 5. *SE-Oracle-Adapt*

(2) **Rapid oracle construction**: When constructing *RC-Oracle*, we regard each POI as a source and use algorithm *FastFly*, i.e., a *SSAD* algorithm, for n times for oracle construction, and we assign a *different* earlier termination criteria for each POI to terminate the *SSAD* algorithm earlier for time-saving. There are two versions of a *SSAD* algorithm. (i) Given a source POI and a set of destination POIs, the *SSAD* algorithm can terminate earlier if it has visited all destination POIs. (ii) Given a source POI and a *termination distance* (denoted by D), the *SSAD* algorithm can terminate earlier if the searching distance from the source POI is larger than D. We use the first version. For each POI, by considering more geometry information of the point cloud, including the Euclidean distance and the distance of the previously calculated shortest paths, we use *different* earlier termination criteria to calculate the corresponding destination POIs, such that the number of destination POIs is minimized, and these destination POIs are closer to the source POI compared with other POIs.

We use an example for illustration. In Figure 3 (a), we have a set of POIs a, b, c, d, e. In Figure 3 (b) - (d), we process these POIs based on their y-coordinate, i.e., we process them in the order of a, b, c, d, e. In Figure 3 (b), for a, we use the *SSAD* algorithm (i.e., *FastFly*) to calculate the shortest paths passing on C from a to all other POIs. We store the paths in M_{path}. In Figure 3 (c), for b, if b is close to a, i.e., judged using the previously calculated $|\Pi^*(a, b|C)|$, and b is far away from d (resp. e), i.e., judged using the Euclidean distance $d_E(b, d)$ (resp. $d_E(b, e)$), we can use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ (resp. $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$) to approximate $\Pi^*(b, d|C)$ (resp. $\Pi^*(b, e|C)$). Thus, we just need to use the *SSAD* algorithm with b as a source, and terminate earlier when it has visited c. We store the path in M_{path}, and b as key and a as value in M_{end}. In Figure 3 (d), for c, we repeat the process as of for a. We store the paths in M_{path}. Similarly, for d, we use $|\Pi^*(c, d|C)|$ and $d_E(c, e)$ to determine whether we can terminate the *SSAD* algorithm earlier with d as a source. We found that there is even no need to use the *SSAD* algorithm with d as the source. For *different* POIs b and d, we use *different* termination criteria (i.e., $|\Pi^*(a, b|C)|$ and $d_E(b, d)$ for b, $|\Pi^*(c, d|C)|$ and $d_E(c, e)$ for d) to calculate a different set of destination POIs for time-saving. We store d as key and c as value in M_{end}. In Figure 3 (e), we have M_{path} and M_{end}.

However, in *SE-Oracle-Adapt*, it has the *bad criterion for algorithm earlier termination* drawback. After the construction of the compressed partition tree, it pre-computes the shortest surface paths passing on T using the *SSAD* algorithm (i.e., *CH-Adapt*) with each POI as a source for n times, to construct the well-separated node pair sets. It uses the second version of the *SSAD* algorithm and sets termination distance $D = \frac{8r}{\epsilon} + 10r$, where r is the radius of the source POI in the compressed partition tree. Given two POIs a and b in the same level of the tree, their termination distances are the same (suppose that the value is d_1). However, for a, it is enough to terminate the *SSAD* algorithm when the searching distance from a is larger than d_2, where $d_2 < d_1$. This results in a large oracle construction time. In Figure 5, when processing d, suppose that b and d are in the same level of the tree, and they use the *same* termination criteria to get the *same* termination distance D. Since $|\Pi^*(d, e|C)| < D$, for d, it cannot terminate the *SSAD* algorithm earlier until e is visited. The two versions of the *SSAD* algorithm are similar, we achieve a small oracle construction time mainly by using *different* termination criteria for different POIs, unlike using the *same* termination criteria for different POIs in *SE-Oracle-Adapt*.

4.2.2 ***Small oracle size.*** We introduce the reason why *RC-Oracle* has a small oracle size. We only store a small number of paths in *RC-Oracle*, i.e., we do not store the paths between any pairs of POIs. In Figure 3 (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.

4.2.3 ***Small shortest path query time.*** We use an example to introduce the reason why *RC-Oracle* has a small shortest path query time. In Figure 3 (f), in the shortest path query phase of *RC-Oracle*, we need to query the shortest path passing on C (1) between a and d, and (2) between b and d. (1) For a and d, since $\langle a, d\rangle \in M_{path}.key$, we can directly return $\Pi^*(a, d|C)$. (2) For b and d, since $\langle b, d\rangle \notin M_{path}.key$, b and d are both keys in M_{end}, we use the key b with a smaller y-coordinate value to retrieve the value a in M_{end}, then in M_{path}, we use $\langle b, a\rangle$ and $\langle a, d\rangle$ to retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, for approximating $\Pi^*(b, d|C)$.

4.3 Implementation Details of *RC-Oracle*

4.3.1 ***Construction Phase.*** Given a point cloud C and a set of POIs P, *RC-Oracle* pre-computes the exact shortest paths passing on C between some selected pairs of POIs, stores them in M_{path}, and stores the non-selected POIs and their corresponding POIs in M_{end}.

Notation: Let $P_{remain} = \{p_1, p_2, \dots\}$ be a set of remaining POIs of P that we have not used algorithm *FastFly* to calculate the exact shortest paths passing on C with $p_i \in P_{remain}$ as a source. P_{remain} is initialized to be P. Given a POI q, let $P_{dest}(q) = \{p_1, p_2, \dots\}$ be a set of POIs of P that we need to use *FastFly* to calculate the exact shortest paths passing on C from q to $p_i \in P_{dest}(q)$. $P_{dest}(q)$ is empty at the beginning. In Figure 3 (c), $P_{remain} = \{c, d, e\}$ since we have not used *FastFly* to calculate the exact shortest paths with c, d, e as source, $P_{dest}(b) = \{c\}$ since we need to use *FastFly* to calculate the exact shortest path from b to c.

Detail and example: Algorithm 1 shows the construction phase in detail, and the following illustrates it with an example.

(1) *POIs sort* (lines 2-3): In Figure 3 (b), since $L_x < L_y$, the sorted POIs are a, b, c, d, e.

(2) *Shortest paths calculation* (lines 4-20): There are two steps.

(i) *Exact shortest paths calculation* (lines 5-9): In Figure 3 (b), a has the smallest y-coordinate based on the sorted POIs in P_{remain}, we delete a from P_{remain} (so $P_{remain} = P'_{remain} = \{b, c, d, e\}$), calculate

Algorithm 1 *Construction* (C, P)

Input: a point cloud C and a set of POIs P
Output: a path map table M_{path} and an endpoint map table M_{end}

1: $P_{remain} \leftarrow P, M_{path} \leftarrow \emptyset, M_{end} \leftarrow \emptyset$
2: **if** $L_x \geq L_y$ (resp. $L_x < L_y$) **then**
3:     sort POIs in P_{remain} in ascending order using x-coordinate (resp. y-coordinate)
4: **while** P_{remain} is not empty **do**
5:     $u \leftarrow$ a POI in P_{remain} with the smallest x-coordinate / y-coordinate
6:     $P_{remain} \leftarrow P_{remain} - \{u\}, P'_{remain} \leftarrow P_{remain}$
7:     calculate the exact shortest paths passing on C from u to each POI in P'_{remain} simultaneously using algorithm *FastFly*
8:     **for** each POI $v \in P'_{remain}$ **do**
9:         $key \leftarrow \langle u, v\rangle$, $value \leftarrow \Pi^*(u, v|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
10:     sort POIs in P'_{remain} in ascending order using the exact distance on C between u and each $v \in P_{remain}$, i.e., $|\Pi^*(u, v|C)|$
11:     **for** each sorted POI $v \in P'_{remain}$ such that $|\Pi^*(u, v|C)| \leq \epsilon L$ **do**
12:         $P_{remain} \leftarrow P_{remain} - \{v\}, P'_{remain} \leftarrow P'_{remain} - \{v\}, P_{dest}(v) \leftarrow \emptyset$
13:         **for** each POI $w \in P'_{remain}$ **do**
14:             **if** $d_E(v, w) > \frac{2}{\epsilon} \cdot |\Pi^*(u, v|C)|$ and $v \notin M_{end}.key$ **then**
15:                 $key \leftarrow v$, $value \leftarrow u$, $M_{end} \leftarrow M_{end} \cup \{key, value\}$
16:             **else if** $d_E(v, w) \leq \frac{2}{\epsilon} \cdot |\Pi^*(u, v|C)|$ **then**
17:                 $P_{dest}(v) \leftarrow P_{dest}(v) \cup \{w\}$
18:         calculate the exact shortest paths passing on C from v to each POI in $P_{dest}(v)$ simultaneously using algorithm *FastFly*
19:         **for** each POI $w \in P_{dest}(v)$ **do**
20:             $key \leftarrow \langle v, w\rangle$, $value \leftarrow \Pi^*(v, w|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
21: **return** M_{path} and M_{end}

the exact shortest paths passing on C from a to b, c, d, e (in purple lines) using algorithm *FastFly*, and store each POIs pair as a key and the corresponding path as a value in M_{path}.

(ii) *Shortest paths approximation* (lines 10-20): In Figure 3 (c), b is the POI in P'_{remain} closest to a, c is the POI in P'_{remain} second closest to a, so the following order is $b, c, \ldots$. There are two cases:

- *Approximation loop start* (lines 11-20): In Figure 3 (c), we first select a's closest POI in P'_{remain}, i.e., b, since $d_E(a, b) \leq \epsilon L$, it means a and b are not far away, we start the approximation loop, delete b from P_{remain} and P'_{remain}, so $P_{remain} = P'_{remain} = \{c, d, e\}$. There are three steps:
  - *Far away POIs selection* (lines 13-15): In Figure 3 (c), $d_E(b, d) > \frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)|$, $d_E(b, e) > \frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)|$, $d \notin M_{end}.key$ and $e \notin M_{end}.key$, it means d and e are far away from b, we can use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ that we have already calculated before to approximate $\Pi^*(b, d|C)$, and use $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$ that we have already calculated before to approximate $\Pi^*(b, e|C)$, so we get $\Pi(b, d|C)$ by concatenating $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, and get $\Pi(b, e|C)$ by concatenating $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$, we store b as key and a as value in M_{end}.
  - *Close POIs selection* (line 13 and lines 16-17): In Figure 3 (c), $d_E(b, c) \leq \frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)|$, it means c is close to b, so we cannot use any existing exact shortest paths passing on C to approximate $\Pi^*(b, c|C)$, then we store c into $P_{dest}(b)$.
  - *Selected exact shortest paths calculation* (lines 18-20): In Figure 3 (c), when we have processed all POIs in P'_{remain} with b as a source, we have $P_{dest}(b) = \{c\}$, we use algorithm *FastFly* to calculate the exact shortest path passing on C between b and c, i.e., $\Pi^*(b, c|C)$ (in green line), and store

$\langle b, c \rangle$ as a key and $\Pi^*(b, c|C)$ as a value in M_{path}. Note that we can terminate algorithm *FastFly* earlier since we just need to visit POIs that are close to b, and we do not need to visit d and e.

- *Approximation loop end* (line 11): In Figure 3 (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 $d_E(a, c) > \epsilon L$, it means a and c are far away, and it is unlikely to have a POI m that satisfies $d_E(c, m) > \frac{2}{\epsilon} \cdot |\Pi^*(a, c|C)|$, we end the approximation loop and terminate the iteration.

(3) *Shortest paths calculation iteration* (lines 4-20): In Figure 3 (d), we repeat the iteration, and calculate the exact shortest paths passing on C with c as a source (in orange lines).

4.3.2 ***Shortest Path Query Phase.*** Given a pair of POIs s and t in P, M_{path} and M_{end}, *RC-Oracle* efficiently answers the associated shortest path $\Pi(s, t|C)$ passing on C, which is a $(1+\epsilon)$-approximated exact shortest path of $\Pi^*(s, t|C)$ in $O(1)$ time. Given a pair of POIs s and t, there are two cases (s and t are interchangeable, i.e., $\langle s, t \rangle = \langle t, s \rangle$):

(1) *Exact shortest path retrieval*: If $\langle s, t \rangle \in M_{path}.key$, we retrieve $\Pi^*(s, t|C)$ using $\langle s, t \rangle$ in $O(1)$ time (in Figures 3 (d) and (e), given a and d, since $\langle a, d \rangle \in M_{path}.key$, we retrieve $\Pi^*(a, d|C)$).

(2) *Approximate shortest path retrieval*: If $\langle s, t \rangle \notin M_{path}.key$, it means $\Pi^*(s, t|C)$ is approximated by two exact shortest paths passing on C in M_{path}, and (i) either s or t is a key in M_{end}, or (ii) both s and t are keys in M_{end}. Without loss of generality, suppose that (i) s exists in M_{end} if either s or t is a key in M_{end}, or (ii) the x- (resp. y-) coordinate of s is smaller than t when $L_x \geq L_y$ (resp. $L_x < L_y$) if both s and t are keys in M_{end}. For both of two cases, we retrieve the value s' using the key s from M_{end} in $O(1)$ 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 $O(1)$ time, and use $\Pi^*(s, s'|C)$ and $\Pi^*(s', t|C)$ to approximate $\Pi^*(s, t|C)$ ((i) in Figures 3 (d) and (e), given b and e, since $\langle b, e \rangle \notin M_{path}.key$, b is a key in M_{end}, so we retrieve the value a using the key b in M_{end}, then in M_{path}, we use $\langle b, a \rangle$ and $\langle a, e \rangle$ to retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$, for approximating $\Pi^*(b, e|C)$, or (ii) in Figure 3 (d), (e) and (f), given b and d, since $\langle b, d \rangle \notin M_{path}.key$, b and d are both keys in M_{end}, and $L_x < L_y$, we use the key b with a smaller y-coordinate value to retrieve the value a in M_{end}, then in M_{path}, we use $\langle b, a \rangle$ and $\langle a, d \rangle$ to retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, for approximating $\Pi^*(b, d|C)$).

4.4 Adaption to *RC-Oracle-A2A*

We can adapt *RC-Oracle* (that answers the P2P query) to be *RC-Oracle-A2A* (that answers the A2A query) on a point cloud, by simply creating POIs that have the same coordinate values as all points in the point cloud. We just need to pre-compute the exact shortest paths passing on the point cloud between *some* selected pairs of points on the point cloud (not *all* pairs of points on the point cloud), so *RC-Oracle-A2A* also has a small oracle construction time, small oracle size and small shortest path query time.

4.5 Proximity Query Algorithms

Given a point cloud C, a set of n' objects O on C, a query object $q \in O$, a user parameter k and a range value r, we can answer other proximity queries, i.e., the *kNN* and range query using *RC-Oracle* and *RC-Oracle-A2A*. In the P2P query (resp. A2A query), these objects are POIs in P (resp. any points on C). A naive algorithm is to perform a linear scan using the shortest path query results. We propose an efficient algorithm for it. Intuitively, when constructing *RC-Oracle* or *RC-Oracle-A2A*, we have used the *SSAD* algorithm to calculate the shortest paths passing on C with q as a source and sorted these paths in ascending order based on their distance in M_{path} (we can use an additional table to store these sorted paths). For these paths, we do not need to perform linear scans over all of them in proximity queries for time-saving. Since the proximity query algorithms for *RC-Oracle* and *RC-Oracle-A2A* are similar, we use *RC-Oracle* as an example for illustration.

Detail and example: There are two cases. For both two cases, we can then return the corresponding *kNN* and range query results.

(1) *Approximation needed in direct result return*: If $q \in M_{end}.key$, it means we need to use two paths in M_{path} to approximate some other paths in a later stage, we retrieve the value q' using the key q from M_{end} (in Figures 3 (d) and (e), $b \in M_{end}.key$, we retrieve the value a using the key b from M_{end}), there are two more cases:

(i) *Linear scan*: For objects with a smaller x- (resp. y-) coordinate compared with q' when $L_x \geq L_y$ (resp. $L_x < L_y$), we perform a linear scan on the shortest path query result between q and these objects (in Figures 3 (d) and (e), since $L_x < L_y$, there is no POI with a smaller y-coordinate compared with a).

(ii) *Direct result return*: For objects (not including q) with a larger x- (resp. y-) coordinate compared with q' when $L_x \geq L_y$ (resp. $L_x < L_y$) (in Figures 3 (d) and (e), since $L_x < L_y$, the POIs with a larger y-coordinate compared with a are $\{c, d, e\}$), there are further more two cases:

- *Direct result return without approximation*: If the endpoint pairs of q and these objects are keys in M_{path}, it means that we have used the *SSAD* algorithm with q as a source for such objects and we have already sorted such paths in order, so we can directly return the corresponding result (in Figures 3 (d) and (e), since $\langle b, c\rangle \in M_{path}.key$, we know that $|\Pi^*(b, c|C)|$ is sorted in order, but since there is only one distance, it does not matter whether itself is sorted in order or not).
- *Direct result return with approximation*: If the endpoint pairs of q and these objects are not keys in M_{path}, it means that we have used the *SSAD* algorithm with q' as a source for such objects and we have already sort such paths in order, we just need to use the exact distance between q' and these objects plus $|\Pi^*(q', q|C)|$, to get the approximate distance between q and o in sorted order, so we can directly return the corresponding result (in Figures 3 (d) and (e), since $\langle b, d\rangle \notin M_{path}.key$ and $\langle b, e\rangle \notin M_{path}.key$, we know that $|\Pi^*(a, d|C)|$ and $|\Pi^*(a, e|C)|$ are sorted in order, so $|\Pi(b, d|C)|$ and $|\Pi(b, e|C)|$ are also sorted in order).

(2) *Approximation not needed in direct result return*: If $q \notin M_{end}.key$, it means we do need to use two paths in M_{path} to approximate all other paths in a later stage (in Figures 3 (d) and (e), $c \notin M_{end}.key$), there are two more cases:

(i) *Linear scan*: For objects with a smaller x- (resp. y-) coordinate compared with q when $L_x \geq L_y$ (resp. $L_x < L_y$), we perform a linear scan on the shortest path query result between q and these objects (in Figures 3 (d) and (e), since $L_x < L_y$, the POIs with a smaller y-coordinate compared with c are $\{a, b\}$, we perform a linear scan on the shortest path query result between c and $\{a, b\}$).

(ii) *Direct result return*: For objects with a larger x- (resp. y-) coordinate compared with q when $L_x \geq L_y$ (resp. $L_x < L_y$), we have used the *SSAD* algorithm with q as a source for such objects and we have already sorted such paths in order, so we can directly return the corresponding result (in Figures 3 (d) and (e), since $L_x < L_y$, the POIs with a larger y-coordinate compared with c are $\{d, e\}$, we know that $|\Pi^*(c, d|C)|$ and $|\Pi^*(c, e|C)|$ are sorted in order).

4.6 Theoretical Analysis

4.6.1 ***Algorithm FastFly, RC-Oracle and RC-Oracle-A2A***. The analysis of Algorithm *FastFly* is in Theorem 4.1, and the analysis of *RC-Oracle* and *RC-Oracle-A2A* are Theorem 4.2.

THEOREM 4.1. *The shortest path query time and memory consumption of algorithm FastFly are $O(N \log N)$ and $O(N)$. Algorithm FastFly returns the exact shortest path passing on the point cloud.*

PROOF. Since algorithm *FastFly* is a Dijkstra algorithm and there are total N points, we obtain the shortest path query time and memory consumption. Since Dijkstra algorithm is guaranteed to return the exact shortest path, algorithm *FastFly* returns the exact shortest path passing on the point cloud. □

Theorem 4.2. *The oracle construction time, oracle size and shortest path query time of (1) RC-Oracle are* $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$, $O(1)$ *and (2) RC-Oracle-A2A are* $O(\frac{N \log N}{\epsilon})$, $O(\frac{N}{\epsilon})$, $O(1)$, *respectively. RC-Oracle always have* $|\Pi(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ *for any pairs of POIs* s *and* t *in* P, *and RC-Oracle-A2A always have* $|\Pi_{\text{RC-Oracle-A2A}}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ *for any pairs of points* s *and* t *on* C, *where* $\Pi_{\text{RC-Oracle-A2A}}(s,t|C)$ *is the shortest path of RC-Oracle-A2A passing on* C *between* s *and* t.

Proof. We give the proof for *RC-Oracle* as follows.

Firstly, we show the *oracle construction time*. (1) In *POIs sort* step, it needs $O(n \log n)$ time. Since there are n POIs, and we use the quick sort for sorting. (2) In *shortest paths calculation* step, it needs $O(\frac{N \log N}{\epsilon} + n)$ time. (i) It needs to use $O(\frac{1}{\epsilon})$ POIs as a source to run algorithm *FastFly* for the exact shortest paths calculation according to standard packing property [27], and each algorithm *FastFly* needs $O(N \log N)$ time. (ii) For other $O(n)$ POIs that there is no need to use them as a source to run algorithm *FastFly*, we just calculate the Euclidean distance from these POIs to other POIs in $O(1)$ time for the shortest paths approximation. (3) So the oracle construction time is $O(\frac{N \log N}{\epsilon} + n \log n)$.

Secondly, we show the *oracle size*. (1) For M_{end}, its size is $O(n)$ since there are n POIs. (2) For M_{path}, its size is $O(\frac{n}{\epsilon})$. We store (i) $O(\frac{n}{\epsilon})$ exact shortest paths passing on C from $O(\frac{1}{\epsilon})$ POIs (that uses algorithm *FastFly* as a source and cover all other POIs) to other $O(n)$ POIs, and (ii) $O(n)$ exact shortest paths passing on C from $O(n)$ POIs (that uses algorithm *FastFly* as a source and cover only some of POIs) to other $O(1)$ POIs. (3) So the oracle size is $O(\frac{n}{\epsilon})$.

Thirdly, we show the *shortest path query time*. (1) If $\Pi^*(s,t|C) \in M_{path}$, the shortest path query time is $O(1)$. (2) If $\Pi^*(s,t|C) \notin M_{path}$, we need to retrieve s' from M_{end} using s in $O(1)$ 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 $O(1)$ time, so the shortest path query time is still $O(1)$. Thus, the shortest path query time of *RC-Oracle* is $O(1)$.

Fourthly, we show the *error bound*. 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}. Suppose that u is a POI close to s, such that $\Pi(s,t|C)$ is calculated by concatenating $\Pi^*(s,u|C)$ and $\Pi^*(u,t|C)$. This means that $d_E(s,t) > \frac{2}{\epsilon} \cdot \Pi^*(u,s|C)$. So we have $|\Pi^*(s,u|C)| + |\Pi^*(u,t|C)| < |\Pi^*(s,u|C)| + |\Pi^*(u,s|C)| + |\Pi^*(s,t|C)| = |\Pi^*(s,t|C)| + 2 \cdot |\Pi^*(u,s|C)| < |\Pi^*(s,t|C)| + \epsilon \cdot d_E(s,t) \leq |\Pi^*(s,t|C)| + \epsilon \cdot |\Pi^*(s,t|C)| = (1+\epsilon)|\Pi^*(s,t|C)|$. The first inequality is due to triangle inequality. The second equation is because $|\Pi^*(u,s|C)| = |\Pi^*(s,u|C)|$. The third inequality is because we have $d_E(s,t) > \frac{2}{\epsilon} \cdot \Pi^*(u,s|C)$. The fourth inequality is because the Euclidean distance between two points is no larger than the distance of the shortest path passing on the point cloud between the same two points.

We give the proof for *RC-Oracle-A2A* as follows. We need to change (1) n to N in the oracle construction time and oracle size, and (2) any pairs of POIs in P to any pairs of points on C in the error bound. □

4.6.2 *The shortest path passing on a point cloud and the shortest surface or network path passing on a TIN*.

We show the relationship of $|\Pi^*(s,t|C)|$ with $|\Pi_N(s,t|T)|$ and $|\Pi^*(s,t|T)|$ in Lemma 4.3.

Lemma 4.3. *Given a pair of points* s *and* t *on* C, *we have (1)* $|\Pi^*(s,t|C)| \leq |\Pi_N(s,t|T)|$ *and (2)* $|\Pi^*(s,t|C)| \leq k \cdot |\Pi^*(s,t|T)|$, *where* $k = \max\{\frac{2}{\sin\theta}, \frac{1}{\sin\theta\cos\theta}\}$.

Proof. (1) In Figure 2 (a), given a green point q on C, it can connect with one of its 8 neighbor points (7 blue points and 1 red point s). In Figure 2 (b), given a green vertex q on T, it can only connect with one of its 6 blue neighbor vertices. So $|\Pi^*(s,t|C)| \leq |\Pi_N(s,t|T)|$. (2) We let $\Pi_E(s,t|T)$ be the shortest path passing on the edges of T (where these edges belong to the faces that $\Pi^*(s,t|T)$ passes) between s and t. According to left hand side equation in Lemma 2 of [31], we have $|\Pi_E(s,t|T)| \leq$

$k \cdot |\Pi^*(s,t|T)|$. Since $\Pi_N(s,t|T)$ considers all the edges on T, $|\Pi_N(s,t|T)| \leq |\Pi_E(s,t|T)|$. Thus, we finish the proof by combining these inequalities. □

4.6.3 ***Proximity query algorithms.*** We provide analysis on the proximity query algorithms using *RC-Oracle* and *RC-Oracle-A2A*. For the *kNN* and range query, both of them return a set of objects. Given a query object q, we let v_f (resp. v'_f) be the furthest object to q among the returned objects calculated using the exact distance on C (resp. the approximated distance on C returned by *RC-Oracle*). In Figure 1 (a), suppose that the exact k nearest POIs ($k = 2$) of a is c, d. And d is the furthest POI to a in these two POIs, i.e., $v_f = d$. Suppose that our *kNN* query algorithm finds the k nearest POIs ($k = 2$) of a is b, c. And b is the furthest POI to a in these two POIs, i.e., $v'_f = b$. We define the error rate of the *kNN* and range query to be $\frac{|\Pi^*(q,v'_f|C)|}{|\Pi^*(q,v_f|C)|}$, which is a real number no smaller than 1. In Figure 1 (a), the error rate is $\frac{|\Pi^*(a,b|C)|}{|\Pi^*(a,d|C)|}$. Then, we show the query time and error rate of *kNN* and range query using *RC-Oracle* and *RC-Oracle-A2A* in Theorem 4.4.

Theorem 4.4. *The query time and error rate of both the kNN and range query by using RC-Oracle and RC-Oracle-A2A are both $O(n')$ and $1+\epsilon$, respectively.*

Proof Sketch. The *query time* is due to the usages of the shortest path query phase of *RC-Oracle* and *RC-Oracle-A2A* for n' times in the worst case. The *error rate* is due to its definition and the error of *RC-Oracle* and *RC-Oracle-A2A*. The detailed proof appears in our technical report [61]. □

5 EMPIRICAL STUDIES

5.1 Experimental Setup

We conducted our experiments on a Linux machine with 2.2 GHz CPU and 512GB memory. All algorithms were implemented in C++. Our experimental setup generally follows the setups in the literature [30, 31, 39, 55, 56]. We conducted experiments with point clouds and *TIN*s as input, separately.

Datasets: (1) Point cloud datasets: We conducted our experiment based on 34 real point cloud datasets in Table 3, where the subscript p means a point cloud. For BH_p and EP_p datasets, they are represented as a point cloud with 8km × 6km covered region. For GF_p, LM_p and RM_p, we first obtained the satellite map from Google Earth [2] with 8km × 6km covered region, and then used Blender [1] to generate the point cloud. These five original datasets have a resolution of 10m × 10m [20, 39, 51, 55, 56]. We extracted 500 POIs using OpenStreetMap [55, 56] for these datasets in the P2P query. For small-version datasets, we use the same region of the original datasets with a (lower) resolution of 70m × 70m and the dataset generation procedure in [39, 55, 56] to generate them. This procedure can be found in our technical report [61]. In addition, we have six sets of multi-resolution datasets with different numbers of points generated using the original and small-version datasets with the same procedure. (2) *TIN* datasets: Based on the 34 point cloud datasets, we triangulate [46] them and generate another 34 *TIN* datasets, and use t as the subscript. For example, BH_t means a *TIN* dataset generated using the BH_p point cloud dataset.

Algorithms: (1) Algorithms that support the shortest path query (and also other proximity queries) on a point cloud (i.e., algorithms for solving the problem studied in this paper): We adapted existing algorithms, originally designed for the problem on *TIN*s, for our problem on point clouds by performing the triangulation approach on the point cloud to obtain a *TIN* [46] (i.e., we store the *TIN* as a data structure in the memory and clear the given point cloud from the memory) so that the existing algorithm could be used. Their algorithm names are appended by "*-Adapt*". We have four on-the-fly algorithms, i.e., (i) *CH-Adapt* [15], (ii) *Kaul-Adapt* [30], (iii) *Dijk-Adapt* [31], and (iv) *FastFly*: our algorithm. We have four oracles, i.e., (v) *SE-Oracle-Adapt*: the best-known

Table 3. Point cloud datasets

Name	$\lvert N \rvert$
Original dataset	
BearHead (BH_p) [5, 55, 56]	0.5M
EaglePeak (EP_p) [5, 55, 56]	0.5M
GunnisonForest (GF_p) [7]	0.5M
LaramieMount (LM_p) [8]	0.5M
RobinsonMount (RM_p) [3]	0.5M
Small-version dataset	
BH_p*-small*	10k
EP_p*-small*	10k
GF_p*-small*	10k
LM_p*-small*	10k
RM_p*-small*	10k
Multi-resolution dataset	
BH_p multi-resolution	1M, 1.5M, 2M, 2.5M
EP_p multi-resolution	1M, 1.5M, 2M, 2.5M
GF_p multi-resolution	1M, 1.5M, 2M, 2.5M
LM_p multi-resolution	1M, 1.5M, 2M, 2.5M
RM_p multi-resolution	1M, 1.5M, 2M, 2.5M
EP_p*-small* multi-resolution	20k, 30k, 40k, 50k

oracle [55, 56] for the P2P query on a point cloud, (vi) *EAR-Oracle-Adapt*: the best-known oracle [28] for the A2A query on a point cloud, (vii) *RC-Oracle-Naive*: the naive version of our oracle *RC-Oracle* without shortest paths approximation step, and (viii) *RC-Oracle*: our oracle.

(2) Algorithms that support the shortest path query (and also other proximity queries) on a *TIN* (i.e., algorithms for solving the problem studied by previous studies [28, 55, 56]): Similarly, we have four on-the-fly algorithms, i.e., (i) *CH* [15], (ii) *Kaul* [30], (iii) *Dijk* [31], (iv) *FastFly-Adapt*: our adapted algorithm (for the queries on a *TIN*) that calculates the shortest path passing on a conceptual graph of a *TIN*, where the vertices of this conceptual graph are formed by the vertices of the given *TIN*, and the edges of this graph are formed by adding edges between each vertex and its 8 neighbor vertices (this conceptual graph is similar to the one in Figure 2 (c), we store it as a data structure in the memory and clear the given *TIN* from the memory). We have four oracles, i.e., (v) *SE-Oracle* [55, 56], (vi) *EAR-Oracle* [28], (vii) *RC-Oracle-Naive-Adapt*: the adapted naive version of our oracle without shortest paths approximation step that calculates the shortest path passing on a conceptual graph of a *TIN*, and (viii) *RC-Oracle-Adapt*: our adapted oracle that calculates the shortest path passing on a conceptual graph of a *TIN*.

Query Generation: We conducted all proximity queries, i.e., (1) shortest path query, (2) all objects *kNN* query, and (3) all objects range query. (1) For the shortest path query, we issued 100 query instances where for each instance, we randomly chose two points (i) in P for the P2P query on a point cloud or a *TIN*, or (ii) on the point cloud (resp. *TIN*) for the A2A query on a point cloud (resp. the AR2AR query on a *TIN*), one as a source and the other as a destination. The average, minimum and maximum results were reported. In the experimental result figures, the vertical bar and the points mean the minimum, maximum and average results. (2 & 3) For all objects *kNN* query and range query, we perform the proximity query algorithm for *RC-Oracle* in Section 4.5 and a linear scan for other baselines (as described in [56]) using all objects as query objects. In the P2P query on a point cloud or a *TIN*, these objects are POIs in P. In the A2A query on a point cloud (resp. the AR2AR query on a *TIN*), we randomly select 2500 points on the point cloud (resp. *TIN*) as objects. Since we perform linear scans or use the sorted distance stored in M_{path} for proximity query algorithms, the value of k and r will not affect their query time, we set $k = 3$ and $r = 1$km.

Factors and Measurements: We studied three factors for the P2P query, namely (1) ϵ (i.e., the error parameter), (2) n (i.e., the number of POIs), and (3) N (i.e., the number of points in a point cloud dataset or the number of vertices in a *TIN* dataset). We studied one factor ϵ for the A2A query. In addition, we used nine measurements to evaluate the algorithm performance, namely (1) *oracle construction time*, (2) *memory consumption* (i.e., the space consumption when running the algorithm), (3) *oracle size*, (4) *query time* (i.e., the shortest path query time), (5) *kNN query time* (i.e., all objects kNN query time), (6) *range query time* (i.e., all objects range query time), (7) *distance error* (i.e., the error of the distance returned by the algorithm compared with the exact distance), (8) *kNN query error* (i.e., the error rate of the *kNN* query defined in Section 4.6.3), and (9) *range query error* (i.e., the error rate of the range query defined in Section 4.6.3).

5.2 Experimental Results for *TIN*s

We first study proximity queries on *TIN*s (studied by previous studies [28, 55, 56]) to justify why our proximity queries on *point clouds* are useful in practice. We have the following settings. (1) The distance of the path calculated by *CH* is used for distance error calculation since the path is the exact shortest surface path passing on the *TIN*. (2) *SE-Oracle*, *EAR-Oracle* and *RC-Oracle-Naive-Adapt* are not feasible on large-version datasets due to their expensive oracle construction time (more than 24 hours), so we (i) compared *SE-Oracle*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt*, *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt* on small-version datasets (with default 50 POIs for the P2P query), and (ii) compared *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt* on large-version datasets (with default 500 POIs for the P2P query). (3) The transformation time from a *TIN* to the conceptual graph of *FastFly-Adapt*, *RC-Oracle-Naive-Adapt* and *RC-Oracle-Adapt* is only counted once (i) in the shortest path query time, the *kNN* and range query time for *FastFly-Adapt*, and (ii) in the oracle construction time for *RC-Oracle-Adapt* and *RC-Oracle-Naive-Adapt*. (4) The transformation time from a *TIN* to the conceptual graph of *Dijk* is also only counted once in its shortest path query time, the *kNN* and range query time.

5.2.1 ***Baseline comparisons***. We study the effect of ϵ and n for the P2P query on a *TIN* in this subsection. We study the effect of N for the P2P query, and the comparisons for the AR2AR query on a *TIN* in our technical report [61].

Effect of ϵ for the P2P query on a *TIN*. In Figure 6, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_p*-small* dataset by setting N to be 10k and n to be 50 for baseline comparisons. Although a *TIN* is given as input, *RC-Oracle-Adapt* performs better than *SE-Oracle*, *EAR-Oracle* and *RC-Oracle-Naive-Adapt* in terms of the oracle construction time, oracle size and shortest path query time. The shortest path query time of *FastFly-Adapt* is 100 times smaller than that of *CH* (although *FastFly-Adapt* needs to construct a conceptual graph from the given *TIN*, and there is no other additional steps for *CH*), since the query region of the path calculated by *FastFly-Adapt* is smaller than that of *CH*. The distance error of *FastFly-Adapt* (i.e., 0.008) is very small compared with that of *CH* (i.e., without error), and much much smaller than that of *Dijk* (i.e., 0.1). This motivates us to conduct experiments on point clouds. The *kNN* query error and range query error are all equal to 0 (due to the small distance error), so their results are omitted.

Effect of n for the P2P query on a *TIN*. In Figure 7, we tested 5 values of n from {50, 100, 150, 200, 250} on EP_t dataset by setting N to be 10k and ϵ to be 0.1 for baseline comparisons. In Figure 7 (a), when n increases, the construction time of all oracles increases. In Figure 7 (b), when n increases, the memory consumption of *RC-Oracle-Adapt* exceeds that of *Dijk* and *FastFly-Adapt*. This is because (1) *RC-Oracle-Adapt* is an oracle which is affected by n, it needs more memory consumption during the oracle construction phase to calculate more shortest paths among these POIs when n increases, but (2) *Dijk* and *FastFly-Adapt* are on-the-fly algorithms which are not affected by n, their memory consumption only measure the space consumption for calculating one shortest path.

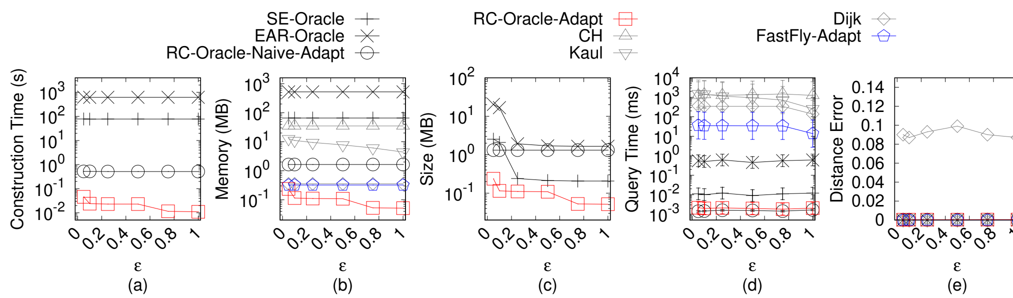

Fig. 6. Baseline comparisons (effect of ϵ on BH_t-*small TIN* dataset for the P2P query)

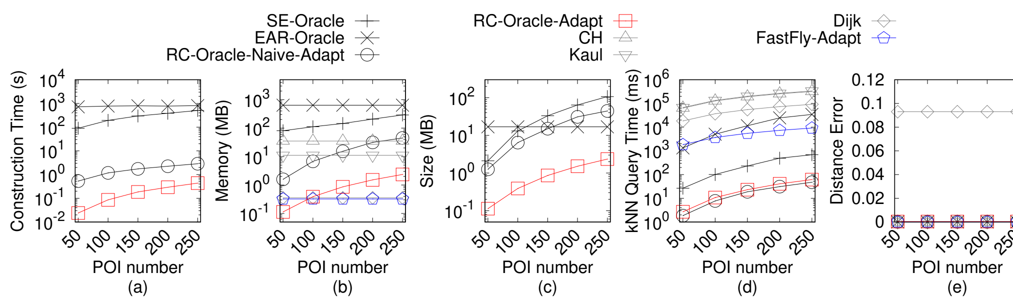

Fig. 7. Baseline comparisons (effect of n on EP_t-*small TIN* dataset for the P2P query)

5.3 Experimental Results for Point Clouds

Now, we understand the effectiveness of proximity queries on *point clouds*. In this section, we then study proximity queries on *point clouds* using the algorithms in Table 2. We have the following setting. (1) The distance of the path calculated by *FastFly* is used for distance error calculation since the path is the exact shortest path passing on the point cloud. (2) *SE-Oracle-Adapt*, *EAR-Oracle-Adapt* and *RC-Oracle-Naive* are not feasible on large-version datasets due to their expensive oracle construction time (more than 24 hours), so we (i) compared *SE-Oracle-Adapt*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive*, *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on small-version datasets (with default 50 POIs for the P2P query), and (ii) compared *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on large-version datasets (with default 500 POIs for the P2P query). (3) Since algorithm *FastFly* uses the Dijkstra algorithm on the conceptual graph of a point cloud, it is the same as the shortest path algorithm on a general graph (constructed by the given point cloud), we do not (and there is no need to) compare them in the experiment. But, there is no existing work discussing how to build a conceptual graph from a point cloud. We fill this gap by proposing algorithm *FastFly*. (4) The transformation time from a point cloud to the conceptual graph of *FastFly*, *RC-Oracle-Naive* and *RC-Oracle* is only counted once (i) in the shortest path query time, the *kNN* and range query time for *FastFly*, and (ii) in the oracle construction time for *RC-Oracle* and *RC-Oracle-Naive*. (5) The transformation time from a point cloud to a *TIN* is also only counted once (i) in the shortest path query time, the *kNN* and range query time for *CH-Adapt* and *Kaul-Adapt*, and (ii) in the oracle construction time for *SE-Oracle-Adapt* and *EAR-Oracle-Adapt*. (6) The transformation time from a point cloud to a *TIN*, and then to the conceptual graph of *Dijk-Adapt* is also only counted once in its shortest path query time, the *kNN* and range query time.

5.3.1 ***Baseline comparisons***. We study the effect of ϵ, n and N for the P2P query on a point cloud, and the comparisons for the A2A query on a point cloud in this subsection.

Effect of ϵ for the P2P query on a point cloud. In Figure 8, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on EP_p*-small* dataset by setting N to be 10k and n to be 50 for baseline comparisons. (1) For *RC-Oracle* and the best-known oracle *SE-Oracle-Adapt*, (i) the oracle construction time and memory consumption, (ii) oracle size, and (iii) shortest path query time of *RC-Oracle* are all smaller than *SE-Oracle-Adapt*, since (i) *SE-Oracle-Adapt* has the *bad criterion for algorithm earlier termination* drawback, it cannot terminate the *SSAD* algorithm earlier, so it requires more time and memory, (ii) *RC-Oracle* can terminate the *SSAD* algorithm earlier and store fewer paths, (iii) *RC-Oracle*'s shortest path query time is $O(1)$, while the time is $O(h^2)$ for *SE-Oracle-Adapt*. (2) *RC-Oracle* performs better than other on-the-fly algorithms in terms of the shortest path query time since it is an oracle. (3) Algorithm *FastFly* performs better than other on-the-fly algorithms in terms of the shortest path query time since it calculates the shortest path passing on a point cloud. (4) In Figures 8 (a) & (b), regarding the oracle construction time and memory consumption, the variation of ϵ (i) has a large effect on *RC-Oracle*, but due to the log scale used in the experimental figures, the effect is not obvious (e.g., the oracle construction time and memory consumption of *RC-Oracle* with $\epsilon = 1$ are both up to 5 times smaller than that of the case when $\epsilon = 0.05$), (ii) has a small effect on *SE-Oracle-Adapt* and *EAR-Oracle-Adapt*, because even when ϵ is large, they cannot terminate the *SSAD* algorithm earlier for most of the cases due to their *bad criterion for algorithm earlier termination* drawback, and (iii) has no effect on *RC-Oracle-Naive* since it is independent of ϵ. (5) The *kNN* and range query time of *RC-Oracle* are much smaller than the on-the-fly algorithms. (6) The distance error of *RC-Oracle* is close to 0.

Effect of n for the P2P query on a point cloud. In Figure 9, we tested 5 values of n from $\{500, 1000, 1500, 2000, 2500\}$ on GF_p dataset by setting N to be 0.5M and ϵ to be 0.25 for baseline comparisons. Since *RC-Oracle* is an oracle, its shortest path query time is smaller than on-the-fly algorithms.

Effect of N (scalability test) for the P2P query on a point cloud. In Figure 10, we tested 5 values of N from $\{0.5M, 1M, 1.5M, 2M, 2.5M\}$ on LM_p dataset by setting n to be 500 and ϵ to be 0.25 for baseline comparisons. The oracle construction time of *RC-Oracle* is only 200s $\approx$ 3.2 min for a point cloud with 2.5M points and 500 POIs, this shows the scalable of *RC-Oracle*. The range query time of *RC-Oracle* is the smallest.

A2A query on a point cloud. In Figure 11, we tested the A2A query by varying ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ and setting N to be 10k on a small-version of EP_p dataset. We adapt *SE-Oracle-Adapt* (resp. *RC-Oracle-Naive*) to be *SE-Oracle-Adapt-A2A* (resp. *RC-Oracle-Naive-A2A*) in a similar way to *RC-Oracle-A2A*. Although *EAR-Oracle-Adapt* is regarded as the best-known oracle in the A2A query on a point cloud, *RC-Oracle-A2A* still performs more efficiently than it due to the *bad criterion for algorithm earlier termination* drawback of *EAR-Oracle-Adapt*.

5.3.2 ***Ablation study for the P2P query on a point cloud***. We denote *SE-Oracle-FastFly-Adapt* (resp. *EAR-Oracle-FastFly-Adapt*) to be another adapted oracle of *SE-Oracle-Adapt* (resp. *EAR-Oracle-Adapt*) that uses algorithm *FastFly* to directly calculate the shortest path passing on a point cloud without constructing a *TIN*. In Figure 12, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on RM_p dataset by setting N to be 0.5M and n to be 500 for ablation study among *SE-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt* and *RC-Oracle*, such that they only differ by the oracle construction. The oracle construction time, oracle size and shortest path query time of *RC-Oracle* perform better than the two baselines.

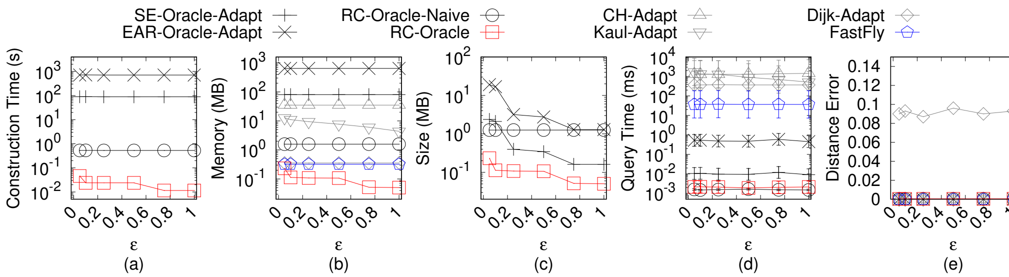

Fig. 8. Baseline comparisons (effect of ϵ on EP_p*-small* point cloud dataset for the P2P query)

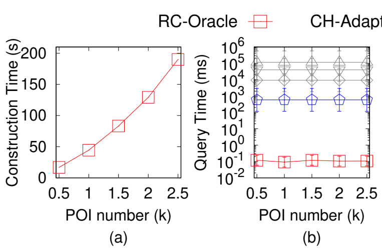

Fig. 9. Baseline comparisons (effect of n on GF_p point cloud dataset for the P2P query)

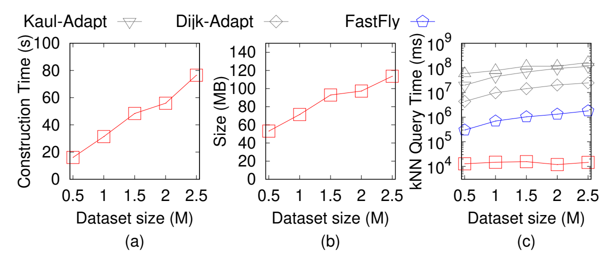

Fig. 10. Baseline comparisons (effect of N on LM_p point cloud dataset for the P2P query)

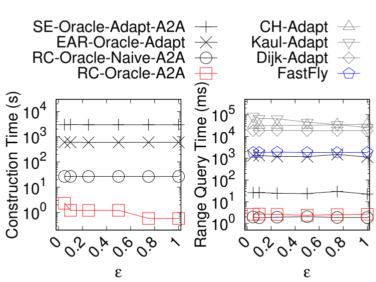

Fig. 11. Baseline comparisons on EP_p point cloud dataset for the A2A query

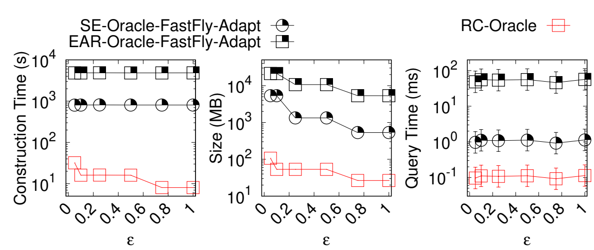

Fig. 12. Ablation study on RM_p point cloud dataset for the P2P query

5.3.3 ***Comparisons with other proximity queries oracle and algorithm on a point cloud***.

We compared *RC-Oracle* using our efficient proximity query algorithm with *SU-Oracle-Adapt* (i.e., the oracle designed for the *kNN* query) and *RC-Oracle* using the naive proximity query algorithm in our technical report [61]. For a point cloud with 2.5M points and 500 objects, the *kNN* query time of *RC-Oracle* using our efficient proximity query algorithm is 12.5s, but the time is 1520s $\approx$ 25 mins for *SU-Oracle-Adapt*, and 25s for *RC-Oracle* using the naive proximity query algorithm (since the shortest path query time of *RC-Oracle* is $O(1)$, and we do not need to perform linear scans over all the objects in our efficient proximity query algorithm).

5.3.4 ***Case study.*** We conducted a case study on an evacuation simulation in Mount Rainier [47] due to the frequent heavy snowfall [48]. The blizzard wreaking havoc across the USA in December 2022 killed more than 60 lives [10] and one may be dead due to asphyxiation [34] if s/he gets buried in the snow. In the case of snowfall, staffs will evacuate tourists in the mountain to the closest hotels immediately for tourists' safety. The time of a human being buried in the snow is expected to be 2.4 hours[1]. The average distance between the viewing platforms and hotels in Mount Rainier National Park is 11.2km [6], and the average human walking speed is 5.1 km/h [9], so the evacuation (i.e., the time of human's walking from the viewing platform to hotels) can be finished in 2.2 ($= \frac{11.2\text{km}}{5.1\text{km/h}}$) hours. Thus, the calculation of the shortest paths is expected to be finished within 12 min ($= 2.4 - 2.2$ hours). Our experimental results show that for a point cloud with 2.5M points and 500 POIs (250 viewing platforms and 250 hotels), (1) the oracle construction time for (i) *RC-Oracle* is 200s $\approx$ 3.2 min and (ii) the best-known oracle *SE-Oracle-Adapt* is 78,000s $\approx$ 21.7 hours, and (2) the query time for calculating 10 nearest hotels of each viewing platform for (i) *RC-Oracle* is 6s, (ii) *SE-Oracle-Adapt* is 75s, and (iii) the best-known on-the-fly approximate shortest surface path query algorithm *Kaul-Adapt* is 80,500s $\approx$ 22.5 hours. Thus, *RC-Oracle* is the best one in the evacuation since 3.2 min + 6s $\leq$ 12 min. *RC-Oracle* also supports real-time responses, i.e., it can construct the oracle in 0.4s and answer the *kNN* query and range query in both 7 ms on a point cloud with 10k points and 250 POIs.

5.3.5 ***Summary.*** In terms of the oracle construction time, oracle size and shortest path query time, *RC-Oracle* is up to 390 times, 30 times and 6 times better than the best-known oracle *SE-Oracle-Adapt* for the P2P query on a point cloud, and up to 500 times, 140 times and 50 times better than the best-known oracle *EAR-Oracle-Adapt* for the A2A query on a point cloud. With the assistance of *RC-Oracle*, our algorithms for the *kNN* and range query are both up to 6 times faster than *SE-Oracle-Adapt* and up to 100 times faster than *EAR-Oracle-Adapt*. For the P2P query on a point cloud with 2.5M points and 500 POIs, the oracle construction time, oracle size and all POIs *kNN* query time for *RC-Oracle* is 200s $\approx$ 3.2 min, 50MB and 12.5s, but the values are 78,000s $\approx$ 21.7 hours, 1.5GB and 150s for the best-known oracle *SE-Oracle-Adapt*, respectively. For the A2A query on a point cloud with 100k points and 5000 objects, the oracle construction time, oracle size and all POIs *kNN* query time for *RC-Oracle-A2A* is 100s $\approx$ 1.6 min, 150MB and 0.25s, but the values are 50,000s $\approx$ 13.9 hours, 21GB and 25s for the best-known oracle *EAR-Oracle-Adapt*, respectively.

6 CONCLUSION

In our paper, we propose an efficient $(1+\epsilon)$-approximate shortest path oracle on a point cloud called *RC-Oracle*, 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*, we propose algorithms for answering other proximity queries, i.e., the *kNN* and range query. For the future work, we can explore how to build a novel index designed for the *kNN* and range query for better performance.

ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their constructive comments on this paper. The research of Yinzhao Yan and Raymond Chi-Wing Wong is supported by GZSTI16EG24.

[1]The time of a human being buried is calculated as 2.4 hours which is computed by $\frac{10\text{centimeters} \times 24\text{hours}}{1\text{meter}}$, since the maximum snowfall rate (which is defined to be the maximum amount of snow accumulates in depth during a given time [16, 50]) in Mount Rainier is 1 meter per 24 hours [49], 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 [26].

REFERENCES

[1] 2022. *Blender*. https://www.blender.org
[2] 2022. *Google Earth*. https://earth.google.com/web
[3] 2022. *Robinson Mountain*. https://www.mountaineers.org/activities/routes-places/robinson-mountain
[4] 2023. *Cyberpunk 2077*. https://www.cyberpunk.net
[5] 2023. *Data Geocomm*. http://data.geocomm.com/
[6] 2023. *Google Map*. https://www.google.com/maps
[7] 2023. *Gunnison National Forest*. https://gunnisoncrestedbutte.com/visit/places-to-go/parks-and-outdoors/gunnison-national-forest/
[8] 2023. *Laramie Mountain*. https://www.britannica.com/place/Laramie-Mountains
[9] 2023. *Preferred walking speed*. https://en.wikipedia.org/wiki/Preferred_walking_speed
[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] Claudine Badue, Rânik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius B Cardoso, Avelino Forechi, Luan Jesus, Rodrigo Berriel, Thiago M Paixao, Filipe Mutz, et al. 2021. Self-driving cars: A survey. *Expert Systems with Applications* 165 (2021), 113816.
[13] 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.
[14] 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.
[15] Jindong Chen and Yijie Han. 1990. Shortest Paths on a Polyhedron. In *SOCG*. New York, NY, USA, 360–369.
[16] 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
[17] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.
[18] 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.
[19] 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.
[20] 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.
[21] Ke Deng, Xiaofang Zhou, Heng Tao Shen, Qing Liu, Kai Xu, and Xuemin Lin. 2008. A multi-resolution surface distance model for k-nn query processing. *The VLDB Journal* 17, 5 (2008), 1101–1119.
[22] 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.
[23] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.
[24] 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.
[25] David Eriksson and Evan Shellshear. 2016. Fast exact shortest distance queries for massive point clouds. *Graphical Models* 84 (2016), 28–37.
[26] 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/
[27] 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.
[28] Bo Huang, Victor Junqiu Wei, Raymond Chi-Wing Wong, and Bo Tang. 2023. EAR-Oracle: on efficient indexing for distance queries between arbitrary points on terrain surface. *Proceedings of the ACM on Management of Data (SIGMOD)* 1, 1 (2023), 1–26.
[29] 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
[30] 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.

[31] 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.

[32] 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.

[33] Baki Koyuncu and Erkan Bostancı. 2009. 3D battlefield modeling and simulation of war games. *Communications and Information Technology proceedings* (2009).

[34] Russell LaDuca. 2020. *What would happen to me if I was buried under snow?* https://qr.ae/prt6zQ

[35] Mark Lanthier, Anil Maheshwari, and J-R Sack. 2001. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica* 30, 4 (2001), 527–562.

[36] 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).

[37] 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).

[38] 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.

[39] 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.

[40] 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.

[41] Joseph SB Mitchell, David M Mount, and Christos H Papadimitriou. 1987. The discrete geodesic problem. *SIAM J. Comput.* 16, 4 (1987), 647–668.

[42] 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

[43] Hoong Kee Ng, Hon Wai Leong, and Ngai Lam Ho. 2004. Efficient algorithm for path-based range query in spatial databases. In *Proceedings. International Database Engineering and Applications Symposium, 2004. IDEAS'04.* IEEE, 334–343.

[44] 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.

[45] 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.

[46] 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).

[47] National Park Service. 2022. *Mount Rainier.* https://www.nps.gov/mora/index.htm

[48] National Park Service. 2022. *Mount Rainier Annual Snowfall Totals.* https://www.nps.gov/mora/planyourvisit/annual-snowfall-totals.htm

[49] National Park Service. 2022. *Mount Rainier Frequently Asked Questionss.* https://www.nps.gov/mora/faqs.htm

[50] National Weather Service. 2023. *Measuring Snow.* https://www.weather.gov/dvn/snowmeasure

[51] 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.

[52] 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.

[53] 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).

[54] 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

[55] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, and David M. Mount. 2017. Distance oracle on terrain surface. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. 1211–1226.

[56] 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).

[57] Shi-Qing Xin and Guo-Jin Wang. 2009. Improving Chen and Han's algorithm on the discrete geodesic problem. *ACM Transactions on Graphics* 28, 4 (2009), 1–8.

[58] 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.

[59] 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.

[60] 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.

[61] Yinzhao Yan and Raymond Chi-Wing Wong. 2023. Proximity queries on point cloud using rapid construction path oracle (technical report). https://github.com/yanyinzhao/PointCloudPathCode/blob/master/TechnicalReport.pdf

[62] Hongchuan Yu, Jian J Zhang, and Zheng Jiao. 2014. Geodesics on point clouds. *Mathematical Problems in Engineering* 2014 (2014).

Received July 2023; revised October 2023; accepted November 2023