An Efficiently Updatable Path Oracle for Terrain Surfaces

Yinzhao Yan, Raymond Chi-Wing Wong, and Christian S. Jensen

Abstract—The booming of computer graphics technology facilitates the growing use of terrain data. Notably, shortest path querying on a terrain surface is central in a range of applications and has received substantial attention from the database community. Despite this, computing the shortest paths on-the-fly on a terrain surface remains very expensive, and all existing oracle-based algorithms are only efficient when the terrain surface is fixed. They rely on large data structures that must be re-constructed from scratch when updates to the terrain surface occur, which is very time-consuming. To advance the state-of-the-art, we propose an efficiently updatable $(1+\epsilon)$-approximate shortest path oracle for a set of *Points-Of-Interests* (*POIs*) on an updated terrain surface, and it can be easily adapted to the case if POIs are not given as input. Our experiments show that when POIs are given (resp. not given), our oracle is up to 88 times, 12 times, and 3 times (resp. 15 times, 50 times, and 100 times) better than the best-known oracle on terrain surfaces in terms of the oracle update time, output size, and shortest path query.

Index Terms—Spatial databases, query processing, shortest path query, terrain

1 Introduction

Calculating shortest paths on terrain surfaces is a topic of widespread interest [1]. In industry, Metaverse [2] and Google Earth [3] use shortest paths on terrain surfaces (e.g., in virtual reality or on Earth) to assist users to reach destinations more quickly. In academia, shortest path querying on terrain surfaces also attracts considerable attention [4], [5], [6], [7], [8], [9], [10], [11], [12]. A terrain surface is represented by a set of *faces*, each of which is captured by a triangle. A face consists of three line segments, called *edges*, connected with each other at three *vertices*. Figures 1 (a) and (b) show a real map of Valais, Switzerland [13] with an area of 20km $\times$ 20km, and Figures 1 (c) and (d) show Valais terrain surface (consisting of vertices, edges and faces).

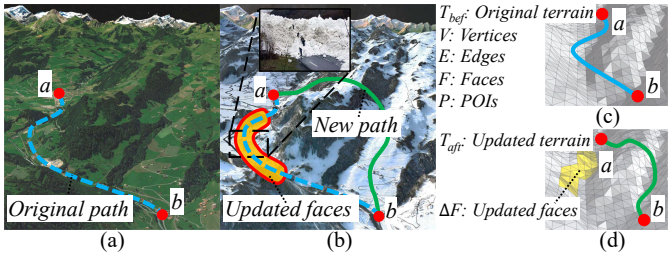

Fig. 1. The real map (a) before and (b) after updates, the terrain surface (c) before and (d) after updates for avalanche in Valais, Switzerland

1.1 Motivation

1) *Updated terrain surface:* The computation of shortest paths on *updated* terrain surfaces occurs in many scenarios.

(i) **Earthquake**: We aim to find the shortest rescue paths for life-saving after an earthquake. The death toll of the 7.8 magnitude earthquake on Feb 6, 2023 in Turkey and Syria exceeded 40,000 [14], and more than 69,000 died in the 7.9 magnitude earthquake on May 12, 2008 in Sichuan, China [15]. A rescue team can save 3 lives every 15 minutes [16], and we expect that the team can arrive at the sites of the quake as early as possible. In practice, (a) satellites or (b) drones can be used to collect the terrain surface after an earthquake, which takes (a) 10s and USD $48.72 [17], and (b) 144s $\approx$ 2.4 min and USD $100 [18] for a 1km^2 region, respectively, which are time and cost efficient.

(ii) **Avalanche**: Earthquakes may cause avalanches. The 4.1 magnitude earthquake on Oct 24, 2016 in Valais [13] caused an avalanche: Figures 1 (a) and (b) (resp. Figures 1 (c) and (d)) show the original and new shortest paths between a (a village) and b (a hospital) on a real map (resp. a terrain surface) before and after terrain surface updates. We need to efficiently find the shortest rescue paths.

(iii) **Marsquake**: As observed by NASA's InSight lander on May 4, 2022 [19], Mars also experienced a marsquake. In NASA's Mars exploration project [20] (with cost USD 2.5 billion [21]), Mars rovers should find the shortest escape paths quickly and autonomously in regions affected by marsquakes to avoid damage.

2) *P2P and A2A query:* (i) Given a set of *Points-Of-Interest* (*POI*) on a terrain surface, we can calculate the shortest path between *pairs of POIs*, i.e., perform the *POI-to-POI* (*P2P*) *query*. For earthquakes and avalanches, POIs can be villages waiting for rescuing [22], hospitals, and expressway exits. For the Marsquake, POIs can be working stations of Mars rover. (ii) If no POIs are given, we calculate the shortest path between *pairs of arbitrary points* (including the vertices of the terrain surface), i.e., perform the *arbitrary points-to-arbitrary points* (*A2A*) *query*. The A2A query generalizes the P2P query because it allows any points on a terrain surface.

3) *Oracle:* Pre-computing shortest paths on a terrain

- *Y. Yan and R.C. Wong are with The Hong Kong University of Science and Technology.*
  E-mail: yyanas@cse.ust.hk; raywong@cse.ust.hk
- *C.S. Jensen is with Aalborg University.*
  E-mail: csj@cs.aau.dk

- *Code: https://github.com/yanyinzhao/UpdatedStructureTerrainCode*

surface using an index, known as an *oracle*, can efficiently reduce the shortest path query time, especially when we need to calculate more than one shortest path with different sources and destinations (where the time taken to pre-compute the oracle is called the *oracle construction time*, the space usage of the output oracle is called the *output size*, and the time taken to return the result is called the *shortest path query time*). We also aim to update the oracle *quickly* when the terrain surface changes (where the time taken to update the oracle is called the *oracle update time*). In earthquakes, avalanches or Marsquakes, if we pre-compute shortest paths (among villages, hospitals or Mars rover working stations) using an *oracle* on terrain surfaces prone to these disasters, and efficiently update the oracle after the disaster, then we can use it to efficiently return shortest paths.

1.2 Challenges

1) *Inefficiency of on-the-fly algorithms:* Consider a terrain surface T with N vertices. All existing *exact on-the-fly* shortest path algorithms [23], [24], [25], [26] on a terrain surface are slow when many shortest path queries are involved. As recognized by existing studies [1], [5], [10], [27], the best-known exact algorithm [24], [25] runs in $O(N^2)$ time. Although *approximate* algorithms [6], [7], [8], [12] can reduce the running time, they are not efficient enough. The best-known approximate algorithm [6], [12] on terrain surfaces runs in $O((N + N')\log(N + N'))$ time, where N' is the number of Steiner points used for the bound guarantee. Our experiments show that the best-known exact algorithm [24], [25] (resp. approximate algorithm [6], [12]) needs 11,600s $\approx$ 3.2 hours (resp. 8,600s $\approx$ 2.4 hours) to calculate 100 shortest paths on a terrain surface with 0.5M faces.

2) *Non-existence of oracles on updated terrain surfaces:* Although existing studies [5], [9], [10] can construct oracles on *static* terrain surfaces, and can then answer P2P or A2A queries efficiently, no study can accommodate updated terrain surfaces, where the oracle needs to be updated efficiently. One study [28] constructs an oracle on *static* point clouds that can be adapted to *static* terrain surfaces for the P2P query by using the on-the-fly algorithm on terrain surfaces [23], [24], [25], [26]. However, this oracle still cannot accommodate updated terrain surfaces. When a terrain surface is updated, straightforward adaptations of the best-known oracle [9], [10] for the P2P query, the best-known oracle [5] for the A2A query, and the oracle [28] for points clouds adapted to terrain surfaces for the P2P query must re-construct the oracles. However, their oracle construction times are $O(nN^2 + c_1 n)$, $O(c_2 N^2)$, and $O(c_3 nN^2 + n\log n)$, respectively, where n is the number of POIs, c_1, c_2, and c_3 are constants depending on T ($c_1 \in [35, 80]$, $c_2 \in [75, 154]$, and $c_3 \in [3, 10]$ on average). In our experiments, oracle [9], [10] (resp. adapted oracle [28]) needs 35,100s $\approx$ 9.8 hours (resp. 28,100s $\approx$ 7.5 hours) for oracle construction on a terrain surface with 0.5M faces and 250 POIs, and oracle [5] needs 35,500s $\approx$ 9.9 hours on a terrain surface with 100k faces.

1.3 Path Oracle on Updated Terrain Surfaces

We propose an efficiently updatable $(1 + \epsilon)$-approximate shortest path oracle, called *Updatable Path Oracle* (*UP-Oracle*), for solving the *updated terrain surfaces problem* (given two terrain surfaces before and after updates, i.e., T_{bef} and T_{aft}, we efficiently answer P2P queries on T_{aft} by using shortest paths on T_{bef}), where $\epsilon > 0$ is the *error parameter*. We construct *UP-Oracle*, efficiently update it, and find the shortest path using it in Figures 2 (a) to (c), Figures 2 (d) to (f), and Figure 2 (g). *UP-Oracle* has state-of-the-art performance in terms of the oracle update time, output size, and shortest path query time (compared with the best-known oracle [9], [10] for the P2P query on terrain surfaces) due to the concise capture of pairwise P2P shortest paths. *UP-Oracle* can be easily adapted to answering A2A queries on T_{aft} (denoted by *UP-Oracle-A2A*) and also has good performance (compared with the best-known oracle [5] for the A2A query on terrain surfaces).

1) *Achieving a short oracle update time:* The ideas for achieving a short oracle update time of *UP-Oracle* follow from (i) a novel property, i.e., the *non-updated terrain shortest path intact* property, and (ii) the useful information on T_{bef}, i.e., the stored pairwise P2P exact shortest paths on T_{bef} when *UP-Oracle* is constructed.

(i) ***Non-updated terrain shortest path intact* property**: In Figure 3, this property implies that given the light blue path between u and v on T_{bef} (with distance d), if the distances from both u and v to the updated faces are large enough (i.e., both larger than $\frac{d}{2}$), then the path between u and v on T_{aft} remains the same and does not need to be updated.

(ii) **Stored pairwise P2P exact shortest paths on T_{bef}**: The *exact* shortest distances are no larger than the *approximate* shortest distances. So given an exact (resp. approximate) shortest path with two endpoints u and v on T_{bef}, in the non-updated terrain shortest path intact property in Figure 3, it is likely (resp. unlikely) that the distances from both u and v to the updated faces are both larger than the exact (resp. approximate) length of this path, and it reduces (resp. increases) the likelihood of updating this path on T_{aft}.

2) *Efficiently achieving a small output size:* Although we have the pairwise P2P exact shortest paths on T_{aft}, we aim to return fewer paths to reduce the output size:

(i) **Earthquake and avalanche**: For the earthquake region with ruins, rescue teams need to transport injured citizens from damaged villages to unaffected hospitals. Since it is time-consuming to build rescue paths in the earthquake region [29], fewer paths imply that the total time to build rescue paths is smaller, enabling the rescue teams to focus on saving lives. For other nearby regions, trucks need to transfer medical materials to hospitals. Rescue teams need to keep roads clear to avoid road blockages caused by panic. Fewer paths imply that the number of rescue teams needed for road maintenance is smaller, enabling an increased focus on saving lives. That is, given a complete graph (with the POIs as vertices and with the exact shortest paths between POIs on T_{aft} as edges), we hope that *UP-Oracle* can efficiently generate a sub-graph of it with a small output size. But, we can store the complete graph in the hard disk for any subsequent changes.

(ii) **Marsquake**: For Mars rovers, we only know the damaged region after a quake, so the coverage of T_{bef} stored in Mars rovers is large (e.g., the entire Mars surface), and similar to T_{aft}. NASA's Mars 2020 rover has 256MB memory and 2GB hard disk space [30], and it autonomously calculates paths [31]. Since the round trip signal delay between

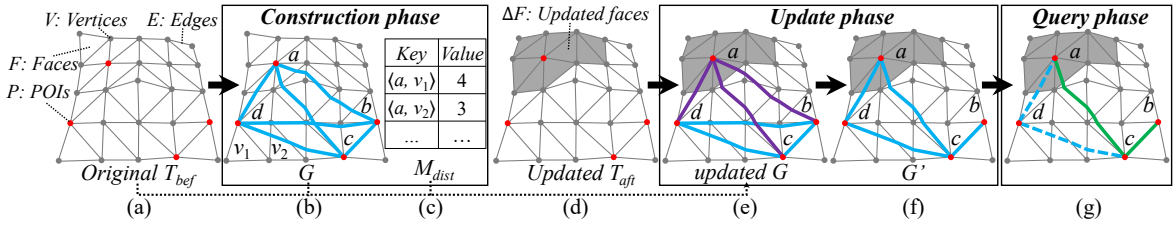

Fig. 2. Framework overview

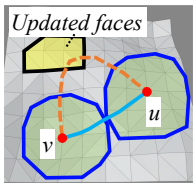

Fig. 3. An unaffected path

Earth and Mars is 40 minutes [32], it is time-consuming to send terrain information captured by a rover after a quake to Earth, ask human experts to find shortest escape paths, and then send the paths to Mars. Before a quake, a rover stores the pairwise P2P exact shortest paths on T_{bef} in the hard disk and transfers this to memory as needed. After a quake, it needs to update the pairwise P2P exact shortest paths on T_{aft} and store this in a complete graph. It then needs to efficiently generate a sub-graph for rescue path calculation. Because our experiments show that for a terrain surface with 250k faces and 250 POIs, the pairwise P2P exact shortest paths on T_{bef} consume 127MB, the complete graph occupies 126MB, and the sub-graph output from *UP-Oracle* occupies 10MB. When a rover starts to escape, it needs 210MB extra memory for different sensors to work [33]. Since 126MB + 210MB = 336MB $>$ 256MB and 10MB + 210MB = 220MB $<$ 256MB, we can only fit the sub-graph in the memory of a rover for escaping. We have sufficient memory for path updating and sub-graph generation since 127MB + 126MB = 253MB $<$ 256MB and 126MB + 10MB = 136MB $<$ 256MB, and we can store the complete graph in the hard disk for subsequent changes since 126MB $<$ 2GB.

Generating a sub-graph from a complete graph is also used in distributed systems for faster network synchronization [34] and in wireless networks for faster signal transmission [35]. Specifically, given a complete graph and ϵ, the best-known $(1+\epsilon)$-sub-graph generation algorithm [36] runs in $O(n^3 \log n)$ time, which is inefficient. A $(1+\epsilon)$-sub-graph has the property that the distance between any pair of its vertices is at most $(1+\epsilon)$ times the exact distance. We propose a faster algorithm called *Hierarchy Greedy Spanner* (*HGSpan*). Given a complete graph and ϵ, we use a simpler structure to approximate the internal graph for faster processing and generate a $(1+\epsilon)$-sub-graph. Our experiments show that when $n = 500$, our algorithm takes 24s, while the best-known algorithm [36] takes 101s.

We can also maintain a multi-layer structure. Long-range queries can utilize the approximate results calculated using the sub-graph, as the complete graph is excessively large. Short-range queries can utilize exact results with higher accuracy. In the earthquake and avalanche (resp. Marsquake) example, dense population villages (resp. Mars rover frequent work regions) can be regarded as short-range query regions, we use the exact results for faster rescuing (resp. escaping). Depending on different areas of villages or work regions, we can select different *Level-Of-Details* (*LODs*) of short-range query regions for customized querying.

1.4 Contributions and Organization

Our major contributions are as follows.

(1) We propose the first oracle, called *UP-Oracle*, for solving the updated terrain surfaces problem. It achieves a short oracle update time by satisfying the novel non-updated terrain shortest path intact property, and by utilizing the useful information on T_{bef} (the pairwise P2P exact shortest paths on T_{bef}). We also propose four additional techniques to further reduce the oracle update time. Designing an oracle on an updated terrain surface with a small oracle update time is challenging: there are no existing solutions, and only limited information about T_{bef} can be re-used. We also develop an efficient algorithm called *HGSpan* to reduce the output size, and we adapt *UP-Oracle* for handling subsequent changes, adapt *UP-Oracle* to *UP-Oracle-A2A* for A2A queries, and adapt *UP-Oracle* to a multi-layer structure called *UP-Oracle Multi Layer* (*UP-Oracle-MuLa*).

(2) We provide a thorough theoretical analysis of the oracle construction time, oracle update time, output size, shortest path query time, and error bound for these oracles.

(3) *UP-Oracle* performs much better than the best-known oracle [9], [10] for the P2P query and *UP-Oracle-A2A* performs much better than the best-known oracle [5] for the A2A query on terrain surfaces in terms of the oracle update time, output size, and shortest path query time. (i) For the P2P query on a terrain surface with 0.5M faces and 250 POIs, the oracle update time and output size of *UP-Oracle* are 400s $\approx$ 6.7 min and 22MB, while the values are 35,100s $\approx$ 9.8 hours, and 250MB for the best-known oracle [9], [10], respectively, and (ii) the shortest path query time for computing 100 shortest paths with different sources and destinations is 0.1s for *UP-Oracle*, while the time is 8,600s $\approx$ 2.4 hours for the best-known approximate on-the-fly algorithm [6], [12] and 0.3s for the best-known oracle [9], [10]. (iii) For the A2A query on a terrain surface with 20k faces, the oracle update time, output size, and shortest path query time for computing 100 shortest paths of *UP-Oracle-A2A* are 480s $\approx$ 7 min, 3MB and 0.05s, while the values are 7,100s $\approx$ 2 hours, 150MB and 5s for the best-known oracle [5], respectively. The adapted *UP-Oracle* for subsequent changes and *UP-Oracle-MuLa* also perform well.

The remainder of the paper is organized as follows. Section 2 provides the preliminary. Section 3 discusses related work. Section 4 presents *UP-Oracle*. Section 5 covers the empirical study, and Section 6 concludes the paper.

2 Preliminary

2.1 Notation and Definitions

1) *Terrain surfaces and POIs:* Consider a terrain surface T_{bef} represented as a *Triangulated Irregular Network* (*TIN*) [8], [9], [10], [27]. Let V, E, and F be the set of vertices, edges,

and faces of T_{bef}, respectively. Let L_{max} be the length of the longest edge in E. Let N be the number of vertices. Each vertex $v \in V$ has three coordinates, x_v, y_v, and z_v. If the positions of vertices in V are updated, we obtain a new terrain surface, T_{aft}. Figures 1 (c) and (d), and Figures 2 (a) and (d) show examples of T_{bef} and T_{aft}, respectively. There is no need to consider the case when N changes because T_{bef} and T_{aft} have the same 2D grid with $\overline{x} \times \overline{y} = N$ vertices [8], [9], [10]. Specifically, for T_{bef} in Figure 2 (a), given a *fixed* region, existing methods [1], [5], [10], [27] sample a *fixed* set of points on T_{bef} on the 2D grid in the x-y plane and use the elevations of these points as the z-coordinates, yielding the final set of vertices on T_{bef}. For T_{aft} in Figure 2 (d), since we focus on the *same* region (although the *shape* of the region on T_{aft} may change), the set of points is *fixed*, and N is also *fixed*. In the P2P query, let P be a set of POIs on T_{bef} and n be the number of POIs. There is no need to consider when n changes, or when $n > N$. The set of red points in Figure 2 (a) is P. When a POI is added, we create an oracle that answers the A2A query, which implies we consider all possible POIs to be added. When a POI is removed, we continue to use the original oracle. When $n > N$, we still create an oracle that answers the A2A query.

2) *Path:* Given s and t in P, and a terrain surface T, let $\Pi(s,t|T)$ be the exact shortest path between s and t on T, and $|\cdot|$ be the distance of a path (e.g., $|\Pi(s,t|T)|$ is the exact distance of $\Pi(s,t|T)$ on T).

3) *Updated and non-updated faces:* Given T_{bef}, T_{aft}, and P, a set of *updated faces*, denoted by ΔF, is defined to be a set of faces $\Delta F = \{f_1, f_2, \ldots, f_{|\Delta F|}\}$, where f_i is a face in F with at least one of its three vertices' coordinates differing between T_{bef} and T_{aft}, and $|\Delta F|$ is the number of faces in ΔF. It is easy to obtain ΔF by comparing T_{bef} and T_{aft}. In Figure 1 (d) (resp. Figure 2 (d)), the yellow (resp. gray) region is ΔF based on T_{bef} and T_{aft}. There is no need to consider the case with two or more *disjoint* non-empty sets of updated faces. If this happens, we can create a larger set of faces that contains these disjoint sets. Thus, in Figures 1 (d) and Figure 2 (d), the set of updated faces that we consider is connected [37]. We say that a point (either a vertex or a POI) is in ΔF if it is on a face in ΔF, and we say that a path passes ΔF if any segment of this path is on a face ΔF. In Figure 2 (e), a is in ΔF, and the purple path between a and b passes ΔF.

4) *Disk:* Given a point p on T_{bef} and a constant $r > 0$, let $D(p,r)$ be the disk centered at p with radius r, which consists of all points on T_{bef} whose exact shortest distance to p is no more than r. Given a face f_i, if a point q exists on f_i such that the shortest distance between p and q is no more than r, then disk $D(p,r)$ *intersects with* face f_i. Figure 3 shows two disks centered at u and v, both with radius $\frac{|\Pi(u,v|T_{bef})|}{2}$, that do not intersect with any updated faces. Table 1 shows a summary of frequently used notation.

2.2 Updated Terrain Surfaces Problem

Given T_{bef}, T_{aft}, and P, the problem is to efficiently answer P2P queries on T_{aft} (using shortest paths on T_{bef}) with $|\Pi'(s,t|T_{aft})| \leq (1+\epsilon)|\Pi(s,t|T_{aft})|$ for any s and t in P, where $\Pi'(s,t|T_{aft})$ is the shortest path result between s and t on T_{aft}.

TABLE 1
Summary of frequently used notation

Notation	Meaning
T_{bef}/T_{aft}	The terrain surface before / after updates
$V/E/F$	The set of vertices / edges / faces of terrain surface
L_{max}	The length of the longest edge in E of T_{bef}
N	The number of vertices of T
P	The set of POI
n	The number of vertices of P
$\Pi(s,t\|T)$	The exact shortest path between s and t on T
$\|\Pi(s,t\|T)\|$	The distance of $\Pi(s,t\|T)$
ΔF	The updated faces of T_{bef} and T_{aft}
$D(p,r)$	A disk centered at p with radius r
ϵ	The error parameter

2.3 Non-updated Terrain Shortest Path Intact Property

Property 1 describes an important property for solving the updated terrain surface surfaces problem.

Property 1 (Non-updated Terrain Shortest Path Intact Property). *In Figure 3, given T_{bef}, T_{aft}, and $\Pi(u,v|T_{bef})$, if two disks $D(u, \frac{|\Pi(u,v|T_{bef})|}{2})$ and $D(v, \frac{|\Pi(u,v|T_{bef})|}{2})$ do not intersect with ΔF, then $\Pi(u,v|T_{aft})$ is the same as $\Pi(u,v|T_{bef})$.*

Proof Sketch. We show by contradiction that the two paths cannot be different. The detailed proofs in the remaining of this paper appear in our technical report [38]. □

3 Related Work

3.1 On-the-fly Algorithms on Terrain Surfaces

Two types of algorithms can compute the shortest path on a terrain surface *on-the-fly*.

1) *Exact algorithms:* The running times of the four exact algorithms [23], [24], [25], [26] are $O(N \log^2 N)$, $O(N^2)$, $O(N^2)$, and $O(N^2 \log N)$, respectively. They are *Single-Source All-Destination* (*SSAD*) algorithms [5], [10], [23], [24], [25], [26], i.e., given a source, they can calculate the shortest path from it to all other vertices *simultaneously*. According to existing studies [1], [5], [10], [27], algorithm [23] that runs in $O(N \log^2 N)$ time is hard to implement (no implementation exists). So, the implementable algorithm *WAVefront on-the-Fly Algorithm* (*WAV-Fly-Algo*) [24], [25] that runs in $O(N^2)$ time is recognized as the practical algorithm of choice. It uses a continuous version of Dijkstra's algorithm and needs to consider continuous points on the edges of the terrain surface by unfolding the 3D terrain surface into a 2D plane (which is not needed in the plain Dijkstra's algorithm) during wavefront propagation, so its running time cannot be reduced to $O(N \log N)$ in the plain Dijkstra's algorithm. *WAV-Fly-Algo* comes in two variants with the same time complexity: an initial version [25] and an extended version [24] with better empirical running time.

2) *Approximate algorithms:* Approximate algorithms [6], [7], [8], [12] aim to reduce the running time. The best-known approximate algorithm *Efficient Steiner Point on-the-Fly Algorithm* (*ESP-Fly-Algo*) [6], [12] on terrain surfaces places Steiner points on edges in E, and then constructs a graph using these points and V to calculate a $(1+\epsilon)$-approximate shortest path on a terrain surface using Dijkstra's algorithm. It 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 T, and θ is the minimum inner angle of any face in F. Algorithm [6] runs on an unweighted terrain surface and algorithm [12] runs on a weighted terrain surface where each terrain surface face is assigned a weight. They are the same if we set the weight of each face in algorithm [12] to be 1, so we regard them as one algorithm.

Drawback: All these algorithms are inefficient for computing multiple shortest path queries. Our experiments show that *WAV-Fly-Algo* and *ESP-Fly-Algo* need 11,600s $\approx$ 3.2 hours, and 8,600s $\approx$ 2.4 hours to compute 100 paths with different sources and destinations on a terrain surface with 0.5M faces, respectively.

3.2 Oracle-based Algorithms on Terrain Surfaces

The *Well-Separated Pair Decomposition Oracle* (*WSPD-Oracle*) [9], [10] (resp. the *Efficiently ARbitrary pints-to-arbitrary points Oracle* (*EAR-Oracle*) [5]) is regarded as the best-known oracle for answering *approximate* P2P (resp. A2A) queries on terrain surfaces. Further, an existing oracle [28], originally designed for answering *approximate* P2P queries on point clouds, can be adapted to the *Rapid-Constuction TIN Oracle* (*RC-TIN-Oracle*) for answering *approximate* P2P queries on terrain surfaces [28]. Yet, no existing oracle can accommodate updated terrain surfaces, where the oracle needs to be updated efficiently. A straightforward adaption is to re-construct them from scratch when the terrain surface is updated. A smart adaption is to leverage Property 1 in *UP-Oracle*, such that we only re-calculate the paths on T_{aft} that require updating to reduce the oracle update time. We denote the adapted oracles as *WSPD-UP-Oracle*, *EAR-UP-Oracle*, and *RC-TIN-UP-Oracle*.

1) *WSPD-Oracle* and *WSPD-UP-Oracle*: These use a *compressed partition tree*, algorithm *SSAD*, and *well-separated node pair sets* to index the $(1+\epsilon)$-approximate pairwise P2P shortest paths. (i) *WSPD-Oracle*'s oracle construction time, output size, and shortest path query time is $O(\frac{nN^2}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$, 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 [39]. (ii) *WSPD-UP-Oracle*'s oracle update time is $O(\mu_1 N^2 + n\log^2 n)$, where μ_1 is a data-dependent variable, and $\mu_1 \in [5, 20]$ in our experiments.

2) *EAR-Oracle* and *EAR-UP-Oracle* These use the same idea as *WSPD-Oracle* and *WSPD-UP-Oracle*, i.e., well-separated pair decomposition. Their differences are that they adapt *WSPD-Oracle* and *WSPD-UP-Oracle* from the P2P query to the A2A query by using Steiner points on the terrain faces and using *highway nodes* (i.e., not POIs in *WSPD-Oracle* and *WSPD-UP-Oracle*) for well-separated pair decomposition. (i) *EAR-Oracle*'s oracle construction time, output size, and shortest path query time is $O(\lambda\xi(mN)^2 + \frac{N^3}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$, and $O(\lambda\xi\log(\lambda\xi))$, respectively, where λ is the number of highway nodes in one square, ξ is the square root of the number of boxes, and m is the number of Steiner points per face. (ii) *EAR-UP-Oracle*'s oracle update time is $O(\mu_2 N^2 + n\log^2 n)$, where μ_2 is a data-dependent variable, and $\mu_2 \in [12, 45]$ in our experiments.

3) *RC-TIN-Oracle* and *RC-TIN-UP-Oracle* These use *path* and *endpoint* map tables to index the $(1+\epsilon)$-approximate pairwise P2P shortest paths. (i) *RC-TIN-Oracle*'s oracle construction time, output size, and shortest path query time is $O(\frac{nN\log N}{\epsilon} + n\log n)$, $O(\frac{nN}{\epsilon})$, and $O(1)$, respectively. (ii) *RC-TIN-UP-Oracle*'s oracle update time is $O(\mu_3 N^2 + n\log^2 n)$, where μ_3 is a data-dependent variable, and $\mu_3 \in [30, 65]$ in our experiments.

Drawbacks: (i) *WSPD-Oracle*, *EAR-Oracle*, and *RC-TIN-Oracle* only support the *static* terrain surface and do not address how to update the oracle on an *updated* terrain surface, since they do not utilize Property 1. (ii) Although *WSPD-UP-Oracle*, *EAR-UP-Oracle*, and *RC-TIN-UP-Oracle* utilize Property 1, they do not *fully utilize it*. Since they only store the pairwise P2P *approximate* shortest paths on T_{bef}, the oracle update time remains large. (iii) In the P2P query, the oracle update time for *WSPD-Oracle*, *WSPD-UP-Oracle*, *RC-TIN-Oracle*, *RC-TIN-UP-Oracle*, and *UP-Oracle* are 35,100s $\approx$ 9.8 hours, 8,400s $\approx$ 2.4 hours, 28,100s $\approx$ 7.5 hours, 10,100s $\approx$ 2.9 hours, and 400s $\approx$ 6.7 min on a terrain dataset with 0.5M faces and 250 POIs, respectively. In the A2A query, the oracle update time for *EAR-Oracle*, *EAR-UP-Oracle*, and *UP-Oracle-A2A* are 7,100s $\approx$ 2 hours, 4,300s $\approx$ 1.2 hours, and 480s $\approx$ 7 min on a terrain surface with 20k faces, respectively.

3.3 Sub-graph Generation Algorithms

Given a complete graph and ϵ, algorithm *Greedy Spanner* (*GSpan*) [36] that runs in $O(n^3\log n)$ time is the best-known $(1+\epsilon)$-sub-graph generation algorithm. Given a $(1+\epsilon')$-sub-graph and ϵ, where $\epsilon \geq \epsilon' > 0$, algorithm *Sparse Greedy Spanner* (*SGSpan*) [40] uses a simpler structure to approximate the internal graph for faster processing and generates a $(1+\epsilon)$-sub-graph.

Drawbacks: (i) Algorithm *GSpan* is very slow since it does not use any simpler structure to approximate the internal graph when performing Dijkstra's algorithm [41] on the sub-graph. (ii) Although algorithm *SGSpan* uses a simpler structure for approximation, it cannot take a complete graph, i.e., $\epsilon' = 0$, as input. If we force $\epsilon' = 0$, algorithm *SGSpan* degenerates to algorithm *GSpan*. (iii) On a terrain surface with 0.5M faces and 500 POIs, algorithm *HGSpan* uses 24s to generate a $(1+\epsilon)$-sub-graph of size 44MB, but algorithm *GSpan* uses 101s to generate a $(1+\epsilon)$-sub-graph of size 41MB.

4 Methodology

4.1 Overview of *UP-Oracle*

1) *Components:* *UP-Oracle* has three components.

(i) **The temporary graph G** is a complete graph that stores the pairwise P2P exact shortest paths. Let $G.V$ and $G.E$ be the sets of vertices and edges of G (where each POI in P is denoted by a vertex in $G.V$). Given a pair of POIs u and v, the exact shortest path $\Pi(u, v|T)$ on T is denoted by an edge $e(u, v|T)$ in $G.E$ with a weight $|e(u, v|T)| = |\Pi(u, v|T)|$, where T can be T_{bef} or T_{aft}. Figure 2 (b) shows a G with 4 vertices and 6 edges. The light blue edge $e(a, c|T_{bef})$ in G denotes a path $\Pi(a, c|T_{bef})$.

(ii) **POI-to-vertex distance table M_{dist}** is a *hash table* [42] that stores the exact shortest distance from each POI in P to each vertex in V on T_{bef}, used for reducing the oracle update time of *UP-Oracle*. A vertex u and a POI v is stored as a key $\langle u, v\rangle$, and the distance between them $|\Pi(u, v|T_{bef})|$ is

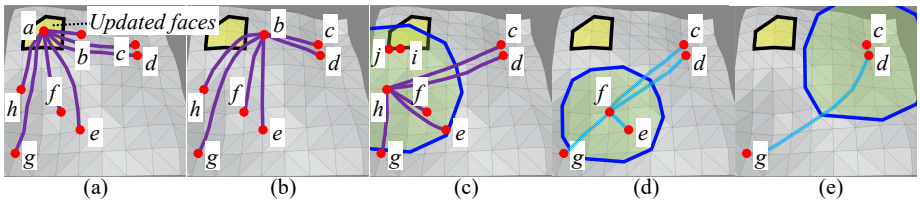

Fig. 4. In the update phase when (a) updating $\Pi(a)$, (b) updating $\Pi(b)$, (c) updating $\Pi(h)$, (d) no need for updating $\Pi(f)$, and (e) no need for updating $\Pi(d)$

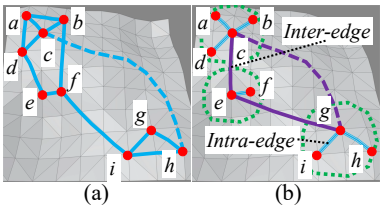

Fig. 5. (a) *UP-Oracle* output graph G' and (b) hierarchy graph H of G'

stored as a value. In Figures 2 (c), the exact shortest distance between POI a and vertex v_1 is 4.

(iii) **The *UP-Oracle* output graph G'** is a sub-graph of G used for answering pairwise P2P $(1+\epsilon)$-approximate shortest paths. Let $G'.V$ and $G'.E$ be the set of vertices and edges of G'. Given a pair of vertices u and v in $G'.V$, let $e'(u,v|T_{aft})$ be an edge with a weight $|e'(u,v|T_{aft})|$, and let $\Pi_{G'}(u,v|T_{aft})$ be the shortest path on G'. Figure 2 (f) shows a G' with 4 edges. The light blue edge $e'(a,c|T_{aft})$ in G' denotes a path $\Pi(a,c|T_{aft})$. The shortest path $\Pi_{G'}(a,b|T_{aft})$ consists of edges $e'(a,c|T_{aft})$ and $e'(c,b|T_{aft})$.

2) *Phases:* *UP-Oracle* has three phases.

(i) **Construction phase**: Given T_{bef} and P, considering each POI in P as the source, we use algorithm *SSAD* [5], [10], [23], [24], [25], [26] to simultaneously: (1) calculate the exact shortest paths between this POI and other POIs on T_{bef}, and store these in G, and (2) calculate the exact shortest distance between this POI and all vertices on T_{bef}, and store these in M_{dist}. In Figures 2 (b) and (c), we first use algorithm *SSAD* with a as the source to calculate paths between a and $\{b,c,d\}$ (the light blue paths), and distances between a and all vertices. Next, we use b,c as sources and repeat this.

(ii) **Update phase**: Given T_{bef}, T_{aft}, P, G, and M_{dist}, we efficiently update paths on T_{aft} in G and produce G':

- *Detect updated terrain surface*: Given T_{bef} and T_{aft}, we compare the coordinates of their vertices to detect ΔF.
- *Update exact shortest path*: Given T_{aft}, P, G, M_{dist}, and ΔF, we select some POIs in P as sources in algorithm *SSAD* to update the exact shortest paths on T_{aft} if Property 1 is not satisfied for paths connecting to these POIs, and we update G. In Figure 2 (e), we use a as the source to update paths between a and $\{b,c,d\}$ (the purple paths). Figure 4 shows more details. In Figures 4 (a) to (c), since a is on a face in ΔF, the path with b as the source passes on ΔF, and the blue disk $D(h, \frac{|\Pi(c,h|T_{bef})|}{2})$ centered at h intersects with ΔF, Property 1 is not satisfied for all possible paths connecting a,b,h. So, we use them as sources in algorithm *SSAD* for path updating on T_{aft}, and we update G. In Figures 4 (d) and (e), Property 1 is satisfied, and path update is not needed. We give more details in Section 4.2.
- *Generate sub-graph*: Given G and ϵ, we use algorithm *HGSpan* to efficiently generate G' for output size reduction, such that $|\Pi_{G'}(s,t|T_{aft})| \leq (1+\epsilon)|\Pi(s,t|T_{aft})|$ for any pair of POIs s and t in P. In Figure 2 (f), we obtain G' from G. We give more details in Section 4.3.

(iii) **Query phase**: Given G', and a pair of POIs s and t in P, we use Dijkstra's algorithm to find the shortest path between s and t on G', i.e., $\Pi_{G'}(s,t|T_{aft})$. In Figure 2 (g), given a source a and a destination b, we calculate $\Pi_{G'}(a,b|T_{aft})$, see the green path.

4.2 Update Phase: Update Exact Shortest Path Step

UP-Oracle has a short oracle update time due to our design in the *update exact shortest path* step of the update phase.

1) *Method:* Recall from Section 1.3 that the short oracle update time is mainly enabled by the *non-updated terrain shortest path intact* property in Property 1, and the stored pairwise P2P exact shortest paths on T_{bef}. We consider three additional issues and propose four techniques (one for each of the first two issues, and two for the third issue) to further reduce the oracle update time.

(i) **Which POI to select first for path updating before Property 1 is utilized - Optimal POI selection sequence:** In Figures 4 (a) to (c), (i) a is in ΔF, (ii) one of b's exact shortest paths, $\Pi(b,h|T_{bef})$, passes ΔF, and (iii) h is near ΔF. As Property 1 is not satisfied for the paths connecting a,b,h, we use a as the source in algorithm *SSAD* to update the paths on T_{aft} to other POIs simultaneously, and repeat this for b and h. In Figures 4 (d) and (e), Property 1 is satisfied, so we do not need to use algorithm *SSAD* with f and d as sources. Here, we only need to use algorithm *SSAD* 3 times, and the optimal sequence is selecting the POIs: (i) on a face in ΔF, (ii) connecting to the path passing ΔF, and (iii) near ΔF (corresponding to the sequence a,b,h). But, if we do not use this optimal sequence, e.g., we first update the paths with c,d,e,f,g as sources, then we still need to update the paths with a,b,h as sources. Here, we need to use algorithm *SSAD* 8 times. Note that the sequence only aims to identify the POI to select first as the source in algorithm *SSAD*, and we still update paths in parallel (as in the construction phase).

(ii) **Which disk radius to use in Property 1 - Minimum disk radius:** In Property 1, we use *half* of the distance between a pair of POIs as the disk radius to reduce the likelihood of re-calculating shortest paths on T_{aft}. But, if we do not use this minimum disk radius, we need to use the *full* distance. This increases the likelihood of updating this path on T_{aft}.

(iii) **How to efficiently determine whether Property 1 is satisfied - Efficient distance approximation:** In Figure 4 (c), given h, let i be the point belonging to ΔF that is closest to h, and let j be the vertex in ΔV that is closest to h. When determining whether Property 1 is satisfied, for the path between h and c, we determine whether the blue disk $D(h,r)$ intersects with ΔF (where $r = \frac{|\Pi(h,c|T_{bef})|}{2}$), by efficiently determining whether $r < |\Pi(h,j|T_{bef})| - L_{max}$ in $O(1)$ time (where $|\Pi(h,j|T_{bef})|$ is stored in M_{dist}). If so, $r < |\Pi(h,i|T_{aft})|$, i.e., Property 1 is satisfied, and we do not need to update the path, since we have the distance

approximation $|\Pi(h, j|T_{bef})| - L_{max} \leq |\Pi(h, i|T_{aft})|$ from the triangle inequality. Otherwise, we update the path. But, if we do not use this efficient approximation, we need to calculate $|\Pi(h, i|T_{aft})|$ using algorithm *SSAD* in $O(N^2)$ time.

(iv) **How to efficiently determine whether Property 1 is satisfied - Efficient disk and updated face intersection check:** In Figure 4 (c), we sort the third type of POI in the optimal POI selection sequence from near to far based on their minimum distance to any vertex in ΔV on T_{bef} using M_{dist}. Thus, we get the ordering h, f, e, d, c, g. When determining whether Property 1 is satisfied, we just need to create one blue disk $D(h, r)$ (where $r = \frac{|\Pi(h,c|T_{bef})|}{2}$ is half of the longest distance of the paths between h and each POI in $\{c, d, e, f, g\}$), and determine whether it intersects with ΔF. If the disk with the largest radius and with the center closest to ΔF intersects with ΔF, Property 1 is not satisfied and there is no need to check other disks. Otherwise, Property 1 is always satisfied. In total, there are $O(n)$ POIs, we need to create $O(n)$ disks for efficient checking. But, if we do not use this efficient checking, we need to create ten disks, i.e., five disks $D(h, \frac{|\Pi(X,h|T_{bef})|}{2})$ and five disks $D(X, \frac{|\Pi(X,h|T_{bef})|}{2})$ for checking, where $X \in \{c, d, e, f, g\}$. In total, there are $O(n^2)$ paths, it needs to create $O(n^2)$ disks.

2) *Algorithm:* Before we provide the algorithm, we introduce some notation. Let $P_{rem} = \{p'_1, p'_2, \ldots, p'_{|P_{rem}|}\}$ be a set of remaining POIs in P on T_{aft} that we have not processed, where $|P_{rem}|$ is the number of POIs in P_{rem}. P_{rem} is initialized to be P. In each update iteration, when we have processed a POI, we remove it from P_{rem}. In Figures 4 (a) and (b), $P_{rem} = \{b, c, d, e, f, g, h\}$ and $P_{rem} = \{c, d, e, f, g, h\}$. Given a POI $u \in P_{rem}$, let $\Pi(u) = \{\Pi(u, v_1|T_{bef}), \Pi(u, v_2|T_{bef}), \ldots, \Pi(u, v_{|\Pi(u)|}|T_{bef})\}$ be a set of the exact shortest paths stored in G on T_{bef} with u as an endpoint and each $v_i \in P_{rem} \setminus \{u\}$ as the other endpoint, such that these paths have not been updated. $\Pi(u)$ is initialized to be all the exact shortest paths stored in G with u as an endpoint, where $|\Pi(u)|$ is the number of paths in $\Pi(u)$. In Figures 4 (a) to (c), the purple paths denote $\Pi(a)$, $\Pi(b)$, and $\Pi(h)$. We summarize the methods in Algorithms 1 and 2.

Algorithm 1 *OnePoiUpdate* (T_{aft}, G, u, P_{rem})

Input: T_{aft}, G, a POI u, and P_{rem}
Output: updated G and updated P_{rem}
1: use u as the source in algorithm *SSAD* to calculate $\Pi(u, v|T_{aft})$ for each POI $v \in P_{rem}$ simultaneously
2: **for** each POI $v \in P_{rem}$ **do**
3: $G.E \leftarrow G.E - \{\Pi(u, v|T_{bef})\} \cup \{\Pi(u, v|T_{aft})\}$
4: $\Pi(v) \leftarrow \Pi(v) - \{\Pi(u, v|T_{bef})\}$
5: $P_{rem} \leftarrow P_{rem} - \{u\}$
6: **return** updated G and P_{rem}

3) *Example:* Algorithm 1 is used in three places in Algorithm 2. The following is an example of them.

(i) **Initialize P_{rem} and $\Pi(u)$:** Lines 1–3. In Figure 4 (a), we initialize $P_{rem} = \{a, b, c, d, e, f, g, h\}$, $\Pi(a) = \{\Pi(a, b|T_{bef}), \Pi(a, c|T_{bef}), \ldots, \Pi(a, h|T_{bef})\}$, $\Pi(b) = \{\Pi(b, a|T_{bef}), \Pi(b, c|T_{bef}), \ldots, \Pi(b, h|T_{bef})\}$, $\ldots$, and $\Pi(h) = \{\Pi(h, a|T_{bef}), \Pi(h, b|T_{bef}), \ldots, \Pi(h, g|T_{bef})\}$. Next, we use the *optimal POI selection sequence* for path updating.

(ii) **Path update for POI in ΔF:** Lines 4–6. In Figure 4 (a), a is on a face in ΔF, so Property 1 is not satisfied. We first use *OnePoiUpdate* (T_{aft}, G, a, P_{rem}) to update the purple paths on T_{aft}, and update G. Then, we remove $\Pi(a, X|T_{bef})$ in $\Pi(X)$ for each $X \in P_{rem}$, so $\Pi(a)$ becomes empty, $\Pi(b) = \{\Pi(b, c|T_{bef}), \Pi(b, d|T_{bef}), \ldots, \Pi(b, h|T_{bef})\}$, $\ldots$, and $\Pi(h) = \{\Pi(h, b|T_{bef}), \Pi(h, c|T_{bef}), \ldots, \Pi(h, g|T_{bef})\}$. Finally, we remove a from P_{rem} to get $P_{rem} = \{b, c, d, e, f, g, h\}$.

Algorithm 2 *Update* $(T_{aft}, P, G, M_{dist}, \Delta F)$

Input: T_{aft}, P, G, M_{dist}, and ΔF
Output: updated G
1: $P_{rem} \leftarrow P$
2: **for** each POI $u \in P_{rem}$ **do**
3: $\Pi(u) \leftarrow$ all the exact shortest paths in G with u as an endpoint
4: **for** each POI $u \in P_{rem}$ **do**
5: **if** u is on a face in ΔF (i.e., Property 1 is not satisfied) **then**
6: *OnePoiUpdate* (u, T_{aft}, G, P_{rem})
7: **for** each POI $u \in P_{rem}$ **do**
8: **if** u is not on any face in ΔF but there exists an exact shortest path in $\Pi(u)$ that passes ΔF (i.e., Property 1 is not satisfied) **then**
9: *OnePoiUpdate* (u, T_{aft}, G, P_{rem})
10: sort each POI in P_{rem} from near to far based on their minimum distance to any vertex in ΔV on T_{bef} using M_{dist}
11: **for** each sorted POI $u \in P_{rem}$ **do**
12: $v \leftarrow$ a POI in P_{rem} such that $\Pi(u, v|T_{bef})$ has the longest distance among all $\Pi(u)$
13: **if** Property 1 is satisfied, i.e., only one disk $D(u, |\frac{\Pi(u,v|T_{bef})}{2}|)$ does not intersect with ΔF by checking $|\frac{\Pi(u,v|T_{bef})}{2}| < \min_{\forall w \in \Delta V} |\Pi(u, w|T_{bef})| - L_{max}$, where $|\Pi(u, w|T_{bef})|$ can be retrieved from M_{dist} **then**
14: $P_{rem} \leftarrow P_{rem} - \{u\}$
15: **else**
16: *OnePoiUpdate* (u, T_{aft}, G, P_{rem})
17: **return** updated G

(iii) **Path update for POI connecting to the path passing ΔF:** Lines 7–9. In Figure 4 (b), b is not on any face in ΔF, but $\Pi(b, g|T_{bef})$ and $\Pi(b, h|T_{bef})$ in $\Pi(u)$ pass ΔF, so Property 1 is not satisfied. We first use *OnePoiUpdate* (T_{aft}, G, b, P_{rem}) to update the purple paths on T_{aft}, and update G. Then, we remove $\Pi(b, c|T_{bef})$ in $\Pi(c)$ for each $X \in P_{rem}$, so $\Pi(a)$ and $\Pi(b)$ become empty, $\Pi(c) = \{\Pi(c, d|T_{bef}), \Pi(c, e|T_{bef}), \ldots, \Pi(c, h|T_{bef})\}$, $\ldots$, and $\Pi(h) = \{\Pi(h, c|T_{bef}), \Pi(h, d|T_{bef}), \ldots, \Pi(h, g|T_{bef})\}$. Finally, we remove b from P_{rem} to get $P_{rem} = \{c, d, e, f, g, h\}$.

(iv) **Path update for POI near ΔF:** Lines 10–16.

- In Figure 4 (c), the sorted POIs are h, f, e, d, c, g. We process h, and the path with the longest distance is $\Pi(c, h|T_{bef})$. Since Property 1 with the *minimum disk radius* is not satisfied, i.e., only one blue disk intersects with ΔF (determined by checking $|\frac{\Pi(h,c|T_{bef})}{2}| > |\Pi(h, j|T_{bef})| - L_{max}$ according to *efficient distance approximation* and *efficient disk and updated face intersection check*), we first use *OnePoiUpdate* (T_{aft}, G, h, P_{rem}) to update the purple paths on T_{aft}, and update G. Then, we remove $\Pi(h, X|T_{bef})$ in $\Pi(X)$ for each $X \in P_{rem}$, so $\Pi(a)$, $\Pi(b)$ and $\Pi(h)$ become empty, $\Pi(c) = \{\Pi(c, d|T_{bef}), \Pi(c, e|T_{bef}), \ldots, \Pi(c, g|T_{bef})\}$, $\ldots$, and $\Pi(g) = \{\Pi(g, c|T_{bef}), \Pi(g, d|T_{bef}), \ldots, \Pi(g, f|T_{bef})\}$. Finally, we remove h from P_{rem} to get $P_{rem} = \{c, d, e, f, g\}$.
- In Figure 4 (d), the sorted POIs are f, e, d, c, g. We process f, and the paths with the longest distance is $\Pi(c, f|T_{bef})$. Since Property 1 is satisfied, i.e., the blue disk does not intersect with ΔF (determined by checking $|\frac{\Pi(f,c|T_{bef})}{2}| < \min_{\forall v \in \Delta V} |\Pi(f, v|T_{bef})| - L_{max}$), we do not need to update the paths connect to f. We remove f from P_{rem} to get $P_{rem} = \{c, d, e, g\}$. Then, the sorted POIs are e, d, c, g, and

we process e similar to above.
- In Figure 4 (e), the case is also similar.

4) *Lemma:* We give three important lemmas as follows.

(i) **Necessity of storing the pairwise P2P exact shortest paths (i.e., G) on T_{bef}:** Let $U(A)$ be the *Update ratio* of an oracle A, which is defined as the number of POIs in P that need to be used as a source in algorithm *SSAD* (for path updating on T_{aft}) divided by the total number of POIs. In Figures 4 (a) to (c), we use algorithm *SSAD* with a, b, h as sources to update shortest paths on T_{aft} for *UP-Oracle*, and *WSPD-UP-Oracle*. In Figure 4 (d), *UP-Oracle* (resp. *WSPD-UP-Oracle*) calculates an *exact* (resp. *approximate*) path between c and f on T_{bef}. The disk radius centered at f is smaller (resp. larger), so the disk does not (resp. may) intersect with ΔF, and *UP-Oracle* does not need to (resp. *WSPD-UP-Oracle* may need to) use algorithm *SSAD* with f as source to update shortest paths on T_{aft}. The case also happens for the paths between c and e. In Figure 4 (e), the case also happens for the path between g and each POI in $\{c, d\}$. Thus, for *UP-Oracle* (resp. *WSPD-UP-Oracle*), we perform algorithm *SSAD* with three POIs a, b, h (resp. seven POIs a, b, c, d, e, f, g) as sources for path updating on T_{aft}. As there is a total of eight POIs, U(*UP-Oracle*) $= \frac{3}{8}$ (resp. U(*WSPD-UP-Oracle*) $= \frac{7}{8}$). The oracle update time of *WSPD-UP-Oracle* is 2.4 times larger than that of *UP-Oracle*. *RC-TIN-UP-Oracle* is similar to *WSPD-UP-Oracle*. Given an oracle A, a higher $U(A)$ means that the oracle update time of A is larger. Lemma 1 shows the necessity of storing G.

Lemma 1. *Given T_{bef}, T_{aft}, P, and an oracle A that does not store the pairwise P2P exact shortest paths on T_{bef}, U(UP-Oracle) $\leq U(A)$.*

Proof. By storing G, we can minimize the likelihood of updating the paths on T_{aft}, so U(*UP-Oracle*) is the smallest. □

(ii) **Correctness of the efficient distance approximation:** In the distance approximation, we have "$|\Pi(h, j|T_{bef})| - L_{max} \leq |\Pi(h, i|T_{aft})|$ due to the triangle inequality" in Figure 4 (c). Lemma 2 shows the correctness of this inequality, implying that the correctness of the efficient distance approximation. In Lemma 2, u can be h, any point on a face in ΔF can be i, and v can be j in Figure 4 (c).

Lemma 2. *The minimum distance from a POI u to any point on a face in ΔF on T_{aft} is at least $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{bef})| - L_{max}$.*

Proof Sketch. We show that the minimum distance from u to a point of e on T_{aft} is the same as on T_{bef}, where e is an edge that belongs to a face in ΔF, and that the exact shortest path from u to ΔF intersects with any point on e for the first time. Then, we use the triangle inequality to prove it. □

(iii) **Correctness of the efficient disk and updated face intersection check:** In the intersection check, we just need to create one blue disk $D(h, r)$ (where $r = \frac{|\Pi(h,c|T_{bef})|}{2}$ is half of the longest distance of the paths between h and each POI in $\{c, d, e, f, g\}$), and determine whether it intersects with ΔF in Figure 4 (c), instead of creating ten disks. Lemma 3 shows the correctness of this check. The disk in Lemma 3 can be $D(h, \frac{|\Pi(h,c|T_{bef})|}{2})$ in Figure 4 (c).

Lemma 3. *If the disk, centered at u, with radius equal to half of the longest distance among all non-updated paths adjacent to u, intersects with ΔF, Property 1 is not satisfied, and we need to use algorithm SSAD to update all non-updated paths adjacent to u. Otherwise, Property 1 is satisfied, and there is no need to update shortest paths adjacent to u.*

Proof Sketch. If the disk with the largest radius intersects with ΔF, Property 1 is not satisfied, and there is no need to check other disks. If the disk with the largest radius and with the center closest to ΔF does not intersect with ΔF, then other disks cannot intersect with ΔF, and Property 1 is satisfied. □

4.3 Update Phase: Generate Sub-graph Step

UP-Oracle can efficiently reduce the output size due to our design in the *generate sub-graph* step (using algorithm *HGSpan*) of the update phase. Due to this, the output size of *UP-Oracle* is 44MB on a terrain surface with 0.5M faces and 500 POIs, but the value is 520MB for *WSPD-Oracle* and 416MB for *RC-TIN-Oracle*.

1) *Concept:* The hierarchy graph H is a graph that has a simpler structure than G', and it is used for efficiently generating G' using G. Let $Q_{G'}$ be a *group* of vertices in $G'.V$ with *group center* $v \in Q_{G'}$ and *radius* r, such that for every vertex $u \in Q_{G'}$, we have $|\Pi_{G'}(u, v|T_{aft})| \leq r$. A set of groups $Q^1_{G'}, Q^2_{G'}, \ldots, Q^k_{G'}$ is a group cover of G' if every vertex in $G'.V$ belongs to at least one group, where k is the number of groups. H can form a set of *groups* by regarding several vertices in G' that are close to each other as one vertex. As a result, H is an approximation of G'. Similar to G', let $H.E$ be the set of edges of H. Given a group $Q^i_{G'}$, let the *intra-edges* be the set of edges connecting the group center of $Q^i_{G'}$ to all other vertices in $Q^i_{G'}$, and let the *inter-edges* be the set of edges connecting two group centers. Given a pair of vertices u and v in $H.E$, let $e_H(u, v|T_{aft})$ be an (intra- or inter-) edge with a weight $|e_H(u, v|T_{aft})|$. Given a pair of group centers s and t, let $\Pi_H(s, t|T_{aft})$ be the shortest path between them in H that only consists of inter-edges. Figures 5 (a) and (b) show a G' and its corresponding H. There are three groups with centers c, e, g in H. The light blue paths are intra-edges and the purple paths are inter-edges. The shortest path $\Pi_H(c, g|T_{aft})$ consists of edges $e_H(c, e|T_{aft})$ and $e_H(e, g|T_{aft})$.

2) *Method:* We introduce algorithm *HGSpan* as follow.

(i) **Why algorithm *HGSpan* is efficient:** Given a complete graph G and ϵ, algorithm *HGSpan* sorts edges in G based on their length in ascending order, and we initialize sub-graph G' to be empty. For each sorted edge in $G.E$, e.g., $|e(c, h|T_{aft})|$ in Figure 5 (a), if $|\Pi_{G'}(c, h|T_{aft})| > (1 + \epsilon)|e(c, h|T_{aft})|$, it is inserted into G', where $|\Pi_{G'}(c, h|T_{aft})|$ is approximated by the distance between the group centers of c and h on H, i.e., $|\Pi_H(c, g|T_{aft})|$ in Figure 5 (b). It is calculated using Dijkstra's algorithm in $O(1)$ time. But, if we do not use H for approximation, we need to use Dijkstra's algorithm on G' to calculate $|\Pi_{G'}(c, h|T_{aft})|$ in $O(n \log n)$ time, as in algorithm *GSpan* [36]. We repeat this for all edges in G, and return G' as the output.

(ii) **How to construct H:** To construct H, we sort the edges in G based on their length in ascending order. We then divide them into $\log n$ intervals, where each interval contains edges with lengths in $(\frac{2^{i-1}D}{n}, \frac{2^i D}{n}]$ for $i \in [1, \log n]$ and D is the length of the longest edge in G. In Figure 5 (b), for the i-th iteration, we select c, e, g as group centers and create

groups with radius $\delta\frac{2^iD}{n}$, where $\delta = \frac{\sqrt{\epsilon^2+36\epsilon+36}-(\epsilon+6)}{24} \in (0, \frac{1}{2})$ since $\epsilon \in (0, \infty)$. We insert intra-edges (light blue paths) into H between each group center and vertices in this group, and we insert inter-edges (solid purple paths) into H between every two group centers such that the distances between them on G' are at most $\frac{2^iD}{n} + 2\delta\frac{2^iD}{n}$. Then, we use H to approximate G'. For each sorted edge in G with length in the current length interval, e.g., $e(c, h|T_{aft})$ in Figure 5 (a), if $|\Pi_{G'}(c, h|T_{aft})| > (1+\epsilon)|e(c, h|T_{aft})|$, where $|\Pi_{G'}(c, h|T_{aft})|$ is approximated by $|\Pi_H(c, g|T_{aft})|$, we insert this edge (dashed light blue path) into G' and also insert an inter-edge between the group centers of c and h, i.e., $e_H(c, g|T_{aft})$ (dashed purple path), into H. Having processed all edges in the current length interval, for the $(i+1)$-st iteration, we repeat the above process to re-construct H, so that H is a valid approximate graph of G'. But, in algorithm *SGSpan* [40], if we force it to use a complete graph as input, the radius of each group in H becomes 0, and it degenerates to algorithm *GSpan*. In algorithm *HGSpan*, we use a different process to construct H (i.e., we insert inter-edges into H at two different stages: one before processing edges in G, and another one during this process), and we set the radius of each group in H to exceed 0 (i.e., $\delta\frac{2^iD}{n} > 0$ since $\delta > 0$), to avoid the degeneration by sacrificing the output size.

3) *Algorithm:* Algorithm 3 details *HGSpan*.

Algorithm 3 *HGSpan* (G, ϵ)

Input: G and ϵ
Output: G' (a sub-graph of G)
1: $D \leftarrow$ the length of the longest edge in $G.E$
2: **for** each edge $e(u, v|T_{aft}) \in G.E$ **do**
3: sort edge length in increasing order
4: create intervals $I_0 = (0, \frac{D}{N}]$, $I_i = (\frac{2^{i-1}D}{n}, \frac{2^iD}{n}]$ for $i \in [1, \log n]$
5: $G.E^i \leftarrow$ sorted edges of $G.E$ with length in I_i
6: $G'.E \leftarrow G.E^0$
7: **for** $i \leftarrow 1$ to $\log n$ **do**
8: $H.E \leftarrow \emptyset$
9: **for** each $u_j \in G.V$ that has not been visited **do**
10: perform Dijkstra's algorithm on G', such that the algorithm never visits vertices further than $\delta\frac{2^iD}{n}$ from u_j
11: create group $Q^j_{G'} \leftarrow \{u_j\}$ with group center u_j, $u_j \leftarrow$ *visited*
12: **for** each $v \in G.V$ such that $|\Pi_{G'}(u_j, v|T_{aft})| \leq \delta\frac{2^iD}{n}$ **do**
13: $Q^j_{G'} \leftarrow \{v\}$, $v \leftarrow$ *visited*
14: H intra-edges $\leftarrow H.E \cup \{e_H(u_j, v|T_{aft})\}$, where $|e_H(u_j, v|T_{aft})| = |\Pi_{G'}(u_j, v|T_{aft})|$
15: $j \leftarrow j + 1$
16: **for** each group center u_j **do**
17: perform Dijkstra's algorithm on G', such that the algorithm never visits vertices further than $\frac{2^iD}{n} + 2\delta\frac{2^iD}{n}$ from u_j
18: H inter-edges $\leftarrow H.E \cup \{e_H(u_j, u|T_{aft})\}$, where u is other group centers and $|e_H(u_j, u|T_{aft})| = |\Pi_{G'}(u_j, u|T_{aft})|$
19: $j \leftarrow j + 1$
20: **for** each edge $e(u, v|T_{aft}) \in G.E^i$ **do**
21: $w \leftarrow$ group center of u, $x \leftarrow$ group center of v
22: $\Pi_H(w, x|T_{aft}) \leftarrow$ the shortest path between w and x calculated using Dijkstra's algorithm on H
23: **if** $|\Pi_H(w, x|T_{aft})| > (1+\epsilon)|e(u, v|T_{aft})|$ **then**
24: $G'.E \leftarrow G'.E \cup \{e(u, v|T_{aft})\}$
25: H inter-edge $\leftarrow H.E \cup \{e_H(w, x|T_{aft})\}$, where $|e_H(w, x|T_{aft})| = |e_H(w, u|T_{aft})| + |e(u, v|T_{aft})| + |e_H(v, x|T_{aft})|$
26: **return** G'

4) *Example:* The following is an example of Algorithm 3.

(i) **Sort edge, split interval, and initialize G':** Lines 2–6. We insert edges of G with lengths in $I_0 = (0, \frac{D}{N}]$ into G'.

(ii) **Construct G':** Lines 7–25, let $i = 1$ and we clear H.

- Lines 9–15 (*construct group and insert intra-edge into H*): When $i = 1$, there is a limited number of edges in G', i.e., most pairs of vertices in $G'.V$ are not connected by an edge. It is likely that every vertex in $G'.V$ forms a group itself in H, and there are no intra-edges in H since each group only contains one vertex.
- Lines 16–19 (*insert first type inter-edge into H*): Similarly, it is likely that there are no inter-edges in H.
- Lines 20–25 (*insert edge into G' and second type inter-edge into H*): For each edge $e(u, v|T_{aft})$ in G with a weight in $I_1 = (\frac{D}{N}, \frac{2D}{N}]$, it is likely that the group center of u (resp. v) in H is u (resp. v) itself, i.e., $w = u$ and $x = v$. Since there is a limited number of edges in G', line 23 is likely true, and we insert $e(u, v|T_{aft})$ into G' and $e_H(w, x|T_{aft})$ into H.

(iii) **Continuing construct G':** Lines 7–25, we repeat the above process. Suppose that we start the i-th iteration and we clear H.

- Lines 9–15: Suppose that we have G' as shown in Figure 5 (a). Based on $G.V$, we create three groups with centers c, e, g in H, see Figure 5 (b). We insert intra-edges $e_H(a, c|T_{aft})$, ..., $e_H(g, i|T_{aft})$ (light blue paths) into H in Figure 5 (b).
- Lines 16–19: We insert inter-edges $e_H(c, e|T_{aft})$ and $e_H(e, g|T_{aft})$ (solid purple paths) into H, see Figure 5 (b).
- Lines 20–25: Suppose that we need to examine edge $e(c, h|T_{aft})$ in G in Figure 5 (a) with a weight in $I_i = (\frac{2^{i-1}D}{n}, \frac{2^iD}{n}]$, and the group center of c (resp. h) in H is c (resp. g). Then, we check whether $|\Pi_H(c, g|T_{aft})| > (1+\epsilon)|e(c, h|T_{aft})|$. If so, we insert edge $e(c, h|T_{aft})$ (dashed light blue path) into G', and insert inter-edge $e_H(c, g|T_{aft})$ (dashed purple path) with a weight $|e(c, g|T_{aft})| + |e_H(g, h|T_{aft})|$ into H. Next, we repeat this by starting the $(i+1)$-st iteration and clear H. This way, we construct G'.

5) *Lemma:* Lemma 4 analyzes algorithm *HGSpan*.

Lemma 4. *The running time of HGSpan is $O(n \log^2 n)$. The output of HGSpan, i.e., G', satisfies $|\Pi_{G'}(u, v|T_{aft})| \leq (1+\epsilon)|\Pi(u, v|T_{aft})|$ for all pairs of vertices u and v in $G'.V$.*

Proof Sketch. The *running time* includes (1) the $O(n)$ time to sort edge, split interval, and initialize G' due to n vertices in G, and (2) the $O(n \log^2 n)$ time to construct G' due to total $\log n$ intervals and $O(n \log n)$ time needed for each interval. For the *error bound*, we use the same notation in Algorithm 3. Since H is a valid approximation of G', in lines 20–25 of Algorithm 3, when we check whether $|\Pi_H(w, x|T_{aft})| > (1+\epsilon)|e(u, v|T_{aft})|$, we are checking $|\Pi_{G'}(u, v|T_{aft})| > (1+\epsilon)|e(u, v|T_{aft})|$. For any edge $e(u, v|T_{aft}) \in G'.E$ that is not inserted to G', we know $|\Pi_{G'}(u, v|T_{aft})| \leq (1+\epsilon)|e(u, v|T_{aft})|$. Since $|e(u, v|T_{aft})| = |\Pi(u, v|T_{aft})|$, we have $|\Pi_{G'}(u, v|T_{aft})| \leq (1+\epsilon)|\Pi(u, v|T_{aft})|$. □

4.4 Handling Subsequent Changes

So far, we have handled a single change. After one change, we have the updated G and the sub-graph G'. There is no old G, since we update G partially by using the new paths on T_{bef} to replace the original paths on T_{aft}. We keep G in the hard disk and use G' for shortest paths queries. To adapt *UP-Oracle* to handle subsequent changes, we also update

M_{dist} simultaneously when using algorithm *SSAD* for path updating in the *update exact shortest path* step of the update phase. Then, if subsequent changes occur, we update G and M_{dist}, and generate G' to support querying.

4.5 Adaption to Multi-layer Structure (*UP-Oracle-MuLa*)

We can adapt *UP-Oracle* to using a multi-layer structure, thus obtaining *UP-Oracle-MuLa*. Long-range queries utilize the approximate paths obtained from G', and short-range queries utilize the exact paths obtained from G with different *LODs*. Suppose that there are l *LODs*. The basic idea is to form temporary hierarchy graphs $H'_1, H'_2, \ldots, H'_l$ from G (similar to H, but H is constructed based on G', while $H'_1, H'_2, \ldots, H'_l$ are constructed based on G) with different group radii that correspond to different *LODs*. Then, we can regard the groups of each temporary hierarchy graph as short-range query regions. Specifically, to adapt *UP-Oracle* to become *UP-Oracle-MuLa*, we add one more step called *generate multi-layer structure* at the end of the update phase of *UP-Oracle*, and we add one more check in the query phase of *UP-Oracle*.

1) *Update phase:* In the *generate multi-layer structure* step of the update phase of *UP-Oracle-MuLa*, we construct a temporary hierarchy graph H'_i from G using a fixed set of group centers with radius $\frac{i \cdot D_{mean}}{2l}$ (where D_{mean} is the mean length of all edges in G) at each $LOD = i$.

(i) When $LOD = 1$, i.e., the most zoomed-in level, we build H'_1 using the *insert intra-edge and first type inter-edge* step of algorithm *HGSpan* with group radius $\frac{1 \cdot D_{mean}}{2l}$. For any pairs of POIs that belong to the same group in H'_1, we store their exact paths in a hash table M_1. We then store M_1 in a hash table M_{LOD} corresponding to $LOD = 1$.

(ii) When $LOD = i > 1$, we build H'_i using the same group centers as of H'_1 with group radius $\frac{i \cdot D_{mean}}{l}$. For any pair of POIs that belong to the same group in H'_i, we store their exact path in a hash table M_i, such that these paths do not exist in any previous tables $M_1, M_2, \ldots, M_{i-1}$. We then store M_i in M_{LOD} corresponding to $LOD = i$. We repeat this until $i = l$. Finally, M_{LOD} and G' are returned as the output.

Note that when $LOD = i > 1$, H'_{i-1} and H'_i have the same group centers, but the group radius of H'_i is larger than that of H'_{i-1}, so if a pair of POIs belong to the same group in H'_{i-1} (such that their corresponding exact path is stored in one of $M_1, M_2, \ldots, M_{i-1}$), they also belong to the same group in H'_i, and we do not need to store this exact path again in M_i. Thus, the sum of exact paths stored in M_{LOD} does not exceed the total number of exact paths in G.

2) *Query phase:* In the query phase of *UP-Oracle-MuLa*, given M_{LOD}, G', a *LOD* i, and a pair of POIs s and t in P, (i) if the exact path between s and t does not exist in $M_1, M_2, \ldots, M_i$ of M_{LOD}, we use Dijkstra's algorithm to find the path between them on G' using the query phase of *UP-Oracle*; (ii) otherwise, we simply return the exact paths.

4.6 Adaption to the A2A Query (*UP-Oracle-A2A*)

We can adapt *UP-Oracle* to be *UP-Oracle-A2A* for the A2A query. We first place Steiner points on T_{bef} using the method in study [43], and then use them as input (not the POIs) to construct *UP-Oracle-A2A* (as of *UP-Oracle*). When T_{bef} changes to T_{aft}, the positions of Steiner points (based on T_{aft}) also change, we update *UP-Oracle-A2A* using these Steiner points accordingly. For the query phase, given arbitrary point s (resp. t) on face f_s (resp. f_t), we let $\mathcal{S}(s)$ (resp. $\mathcal{S}(t)$) be a set of Steiner points on f_s (resp. f_t) and its adjacent faces [43]. Then, we return $\Pi_{G'}(s, t|T_{aft})$ in *UP-Oracle-A2A* (which has the same definition in *UP-Oracle*) by concatenating $\Pi(s, p|T_{aft})$, $\Pi_{G'}(p, q|T_{aft})$, and $\Pi(q, t|T_{aft})$ such that $|\Pi_{G'}(s, t|T_{aft})| = \min_{p \in \mathcal{S}(s), q \in \mathcal{S}(t)}[|\Pi(s, p|T_{aft})| + |\Pi_{G'}(p, q|T_{aft})| + |\Pi(q, t|T_{aft})|]$, where $|\Pi(s, p|T_{aft})|$ and $|\Pi(q, t|T_{aft})|$ can be calculated in $O(1)$ time using algorithm *SSAD* and $|\Pi_{G'}(p, q|T_{aft})|$ is distance of the path between p and q returned by *UP-Oracle-A2A*.

4.7 Theoretical Analysis

Theorem 1 analyzes *UP-Oracle* and its two adaptions.

Theorem 1. *The oracle construction time, oracle update time, output size, and shortest path query time of (1) UP-Oracle and (2) UP-Oracle-MuLa are both $O(nN^2)$, $O(N^2 + n \log^2 n)$, $O(n)$, and $O(\log n)$, and (3) UP-Oracle-A2A are $O(\frac{N^3}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon})$, $O(N^2 + \frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon} \log^2(\frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon}))$, $O(\frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon})$, and $O(\log(\frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon}))$, respectively. (1) UP-Oracle, (2) UP-Oracle-MuLa, and (3) UP-Oracle-A2A satisfy $|\Pi_{G'}(s, t|T_{aft})| \leq (1+\epsilon)|\Pi(s, t|T_{aft})|$ for all pairs of (1 & 2) POIs s and t in P, and (3) points s and t on T_{aft}, respectively.*

Proof Sketch. We first discuss *UP-Oracle*. The *oracle construction time* includes the $O(nN^2)$ time to calculate the pairwise P2P exact shortest paths time due to total n POIs and the use of algorithm *SSAD* in $O(N^2)$ time for each POI. The *oracle update time* includes (1) $O(N)$ time to detect updated terrain surface due to $O(N)$ faces, (2) $O(N^2)$ time to update exact shortest paths due to total $O(1)$ updated POIs and the use of algorithm *SSAD* in $O(N^2)$ time for each POI, (3) and $O(n \log^2 n)$ time to generate sub-graph time due to algorithm *HGSpan* in Lemma 4. The *output size* is $O(n)$ due to the output graph size of algorithm *HGSpan*. The *shortest path query time* is $O(\log n)$ due to the use of Dijkstra's algorithm on G' (in our experiments, G' has a constant number of edges and n vertices). The *error bound* is due to algorithm *HGSpan*'s error.

We then discuss *UP-Oracle-MuLa*. The *oracle construction time* is the same as of *UP-Oracle*. The *oracle update time* includes the oracle update time in *UP-Oracle*, and also the $O(n \log n)$ time to generate multi-layer structure due to total $O(1)$ temporary hierarchy graphs and the $O(n \log n)$ time needed for constructing each graph. The *output size* is $O(n)$ due to the output graph size of algorithm *HGSpan* and the $O(n)$ size of M_{LOD}. The experimental *shortest path query time* and *error bound* are better than those of *UP-Oracle* since *UP-Oracle-MuLa* stores some exact paths in M_{LOD}, but the theoretical time, and error are the same as of *UP-Oracle*.

We then discuss *UP-Oracle-A2A*. Since there is a total of $\frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon}$ Steiner points [43], we use this value to substitute n in *UP-Oracle* to obtain the new *oracle construction time*, *oracle update time*, *output size*, and *shortest path query time*. The *error bound* is due to the error bound of *UP-Oracle* and the proof in study [43]. □

Since the adapted *UP-Oracle* for subsequent changes has the same update phase as *UP-Oracle*, they share the same complexity analysis.

TABLE 2
Real earthquake terrain datasets

Name	Magnitude	Date
Tohoky, Japan (TJ) [45]	9.0	Mar 11, 2011
Sichuan, China (SC) [15]	8.0	May 12, 2008
Gujarat, India (GI) [46]	7.6	Jan 26, 2001
Alaska, USA (AU) [47]	7.1	Nov 30, 2018
Leogane, Haiti (LH) [48]	7.0	Jan 12, 2010
Valais, Switzerland (VS) [13]	4.1	Oct 24, 2016

TABLE 3
Comparison of algorithms

Algorithm	Oracle construction time	Oracle update time	Output size	Shortest path query time
Oracle-based algorithm				
WSPD-Oracle [9], [10]	Large	Large	Large	Small
WSPD-UP-Oracle [9], [10]	Large	Large	Small	Small
EAR-Oracle [5]	Large	Large	Large	Medium
EAR-UP-Oracle [5]	Large	Large	Small	Small
RC-TIN-Oracle [28]	Small	Large	Medium	Small
RC-TIN-UP-Oracle [28]	Small	Large	Small	Small
***UP-Oracle* (ours)**	**Small**	**Small**	**Small**	**Small**
On-the-fly algorithm				
WAV-Fly-Algo [24], [25]	N/A	N/A	N/A	Large
ESP-Fly-Algo [6], [12]	N/A	N/A	N/A	Large

5 Empirical Study

5.1 Experimental Setup

We conduct experiments on a Linux machine with a 2.20 GHz CPU and 512GB memory. All algorithms are implemented in C++. Our experimental setup generally follows the setups in the literature [6], [7], [8], [9], [10], [12].

1) *Datasets:* We conduct our experiments on 30 real before and after earthquake terrain datasets listed in Table 2 with 0.5M faces [1]. We obtain the earthquake terrain satellite maps with a 5km $\times$ 5km region from Google Earth with a resolution of 10m [8], [9], [10], [27], and then we use Blender [44] to generate the terrain model. To study the scalability, we follow an existing multi-resolution terrain dataset generation procedure [8], [9], [10] to obtain different resolutions of these datasets with 1M, 1.5M, 2M, 2.5M faces. This procedure appears in our technical report [38]. We extract 500 POIs using OpenStreetMap [9], [10].

2) *Algorithms:* We include the best-known exact on-the-fly algorithm *WAV-Fly-Algo* [24], [25], the best-known approximate on-the-fly algorithm *ESP-Fly-Algo* [6], [12], the best-known oracle *WSPD-Oracle* [9], [10] for the P2P query on terrain surfaces, its adaption *WSPD-UP-Oracle*, the best-known oracle *EAR-Oracle* [5] for the A2A query on terrain surfaces, its adaption *EAR-UP-Oracle*, the adapted oracle from point clouds to terrain surfaces *RC-TIN-UP-Oracle* [28] and its adaption *EAR-UP-Oracle* as baselines. In Table 3, we compare these algorithms with *UP-Oracle*. The comparisons of all algorithms (using big-O notation) can be found in our technical report [38]. Since the adapted *UP-Oracle* for subsequent changes has the same update phase as *UP-Oracle*, they have the same complexity and we omit the former oracle.

1. We upload the datasets at IEEE DataPort https://dx.doi.org/10.21227/7ras-ng51

3) *Query generation:* We randomly choose pairs of POIs in P for the P2P query, or arbitrary points on T_{aft} for the A2A query, and we report the average, minimum, and maximum results of 100 queries.

4) *Parameters and performance metrics:* We study the effect of three parameters, namely (i) ϵ, (ii) n, and (iii) dataset size DS (i.e., the number of faces in a terrain model). We consider six performance metrics, namely (i) *oracle construction time*, (ii) *oracle update time*, (iii) *oracle size* (i.e., the space usage of G, M_{dist}, and H), (iv) *output size* (i.e., the space usage of G'), (v) *shortest path query time*, and (vi) *distance error* (i.e., the error of the distance returned by the algorithm compared with the exact shortest distance).

5.2 Experimental Results

Our experiments show that *WSPD-Oracle*, *WSPD-UP-Oracle*, *EAR-Oracle*, *EAR-UP-Oracle*, *RC-TIN-Oracle*, and *RC-TIN-UP-Oracle* have excessive oracle update times with 500 POIs (more than 1 days), so we compare (1) all algorithms on 30 datasets with fewer POIs (50 by default), and (2) *UP-Oracle*, *WAV-Fly-Algo*, and *ESP-Fly-Algo* on 30 datasets with more POIs (500 by default). For the shortest path query time, the vertical bar and the points denote the minimum, maximum, and average results.

1) *Ablation study for the P2P query:* We consider 6 variations of *UP-Oracle*, i.e., (i) we use a random POI selection sequence, instead of using our optimal POI selection sequence, (ii) we use the full shortest distance of a shortest path as the disk radius, instead of using our minimum disk radius, (iii) we do not store the POI-to-vertex distance information and re-calculate the shortest path on T_{aft} for determining whether the disk intersects with ΔF, instead of using our efficient distance approximation, (iv) we create two disks for each path when checking whether we need to re-calculate the shortest path between a pair of POIs, instead of using our efficient disk and updated face intersection check, (v) we remove the generate sub-graph step, i.e., algorithm *HGSpan* in the update phase and use a hash table to store the pairwise P2P exact shortest paths on T_{aft} in G, and (vi) we use algorithm *GSpan* [36] or algorithm *SGSpan* [40] (degenerates to algorithm *GSpan* when the input is a complete graph), instead of using algorithm *HGSpan* in the generate sub-graph step of the update phase. We use *UP-Oracle-*$\boldsymbol{X}$ where $\boldsymbol{X} \in$ {*RanSelSeq, FullRad, NoDistAppr, NoEffIntChe, NoEdgPru, NoEffEdgPru*} to denote these variations. The first four oracles correspond to the four techniques in Section 4.2. The last two oracles correspond to the idea covered in Section 4.3.

In Figure 6 (resp. Figure 7), we test the 5 values of n in {50, 100, 150, 200, 250} on *TJ* (resp. {500, 1000, 1500, 2000, 2500} on *SC*) dataset while fixing ϵ at 0.1 and DS at 0.5M (resp. ϵ to 0.25 and DS to 0.5M) for the ablation study involving 6 variations (resp. the last 3 variations, since the first 3 variations have excessive oracle update times with 500 POIs) and *UP-Oracle*. The oracle update time for *UP-Oracle-*$\boldsymbol{X}$, where $\boldsymbol{X} \in$ {*RanSelSeq, FullRad, NoDistAppr, NoEffIntChe, NoEffEdgPru*} exceeds that of *UP-Oracle* due to the four techniques from Section 4.2 and the use of algorithm *HGSpan* from Section 4.3. Although the oracle update time and the shortest path query time of *UP-Oracle-NoEdgPru* are slightly smaller than those of *UP-Oracle*, the

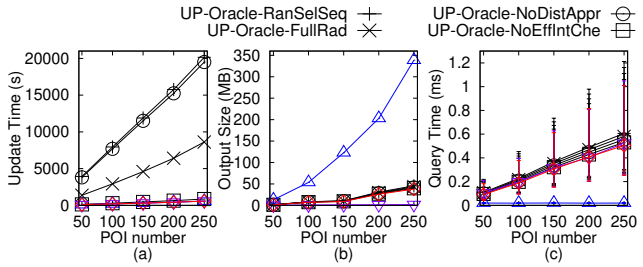

Fig. 6. Ablation study on *TJ* dataset with fewer POIs for the P2P query

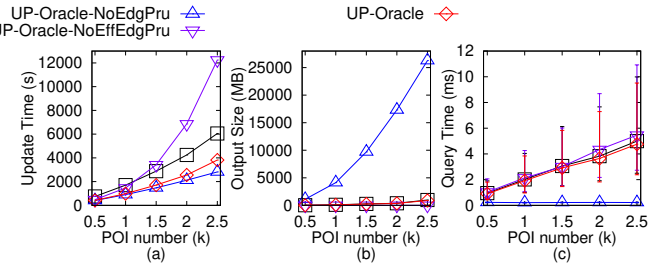

Fig. 7. Ablation study on *SC* dataset with more POIs for the P2P query

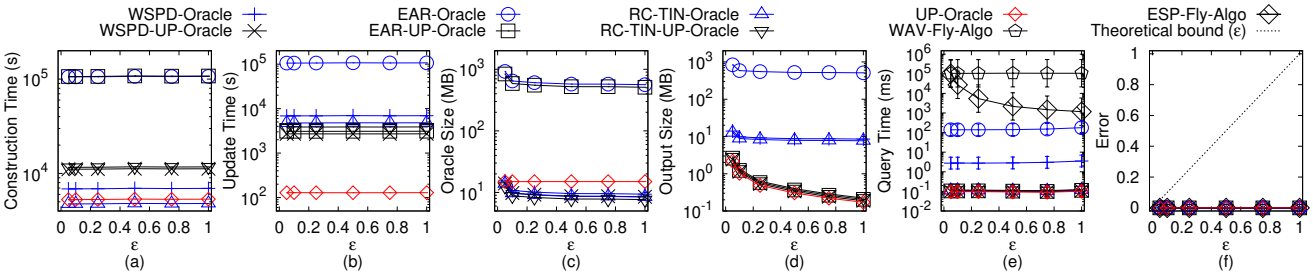

Fig. 8. Baseline comparisons (effect of ϵ on *GI* dataset with fewer POIs) for the P2P query

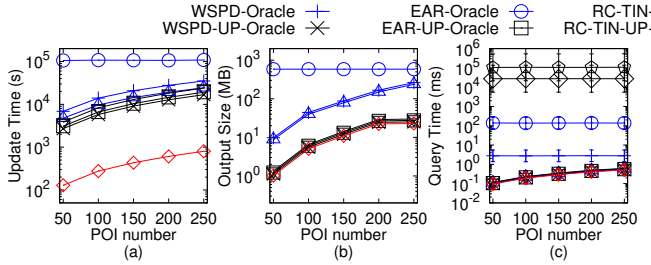

Fig. 9. Baseline comparisons (effect of n on *AU* dataset with fewer POIs) for the P2P query

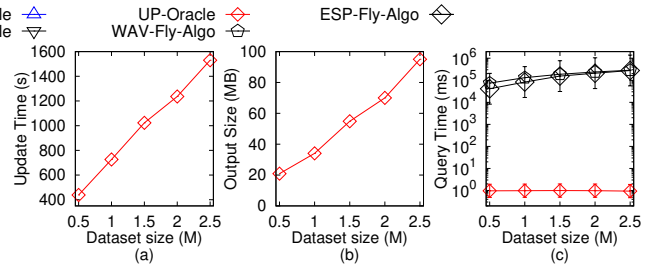

Fig. 10. Scalability test (effect of *DS* on *LH* dataset with more POIs) for the P2P query

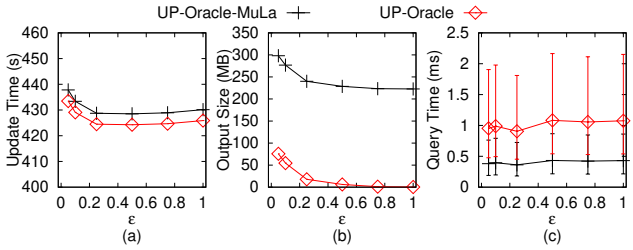

Fig. 11. Multi-layer structure comparison on *SC* dataset

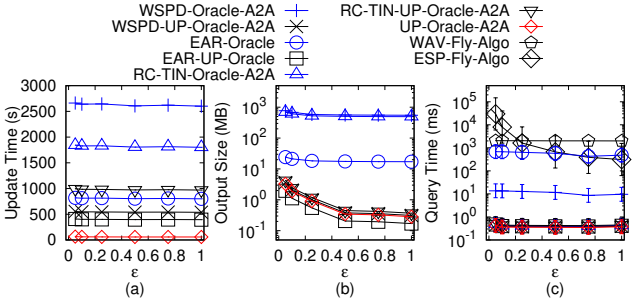

Fig. 12. A2A query on *SC* dataset

output size for *UP-Oracle-NoEdgPru* is 10^4 times due to the use of algorithm *HGSpan*. Thus, *UP-Oracle* is the best oracle among the variations.

2) *Baseline comparisons for the P2P query:* We proceed to compare different baselines with *UP-Oracle*.

Effect of ϵ: In Figure 8, we test the 6 values of ϵ in $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on *GI* dataset with fewer POIs while fixing n at 50 and *DS* at 0.5M. Although all algorithms have errors close to 0%, *UP-Oracle* offers superior performance over other baselines in terms of the oracle construction time, oracle update time, output size, and shortest path query time due to the *non-updated terrain shortest path intact* property, the stored pairwise P2P exact shortest paths on T_{bef}, and the use of algorithm *HGSpan* in *UP-Oracle*. Although the oracle size of *UP-Oracle* is slightly larger than those of *WSPD-Oracle*, *WSPD-UP-Oracle*, *RC-TIN-Oracle*, and *RC-TIN-UP-Oracle*, the oracle update time of *UP-Oracle* is 88 times, 21 times, 70 times, and 17 times smaller, respectively. Varying ϵ has (i) no impact on the oracle construction time of *UP-Oracle* since it is independent of ϵ, (ii) a small impact on the oracle update time of *UP-Oracle*, since when n is small, the update exact shortest path step dominates the generate sub-graph step, and the former step is independent of ϵ, and (iii) a small impact on the oracle construction time

and oracle update time of other oracles since their early termination criteria of using algorithm *SSAD* are loose (i.e., they need to use algorithm *SSAD* to cover most of the POIs or highway nodes as destinations even when ϵ is large).

Effect of n: In Figure 9, we test the 5 values of n in {50, 100, 150, 200, 250} on *AU* dataset while fixing ϵ at 0.1 (we also have the results with 5 values of n in {500, 1000, 1500, 2000, 2500} while fixing ϵ at 0.25 in our technical report [38]) and *DS* at 0.5M. The oracle update time, output size, and shortest path query time of *UP-Oracle* remain better than those of the baselines. Specifically, the oracle update time of *UP-Oracle* is 21 times, 23 times, and 17 times smaller than those of *WSPD-UP-Oracle*, *EAR-UP-Oracle*, and *RC-TIN-UP-Oracle*, respectively. Since *WSPD-UP-Oracle*, *EAR-UP-Oracle*, and *RC-TIN-UP-Oracle* have output graph G' (which is similar to *UP-Oracle*), their output size and shortest path query time are similar to those of *UP-Oracle*.

3) *Scalability test for the P2P query* (effect of *DS*): In Figure 10, we test 5 values of *DS* in {0.5M, 1M, 1.5M, 2M, 2.5M} on *LH* dataset with more POIs while fixing ϵ at 0.25 and n at 500. *UP-Oracle* can scale up to a large dataset with 2.5M points. Since *UP-Oracle* is an oracle, its shortest path query time is 10^5 times smaller than that of *ESP-Fly-Algo*.

4) *Multi-layer structure:* In Figure 11, we compare *UP-Oracle*, and *UP-Oracle-MuLa* by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and fixing n at 500 and *DS* at 0.5M on *SC* dataset. The oracle update time, output size, and shortest path query time of *UP-Oracle-MuLa* are 1.1 times larger, 10 times larger, and 2 times smaller than those of *UP-Oracle*, since *UP-Oracle-MuLa* uses M_{LOD} to store some exact paths to accelerate shortest-range queries.

5) *A2A query:* In Figure 12, we test the A2A query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} while fixing *DS* at 2k on a multi-resolution of *SC* dataset. We adapt *WSPD-Oracle*, *WSPD-UP-Oracle*, *RC-TIN-Oracle*, and *RC-TIN-UP-Oracle* that answer the P2P query in a similar way to *UP-Oracle-A2A*, and denote them as *WSPD-Oracle-A2A*, *WSPD-UP-Oracle-A2A*, *RC-TIN-Oracle-A2A*, and *RC-TIN-UP-Oracle-A2A* (such that they can answer the A2A query). The oracle update time of *UP-Oracle-A2A* is 15 times better than the best-known oracle *EAR-Oracle* on terrain surfaces for the A2A query.

6) *Case study:* We conduct a case study on the 4.1 magnitude earthquake (which caused an avalanche) in Valais as mentioned in Section 1.1. In this case study, on a terrain surface with 0.5M faces and 250 POIs, *UP-Oracle* just needs 400s $\approx$ 6.7 min to update the oracle, but the best-known oracle *WSPD-Oracle* for the P2P query needs 35,100s $\approx$ 9.8 hours. Answering 100 paths takes 0.1s for *UP-Oracle*, 8,600s $\approx$ 2.4 hours for the best-known on-the-fly algorithm *ESP-Fly-Algo*, and 0.3s for *WSPD-Oracle*. Thus, only *UP-Oracle* is suitable for earthquake rescuing to save lives.

7) *Summary:* In terms of the oracle update time, output size, and shortest path query time, *UP-Oracle* is up to 88 times, 12 times, and 3 times (resp. 15 times, 50 times, and 100 times) better than the best-known oracle *WSPD-Oracle* for the P2P query (resp. *EAR-Oracle* for the A2A query) on terrain surfaces. (i) For the P2P query on a terrain dataset with 0.5M faces and 250 POIs, *UP-Oracle*'s oracle update time is 400s $\approx$ 6.7 min, while *WSPD-Oracle* and *RC-TIN-Oracle* take 35,100s $\approx$ 9.8 hours and 28,100s $\approx$ 7.5 hours. (ii) the shortest path query time for computing 100 paths is 0.1s for *UP-Oracle*, while the time is 8,600s $\approx$ 2.4 hours for *ESP-Fly-Algo*, 0.3s for *WSPD-Oracle*, and 0.1s for *RC-TIN-Oracle*. (iii) For the A2A query on a terrain dataset with 20k faces, the oracle update time and shortest path query time for computing 100 shortest paths of *UP-Oracle-A2A* are 480s $\approx$ 7 min and 0.05s, while the values are 7,100s $\approx$ 2 hours and 5s for *EAR-Oracle*.

6 Conclusion

We propose an efficient $(1+\epsilon)$-approximate shortest path oracle on an updated terrain surface called *UP-Oracle*, which has state-of-the-art performance in terms of the oracle update time, output size, and shortest path query time compared with the best-known oracle on terrain surfaces. In future work, it is of interest to explore new pruning steps in *UP-Oracle* to further reduce the oracle update time (e.g., it may be possible to reduce the likelihood of using algorithm *SSAD* when updating *UP-Oracle* by reducing the disk radius in the *non-updated terrain shortest path intact* property).

Acknowledgments

The research of Yinzhao Yan and Raymond Chi-Wing Wong is supported by GZSTI16EG24. The research of Christian S. Jensen is supported in part by the Innovation Fund Denmark project DIREC (9142-00001B).

References

[1] S. Xing, C. Shahabi, and B. Pan, "Continuous monitoring of nearest neighbors on land surface," *VLDB*, vol. 2, no. 1, pp. 1114–1125, 2009.
[2] "Metaverse," 2023. [Online]. Available: https://about.facebook.com/meta
[3] "Google Earth," 2023. [Online]. Available: https://earth.google.com/web
[4] K. Deng, H. T. Shen, K. Xu, and X. Lin, "Surface kNN query processing," *ICDE*, pp. 78–78, 2006.
[5] B. Huang, V. J. Wei, R. C.-W. Wong, and B. Tang, "EAR-oracle: on efficient indexing for distance queries between arbitrary points on terrain surface," *SIGMOD*, vol. 1, no. 1, pp. 1–26, 2023.
[6] M. Kaul, R. C.-W. Wong, and C. S. Jensen, "New lower and upper bounds for shortest distance queries on terrains," *VLDB*, vol. 9, no. 3, pp. 168–179, 2015.
[7] M. Kaul, R. C.-W. Wong, B. Yang, and C. S. Jensen, "Finding shortest paths on terrains by killing two birds with one stone," *VLDB*, vol. 7, no. 1, pp. 73–84, 2013.
[8] L. Liu and R. C.-W. Wong, "Finding shortest path on land surface," *SIGMOD*, pp. 433–444, 2011.
[9] V. J. Wei, R. C.-W. Wong, C. Long, D. Mount, and H. Samet, "Proximity queries on terrain surface," *TODS*, vol. 47, no. 4, pp. 1–59, 2022.
[10] V. J. Wei, R. C.-W. Wong, C. Long, and D. M. Mount, "Distance oracle on terrain surface," *SIGMOD*, pp. 1211–1226, 2017.
[11] Y. Yan and R. C.-W. Wong, "Path Advisor: a multi-functional campus map tool for shortest path," *VLDB*, vol. 14, no. 12, pp. 2683–2686, 2021.
[12] ——, "Efficient shortest path queries on 3D weighted terrain surfaces for moving objects," *MDM*, 2024.
[13] "Moderate mag. 4.1 earthquake - 6.3 km northeast of Sierre, Valais, Switzerland," 2023. [Online]. Available: https://www.volcanodiscovery.com/earthquakes/quake-info/1451397/mag4quake-Oct-24-2016-Leukerbad-VS.html
[14] "Turkey–Syria earthquakes 2023," 2023. [Online]. Available: https://www.bbc.com/news/topics/cq0zxdd0y39t
[15] K. Pletcher and J. P. Rafferty, "Sichuan earthquake of 2008," 2023. [Online]. Available: https://www.britannica.com/event/Sichuan-earthquake-of-2008

[16] H. Li and Z. Huang, "82 die in Sichuan quake, rescuers race against time to save lives," 2022. [Online]. Available: https://www.chinadailyhk.com/article/289413#82-die-in-Sichuan-quake-rescuers-race-against-time-to-save-lives

[17] J. E. Nichol, A. Shaker, and M.-S. Wong, "Application of high-resolution stereo satellite images to detailed landslide hazard assessment," *Geomorphology*, vol. 76, no. 1-2, pp. 68–75, 2006.

[18] A. Annis, F. Nardi, A. Petroselli, C. Apollonio, E. Arcangeletti, F. Tauro, C. Belli, R. Bianconi, and S. Grimaldi, "UAV-DEMs for small-scale flood hazard mapping," *Water*, vol. 12, no. 6, p. 1717, 2020.

[19] T. Kawamura, J. F. Clinton, G. Zenhäusern, S. Ceylan, A. C. Horleston, N. L. Dahmen, C. Duran, D. Kim, M. Plasman, S. C. Stähler *et al.*, "S1222a—the largest marsquake detected by insight," *Geophysical research letters*, vol. 50, no. 5, p. e2022GL101543, 2023.

[20] "NASA Mars exploration," 2023. [Online]. Available: https://mars.nasa.gov

[21] N. McCarthy, "Exploring the red planet is a costly undertaking," 2021. [Online]. Available: https://www.statista.com/chart/24232/life-cycle-costs-of-mars-missions/

[22] S. Pan and M. Li, "Construction of earthquake rescue model based on hierarchical voronoi diagram," *Mathematical problems in engineering*, vol. 2020, pp. 1–13, 2020.

[23] S. Kapoor, "Efficient computation of geodesic shortest paths," in *ACM symposium on theory of computing*, 1999, pp. 770–779.

[24] V. J. Wei, R. C.-W. Wong, C. Long, D. M. Mount, and H. Samet, "On efficient shortest path computation on terrain surface: A direction-oriented approach," *TKDE*, no. 1, pp. 1–14, 2024.

[25] J. Chen and Y. Han, "Shortest paths on a polyhedron," in *Symposium on computational geometry*, 1990, p. 360–369.

[26] S.-Q. Xin and G.-J. Wang, "Improving Chen and Han's algorithm on the discrete geodesic problem," *ACM Transactions on graphics*, vol. 28, no. 4, pp. 1–8, 2009.

[27] C. Shahabi, L.-A. Tang, and S. Xing, "Indexing land surface for efficient kNN query," *VLDB*, vol. 1, no. 1, pp. 1020–1031, 2008.

[28] Y. Yan and R. C.-W. Wong, "Proximity queries on point clouds using rapid construction path oracle," *SIGMOD*, vol. 2, no. 1, pp. 1–26, 2024.

[29] Y. Hong and J. Liang, "Excavators used to dig out rescue path on cliff in earthquake-hit Luding of sw China's Sichuan," *People's daily misc*, 2022. [Online]. Available: http://en.people.cn/n3/2022/0909/c90000-10145381.html

[30] "Mars 2020 mission perseverance rover brains," 2023. [Online]. Available: https://mars.nasa.gov/mars2020/spacecraft/rover/brains/

[31] "NASA's self-driving perseverance mars rover 'takes the wheel'," 2021. [Online]. Available: https://www.nasa.gov/solar-system/nasas-self-driving-perseverance-mars-rover-takes-the-wheel/

[32] "Mars 2020 mission perseverance rover communications," 2023. [Online]. Available: https://www.statista.com/chart/24232/life-cycle-costs-of-mars-missions/

[33] J. A. Crisp, M. Adler, J. R. Matijevic, S. W. Squyres, R. E. Arvidson, and D. M. Kass, "Mars exploration rover mission," *Journal of geophysical research: planets*, vol. 108, no. E12, 2003.

[34] D. Peleg and J. D. Ullman, "An optimal synchronizer for the hypercube," in *ACM Symposium on principles of distributed computing*, 1987, pp. 77–85.

[35] H. Shpungin and M. Segal, "Near-optimal multicriteria spanner constructions in wireless ad hoc networks," *IEEE/ACM Transactions on networking*, vol. 18, no. 6, pp. 1963–1976, 2010.

[36] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares, "On sparse spanners of weighted graphs," *Discrete & computational geometry*, vol. 9, no. 1, pp. 81–100, 1993.

[37] B. Padlewska, "Connected spaces," *Formalized mathematics*, vol. 1, no. 1, pp. 239–244, 1990.

[38] Y. Yan, R. C.-W. Wong, and C. S. Jensen, "An efficiently updatable path oracle for terrain surfaces (technical report)," 2023. [Online]. Available: https://github.com/yanyinzhao/UpdatedStructureTerrainCode/blob/master/TechnicalReport.pdf

[39] M. Fan, H. Qiao, and B. Zhang, "Intrinsic dimension estimation of manifolds by incising balls," *Pattern recognition*, vol. 42, no. 5, pp. 780–787, 2009.

[40] G. Das and G. Narasimhan, "A fast algorithm for constructing sparse Euclidean spanners," in *Symposium on computational geometry*, 1994, pp. 132–139.

[41] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[42] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[43] H. N. Djidjev and C. Sommer, "Approximate distance queries for weighted polyhedral surfaces," in *Proceedings of the European Symposium on Algorithms*, 2011, pp. 579–590.

[44] "Blender," 2023. [Online]. Available: https://www.blender.org

[45] "Mar 11, 2011: Tohoku earthquake and Tsunami," 2023. [Online]. Available: https://education.nationalgeographic.org/resource/tohoku-earthquake-and-tsunami/

[46] "Gujarat earthquake, 2001," 2023. [Online]. Available: https://www.actionaidindia.org/emergency/gujarat-earthquake-2001/

[47] "2018 Anchorage earthquake," 2023. [Online]. Available: https://www.usgs.gov/news/featured-story/2018-anchorage-earthquake

[48] R. Pallardy, "2010 Haiti earthquake," 2023. [Online]. Available: https://www.britannica.com/event/2010-Haiti-earthquake

Yinzhao Yan is a PhD candidate in Computer Science and Engineering of The Hong Kong University of Science and Technology supervised by Prof. Raymond Chi-Wing Wong. He obtained BSc degree in Computer Science in Hong Kong Baptist University in 2020. He published several papers as the first author in SIGMOD, VLDB, and MDM. He received MDM 2024 Best Paper Award. His research interest is spatial databases.

Raymond Chi-Wing Wong is a Professor in Computer Science and Engineering of The Hong Kong University of Science and Technology. He is currently the associate head of Department of Computer Science and Engineering and the director of Undergraduate Research Opportunities Program. He received the BSc, MPhil, and PhD degrees in Computer Science and Engineering in the Chinese University of Hong Kong in 2002, 2004, and 2008, respectively. He published over 160 papers (e.g., SIGMOD, VLDB, ICDE, TODS, TKDE, and VLDB journal). His research interests include database and data mining.

Christian S. Jensen is a Professor in Computer Science of Aalborg University, Denmark. He is an ACM and an IEEE fellow and a member of the Academia Europaea, the Royal Danish Academy of Sciences and Letters, and the Danish Academy of Technical Sciences. He has received several national and international awards for his research, most recently the 2019 IEEE TCDE Impact Award and the 2022 ACM SIGMOD Contributions Award. His research concerns data analytics and management with focus on temporal and spatio-temporal data.