Fast Update Path Oracle on Updated Terrain Surfaces

Yinzhao Yan
The Hong Kong University of Science and Technology
yyanas@cse.ust.hk

Raymond Chi-Wing Wong
The Hong Kong University of Science and Technology
raywong@cse.ust.hk

Christian S. Jensen
Aalborg University
csj@cs.aau.dk

***Abstract*—The booming of computer graphics technology and geo-spatial positioning 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 efficient $(1+\epsilon)$-approximate shortest path oracle on an updated terrain surface. This oracle is capable of improved performance in terms of oracle construction time, oracle update time, output size, and shortest path query time due to the concise information it maintains about the shortest paths between all pairs of points-of-interest stored in the oracle. Our empirical study shows that in realistic settings, when compared to the best-known existing oracle, our oracle is capable of improvements in oracle construction time and oracle update time of up to 1.3 times and 88 times, and of improvements in output size and shortest path query time of up to 12 times and 3 times.**

I. Introduction

It is increasingly important to calculate the shortest paths on terrain surfaces [57]. Well-known companies and applications, including Metaverse [11], Google Earth [6], and Cyberpunk 2077 (a 3D computer game) [4], all rely on the ability to find the shortest paths on terrain surfaces (e.g., Earth or in virtual reality) to assist users to reach destinations more quickly. In academia, shortest path querying on terrain surfaces also attracts considerable attention [26], [32], [35], [36], [39], [41], [54], [55], [58], [59]. A terrain surface is represented by a set of *faces*, each of which is captured by a triangle. A face thus consists of three line segments, called *edges*, connected with each other at three *vertices*. Figure 1 (a) shows an example of a terrain surface consisting of vertices, edges, and faces.

A. Motivation

1) ***Updated terrain surface****:* Being able to compute the shortest paths on *updated* terrain surfaces is critical.

(1a) **Earthquake**: We aim at finding the shortest rescue paths for life-saving after an earthquake. The death toll of the 7.8 magnitude earthquake on February 6, 2023 in Turkey and Syria exceeded 40,000 [15], and more than 69,000 died in the 7.9 magnitude earthquake on May 12, 2008 in Sichuan, China [49]. Annually, there are 15 earthquakes with magnitude 7 and one with 8 or higher on average [16]. The rescue team can save 3 lives every 15 minutes [40], so early arrival at the sites of the quake was of essence. In practice, (1) satellites or (2) drones can be used to collect the terrain surface after an earthquake, which takes (1) 10s and USD \$48.72 [44], and (2) 144s $\approx$ 2.4 min and USD \$100 [20] for a 1km^2 region, respectively, which are both time-efficient and cost-efficient.

(1b) **Avalanche**: Earthquakes may also cause avalanches. The 4.1 magnitude earthquake on October 24, 2016 in Valais, Switzerland [12] causes an avalanche: Figure 3 (a) and (b) (resp. Figure 3 (c) and (d)) shows the original and new shortest paths between a and c, and b and d on a real map (resp. a terrain model) before and after terrain surface updates, where a and b are villages, c and d are hotels. We need to efficiently calculate the new shortest paths for recusing.

(1c) **Marsquake**: As observed by NASA's InSight lander on May 4, 2022 [37], Mars also experienced a marsquake. For NASA's Mars exploration project [13] (with cost USD 2.5 billion [42]), China National Space Administration's Mars mission project [3] (with an annual budget of USD 8.9 billion [50]), and the SpaceX Mars project [14] (with cost USD 67 million per launch [5]), it is essential that Mars rovers can find the shortest escape paths quickly and autonomously in regions affected by marsquakes to avoid damage[1].

2) ***POIs****:* Given a set of *points-of-interest (POIs)* on a terrain surface, computing the shortest path between *pairs of POIs*, i.e., *POI-to-POI (P2P) path query*, is important. For the earthquake and avalanche, POIs can be villages waiting for rescuing [47], hospitals, and expressway exits. For the Marsquake, POIs can be Mars rover's working stations.

3) ***Oracle****:* Pre-computing the shortest paths on a terrain surface among POIs using an indexing technique, known as an *oracle*, is a critical step in achieving fast shortest path query time, especially when we need to calculate more than one shortest paths with different sources and destinations (where the time taken to pre-compute the oracle is called the *oracle construction time*, the time taken to update the oracle is called the *oracle update time*, the space complexity of the output

[1]A Mars rover costs USD 2.5 billion [17] and cannot be repaired remotely [33]. The round trip signal delay between Earth and Mars is 40 minutes [10], so it is impossible to pass terrain information captured by A Mars rover after a quake from Mars to Earth, ask human experts to find the shortest escape paths, and then pass the paths from Earth to Mars.

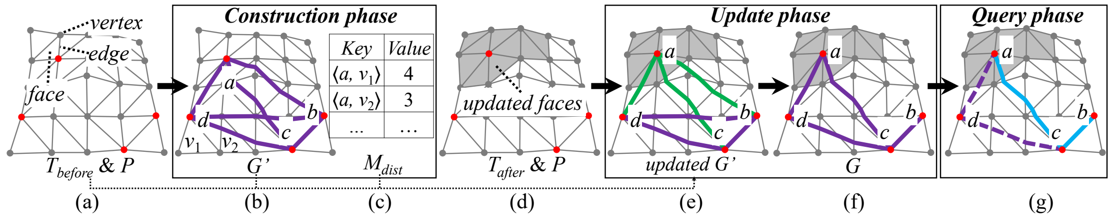

Fig. 1. Framework overview

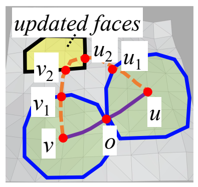

Fig. 2. An unaffected path

oracle is called the *output size*, and the time taken to return the result is called the *shortest path query time*). It is also important to update the oracle *quickly* when the terrain surface updates. In the earthquake, if we can pre-compute the shortest paths (among some villages and hospitals) using an *oracle* on terrain surfaces prone to earthquakes, and efficiently update the oracle after an earthquake, then we can use it to efficiently return shortest paths with different sources and destinations (e.g., villages, hospitals, and expressway exits).

B. 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], [34], [43], [56] on a terrain surface are slow when many shortest path queries are involved. The best-known exact algorithm [23] runs in $O(N^2)$. Although *approximate* algorithms [35], [36], [39], [41] were proposed for reducing the running time, they are still not efficient enough. The best-known approximate algorithm [35] runs in $O((N + N')\log(N + N'))$ time, where N' is the number of additional points introduced for the bound guarantee. Our experiments show that the best-known exact algorithm [23] (resp. approximate algorithm [35]) needs 116s (resp. 86s) to calculate one shortest path on a terrain surface with 0.5M faces, and computing 100 shortest paths takes 11,600s $\approx$ 3.2 hours (resp. 8,600s $\approx$ 2.4 hours).

*2) **Non-existence of oracles on updated terrain surfaces**:* Although existing studies [32], [54], [55] can construct oracles on *static* terrain surfaces, and can then return the P2P shortest path query results efficiently, no existing study can accommodate updated terrains, where the oracle needs to be updated efficiently. A straightforward adaptation of the best-known P2P path query oracle [54], [55] is to re-construct the oracle when the terrain surface is updated. However, its oracle construction time is $O(nN\log^2 N + cn)$, where n is the number of POIs on T and c is a constant depending on T (where $c \in [35, 80]$ on a terrain surface with 0.5M faces on average). In our experiments, its oracle construction time is 35,100s $\approx$ 9.8 hours for a terrain surface with 0.5M faces and 250 POIs, which is not acceptable.

C. Path Oracle on Updated Terrain Surfaces

We propose an efficient $(1 + \epsilon)$-approximate shortest path oracle on an updated terrain surface called Fast Update path Oracle (*FU-Oracle*), which has state-of-the-art performance in terms of oracle construction time, oracle update time, output size, and shortest path query time (compared with the best-known oracle [54], [55]) due to the concise information about pairwise shortest paths between any pair of POIs stored in the oracle, where ϵ is a non-negative real user parameter for controlling the error ratio, called the *error parameter*.

*1) **Key ideas for achieving a short oracle update time**:* Consider two terrain surfaces before and after updates, i.e., T_{before} and T_{after}, respectively. The key ideas of achieving a short oracle update time of *FU-Oracle* are due to (1) a novel property of *FU-Oracle*, called the *non-updated terrain shortest path intact* property, and (2) the stored pairwise P2P exact shortest paths on T_{before} when *FU-Oracle* is constructed.

(1a) ***Non-updated terrain shortest path intact* property**: In Figure 2, this property implies that given the purple path between u and v on T_{before} (with path distance d_1), if the distances from both u and v to the updated faces are large enough (i.e., both larger than $\frac{d_1}{2}$), then the path between u and v on T_{after} is the same, and we do not need to update it.

(1b) **Necessity of storing the pairwise P2P exact shortest paths on T_{before}**: To minimize the updates to *FU-Oracle* for it to accommodate T_{after}, we need to store the pairwise P2P *exact* shortest paths on T_{before} when *FU-Oracle* is constructed. This is because the *exact* shortest distances are no larger than the *approximate* shortest distances. Given an exact (resp. approximate) shortest path with two endpoints u and v on T_{before}, in the non-updated terrain shortest path intact property, 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 chances of updating this path on T_{after}.

*2) **Key idea for efficiently achieving a small output size**:* We are not interested in returning the *pairwise* P2P exact shortest paths on T_{after} as the oracle output.

(2a) **Earthquake and avalanche**: In Figure 1 (f), given three POIs a, b, and c, suppose that a is a damaged village, b and c are unaffected hospitals, and the rescue teams need to transport injured citizens to the hospitals. We hope that *FU-Oracle* can output *fewer* paths among these POIs, so that we can dig out *fewer* paths (since it is time-consuming to dig out a rescue path in the earthquake region [31]). In other words, given a complete graph (where the POIs are the vertices of the complete graph, and the exact shortest path between POIs are the edges of the complete graph), we aim at efficiently generating a sub-graph of it.

(2b) **Marsquake**: The memory size of NASA's Mars 2020 rover is 256MB [9]. Our experimental results show that for a terrain surface with 2.5M faces and 250 POIs, the sub-graph output by *FU-Oracle* is 110MB, while the complete graph is 1.3GB. Thus, we can only store the sub-graph in a Mars rover.

Generating a sub-graph from a complete graph is also used widely in distributed systems for faster network synchronization [21], [48], in wireless and sensor networks for faster signal transmission [53], [52], etc. The best-known algorithm [18], [19] for generating a sub-graph from a complete graph runs in $O(n^3 \log n)$ time, which is inefficient. We propose an algorithm called *Hierarchy Greedy Spanner* (*HieGreSpan*) that considers several vertices of the complete graph in one group to achieve a smaller running time. Our experimental results show that when $n = 500$, our algorithm takes 24s, while the best-known algorithm [18], [19] takes 101s.

D. Contributions and Organization

We summarize our major contributions as follows.

(1) We propose the first oracle that answers P2P path queries efficiently on an updated terrain surface, i.e., *FU-Oracle*. It achieves a short oracle update time by satisfying the novel non-updated terrain shortest path intact property, and utilizing the pairwise P2P exact shortest paths on T_{before}. We also propose four additional novel techniques to further reduce the oracle update time. Due to these, *FU-Oracle* is very different from the best-known oracle [54], [55]. We also develop an efficient algorithm *HieGreSpan* to reduce the output size.

(2) We provide a thorough theoretical analysis on the oracle construction time, oracle update time, output size, shortest path query time, and error bound of *FU-Oracle*.

(3) *FU-Oracle* performs much better than the best-known oracle [54], [55] in terms of oracle construction time, oracle update time, output size, and shortest path query time, and *FU-Oracle* is the most suitable oracle for real-world application (e.g., earthquake rescue) in the updated terrain surface setting. Our experiments show that for a terrain surface with 0.5M faces and 250 POIs, (1) the oracle update time of *FU-Oracle* is 400s $\approx$ 7 min, while the best-known oracle needs 35,100s $\approx$ 9.8 hours; (2) the shortest path query time for computing 100 shortest paths with different sources and destinations is 0.1s for *FU-Oracle*, while the time is 8,600s $\approx$ 2.4 hours for the best-known approximate on-the-fly algorithm [35] and 0.3s for the best-known oracle.

The remainder of the paper is organized as follows. Section II provides the problem definition. Section III covers related work. Section IV presents *FU-Oracle*. Section V covers the empirical study, and Section VI concludes the paper.

II. Problem Definition

A. Notations and Definitions

1) ***Terrain surfaces and POIs:*** Consider a terrain surface T_{before} represented as a *Triangulated Irregular Network* (*TIN*), which is a 3D terrain representation that is used commonly [27], [41], [51], [54], [55] (the terrain surface in Figure 1 (a) is represented as a *TIN*). Let V, E, and F be the set of vertices, edges, and faces of T_{before}, 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_{after}. There is no need to consider the case when new vertices are added or original vertices are deleted. This is because we consider a *TIN* generated first by creating an $\overline{x} \times \overline{y}$ 2D grid with $\overline{x} \times \overline{y} = N$ vertices, and then project these N vertices onto the 3D model to obtain the generated terrain surface [41], [54], [55]. In Figure 1 (a), the terrain surface is constructed by mapping the 2D grid into a 3D model. Our experimental results show that this procedure just needs 1.3s to obtain a terrain surface with 2.5M faces, which is very efficient. Figure 1 (a) and (d) show an example of T_{before} and T_{after}, respectively. Although T_{before} and T_{after} are different (due to the updated face in gray), the x- and y-coordinates of each vertex in V in T_{before} and T_{after} are same. Let P be a set of POIs on the surface of the terrain and n be the size of P. We focus on the case when $n \leq N$. We discuss the case when $n > N$ in the appendix.

2) ***Path:*** Given two points s and t in P, and a terrain surface T, we define $\Pi(s,t|T)$ to be the exact shortest path between s and t on T, and $|\cdot|$ to 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 components:*** Given T_{before}, T_{after}, and P, a set of (a) *updated vertices*, (b) *updated edges*, (c) *updated faces*, and (d) *updated POIs* of T_{before} and T_{after}, denoted by (a) ΔV, (b) ΔE, (c) ΔF, and (d) ΔP, is defined to be a set of (a) vertices $\Delta V = \{v_1, v_2, \dots\}$, where v_i is a vertex in V that has coordinate values that differ between T_{before} and T_{after}, (b) edges $\Delta E = \{e_1, e_2, \dots\}$, where e_i is an edge in E that has any one of its two vertices' coordinate values that differ between T_{before} and T_{after}, (c) faces $\Delta F = \{f_1, f_2, \dots\}$, where f_i is a face in F that has any one of its three vertices' coordinate values that differ between T_{before} and T_{after}, (d) and POIs $\Delta P = \{p_1, p_2, \dots\}$, where p_i is a POI in P that has coordinate values that differ between T_{before} and T_{after}. It is easy to obtain ΔV, ΔE, ΔF, and ΔP by comparing T_{before}, T_{after}, and P. In Figure 1 (d), the gray area is ΔF based on T_{before} and T_{after}. The vertices and edges in ΔF are ΔV and ΔE, $\Delta P = \{a\}$. Figure 3 (c) and (d) show an example of these sets. In addition, 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, the set of updated faces that we consider is connected [45] (in Figure 1 (d), the set of updated faces is connected). Furthermore, a point (either a vertex or a POI) is said to be in ΔF if it is on a face in ΔF, and a path is said to pass ΔF if this path intersects with ΔF. In Figure 5 (a) and (b), a is in ΔF, $\Pi(a,h|T_{before})$ and $\Pi(b,h|T_{before})$ pass ΔF.

4) ***Three queries:*** We study three queries, (a) *POI-to-POI (P2P) path query*, (b) *vertex-to-vertex (V2V) path query*, and (c) *arbitrary point-to-arbitrary point (A2A) path query*, i.e., returning the shortest path between pairs of (a) POIs, (b) vertices, and (c) arbitrary points on a terrain surface. The P2P path query is more general than the V2V path query. By creating POIs with the same coordinate values as all vertices

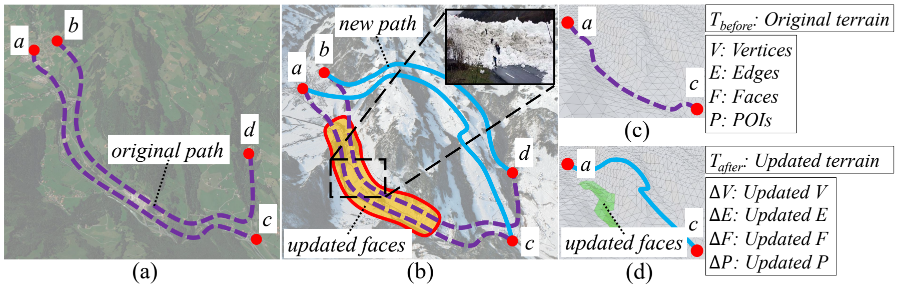

Fig. 3. The (a) real map before updates, (b) real map after updates, (c) terrain model before updates, and (d) terrain model after updates for the avalanche in Switzerland

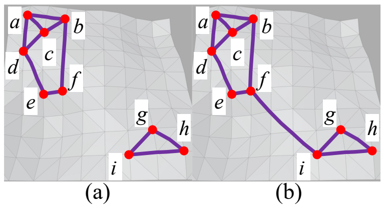

Fig. 4. *FU-Oracle* output graph G

in V, the V2V path query can be regarded as one form of the P2P path query. The A2A path query generalizes both the P2P and V2V path queries because it allows all possible points on a terrain surface. In the main body of this paper, we focus on the P2P path query. We study the V2V and A2A path queries in the appendix. In the P2P path query, there is no need to consider when n changes. When a POI is added, we create an oracle that answers the A2A path query, which implies we consider all possible POIs to be added. When a POI is removed, we continue to use the original oracle. A notation table appears in the appendix.

B. Problem

The problem is to construct a $(1+\epsilon)$-approximate shortest path oracle on an updated terrain surface with state-of-the-art performance in terms of oracle construction time, oracle update time, output size, and shortest path query time.

III. Related Work

A. On-the-fly Algorithms

There are two types of algorithms for computing the shortest path on a terrain surface *on-the-fly*: (1) *exact* [23], [34], [43], [56] and (2) *approximate* [35], [36], [39], [41] algorithms.

Exact algorithms: The time complexities of the exact algorithms [23], [34], [43], [56] are $O(N^2)$, $O(N\log^2 N)$, $O(N^2\log N)$, and $O(N^2\log N)$, respectively. They are *Single-Source All-Destination* (*SSAD*) algorithms, i.e., given a source, they can calculate the shortest path from it to all other vertices *simultaneously*. According to several studies [35], [36], [51], [57], the *Chen and Han on-the-Fly Algorithm* (*CH-Fly-Algo*) [23] is recognized as the best-known exact algorithm. It uses a sequence tree for the shortest path query. Our experimental results show that it needs 11,600s ≈ 3.2 hours to compute 100 paths with different sources and destinations on a terrain surface with 0.5M faces.

Approximate algorithms: Approximate algorithms [35], [36], [39], [41] aim at reducing the running time. The *Kaul on-the-Fly Algorithm* (*K-Fly-Algo*) [35] can return a $(1+\epsilon)$-approximate shortest path on a terrain surface, and it is recognized as the best-known approximate algorithm by work [54], [55]. It places Steiner points on edges in E, and then constructs a graph using these points and V to calculate the shortest path. 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. Under the same setting of the experiment in the last paragraph, *K-Fly-Algo* runs in 8,600s ≈ 2.4 hours.

Drawbacks of the on-the-fly algorithms: All exact and approximate on-the-fly algorithms are not efficient enough when multiple shortest path queries are involved.

B. Oracle-based Algorithms

*1) **WSPD-Oracle**:* Due to the expensive shortest path query time of on-the-fly algorithms, *Well-Separated Pair Decomposition Oracle* (*WSPD-Oracle*) [54], [55] uses an *oracle* to pre-compute the shortest paths on a terrain surface for answering the *approximate* P2P shortest path queries. It is recognized as the best-known P2P path query oracle. It uses algorithm *SSAD*, *compressed partition tree* [54], [55], and *well-separated node pair sets* [22] to index the $(1+\epsilon)$-approximation pairwise P2P shortest paths. The oracle construction time, output size, and shortest path query time of *WSPD-Oracle* is $O(\frac{nN\log^2 N}{\epsilon^{2\beta}}+\frac{nh}{\epsilon^{2\beta}}+nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$, respectively, where h is the height of the compressed partition tree and β is the largest capacity dimension [30], [38] ($\beta\in[1.5,2]$ in practice [54], [55]).

Drawback of *WSPD-Oracle*: It *only supports the static terrain surface* and does not address how to update the oracle on an *updated* terrain surface. If we use the straightforward adaptation, i.e., re-construct *WSPD-Oracle* from scratch when the terrain surface is updated, an update takes 35,100s ≈ 9.8 hours for a terrain dataset with 0.5M faces and 250 POIs, but the time is just 400s ≈ 7 min for *FU-Oracle*.

*2) **WSPD-Oracle-Adapt**:* To handle this, we employ a smart adaption by leveraging the *non-updated terrain shortest path intact* property, such that we only re-calculate the paths on T_{after} that require updating to reduce the oracle update time. We denote it as *WSPD-Oracle-Adapt*, and its oracle update time is $O(\mu_1 N\log^2 N + n\log^2 n)$, where μ_1 is a data-dependent variable and $\mu_1\in[5,20]$ in our experiment.

Drawbacks of *WSPD-Oracle-Adapt*: (1) *Not fully utilizing the non-updated terrain shortest path intact property during update*: Since *WSPD-Oracle-Adapt* only stores the pairwise P2P *approximate* shortest paths on T_{before}, the oracle update time remains large. In Figure 5 (d), suppose that *WSPD-Oracle-Adapt* calculates an approximate path between c and

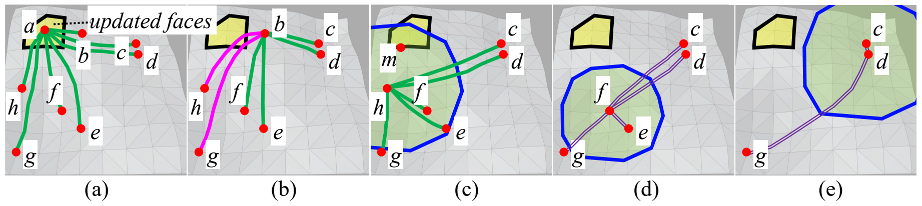

Fig. 5. In the update phase when (a) updating $\Pi(a)$, (b) updating $\Pi(b)$, (c) updating $\Pi(f)$, (d) no need for updating $\Pi(c)$, and (e) no need for updating $\Pi(e)$

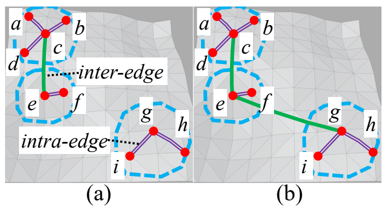

Fig. 6. Hierarchy graph H

f on T_{before} (with path distance d_2), whose distance is longer than the exact shortest distance between c and f on T_{before} (i.e., $d_2 > |\Pi(c, f|T_{before})|$). It may happen that the distance from f to ΔF is smaller than $\frac{d_2}{2}$, but larger than $\frac{|\Pi(c,f|T_{before})|}{2}$. In this case, they need to use algorithm *SSAD* with f as source to update the shortest paths on T_{after}. The case also happens for the paths between c and e. In Figure 5 (e), the case also happens for the path between g and each POI in $\{c, d\}$. The oracle update time is 8,400s $\approx$ 2.4 hours on a terrain dataset with 0.5M faces and 250 POIs for *WSPD-Oracle-Adapt*, while the time is 400s $\approx$ 7 min for *FU-Oracle*. (2) *Additional information needed during construction*: In order to fully utilize this property, *WSPD-Oracle-Adapt* needs to calculate the shortest distance between each POI and vertex on T_{before} when the oracle is constructed, which increases its oracle construction time, but *FU-Oracle* can calculate this information and the pairwise P2P exact shortest paths on T_{before} *simultaneously*.

*3) **EAR-Oracle**:* *Efficiently ARbitrary pints-to-arbitrary points Oracle* (*EAR-Oracle*) [32] uses an *oracle* on a terrain surface to answer the *approximate* A2A shortest path queries. It uses the same idea as *WSPD-Oracle*, i.e., well-separated pair decomposition. Their differences are that *EAR-Oracle* adapts *WSPD-Oracle* from the P2P path query to the A2A path query by using Steiner points on the terrain faces and using *highway nodes* as POIs in well-separated pair decomposition. Since the A2A path query generalizes the P2P path query, *EAR-Oracle* can also be used in the P2P path query. The oracle construction time, output size, and shortest path query time of *EAR-Oracle* is $O(\lambda\xi mN\log^2(mN) + \frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$, and $O(\lambda\xi\log(\lambda\xi))$, respectively, where λ is the number of highway nodes covered by a minimum square, ξ is the square root of the number of boxes, and m is the number of Steiner points per face.

Drawbacks of *EAR-Oracle*: (1) *Numerous highway nodes*: In the P2P path query, n is much smaller than the number of highway nodes, so the oracle construction time of *EAR-Oracle* is much larger than that of *WSPD-Oracle*. Thus, *EAR-Oracle* does not perform well in the P2P path query, and it is only regarded as the best-known oracle in the A2A path query. (2) *Only supporting the static terrain surface*: *EAR-Oracle* has the same drawback as of *WSPD-Oracle*. Even for the A2A path query, our experimental results show that the oracle update time of *EAR-Oracle* is 710s $\approx$ 12 min, while the value is 48s for *FU-Oracle* on a terrain surface with 2k faces.

*4) **EAR-Oracle-Adapt**:* We can adapt *EAR-Oracle* to *EAR-Oracle-Adapt* in the same way as of *WSPD-Oracle-Adapt*. The oracle update time of *WSPD-Oracle-Adapt* is $O(\mu_2 N\log^2 N + n\log^2 n)$, where μ_2 is a data-dependent variable and $\mu_1 \in [12, 45]$ in our experiment.

Drawback of *EAR-Oracle-Adapt*: *EAR-Oracle-Adapt* does not *fully utilize the non-updated terrain shortest path intact property during update*, i.e., it has the first drawback as of *WSPD-Oracle-Adapt*. Even in the A2A path query, our experimental results show that the oracle update time of *EAR-Oracle-Adapt* is 430s $\approx$ 7.2 min, while the value is 48s for *FU-Oracle* on a terrain surface with 2k faces.

C. Sub-graph Generation Algorithm

Algorithm *Greedy Spanner* (*GreSpan*) [18], [19] is the best-known algorithm for generating a sub-graph from a complete graph. However, it is very time-consuming because it does not consider using any simpler structure to approximate the sub-graph G when performing Dijkstra's algorithm [28] on G. Its time complexity is $O(n^3 \log n)$.

IV. Methodology

A. Overview of FU-Oracle

*1) **Four components of FU-Oracle**:* (1a) *FU-Oracle output graph*, (1b) *temporary complete graph*, (1c) *POI-to-vertex distance mapping table*, and (1d) *hierarchy graph*.

(1a) **The *FU-Oracle* output graph *G***: This is a graph used for answering $(1+\epsilon)$-approximate shortest path between any pair of POIs in P. 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 $G.V$). Then an exact shortest path $\Pi(u, v|T_{after})$ between a pair of POIs u and v on T_{after} is denoted by a weighted edge $e(u, v|T_{after})$ in $G.E$, and the distance of this path $|\Pi(u, v|T_{after})|$ is denoted by the weight of the edge $|e(u, v|T_{after})|$. Given two vertices s and t in $G.V$, we define $\Pi_G(s, t|T_{after}) = (v_1, v_2, \ldots, v_l)$ to be a shortest path of *FU-Oracle*, such that the weighted length $\sum_{i=1}^{l-1} |e(v_i, v_{i+1}|T_{after})|$ is the minimum, where $v_1 = s$, $v_l = t$, and for each $i \in [1, l-1]$, $(v_i, v_{i+1}) \in G.E$. We define $|\Pi_G(s, t|T_{after})|$ to be the distance of $\Pi_G(s, t|T_{after})$. *FU-Oracle* guarantees that $|\Pi_G(s, t|T_{after})| \leq (1+\epsilon)|\Pi(s, t|T_{after})|$ for any s and t in P. The purple lines in Figure 1 (f) show an example of G with 4 POIs. The purple line between a and c is the exact shortest path $\Pi(a, c|T_{after})$ on T_{after}, and also an edge $e(a, c|T_{after})$ in

G. The shortest path $\Pi_G(a,b|T_{after})$ in G is (a,b,c), which consists of edges $e(a,c|T_{after})$ and $e(c,b|T_{after})$.

(1b) **The temporary complete graph G'**: This is a complete graph that stores the pairwise exact shortest path between all pairs of POIs in P. The difference between G' and G is that G is a sub-graph of G' with fewer edges. Similar to G, let $G'.V$ and $G'.E$ be the set of vertices and edges of G', let $e'(u,v|T)$ be an edge between two vertices u and v in $G'.V$, and let $|e'(u,v|T)|$ be the weight of this edge, where T can be T_{before} or T_{after}. The purple lines in Figure 1 (b) show a complete graph G' with 4 vertices and 6 edges.

(1c) **POI-to-vertex distance mapping table M_{dist}**: This is a *hash table* [24] that stores the exact shortest distance from each POI in P to each vertex in V on T_{before} (calculated when *FU-Oracle* is constructed), used for reducing the oracle udpate time of *FU-Oracle*. A pair of vertex u and v is stored as a key $\langle u,v\rangle$, and their corresponding exact shortest distance $|\Pi(u,v|T_{before})|$ is stored as a value. M_{dist} needs linear space in terms of the number of distances to be stored. Given a POI u and a vertex v, M_{dist} can return the associated exact shortest distance $|\Pi(u,v|T_{before})|$ in $O(1)$ time. In Figure 1 (c), the exact shortest distance between POI a and vertex v_1 is 4.

(1d) **The hierarchy graph H**: This is a graph similar to G, but has a simpler structure compared with G and is maintained simultaneously with G, which is used for efficiently generating a G using G'. We define a *group*, with *group center* v and *radius* r, to be a set of vertices $Q_G \subseteq G.V$, such that for every vertex $u \in Q_G$, we have $|\Pi_G(u,v|T_{after})| \leq r$, where $v \in Q_G$. A set of groups $Q_G^1, Q_G^2, \ldots, Q_G^k$ is a group cover of G if every vertex in $G.V$ belongs to at least one group. H can form a set of *groups* by regarding several vertices in G that are close to each other as one vertex (see Figure 4 and Figure 6). As a result, the shortest distance between u and v on H is an approximation of the shortest distance between u and v on G. Similar to G, let $H.E$ be the set of edges of H. Given a group Q_G^i, we define *intra-edges* to be a set of edges connecting the group center of Q_G^i to all other vertices in Q_G^i, and we define *inter-edges* to be a set of edges connecting two group centers. H can be constructed from a group cover by adding these two types of edges. For each (intra- or inter-) edge $e_H(u,v|T_{after})$ in H, the weight of this edge is denoted as $|e_H(u,v|T_{after})|$. Given two group centers s and t, we define $\Pi_H(s,t|T_{after}) = (v_1, v_2, \ldots, v_l)$ to be a shortest path of H, such that the weighted length of the inter-edges $\sum_{i=1}^{l-1} |e(v_i, v_{i+1}|T_{after})|$ is minimum, where $v_1 = s$, $v_l = t$, and for each $i \in [1, l-1]$, v_i is a group center of H. Figure 6 (b) shows an example of H, there are three groups with centers c, e, and g. The purple lines are intra-edges, and the green lines are inter-edges. The shortest path of inter-edges $\Pi_H(c,g|T_{after})$ is (c,e,g).

*2) **Three phases of FU-Oracle**:* (2a) *Construction phase*, (2b) *update phase*, and (2c) *query phase* (see Figure 1).

(2a) **Construction phase**: Given T_{before} and P, we use algorithm *SSAD* for n times to calculate the pairwise P2P exact shortest paths on T_{before} (stored in G') and the POI-to-vertex distance information (store in M_{dist}).

(2b) **Update phase**: Given T_{before}, T_{after}, P, G', and M_{dist}, we efficiently update the pairwise P2P exact shortest paths on T_{after} in G' and produce G (a sub-graph of G') in three steps:

- *Terrain surface and POI update detection*: Given T_{before}, T_{after}, and P, we detect ΔF and ΔP.
- *Pairwise P2P exact shortest paths updating*: Given G', P, M_{dist}, ΔF, and ΔP, we update the exact shortest path between all pairs of POIs in P on T_{after} in G' using algorithm *SSAD* exploiting the *non-updated terrain shortest path intact* property.
- *Sub-graph generating*: Given G', we use algorithm *HieGreSpan* to generate a sub-graph of G', i.e., G, for reducing the output size of *FU-Oracle* with the assistance of H, such that $|\Pi_G(s,t|T_{before})| \leq (1+\epsilon)|\Pi(s,t|T_{before})|$ for any pair of POIs s and t in P on T_{after}.

(2c) **Query phase**: Given two query POIs and G, we answer the path between these two POIs on T_{after} using G efficiently.

B. Key Ideas of Short Oracle Update Time

One major contribution of this paper is the short oracle update time of *FU-Oracle*, which comes from our design in the pairwise P2P exact shortest paths updating step of the update phase. In this step, the key reasons for the short oracle update time for *FU-Oracle* are due to (1) the *non-updated terrain shortest path intact* property, and (2) the stored pairwise P2P exact shortest paths on T_{before} when *FU-Oracle* is constructed. We define the concept of a *disk* first. Given a point p on a terrain surface and a non-negative real number r, a disk centered at p with radius r on the terrain surface, denoted by $D(p,r)$, consists of all points on the terrain surface whose exact shortest distance to p is at most r. Given a face f_i, if a point q exists on f_i such that the shortest distance between p and q is at most r, then disk $D(p,r)$ is said to be *intersect with* face f_i. Figure 2 shows a disk centered at u with radius equal to the shortest distance between u and o, and it does not intersect with any updated faces.

Firstly, we give the *non-updated terrain shortest path intact* property in Property 1. The property is satisfied in Figure 2.

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

Proof Sketch. We prove it by contradiction and show that the two paths cannot be different. All detailed proofs in the paper appear in the appendix. □

Secondly, we illustrate the necessity of storing the pairwise P2P exact shortest paths on T_{before}. In Figure 5 (d), recall that *WSPD-Oracle-Adapt* may calculate an *approximate* path between c and f on T_{before} (with path distance d_2). So the disk $D(f, \frac{|d_2|}{2})$ may intersect with ΔF, and it needs to update the path on T_{after}. But, if we use the exact path, the disk $D(f, \frac{|\Pi(c,f|T_{before})|}{2})$ does not intersect with ΔF, and there is no need to update the path.

Apart from these two ideas, we provide four additional novel techniques to further reduce the oracle update time.

*1) **Novel path update sequence**:* We propose a novel path update sequence before utilizing the non-updated terrain shortest path intact property, to minimize the oracle update time. In Figure 5 (a), we need to update the shortest paths between a and two POIs in $\{e, g\}$ on T_{after}. By using algorithm *SSAD*, when we update the paths with a as the source POI, we can update these two paths simultaneously (since e and g are far away from ΔF, we can avoid using algorithm *SSAD* to update the paths with e and g as the source POIs according to the non-updated terrain shortest path intact property). But, if we first update the paths with e as the source POI, we still need to update the paths with g as the source POI, which increases the oracle update time. The path update sequences that results in smallest oracle update time are: (1) updating the paths connect to those POIs in ΔF, (2) updating the paths pass ΔF, and (3) updating the paths connect to those POIs near ΔF. After we update all the paths belonging to one type, we process to the next type. For example, (1) a is in ΔF in Figure 5 (a), (2) one of b's exact shortest path $\Pi(b, h|T_{\mathit{before}})$ pass ΔF in Figure 5 (b), and (3) h is near ΔF in Figure 5 (c), so we use a, b, and h as source point in algorithm *SSAD* and update the shortest paths on T_{after} in sequence for these three figures.

*2) **Novel disk radius selection strategy**:* We design a novel disk radius selection strategy (i.e., *half* of the shortest distance between a pair of POIs as the disk radius) when updating the paths connect to those POIs near ΔF to minimize the chances of re-calculating the shortest paths on T_{after}. In Figure 2, a naive approach is to create two disks centered at u and v with radius equal to the *full* shortest distance between u and v. It increases the chance of re-calculating this path on T_{after} and increases the oracle update time.

*3) **Novel distance approximation approach**:* We propose a novel distance approximation approach, to avoid performing the expensive shortest path query algorithm on T_{after}, for determining whether the disk intersects with ΔF on T_{after} (i.e., whether the minimum distances from the disk center to any point in ΔF on T_{after} is smaller than the disk radius), by using the POI-to-vertex distance information stored in M_{dist}. In Figure 2, we do not want to perform the shortest path query algorithm between v and v_2 on T_{after} again, for determining whether the disk centered at v intersects with the updated faces, where v_2 is a point belonging to the updated faces that is the closest point to v (among other points belonging to the updated faces). Instead, in Lemma 1, we show that we can use M_{dist} to obtain the lower bound of the minimum distances from the disk center (i.e., a POI) to any point in ΔF on T_{after}, to approximate the shortest distance on T_{after} in $O(1)$ time.

Lemma 1. *The minimum distance from a POI u to any point in ΔF on T_{after} is no less than $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{\mathit{before}})| - L_{\mathit{max}}$.*

Proof Sketch. We show that the minimum distance from u to a point of e on T_{after} is the same as on T_{before}, where e is the edge belongs to a face in ΔF, and the exact shortest path from u to ΔF intersects with any point on e for the first time. □

If the lower bound is larger than the disk radius, then the minimum distances from this radius center to any point in ΔF must be larger than the disk radius, i.e., there is no need to update the corresponding paths. In Figure 5 (c), the exact shortest distance between h and m can be calculated in $O(1)$ time. Calculating the POI-to-vertex distance information will not increase the oracle construction time. When *FU-Oracle* is constructed, each POI is given as a source point, then we can use algorithm *SSAD* for n times to calculate the pairwise P2P exact shortest paths on T_{before} and the POI-to-vertex distance information *simultaneously*.

*4) **Novel disk & updated faces intersection checking approach**:* We design a novel disk & updated faces intersection checking approach to minimize the intersection checking for each shortest path on T_{after} when updating the paths connect to those POIs near ΔF. In Figure 5 (d), when checking whether we need to re-calculate the paths between f and each POI in X on T_{after}, a naive approach is creating disks centered at f and each POI in X with radius equal to half of the shortest distance between f and each POI in X, and check whether these eight disks intersect with ΔF, where $X = \{c, d, e, g\}$. Since there are total $O(n^2)$ paths, it needs to create $O(n^2)$ disks. But, we just need to create *one* disk centered at f with radius equal to half of the longest distance of the paths between f and each POI in $\{c, d, e, g\}$, and check whether this disk intersects with ΔF. Since there are total $O(n)$ POIs, we just need to create $O(n)$ disks. Specifically, for each POI not in ΔF and not the endpoint of the paths pass ΔF, we sort them from near to far according to their minimum distance to any vertex in ΔV on T_{before}. We then determine whether there is a need to update the shortest paths adjacent to u using Lemma 2.

Lemma 2. *If the disk centered at u with radius equal to half of the longest distance of all paths (that have not been updated) adjacent to u intersects with ΔF, we use algorithm SSAD to update all the shortest paths adjacent to u that have not been updated. Otherwise, there is no need to update the shortest paths adjacent to u.*

Proof Sketch. If the disk with the largest radius intersects with ΔF, we just need to update the paths 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, so there is no need to update the paths. □

(4a) In Figure 5 (c), the sorted POIs are h, f, e, d, c, g. We create one disk $D(h, \frac{|\Pi(c,h|T_{\mathit{before}})|}{2})$, since it *intersects* with ΔF, we use algorithm *SSAD* to update all the shortest paths adjacent to h that have not been updated. We do not need to create ten disks, i.e., five disks $D(h, \frac{|\Pi(X,h|T_{\mathit{before}})|}{2})$ and five disks $D(X, \frac{|\Pi(X,h|T_{\mathit{before}})|}{2})$, where $X = \{c, d, e, f, g, h\}$.

(4b) In Figure 5 (d), the sorted POIs are f, e, d, c, g. We create one disk $D(f, \frac{|\Pi(c,f|T_{\mathit{before}})|}{2})$, since it *does not intersect* with ΔF, there is no need to update the shortest paths adjacent to f. We do not need to create eight disks, i.e., four disks $D(f, \frac{|\Pi(X,f|T_{before})|}{2})$ and four disks $D(X, \frac{|\Pi(X,f|T_{before})|}{2})$, where $X = \{c, d, e, f, g\}$.

C. Key Ideas of Efficiently Achieving Small Output Size

Another major contribution of this paper is to efficiently reduce the output size of *FU-Oracle*, which comes from our design in the sub-graph generating step (using algorithm *HieGreSpan*) of the update phase. In *HieGreSpan*, unlike *GreSpan*, we perform Dijkstra's algorithm to calculate the shortest distance between two vertices on H (not G).

We first illustrate algorithm *GreSpan*. Given a complete graph G', it first sorts the edge in G' based on the weight of each edge from minimum to maximum, and initializes a sub-graph G to be empty. Then, for each sorted edge $e'(u, v) \in G'$ between two vertices u and v, if the length of $e'(u, v)$ is longer than $(1 + \epsilon)$ times the distance between u and v on G (calculated using Dijkstra's algorithm on G), then $e'(u, v)$ is added into G (see Figure 4). It iterates until all the paths have been processed, and returns G as output.

We then illustrate algorithm *HieGreSpan*. The main difference between *HieGreSpan* and *GreSpan* is the usage of H. To construct H, we first sort the edges of G', i.e., $G'.E$, in increasing order, and then divide them into $\log n$ intervals, where each interval contains edges with weights in $(\frac{2^{i-1}D}{n}, \frac{2^iD}{n}]$ for $i \in [1, \log n]$ and D is the longest edge's weight in $G'.E$. When processing each interval of edges, we group some vertices in $G'.V$ into one vertex (the radius of each group of vertices is $\delta\frac{2^iD}{n}$, where $\delta \in (0, \frac{1}{2})$ is a small constant depending on ϵ), such that the shortest distance between the vertices in the same group is very small (and can be regarded as 0) compared with the current processing interval edges' weights. Thus, when checking whether the length of $e'(u, v)$ is longer than $(1 + \epsilon)$ times the distance between u and v on G, we use Dijkstra's algorithm between the group centers of u and v on H, to approximate the distance between u and v on G. This takes $O(1)$ time on H, but takes $O(n \log n)$ time on G in algorithm *GreSpan*. When we need to process the next interval of edges with larger weight, we update H such that the radius of each group of vertices will also increase, and H is a valid approximated graph of G. In Figure 4, we are adding the purple edge between f and i in G. In Figure 6, f belongs to e, and i belongs to g, so we add the green edge between e and g in H. The length of the green edges between c and e, and e and g in H are much longer than the purple edges in the light blue dashed circle. Our experimental results show that when $n = 500$, algorithm *HieGreSpan* needs 24s, but algorithm *GreSpan* needs 101s. Due to algorithm *HieGreSpan*, the output size of *FU-Oracle* is only 22MB on a terrain surface with 0.5M faces and 250 POIs, but the value is 260MB for *WSPD-Oracle*.

D. Implementation Details in the Construction Phase

We give the implementation detail in the construction phase. Given T_{before} and P, by regarding each POI $p_i \in P$ as a source point, we use algorithm *SSAD* to (1) calculate the exact shortest paths between p_i and other POIs in P on T_{before}, and then store them in G'; (2) calculate the exact shortest distance between p_i and each vertex in V on T_{before}, and then store them in M_{dist}. In Figure 1 (b), we first take a as a source point, and then use algorithm *SSAD* to calculate the exact shortest path between a and $\{b, c, d\}$ (the purple lines), and the exact shortest distance between a and all vertices. Next, we take b as a source point, and use algorithm *SSAD* to calculate the exact shortest path between b and $\{c, d\}$, and the exact shortest distance between b and all vertices.

E. Implementation Details in the Update Phase

We give the implementation detail in the update phase for (1) the pairwise P2P exact shortest paths updating step and (2) the sub-graph generating step.

1) ***Pairwise P2P exact shortest paths updating****:* Given two POIs u and v in P, after we have updated an exact shortest path $\Pi(u, v|T_{before})$ (stored in G') between u and v on T_{before}, the updated exact shortest path between u and v on T_{after} is denoted as $\Pi(u, v|T_{after})$. Let $P_{remain} = \{p_1, p_2, \ldots\}$ be a set of remaining POIs of P on T_{after} that we have not processed. P_{remain} is initialized to be P. In Figure 5 (c), $P_{remain} = \{c, d, e, f, g\}$. Given a POI $u \in P_{remain}$, we let $\Pi(u) = \{\Pi(u, v_1|T_{before}), \Pi(u, v_2|T_{before}), \ldots, \Pi(u, v_l|T_{before})\}$ be a set of the exact shortest paths stored in G' on T_{before} with u as an endpoint and $v_i \in P_{remain} \setminus \{u\}$, $i \in \{1, l\}$ as the other endpoint, such that all 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. In Figure 5 (a) - (c), the green and pink lines denote $\Pi(a)$, $\Pi(b)$, and $\Pi(h)$, respectively.

Detail and example: Algorithm 1 and 2 show this step. In Algorithm 1, see Figure 5 (c), we compute *Update* $(h, T_{after}, G', P_{remain} = \{c, d, e, f, g\})$. The following illustrates Algorithm 2 with an example.

Algorithm 1 *Update* $(u, T_{after}, G', P_{remain})$

```
Input: a POI $u$, $T_{after}$, temporary complete graph $G'$, and $P_{remain}$
Output: updated $G'$ and updated $P_{remain}$
1: use $u$ as source point in algorithm SSAD to calculate $\Pi(u, v|T_{after})$ for
   each POI $v \in P_{remain}$ simultaneously
2: for each POI $v \in P_{remain}$ do
3:     $G'.E \leftarrow G'.E - \{\Pi(u, v|T_{before})\} \cup \{\Pi(u, v|T_{after})\}$
4:     $\Pi(v) \leftarrow \Pi(v) - \{\Pi(u, v|T_{before})\}$
5: $P_{remain} \leftarrow P_{remain} - \{u\}$
6: return updated $G'$ and $P_{remain}$
```

(1.1) *Path updating for POIs in updated faces* Lines 4-6. In Figure 5 (a), $a \in \Delta P$, we update the paths in green on T_{after}.

(1.2) *Path updating for paths passing updated faces*: Lines 7-9. In Figure 5 (b), $b \notin \Delta P$ but one exact shortest path $\Pi(b, h|T_{before}) \in \Pi(u)$ passes ΔF (the black circle), so we update the paths in green and pink on T_{after}.

(1.3) *Path updating for POIs near updated faces*: Lines 10-16. Specifically, In lines 13-14 and Figure 5 (c), the sorted POIs are h, f, e, d, c, g, the path with the longest distance is $\Pi(c, h|T_{before})$. Since the disk with blue circle intersects with ΔF, we update the paths in green on T_{after}. In lines 15-16 and

Algorithm 2 *PairwiseP2PUpdate* (G', P, M_{dist}, ΔF, ΔP)

```
Input: $G'$, a set of POIs $P$, $M_{dist}$, $\Delta F$, and $\Delta P$
Output: updated $G'$
1: $P_{remain} \leftarrow P$
2: for each POI $u \in P_{remain}$ do
3:     $\Pi(u) \leftarrow$ all the exact shortest paths in $G'$ with $u$ as an endpoint
4: for each POI $u \in P_{remain}$ do
5:     if $u \in \Delta P$ then
6:         Update $(u, T_{after}, G', P_{remain})$
7: for each POI $u \in P_{remain}$ do
8:     if $u \notin \Delta P$ but there exists an exact shortest path in $\Pi(u)$ passes $\Delta F$ then
9:         Update $(u, T_{after}, G', P_{remain})$
10: sort each POI in $P_{remain}$ from near to far according to their minimum distance to any vertex in $\Delta V$ on $T_{before}$ using $M_{dist}$
11: for each sorted POI $u \in P_{remain}$ do
12:     $v \leftarrow$ a POI in $P_{remain}$ such that $\Pi(u, v|T_{before})$ has the longest distance among all $\Pi(u)$
13:     if disk $D(u, |\frac{\Pi(u,v|T_{before})}{2}|)$ intersects with $\Delta F$ then
14:         Update $(u, T_{after}, G', P_{remain})$
15:     else
16:         $P_{remain} \leftarrow P_{remain} - \{u\}$
17: return updated $G'$
```

Figure 5 (d), the sorted POIs are f, e, d, c, g, the paths with the longest distance is $\Pi(c, f|T_{before})$. Since the disk with blue circle does not intersect with ΔF, we do not need to update the paths.

2) ***Sub-graph generating using algorithm HieGreSpan****:* Recall that the radius of each group of vertices is $\delta\frac{2^i D}{n}$, we set $\delta = \frac{1}{2}(\frac{\sqrt{\epsilon+1}-1}{\sqrt{\epsilon+1}+3})$. Since $\epsilon \in (0, \infty)$, we have $\delta \in (0, \frac{1}{2})$.

Detail and example: Algorithm 3 shows *HieGreSpan*, and the following illustrates it with an example.

(2.1) *Edge sorting, interval splitting, and G initialization*: Lines 2-6.

(2.2) *G maintenance*: Lines 7-25. Specifically, in lines 9-15 and Figure 6 (a), it is called *groups construction and intra-edges adding for H*, we have three groups with group center c, e, and g. We then add purple lines $e(a, c|T_{after})$, $e(b, c|T_{after})$, $\ldots$ in H. In lines 16-19 and Figure 6 (a), it is called *first type inter-edges adding for H*, we add green lines $e'(c, e|T_{after})$ in H. In lines 20-22 and Figure 4 (a), it is called *edges examining on H*, we need to examine edge $e'(f, i|T_{after})$, the corresponding shortest path on H in Figure 6 (a) is $\Pi_H(e, g|T_{after})$, and $|\Pi_H(e, g|T_{after})| = \infty > (1+\epsilon)|e'(f, i|T_{after})|$. In lines 24 and Figure 4 (b), it is called *Edges adding for G*, we add $e'(f, i|T_{after})$ into G. In lines 25 and Figure 6 (b), it is called *second type inter-edges adding for H*, we add $e_H(e, g|T_{after})$ with weight $|e_H(e, f|T_{after})| + |e'(e, g|T_{after})| + |e_H(g, i|T_{after})|$ in H.

F. Implementation Details in the Query Phase

We give the implementation detail in the query phase. Given G, and two query POIs s and t in P (i.e., two query vertices s and t in $G.V$), we use Dijkstra's algorithm [28] to find the shortest path between s and t on G, i.e., $\Pi_G(s, t|T_{after})$, which is a $(1+\epsilon)$-approximate path of $\Pi(s, t|T_{after})$. In Figure 1 (g), given two query POIs a and b, we use Dijkstra's algorithm to find $\Pi_G(a, b|T_{after})$, which consists of two blue lines, i.e., $\Pi(a, c|T_{after})$ and $\Pi(c, b|T_{after})$.

Algorithm 3 *HieGreSpan* (G', ϵ)

```
Input: temporary complete graph $G'$ and error parameter $\epsilon$
Output: FU-Oracle output graph $G$ (a sub-graph of $G'$)
1: $D \leftarrow$ the weight of the longest edge in $G'.E$
2: for each edge $e'(u, v|T_{after}) \in G'.E$ do
3:     sort edge weights in increasing order
4:     create intervals $I_0 = (0, \frac{D}{N}]$, $I_i = (\frac{2^{i-1}D}{n}, \frac{2^i D}{n}]$ for $i \in [1, \log n]$
5:     $G'.E^i \leftarrow$ sorted edges of $G'.E$ with weight 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 $\delta\frac{2^i D}{n}$ from $u_j$
11:        create a group $Q_G^j \leftarrow \{u_j\}$ with group center $u_j$, $u_j \leftarrow$ visited
12:        for each $v \in G'.V$ such that $|\Pi_G(u_j, v|T_{after})| \le \delta\frac{2^i D}{n}$ do
13:            $Q_G^j \leftarrow \{v\}$, $v \leftarrow$ visited
14:            $H$ intra-edges $\leftarrow H.E \cup \{e_H(u_j, v|T_{after})\}$, where $|e_H(u_j, v|T_{after})| = |\Pi_G(u_j, v|T_{after})|$
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 $\frac{2^i D}{n} + 2\delta\frac{2^i D}{n}$ from $u_j$
18:        $H$ inter-edges $\leftarrow H.E \cup \{e_H(u_j, u|T_{after})\}$, where $u$ is other group centers, $|\{e_H(u_j, U|T_{after})\}| = |\Pi_G(u_j, u|T_{after})| \le \frac{2^i D}{n} + 2\delta\frac{2^i D}{n}$
19:        $j \leftarrow j + 1$
20:    for each edge $e'(u, v|T_{after}) \in G'.E^i$ do
21:        $w \leftarrow$ group center of $u$, $x \leftarrow$ group center of $v$
22:        $\Pi_H(w, x|T_{after}) \leftarrow$ the shortest path between $w$ and $x$ calculated using Dijkstra's algorithm on $H$
23:        if $|\Pi_H(w, x|T_{after})| > (1+\epsilon)|e'(u, v|T_{after})|$ then
24:            $G.E \leftarrow G.E \cup \{e'(u, v|T_{after})\}$
25:            $H$ inter-edge $\leftarrow H.E \cup \{e_H(w, x|T_{after})\}$, where $|e_H(w, x|T_{after})| = |e_H(w, u|T_{after})| + |e'(u, v|T_{after})| + |e_H(v, x|T_{after})|$
26: return $G$
```

G. Theoretical Analysis

Theorem 1 shows the analysis of algorithm *HieGreSpan*.

Theorem 1. *The running time of HieGreSpan is $O(n \log^2 n)$. The output of HieGreSpan, i.e., G, satisfies $|\Pi_G(u, v|T_{after})| \le (1+\epsilon)|\Pi(u, v|T_{after})|$ for all pairs of vertices u and v in $G.V$.*

Proof Sketch. The *running time* includes (1) the edge sorting, interval splitting time, and G initialization $O(n)$ due to n vertices in G', and (2) G maintenance time $O(n \log^2 n)$ due to total $\log n$ intervals and $O(n \log n)$ time for each interval. For the *error bound*, we use the same notations in Algorithm 3. Since H is a valid approximation of G, in the edges examining on H step of algorithm *HieGreSpan*, when we check whether $|\Pi_H(w, x|T)| > (1+\epsilon)|e'(u, v|T)|$, we are checking $|\Pi_G(u, v|T)| > (1+\epsilon)|e'(u, v|T)|$. For any edge $e'(u, v|T) \in G.E$ that is not added to G, we know $|\Pi_G(u, v|T)| \le (1+\epsilon)|e'(u, v|T)|$. Since $|e'(u, v|T)| = |\Pi(u, v|T)|$, we have $|\Pi_G(u, v|T)| \le (1+\epsilon)|\Pi(u, v|T)|$. ☐

Theorem 2 shows the analysis of *FU-Oracle*.

Theorem 2. *The oracle construction time, oracle update time, output size, and shortest path query time of FU-Oracle are $O(nN \log^2 N)$, $O(N \log^2 N + n \log^2 n)$, $O(n)$, and*

$O(\log n)$, respectively. FU-Oracle satisfies $|\Pi_G(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ for all pairs of POIs u and v in P.

Proof Sketch. The *oracle construction time* includes the pairwise P2P exact shortest paths calculation time $O(nN\log^2 N)$ due to total n POIs and the usage of algorithm *SSAD* in $O(N\log^2 N)$ time for each POI. The *oracle update time* includes (1) terrain surface and POIs update detection time $O(N+n)$ due to $O(N)$ faces and n POIs, (2) the pairwise P2P exact shortest paths update time $O(N\log^2 N)$ due to $O(1)$ number of updated POIs (shown by our experimental result) and the usage of algorithm *SSAD* in $O(N\log^2 N)$ time for each POI, (3) and the sub-graph generating time $O(n\log^2 n)$ due to algorithm *HieGreSpan* in Theorem 1. The *output size* is $O(n)$ due to the output graph size of algorithm *HieGreSpan*. The *shortest path query time* is $O(\log n)$ due to the use of Dijkstra's algorithm on G (in our experiment, G has a constant number of edges and n vertices). The *error bound* of *FU-Oracle* is due to the error bound of algorithm *HieGreSpan*. □

H. Discussion on the Necessity of Storing G'

Recall that one key reason for the short oracle update time for *FU-Oracle* is due to the stored pairwise P2P exact shortest paths on T_{before} when *FU-Oracle* is constructed (i.e., the information stored in G'). We discuss the necessity of storing G'. Let *UR*(A) be the Update Ratio of an oracle A, which is defined to be the number of POIs in P that we need to perform algorithm *SSAD* as source (for path updating on T_{after}) divided by the total number of POIs. In Figure 5, for *FU-Oracle*, we need to perform algorithm *SSAD* with a, b, h (3 POIs) as a source for path updating on T_{after}, and there is a total of 8 POIs, so *UR*(*FU-Oracle*) $= \frac{3}{8}$. Given an oracle A, a higher *UR*(A) means that the oracle update time of A is larger. Corollary 1 shows the necessity of storing G'.

Corollary 1. *Given T_{before}, T_{after}, P, and an oracle A that does not store the pairwise P2P exact shortest paths on T_{before}, UR(FU-Oracle) $\leq$ UR(A).*

For example, *WSPD-Oracle-Adapt* is an instance of A, we use it for illustration. In Figure 5, for *WSPD-Oracle-Adapt*, recall that the disk centered at each POI has a larger radius, and these disks intersect ΔF, so we need to perform algorithm *SSAD* with a, b, c, d, e, f, g (7 POIs) as a source for path updating on T_{after}. We have *UR*(*FU-Oracle*) $= \frac{7}{8}$, so the oracle update time of *WSPD-Oracle-Adapt* is 2.4 times larger than that of *FU-Oracle*. In our experiments, the oracle update time of *WSPD-Oracle-Adapt* is up to 21 times larger than that of *FU-Oracle*. This is because some of the selected POIs (e.g., villages, hospitals, and expressway exits) are close to each other. In the earthquake, we aim at minimizing the oracle update time for finding rescue paths faster, so it is necessary to store the pairwise P2P exact shortest paths on T_{before} in G'.

V. Empirical Study

A. Experimental Setup

We conduct our 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 [35], [36], [41], [54], [55].

TABLE I
Real earthquake terrain datasets

Name	Magnitude	Date	$\|F\|$
Tohoky, Japan (*TJ*) [8]	9.0	Mar 11, 2011	**0.5M**, 1M, 1.5M, 2M, 2.5M
Sichuan, China (*SC*) [49]	8.0	May 12, 2008	**0.5M**, 1M, 1.5M, 2M, 2.5M
Gujarat, India (*GI*) [7]	7.6	Jan 26, 2001	**0.5M**, 1M, 1.5M, 2M, 2.5M
Alaska, USA (*AU*) [1]	7.1	Nov 30, 2018	**0.5M**, 1M, 1.5M, 2M, 2.5M
Leogane, Haiti (*LH*) [46]	7.0	Jan 12, 2010	**0.5M**, 1M, 1.5M, 2M, 2.5M
Valais, Switzerland (*VS*) [12]	4.1	Oct 24, 2016	**0.5M**, 1M, 1.5M, 2M, 2.5M

TABLE II
Comparison of algorithms

Algorithm	Oracle construction time	Oracle update time	Output size	Shortest path query time
Oracle-based algorithm				
WSPD-Oracle [54], [55]	Large	Large	Large	Small
WSPD-Oracle-Adapt [54], [55]	Large	Large	Small	Small
EAR-Oracle [32]	Large	Large	Large	Medium
EAR-Oracle-Adapt [32]	Large	Large	Small	Small
FU-Oracle-RanUpdSeq	Small	Large	Small	Small
FU-Oracle-FullRad	Small	Medium	Small	Small
FU-Oracle-NoDistAppr	Small	Large	Small	Small
FU-Oracle-NoEffIntChe	Small	Medium	Small	Small
FU-Oracle-NoEdgPru	Small	Small	Large	Small
FU-Oracle-NoEffEdgPru [18], [19]	Small	Medium	Small	Small
***FU-Oracle* (ours)**	**Small**	**Small**	**Small**	**Small**
On-the-fly algorithm				
CH-Fly-Algo [23]	N/A	N/A	N/A	Large
K-Fly-Algo [35]	N/A	N/A	N/A	Large

1) ***Datasets:*** We conduct our experiment on 30 real before and after earthquake terrain datasets listed in Table I with 0.5M faces by default. We first obtain the earthquake terrain satellite maps with a 5km × 5km covered region from Google Earth [6] with a resolution of 10m [27], [41], [51], [54], [55], and then we use Blender [2] to generate the terrain model. In order to study the scalability, we follow an existing generation procedure for multi-resolution terrain datasets [41], [54], [55] to obtain different resolutions of these datasets with 1M, 1.5M, 2M, 2.5M faces. This procedure appears in the appendix. We extract 500 POIs using OpenStreetMap [54], [55].

2) ***Algorithms:*** We include the the best-known exact on-the-fly algorithm *CH-Fly-Algo* [23], the best-known approximate on-the-fly algorithm *K-Fly-Algo* [35], the best-known P2P path query oracle *WSPD-Oracle* [54], [55], its adaption *WSPD-Oracle-Adapt*, the most recent oracle *EAR-Oracle* [32], and its adaption *EAR-Oracle-Adapt* as baselines. In the update phase of *WSPD-Oracle-Adapt* and *EAR-Oracle-Adapt*, the only implementation detail difference between them and *FU-Oracle* is that, for the *sub-graph generating* step, we always add the non-updated *approximate* paths into G and H for error guarantee in the *edges examining on H* step (but in *FU-Oracle*, we need to examine the non-updated *exact* paths before adding them into G and H). *WSPD-Oracle* and *WSPD-Oracle-Adapt* reveal the two major reasons for the short oracle update time of *FU-Oracle*, i.e., (1) *WSPD-Oracle* does not utilize the non-updated terrain shortest path intact property, and (2) *WSPD-Oracle-Adapt* does not store the pairwise P2P exact shortest paths on T_{before} when the oracle is constructed.

Furthermore, in *FU-Oracle*, we (1) use a random path update sequence instead of our novel path update sequence,

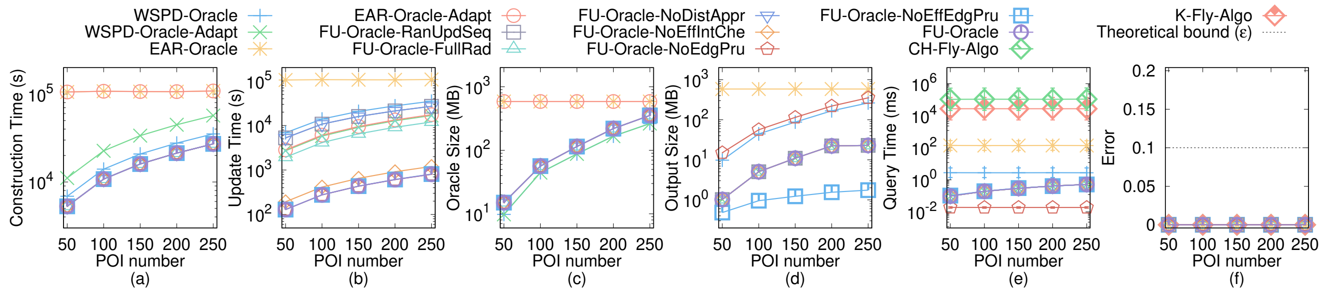

Fig. 7. Effect of n on *GI* dataset (fewer POIs)

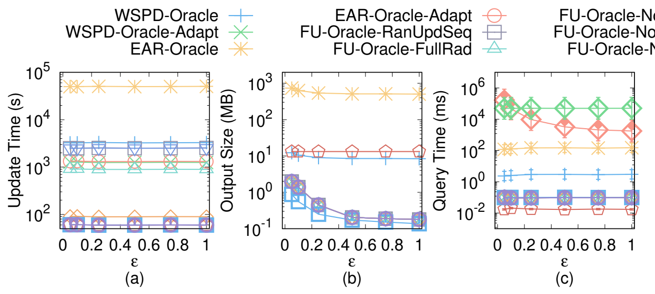

Fig. 8. Effect of ϵ on *LH* dataset (fewer POIs)

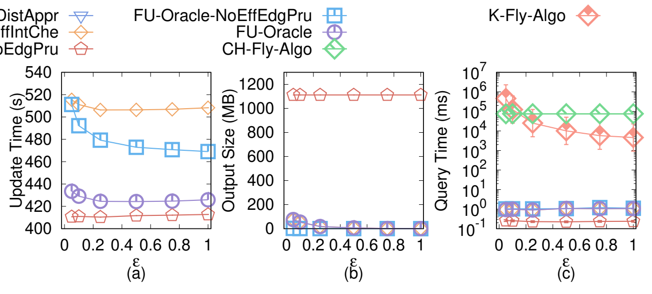

Fig. 9. Effect of ϵ on *TJ* dataset (more POIs)

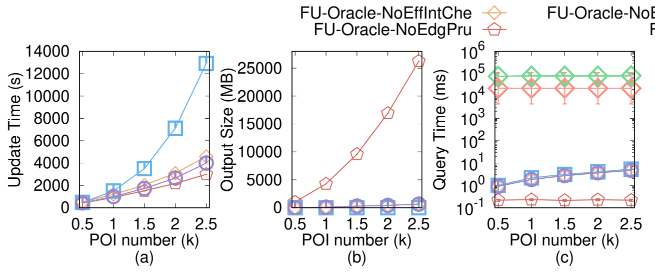

Fig. 10. Effect of n on *AU* dataset (more POIs)

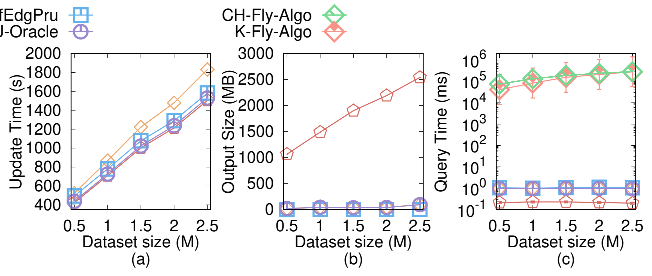

Fig. 11. Effect of *DS* on *VS* dataset (more POIs)

(2) use the full shortest distance of a shortest path as the disk radius instead of our novel disk radius selection strategy, (3) do not store the POI-to-vertex distance information and re-calculate the shortest path on T_{after} for determining whether the disk intersects with ΔF instead of our novel distance approximation approach, (4) create two disks for each path when checking whether we need to re-calculate the shortest path between a pair of POIs instead of our disk & updated faces intersection checking approach, (5) remove the sub-graph generating step, i.e., algorithm *HieGreSpan* in the update phase and use a hash table to store the pairwise P2P exact shortest paths on T_{after} in G', and (6) use algorithm *GreSpan* [18], [19] instead of algorithm *HieGreSpan* in the sub-graph generating step of the update phase, in an **ablation study**. We use *FU-Oracle-X* where ***X*** = {*RanUpdSeq, FullRad, NoDistAppr, NoEffIntChe, NoEdgPru, NoEffEdgPru*} to denote these baseline oracles. The first four baseline oracles correspond to the four techniques in Section IV-B. The last two baseline oracles correspond to the idea in Section IV-C.

In Table II, we compare 13 algorithms in terms of oracle construction time, oracle update time, output size, and shortest path query time. The detailed theoretical analysis with proofs for the baselines appear in the appendix. *FU-Oracle* performs better than existing oracles in terms of all performance metrics.

*3) **Query generation**:* We randomly choose pairs of POIs in P on T_{after}, as source and destination, 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 (1) n, (2) ϵ, and (3) dataset size *DS* (i.e., the number of faces in a terrain model). We consider six performance metrics, namely (1) *oracle construction time*, (2) *oracle update time*, (3) *oracle size* (i.e., the space consumption of G', M_{dist}, and H), (4) *output size* (i.e., the space consumption of G), (5) *shortest path query time*, and (6) *distance error* (i.e., the error of the distance returned by the algorithm compared with the exact shortest distance).

B. Experimental Results

Figure 7 and Figure 8 show the P2P path query results on *GI* and *LH* datasets (with fewer POIs) when varying n and ϵ, respectively. Figure 9 - 11 show the results on *TJ*, *AU*, and *VS* datasets (with more POIs) when varying ϵ, n, and *DS*, respectively. For the shortest path query time, the vertical bar and the points denote the minimum, maximum, and average results. The results on (1) other combinations of datasets and the variation of n, ϵ, and *DS*, (2) the P2P path query in the case $n > N$, (3) the V2V, and (4) the A2A path queries appear in the appendix. Our experimental results show that *WSPD-Oracle*, *WSPD-Oracle-Adapt*, *EAR-Oracle*, *EAR-Oracle-Adapt*, and *FU-Oracle-**X*** where ***X*** = {*RanUpdSeq, FullRad, NoDistAppr*} have excessive oracle update times with 500 POIs (more than 1 days), we (1) compare these 13 algorithms on 30 datasets with fewer POIs (50 by default), and (2) compare *FU-Oracle-**X*** where ***X*** = {*NoEffIntChe, NoEdgPru, NoEffEdgPru*}, *FU-Oracle*, *CH-Fly-Algo*, and *K-Fly-Algo* on 30 datasets with more POIs (500 by default).

*1) **Effect of n for the P2P path query**:* In Figure 7 (resp. Figure 10), we tested the 5 values of n in {50, 100, 150, 200, 250} on *GI* (resp. {500, 1000, 1500, 2000, 2500} on *AU*) dataset while fixing ϵ at 0.1 and *DS* at 0.5M (resp. ϵ to 0.25 and *DS* to 0.5M). Although *FU-Oracle* and other baselines have the similar small error (close to 0%) which are much smaller than the theoretical bound, *FU-Oracle* offers superior performance over *WSPD-Oracle*, *WSPD-Oracle-Adapt*, *EAR-Oracle*, *EAR-Oracle-Adapt*, *CH-Fly-Algo*, and *K-Fly-Algo* in terms of oracle construction time, oracle update time, output size, and shortest path query time. In Figure 7 (b) (resp. Figure 10 (a)), the oracle update time for *FU-Oracle-**X***, where ***X*** = {*RanUpdSeq, FullRad, NoDistAppr, NoEffIntChe*} (resp. ***X*** = {*NoEffIntChe, NoEffEdgPru*}) exceed that of *FU-Oracle*. In Figure 10 (a), the oracle update time of *FU-Oracle-NoEffEdgPru* is 4 times larger than that of *FU-Oracle* when n is large, although the output size of *FU-Oracle-NoEffEdgPru* is slightly smaller than that of *FU-Oracle* in Figure 10 (b). In Figure 7 (c), although the oracle size of *FU-Oracle* is slightly larger than that of *WSPD-Oracle* and *WSPD-Oracle-Adapt*, the oracle update time of *FU-Oracle* is 88 times and 21 times smaller that of *WSPD-Oracle* and *WSPD-Oracle-Adapt*. In Figure 10 (b), the output size for *FU-Oracle-NoEdgPru* is 10^4 times larger than *FU-Oracle*, although the shortest path query time of *FU-Oracle-NoEdgPru* is slightly smaller than that of *FU-Oracle* in Figure 10 (c).

*2) **Effect of ϵ for the P2P path query**:* In Figure 8 (resp. Figure 9), we tested the 6 values of ϵ in {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *LH* dataset with fewer POIs while fixing n at 50 (resp. on *TJ* dataset with more POIs while fixing n at 500) and *DS* at 0.5M. The oracle update time, output size, and shortest path query time of *FU-Oracle* remain better than those of the baselines. In Figure 9, although the oracle update time and the shortest path query time of *FU-Oracle* is slightly larger than that of *FU-Oracle-NoEdgPru*, the latter one's output size is larger. The oracle update time of *FU-Oracle-NoEffEdgPru* is larger than that of *FU-Oracle*. Varying ϵ has a small impact on the oracle update time, since when n is small, the pairwise P2P exact shortest paths updating step dominates the sub-graph generating step, and the former step is independent of ϵ.

*3) **Effect of DS (scalability test) for the P2P path query**:* In Figure 11, we tested the 5 values of *DS* in {0.5M, 1M, 1.5M, 2M, 2.5M} on *VS* dataset with more POIs while fixing ϵ to 0.25 and n to 500 to study scalability (we also have the results with fewer POIs while fixing ϵ to 0.1 and n to 50 in the appendix). Varying *DS* has a small impact on the shortest path query time of *FU-Oracle*, but has a large impact on that of *CH-Fly-Algo* and *K-Fly-Algo*. The shortest path query time of *FU-Oracle* is 10^5 times smaller than that of *K-Fly-Algo*.

*4) **P2P path query in the case $n > N$, V2V path query, and A2A path query**:* We tested the P2P path query in the case $n > N$, V2V path query, and A2A path query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and fixing N at 2k on a small-version of *SC* dataset. The result appears in the appendix. The results are similar to those for the P2P path query in the case $n \leq N$. For the A2A path query, although *EAR-Oracle* is regarded as the best-known oracle, *FU-Oracle* still performs better than *EAR-Oracle* and *EAR-Oracle-Adapt* in terms of oracle update time. In particular, the oracle update time of *FU-Oracle* is up to 15 times and 9 times smaller than that of *EAR-Oracle* and *EAR-Oracle-Adapt*.

*5) **Case study**:* We conducted a case study on the 4.1 magnitude earthquake (which caused an avalanche) that occurred on October 24, 2016 in Valais, Switzerland [12], i.e., the *VS* dataset used in our experiment. Figure 3 (b) shows an example of the avalanche that caused the terrain surface updates. After the avalanche, the original paths (i.e., the purple dashed lines) between a (a village) and b (a hotel), and c (a village) and d (a hotel) pass ΔF (i.e., the orange area). In order to let the rescue team go from a to b, and from c to d, the new paths (i.e., the blue lines) are calculated using *FU-Oracle*. On a terrain dataset with 0.5M faces and 250 POIs, *FU-Oracle* just needs 400s ≈ 7 min to update the oracle, while the best-known oracle *WSPD-Oracle* needs 35,100s ≈ 9.8 hours.

*6) **Summary**:* *FU-Oracle* outperforms all existing oracles, i.e., *WSPD-Oracle*, *WSPD-Oracle-Adapt*, *EAR-Oracle*, and *EAR-Oracle-Adapt* in terms of oracle construction time, oracle update time, output size and shortest path query time. Specifically, *FU-Oracle* is up to 1.3 times, 88 times, 12 times, and 3 times better than the best-known oracle i.e., *WSPD-Oracle*, in terms of these performance metrics. For a terrain dataset with 0.5M faces and 250 POIs, *FU-Oracle*'s oracle update time is 400s ≈ 7 min, while the best-known oracle *WSPD-Oracle* needs 35,100s ≈ 9.8 hours, and the adapted best-known oracle *WSPD-Oracle-Adapt* needs 8,400s ≈ 2.4 hours. When dataset size is 0.5M and with $\epsilon = 0.05$, the shortest path query time for computing 100 paths is 0.1s for *FU-Oracle*, while the time is 8,600s ≈ 2.4 hours for *K-Fly-Algo*, and both 0.3s for *WSPD-Oracle* and *WSPD-Oracle-Adapt*.

VI. Conclusion

We propose an efficient $(1+\epsilon)$-approximate shortest path oracle on an updated terrain surface called *FU-Oracle*, which has state-of-the-art performance in terms of oracle construction time, oracle update time, output size, and shortest path query time compared with the best-known oracle. Future work can be proposing additional new pruning steps in *FU-Oracle* to further reduce the oracle update time.

References

[1] "2018 anchorage earthquake," 2023. [Online]. Available: https://www.usgs.gov/news/featured-story/2018-anchorage-earthquake

[2] "Blender," 2023. [Online]. Available: https://www.blender.org
[3] "China national space administration," 2023. [Online]. Available: https://www.cnsa.gov.cn/english/
[4] "Cyberpunk 2077," 2023. [Online]. Available: https://www.cyberpunk.net
[5] "Falcon 9," 2023. [Online]. Available: https://www.spacex.com/vehicles/falcon-9/
[6] "Google earth," 2023. [Online]. Available: https://earth.google.com/web
[7] "Gujarat earthquake, 2001," 2023. [Online]. Available: https://www.actionaidindia.org/emergency/gujarat-earthquake-2001/
[8] "Mar 11, 2011: Tohoku earthquake and tsunami," 2023. [Online]. Available: https://education.nationalgeographic.org/resource/tohoku-earthquake-and-tsunami/
[9] "Mars 2020 mission perseverance rover brains," 2023. [Online]. Available: https://mars.nasa.gov/mars2020/spacecraft/rover/brains/
[10] "Mars 2020 mission perseverance rover communications," 2023. [Online]. Available: https://www.statista.com/chart/24232/life-cycle-costs-of-mars-missions/
[11] "Metaverse," 2023. [Online]. Available: https://about.facebook.com/meta
[12] "Moderate mag. 4.1 earthquake - 6.3 km northeast of sierre, valais, switzerland, on monday, october 24, 2016 at 16:44 gmt," 2023. [Online]. Available: https://www.volcanodiscovery.com/earthquakes/quake-info/1451397/mag4quake-Oct-24-2016-Leukerbad-VS.html
[13] "Nasa mars exploration," 2023. [Online]. Available: https://mars.nasa.gov
[14] "Spacex," 2023. [Online]. Available: https://www.spacex.com
[15] "Turkey–syria earthquakes 2023," 2023. [Online]. Available: https://www.bbc.com/news/topics/cq0zxdd0y39t
[16] "Why are we having so many earthquakes?" 2023. [Online]. Available: https://www.usgs.gov/faqs/why-are-we-having-so-many-earthquakes-has-naturally-occurring-earthquake-activity-been
[17] C. Alicia, "Cost of nasa's next mars rover soars to $2.5 billion," 2011. [Online]. Available: https://www.nbcnews.com/id/wbna41374241
[18] I. Althöfer, G. Das, D. Dobkin, and D. Joseph, "Generating sparse spanners for weighted graphs," in *Proceedings of the Scandinavian Workshop on Algorithm Theory*, 1990, pp. 26–37.
[19] 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.
[20] 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.
[21] B. Awerbuch, "Communication-time trade-offs in network synchronization," in *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, 1985, pp. 272–276.
[22] P. B. Callahan and S. R. Kosaraju, "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *Journal of the ACM*, vol. 42, no. 1, pp. 67–90, 1995.
[23] J. Chen and Y. Han, "Shortest paths on a polyhedron," in *Proceedings of the sixth annual symposium on Computational geometry*, New York, NY, USA, 1990, p. 360–369.
[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.
[25] G. Das and G. Narasimhan, "A fast algorithm for constructing sparse euclidean spanners," in *Proceedings of the tenth annual symposium on Computational geometry*, 1994, pp. 132–139.
[26] K. Deng, H. T. Shen, K. Xu, and X. Lin, "Surface k-nn query processing," in *Proceedings of the International Conference on Data Engineering*. IEEE, 2006, pp. 78–78.
[27] K. Deng and X. Zhou, "Expansion-based algorithms for finding single pair shortest path on surface," in *Proceedings of the International Workshop on Web and Wireless Geographical Information Systems*, 2004, pp. 151–166.
[28] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
[29] 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.
[30] 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.
[31] 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
[32] 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," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.
[33] B. M. Jaime, "Is nasa able to remotely repair the mars rover?" 2021. [Online]. Available: https://www.quora.com/Is-NASA-able-to-remotely-repair-the-Mars-rover
[34] S. Kapoor, "Efficient computation of geodesic shortest paths," in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, 1999, pp. 770–779.
[35] M. Kaul, R. C.-W. Wong, and C. S. Jensen, "New lower and upper bounds for shortest distance queries on terrains," *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 168–179, 2015.
[36] M. Kaul, R. C.-W. Wong, B. Yang, and C. S. Jensen, "Finding shortest paths on terrains by killing two birds with one stone," *Proceedings of the VLDB Endowment*, vol. 7, no. 1, pp. 73–84, 2013.
[37] 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.
[38] B. Kégl, "Intrinsic dimension estimation using packing numbers," *Advances in neural information processing systems*, vol. 15, 2002.
[39] M. Lanthier, A. Maheshwari, and J.-R. Sack, "Approximating shortest paths on weighted polyhedral surfaces," *Algorithmica*, vol. 30, no. 4, pp. 527–562, 2001.
[40] 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
[41] L. Liu and R. C.-W. Wong, "Finding shortest path on land surface," in *Proceedings of the ACM SIGMOD International Conference on Management of data*, 2011, pp. 433–444.
[42] 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/
[43] J. S. Mitchell, D. M. Mount, and C. H. Papadimitriou, "The discrete geodesic problem," *SIAM Journal on Computing*, vol. 16, no. 4, pp. 647–668, 1987.
[44] 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.
[45] B. Padlewska, "Connected spaces," *Formalized Mathematics*, vol. 1, no. 1, pp. 239–244, 1990.
[46] R. Pallardy, "2010 haiti earthquake," 2023. [Online]. Available: https://www.britannica.com/event/2010-Haiti-earthquake
[47] 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.
[48] D. Peleg and J. D. Ullman, "An optimal synchronizer for the hypercube," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 77–85.
[49] K. Pletcher and J. P. Rafferty, "Sichuan earthquake of 2008," 2023. [Online]. Available: https://www.britannica.com/event/Sichuan-earthquake-of-2008
[50] C. Power, "What's driving china's race to build a space station?" 2023. [Online]. Available: https://chinapower.csis.org/chinese-space-station/
[51] C. Shahabi, L.-A. Tang, and S. Xing, "Indexing land surface for efficient knn query," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1020–1031, 2008.
[52] 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.
[53] P. Von Rickenbach and R. Wattenhofer, "Gathering correlated data in sensor networks," in *Proceedings of the joint workshop on Foundations of mobile computing*, 2004, pp. 60–66.
[54] V. J. Wei, R. C.-W. Wong, C. Long, D. Mount, and H. Samet, "Proximity queries on terrain surface," *ACM Transactions on Database Systems*, vol. 47, no. 4, pp. 1–59, 2022.
[55] V. J. Wei, R. C.-W. Wong, C. Long, and D. M. Mount, "Distance oracle on terrain surface," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1211–1226.

[56] 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.

[57] S. Xing, C. Shahabi, and B. Pan, "Continuous monitoring of nearest neighbors on land surface," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1114–1125, 2009.

[58] D. Yan, Z. Zhao, and W. Ng, "Monochromatic and bichromatic reverse nearest neighbor queries on land surfaces," in *Proceedings of the 21st ACM international conference on Information and knowledge management*, 2012, pp. 942–951.

[59] Y. Yan and R. C.-W. Wong, "Path advisor: a multi-functional campus map tool for shortest path," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 2683–2686, 2021.

Appendix A
Summary of Frequent Used Notations

Table III shows a summary of frequent used notations.

TABLE III
SUMMARY OF FREQUENT USED NOTATIONS

Notation	Meaning
T_{before}/T_{after}	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_{before}
N	The number of vertices of T
ΔV	The updated vertices of T_{before} and T_{after}
ΔE	The updated edges edges of T_{before} and T_{after}
ΔF	The updated faces of T_{before} and T_{after}
P	The set of POI
n	The number of vertices of P
ΔP	The updated POIs on T_{before} and T_{after}
$\Pi(s,t\|T)$	The exact shortest path between s and t on the surface of T
$\|\Pi(s,t\|T)\|$	The distance of $\Pi(s,t\|T)$
G	The *FU-Oracle* output graph
$G.V/G.E$	The set of vertices / edges of G
$e(u,v\|T)$	An edge between u and v in $G.E$
$\Pi_G(s,t\|T)$	The shortest path between s and t in G
$\|\Pi_G(s,t\|T)\|$	The distance of $\Pi_G(s,t\|T)$
ϵ	The error parameter
G'	The temporary complete graph
$G'.V/G'.E$	The set of vertices / edges of G'
$e'(u,v\|T)$	An edge between u and v in $G'.E$
$\Pi(u)$	A set of the exact shortest paths stored in G' on T_{before} with u as an endpoint and $v_i \in P_{remain} \setminus u$, $i \in \{1,l\}$ as the other endpoint, such that all these paths has not been updated
P_{remain}	A set of remaining POIs of P on T_{after} that we have not processed
D	The longest edge's weight in $G'.E$
Q_G	A group of vertices in G on H
$e_H(u,v\|T)$	An edge between u and v in H
$\Pi_H(s,t\|T)$	The shortest path of inter-edges between s and t in H

Appendix B
Comparison of All Algorithms

Table IV shows a comparison of all algorithms in terms of the oracle construction time, oracle update time, output size, and shortest path query time.

Appendix C
V2V Path Query

Apart from the P2P path query that we discussed in the main body of this paper, we also present an oracle to answer the *vertex-to-vertex (V2V) path query* based on our oracle *FU-Oracle*. This adapted oracle is similar to the one presented in Section IV, the only difference is that we need to create POIs which has the same coordinate values as vertices in V, then *FU-Oracle* can answer the V2V path query. In this case, the number of POI becomes N. Thus, for the V2V path query, the oracle construction time, oracle update time, output size, and shortest path query time of *FU-Oracle* that answers the V2V path query are $O(N^2 \log^2 N)$, $O(N \log^2 N)$, $O(N)$, and $O(\log N)$, respectively. *FU-Oracle* satisfies $|\Pi_G(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ for all pairs of vertices u and v in V.

Appendix D
A2A Path Query

Apart from the P2P path query that we discussed in the main body of this paper, we also present an oracle to answer the *arbitrary point-to-arbitrary point (A2A) path query* based on our oracle *FU-Oracle*. This adapted oracle is similar to the one presented in Section IV, the only difference is that we need to use Steiner points as input instead of using POIs as input, where the Steiner points are introduced using the method in [29]. Specifically, [29] places some Steiner points on the terrain surface (there are total $O(\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon})$, where θ means the minimum inner angle of any face), and by using these Steiner points as input, we follow the construction phase and the update phase of *FU-Oracle* for oracle construction and update. For the query phase, given two arbitrary points u and v, we first find the neighborhood of u (resp. v), denoted by $\mathcal{N}(u)$ (resp. $\mathcal{N}(v)$), which is a set of Steiner points on the same face containing u (resp. v) and its adjacent faces [29]. Then, we return $\Pi_G(u,v|T) = \min_{p\in\mathcal{N}(u),q\in\mathcal{N}(v)}[\Pi(u,p|T) + \Pi_G(p,q|T) + \Pi(q,v|T)]$, where $\Pi(u,p|T)$ and $\Pi(q,v|T)$ can be calculated in $O(1)$ time using algorithm *SSAD* and $\Pi_G(p,q|T)$ is the distance between p and q returned by *FU-Oracle*. According to [29], $|\mathcal{N}(u)| \cdot |\mathcal{N}(v)| = \frac{1}{\sin\theta\epsilon}$, and if $|\Pi_G(p,q|T)| \leq (1+\epsilon)|\Pi(p,q|T)|$, then $|\Pi_G(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$. Thus, for the A2A path query, by setting $n = \frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon}$ in the original *FU-Oracle* that answers the P2P path query, we obtain that the oracle construction time, oracle update time, output size, and shortest path query time of *FU-Oracle* that answers the A2A path query are $O(\frac{N^2\log^2 N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon})$, $O(N\log^2 N + \frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log^2(\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$, $O(\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon})$, and $O(\log(\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$, respectively. *FU-Oracle* satisfies $|\Pi_G(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ for all pairs of arbitrary points u and v on T.

Appendix E
P2P Path Query in the Case $n > N$

Apart from the P2P path query when $n \leq N$ that we discussed in the main body of this paper, we also present an oracle to answer the P2P path query in the case $n > N$ based on our oracle *FU-Oracle*. We adopt the same oracle for answering the A2A path query, which is POI-independent.

TABLE IV
COMPARISON OF ALGORITHMS WITH DETAILS

Algorithm	Oracle construction time		Oracle update time		Output size		Shortest path query time	
Oracle-based algorithm								
WSPD-Oracle [54], [55]	$O(\frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Large	$O(h^2)$	Small
WSPD-Oracle-Adapt [54], [55]	$O(\frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\mu_1 N\log^2 N + n\log^2 n)$	Large	$O(n)$	Small	$O(\log n)$	Small
EAR-Oracle [32]	$O(\lambda\xi mN\log^2(mN) + \frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\lambda\xi mN\log^2(mN) + \frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi\log(\lambda\xi))$	Medium
EAR-Oracle-Adapt [32]	$O(\lambda\xi mN\log^2(mN) + \frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\mu_2 N\log^2 N + n\log^2 n)$	Large	$O(n)$	Small	$O(\log n)$	Small
FU-Oracle-RanUpdSeq	$O(nN\log^2 N)$	Small	$O(nN\log^2 N + n\log^2 n)$	Large	$O(n)$	Small	$O(\log n)$	Small
FU-Oracle-FullRad	$O(nN\log^2 N)$	Small	$O(\mu_3 N\log^2 N + n\log^2 n)$	Medium	$O(n)$	Small	$O(\log n)$	Small
FU-Oracle-NoDistAppr	$O(nN\log^2 N)$	Small	$O(nN\log^2 N + n\log^2 n)$	Large	$O(n)$	Small	$O(\log n)$	Small
FU-Oracle-NoEffIntChe	$O(nN\log^2 N)$	Small	$O(N\log^2 N + n^2)$	Medium	$O(n)$	Small	$O(\log n)$	Small
FU-Oracle-NoEdgPru	$O(nN\log^2 N + n^2)$	Small	$O(N\log^2 N + n)$	Small	$O(n^2)$	Large	$O(1)$	Small
FU-Oracle-NoEffEdgPru [18], [19]	$O(nN\log^2 N)$	Small	$O(N\log^2 N + n^3\log n)$	Medium	$O(n)$	Small	$O(\log n)$	Small
***FU-Oracle* (ours)**	$\boldsymbol{O(nN\log^2 N)}$	**Small**	$\boldsymbol{O(N\log^2 N + n\log^2 n)}$	**Small**	$\boldsymbol{O(n)}$	**Small**	$\boldsymbol{O(\log n)}$	**Small**
On-the-fly algorithm								
CH-Fly-Algo [23]	-	N/A	-	N/A	-	N/A	$O(N^2)$	Large
K-Fly-Algo [35]	-	N/A	-	N/A	-	N/A	$O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}} \log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$	Large

Remark: $n < N$, h is the height of the compressed partition tree, β is the largest capacity dimension [38], [30], μ_1, μ_2, and μ_3 are data-dependent variables, $\mu_1 \in [5, 20]$, $\mu_2 \in [12, 45]$ and $\mu_3 \in [5, 10]$ in our experiment.

This oracle can not only answer A2A path query, but also P2P path query (no matter whether $n \leq N$ or $n > N$) and V2V path query, since A2A path query generalizes both P2P and V2V path query.

APPENDIX F
EMPIRICAL STUDIES

A. Experimental Results on the P2P Path Query

(1) Figure 12, (2) Figure 13, (3) Figure 14 show the result on the *TJ* dataset (with fewer POIs) when varying n, ϵ, and *DS*, respectively. (4) Figure 15, (5) Figure 16, (6) Figure 17 show the result on the *SC* dataset (with fewer POIs) when varying n, ϵ, and *DS*, respectively. (7) Figure 7, (8) Figure 18, (9) Figure 19 show the result on the *GI* dataset (with fewer POIs) when varying n, ϵ, and *DS*, respectively. (10) Figure 20, (11) Figure 21, (12) Figure 22 show the result on the *AU* dataset (with fewer POIs) when varying n, ϵ, and *DS*, respectively. (13) Figure 23, (14) Figure 24, (15) Figure 25 show the result on the *LH* dataset (with fewer POIs) when varying n, ϵ, and *DS*, respectively. (16) Figure 26, (17) Figure 27, (18) Figure 28 show the result on the *VS* dataset (with fewer POIs) when varying n, ϵ, and *DS*, respectively. (19) Figure 29, (20) Figure 30, (21) Figure 31 show the result on the *TJ* dataset (with more POIs) when varying n, ϵ, and *DS*, respectively. (22) Figure 32, (23) Figure 33, (24) Figure 34 show the result on the *SC* dataset (with more POIs) when varying n, ϵ, and *DS*, respectively. (25) Figure 35, (26) Figure 36, (27) Figure 37 show the result on the *GI* dataset (with more POIs) when varying n, ϵ, and *DS*, respectively. (28) Figure 38, (29) Figure 39, (30) Figure 40 show the result on the *AU* dataset (with more POIs) when varying n, ϵ, and *DS*, respectively. (31) Figure 41, (32) Figure 42, (33) Figure 43 show the result on the *LH* dataset (with more POIs) when varying n, ϵ, and *DS*, respectively. (34) Figure 44, (35) Figure 45, (36) Figure 46 show the result on the *VS* dataset (with more POIs) when varying n, ϵ, and *DS*, respectively.

Effect of n. In Figure 12, Figure 15, Figure 7, Figure 20, Figure 23 and Figure 26, we tested the 5 values of n in {50, 100, 150, 200, 250} on *TJ*, *SC*, *GI*, *AU*, *LH* and *VS* dataset while fixing ϵ at 0.1 and *DS* at 0.5M. In Figure 29, Figure 32, Figure 35, Figure 38, Figure 41 and Figure 44, we tested the 5 values of n in {500, 1000, 1500, 2000, 2500} on *TJ*, *SC*, *GI*, *AU*, *LH* and *VS* datasets while fixing ϵ at 0.25 and *DS* at 0.5M. *FU-Oracle* superior performance of *WSPD-Oracle*, *WSPD-Oracle-Adapt*, *EAR-Oracle*, *EAR-Oracle-Adapt*, *CH-Fly-Algo*, and *K-Fly-Algo* in terms of oracle construction time, oracle update time, output size, and shortest path query time. When n is small, it is clear that the oracle update time for *FU-Oracle-**X*** where ***X*** = {*RanUpdSeq, FullRad, NoDistAppr, NoEffIntChe*} are larger than *FU-Oracle*. The oracle update time of *FU-Oracle-NoEffEdgPru* is 4 times larger than that of *FU-Oracle* when n is large, although the output size of *FU-Oracle-NoEffEdgPru* is slightly smaller than that of *FU-Oracle*. The output size for *FU-Oracle-NoEdgPru* is larger than *FU-Oracle*, although the shortest path query time of *FU-Oracle-NoEdgPru* is slightly smaller than that of *FU-Oracle*. Thus, these show the superior performance of *FU-Oracle* in terms of oracle update time and output size.

Effect of ϵ. In Figure 13, Figure 16, Figure 18, Figure 21, Figure 24 and Figure 27, we tested the 6 values of ϵ in {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *TJ*, *SC*, *GI*, *AU*, *LH* and *VS* datasets (with fewer POIs) while fixing n at 50 and *DS* at 0.5M. In Figure 30, Figure 33, Figure 36, Figure 39, Figure 42 and Figure 45, we tested the 6 values of ϵ in {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *TJ*, *SC*, *GI*, *AU*, *LH* and *VS* datasets (with fewer POIs) while fixing n at 500 and *DS* at 0.5M. The oracle

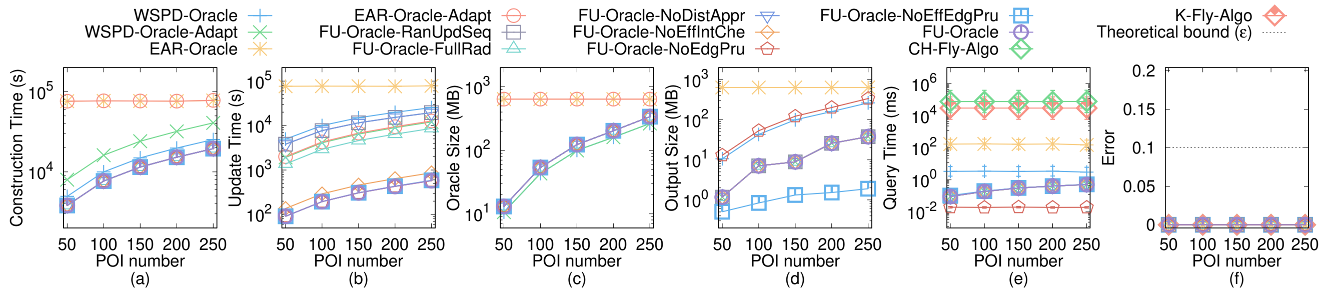

Fig. 12. Effect of n on *TJ* dataset (fewer POIs) for the P2P path query

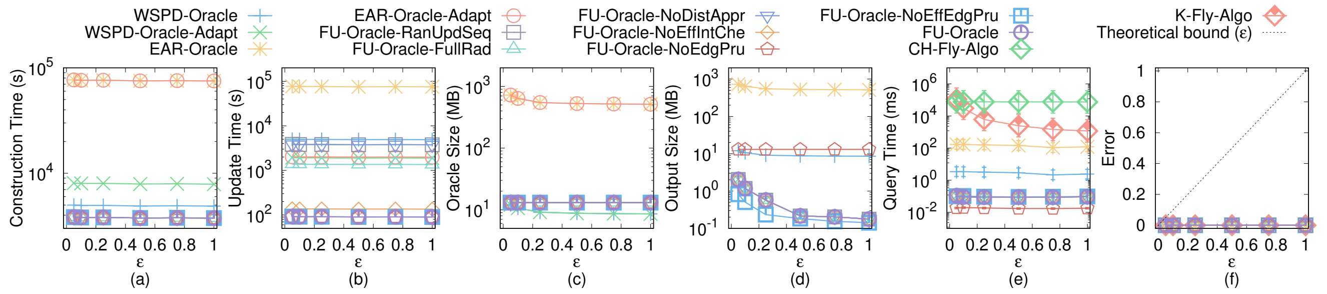

Fig. 13. Effect of ϵ on *TJ* dataset (fewer POIs) for the P2P path query

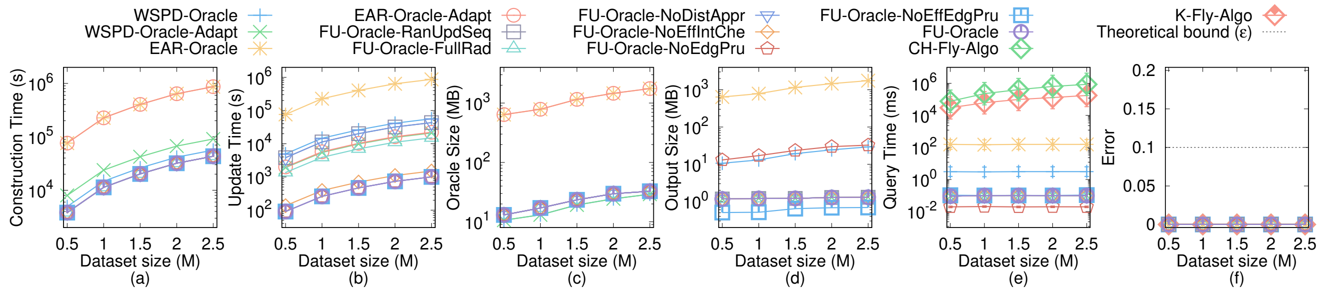

Fig. 14. Effect of *DS* on *TJ* dataset (fewer POIs) for the P2P path query

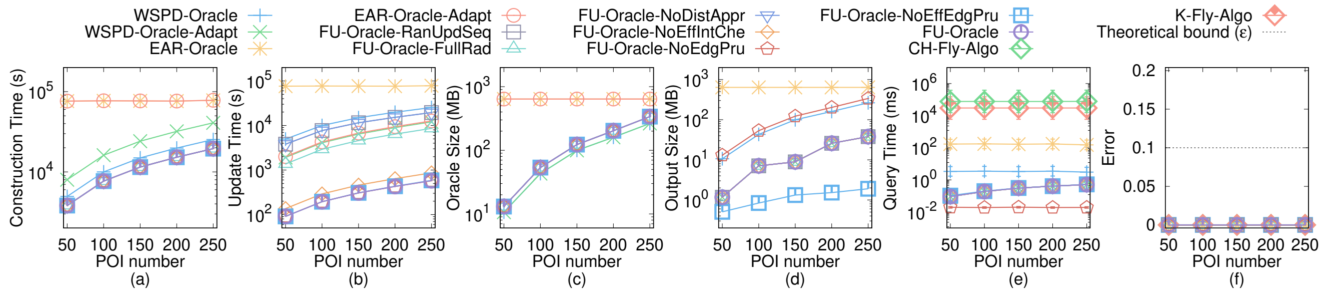

Fig. 15. Effect of n on *SC* dataset (fewer POIs) for the P2P path query

update time, output size, and shortest path query time of *FU-Oracle* still perform better than other baselines. Although the oracle update time and the shortest path query time of *FU-Oracle* is slightly larger than that of *FU-Oracle-NoEdgPru*, the latter one's output size is larger. The oracle update time of *FU-Oracle-NoEffEdgPru* is larger than that of *FU-Oracle*. The errors of all the algorithms are very small (close to 0%) and much smaller than the theoretical bound.

Effect of *DS* (scalability test). In Figure 14, Figure 17, Figure 19, Figure 22, Figure 25 and Figure 28, we tested the 5 values of *DS* in {0.5M, 1M, 1.5M, 2M, 2.5M} on *TJ*, *SC*, *GI*, *AU*, *LH* and *VS* datasets (with fewer POIs) while fixing ϵ at 0.1 and n at 50. In Figure 31, Figure 34, Figure 37, Figure 40, Figure 43 and Figure 46, we the tested 5 values of *DS* in {0.5M, 1M, 1.5M, 2M, 2.5M} on *TJ*, *SC*, *GI*, *AU*, *LH* and *VS* datasets (with fewer POIs) while fixing ϵ at 0.25 and n at 500. Varying *DS* has a small impact on the shortest path query time of *FU-Oracle*, but has a large impact on that of *K-Fly-Algo*. The shortest path query time of *FU-Oracle* is 10^5 smaller than *K-Fly-Algo*.

B. Experimental Results on the V2V Path Query

In Figure 47, we tested the V2V path query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and fixing N at 2k on a multi-resolution of *SC* dataset. It still shows that *FU-Oracle* superior performance of *WSPD-Oracle*, *WSPD-Oracle-Adapt*, *EAR-Oracle*, *EAR-Oracle-Adapt*, *CH-Fly-Algo*, and *K-*

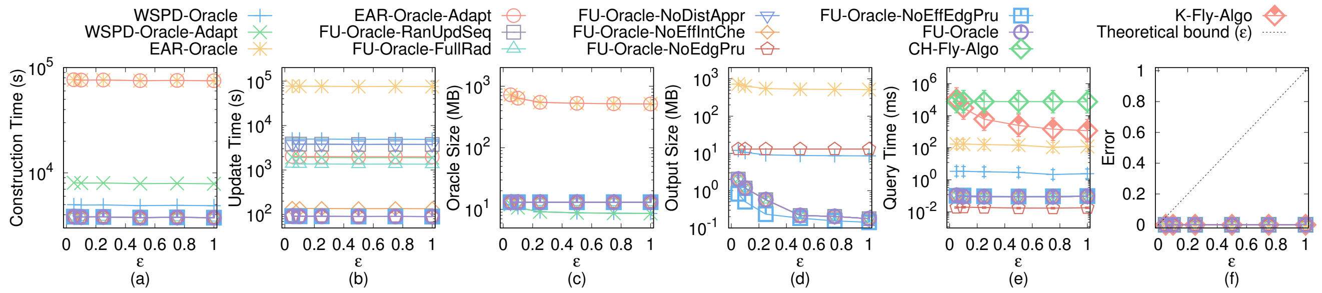

Fig. 16. Effect of ϵ on *SC* dataset (fewer POIs) for the P2P path query

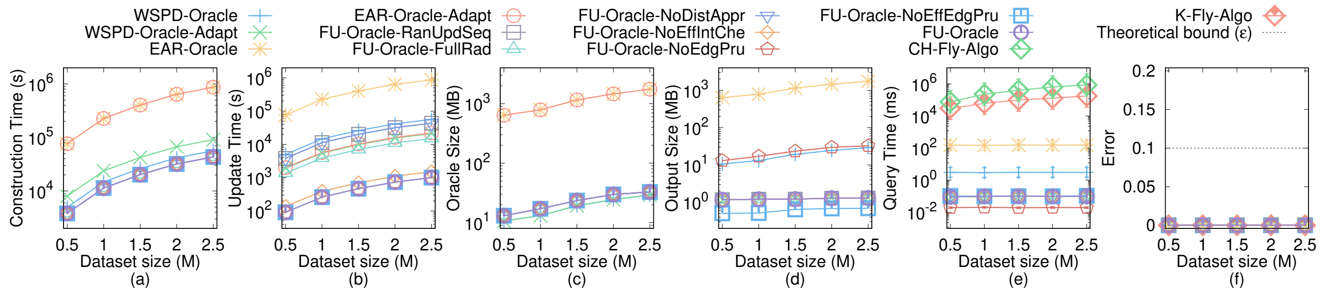

Fig. 17. Effect of *DS* on *SC* dataset (fewer POIs) for the P2P path query

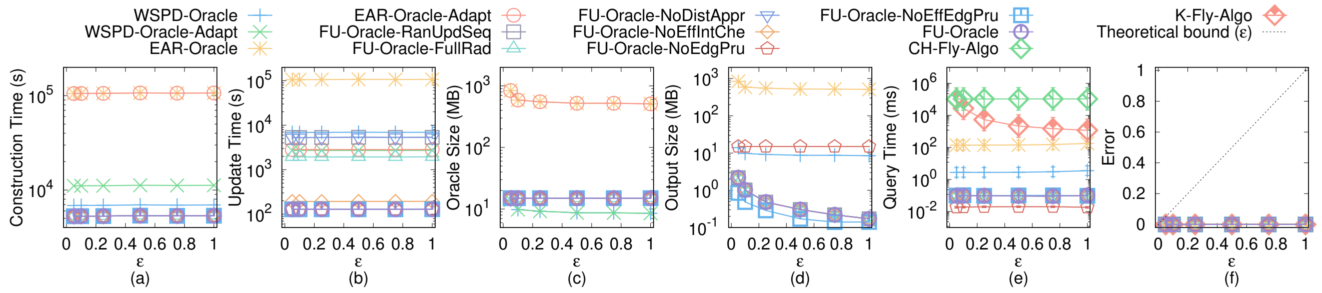

Fig. 18. Effect of ϵ on *GI* dataset (fewer POIs) for the P2P path query

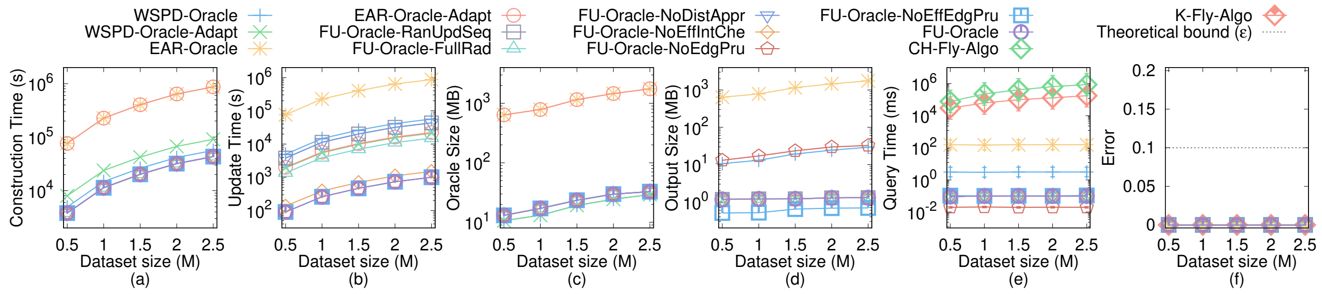

Fig. 19. Effect of *DS* on *GI* dataset (fewer POIs) for the P2P path query

Fly-Algo in terms of oracle construction time, oracle update time, output size, and shortest path query time. In addition, it is clear that the oracle update time for *FU-Oracle-**X*** where $\boldsymbol{X}$ = {*RanUpdSeq, FullRad, NoDistAppr, NoEffIntChe*} are larger than *FU-Oracle*. Even though the output size of *FU-Oracle* is slightly larger than *FU-Oracle-NoEffEdgPru*, the oracle update time of *FU-Oracle* is better than *FU-Oracle-NoEffEdgPru*. Furthermore, the output size for *FU-Oracle-NoEdgPru* is larger than *FU-Oracle*.

C. Experimental Results on the P2P Path Query in the Case $\boldsymbol{n > N}$ and the A2A Path Query

In Figure 48, we tested the P2P path query in the case $n > N$ and the A2A path query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and fixing N at 2k on a multi-resolution of *SC* dataset. It still shows that *FU-Oracle* superior performance of *WSPD-Oracle*, *WSPD-Oracle-Adapt*, *EAR-Oracle*, *EAR-Oracle-Adapt*, *CH-Fly-Algo*, and *K-Fly-Algo* in terms of oracle construction time, oracle update time, output size, and shortest path query time. For *EAR-Oracle* and *EAR-Oracle-Adapt*, although their oracle construction time is slightly smaller than *FU-Oracle*, their oracle update time is still large. In addition, it is clear that the oracle update time for *FU-Oracle-**X*** where $\boldsymbol{X}$ = {*RanUpdSeq, FullRad, NoDistAppr, NoEffIntChe*} are larger than *FU-Oracle*. Even though the output size of *FU-Oracle* is slightly larger than *FU-Oracle-NoEffEdgPru*, but the oracle update time of *FU-Oracle* is better than *FU-Oracle-*

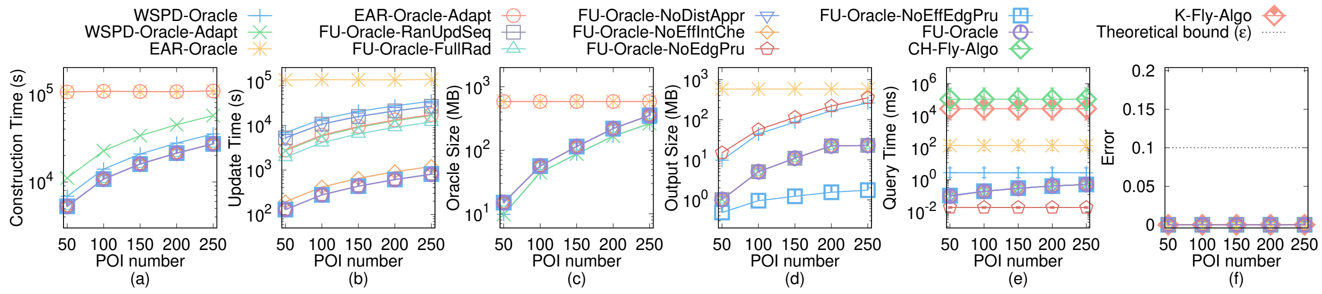

Fig. 20. Effect of n on *AU* dataset (fewer POIs)

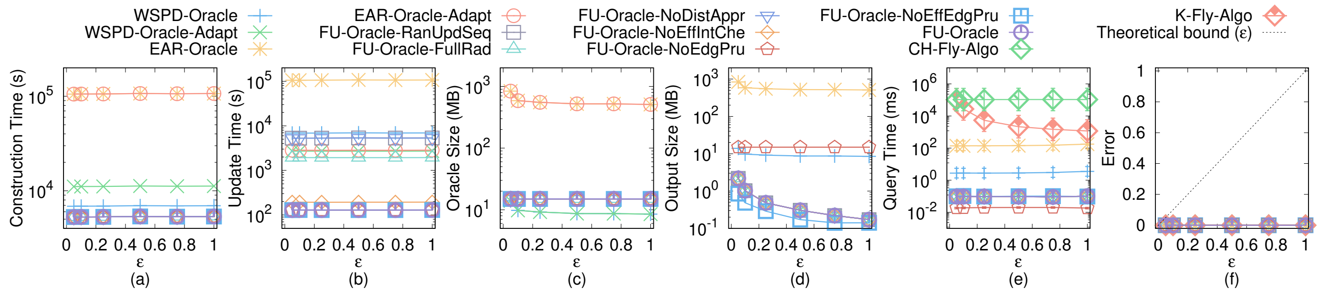

Fig. 21. Effect of ϵ on *AU* dataset (fewer POIs) for the P2P path query

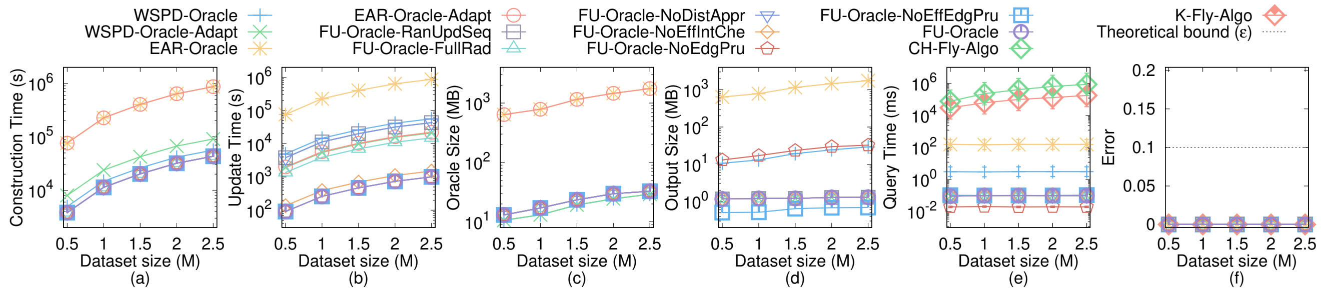

Fig. 22. Effect of *DS* on *AU* dataset (fewer POIs) for the P2P path query

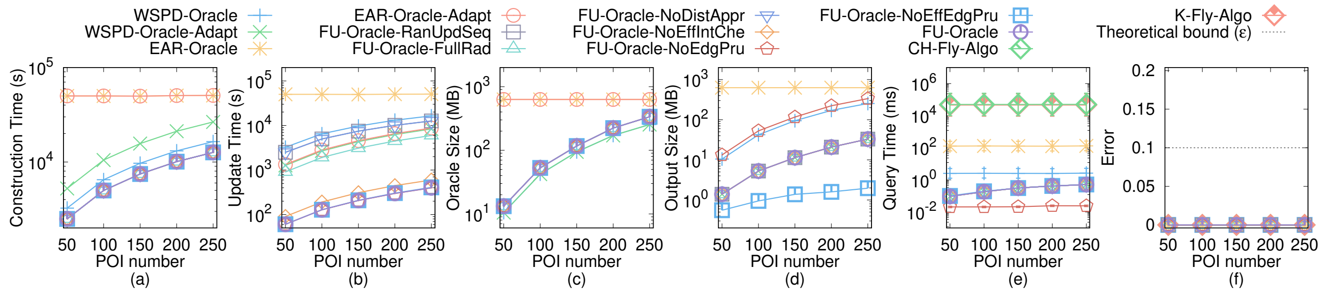

Fig. 23. Effect of n on *LH* dataset (fewer POIs) for the P2P path query

NoEffEdgPru. Furthermore, the output size for *FU-Oracle-NoEdgPru* is larger than *FU-Oracle*.

D. Generating datasets with different dataset sizes

The procedure for generating the datasets with different dataset sizes is as follows. We mainly follow the procedure for generating datasets with different dataset sizes in the work [41], [54], [55]. Let $T_t = (V_t, E_t, F_t)$ be our target terrain surface that we want to generate with ex_t edges along x-coordinate, ey_t edges along y-coordinate and dataset size of DS_t, where $DS_t = 2 \cdot ex_t \cdot ey_t$. Let $T_o = (V_o, E_o, F_o)$ be the original terrain surface that we currently have with ex_o edges along x-coordinate, ey_o edges along y-coordinate and dataset size of DS_o, where $DS_o = 2 \cdot ex_o \cdot ey_o$. We then generate $(ex_t + 1) \cdot (ey_t + 1)$ 2D points (x, y) based on a Normal distribution $N(\mu_N, \sigma_N^2)$, where $\mu_N = (\overline{x} = \frac{\sum_{v_o \in V_o} x_{v_o}}{(ex_o+1)\cdot(ey_o+1)}, \overline{y} = \frac{\sum_{v_o \in V_o} y_{v_o}}{(ex_o+1)\cdot(ey_o+1)})$ and $\sigma_N^2 = (\frac{\sum_{v_o \in V_o}(x_{v_o}-\overline{x})^2}{(ex_o+1)\cdot(ey_o+1)}, \frac{\sum_{v_o \in V_o}(y_{v_o}-\overline{y})^2}{(ex_o+1)\cdot(ey_o+1)})$. In the end, we project each generated point (x, y) to the surface of T_o and take the projected point (also add edges between neighbours of two points to form edges and faces) as the newly generate T_t.

Appendix G
Proof

Proof of Property 1. We prove by contradiction. Suppose that two disks $D(u, \frac{|\Pi(u,v|T_{before})|}{2})$ and $D(v, \frac{|\Pi(u,v|T_{before})|}{2})$ do not intersect with ΔF, but $\Pi(u, v|T_{after})$ is different from

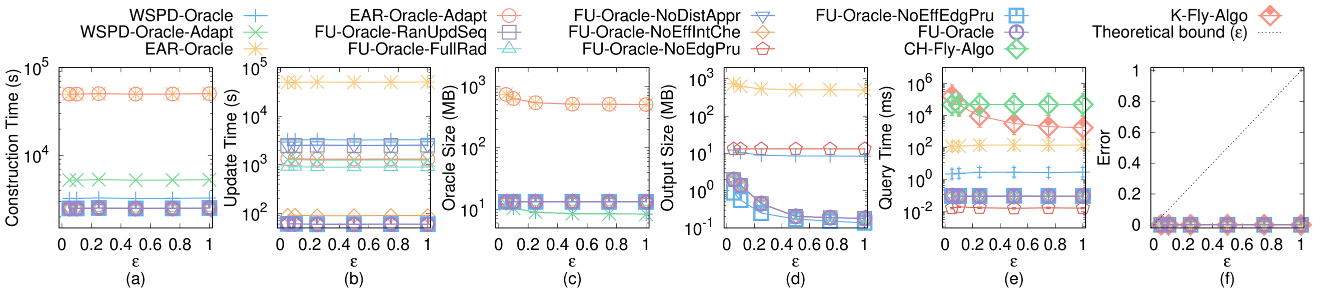
Fig. 24. Effect of ϵ on *LH* dataset (fewer POIs) for the P2P path query

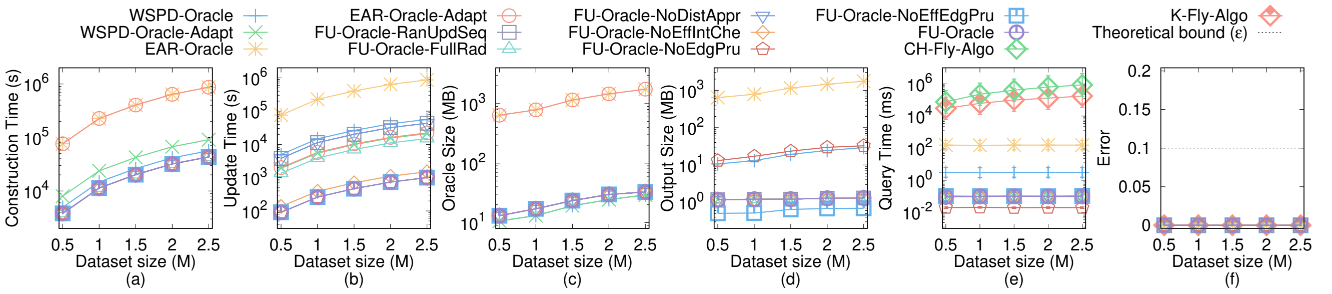
Fig. 25. Effect of *DS* on *LH* dataset (fewer POIs) for the P2P path query

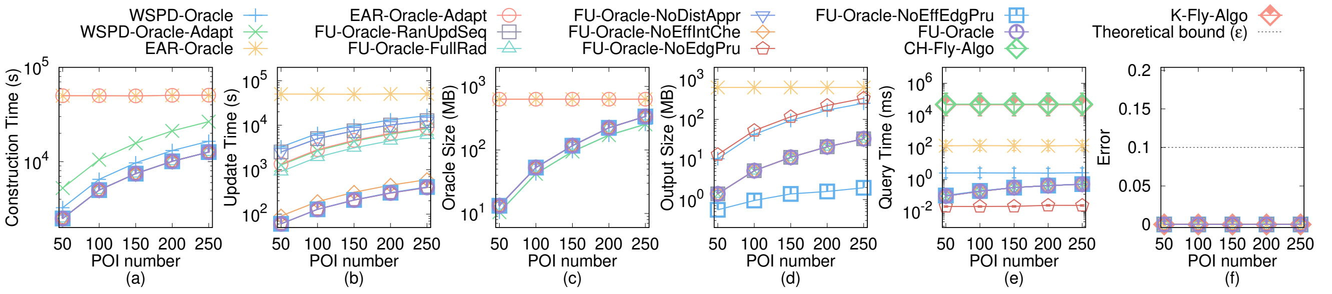
Fig. 26. Effect of n on *VS* dataset (fewer POIs) for the P2P path query

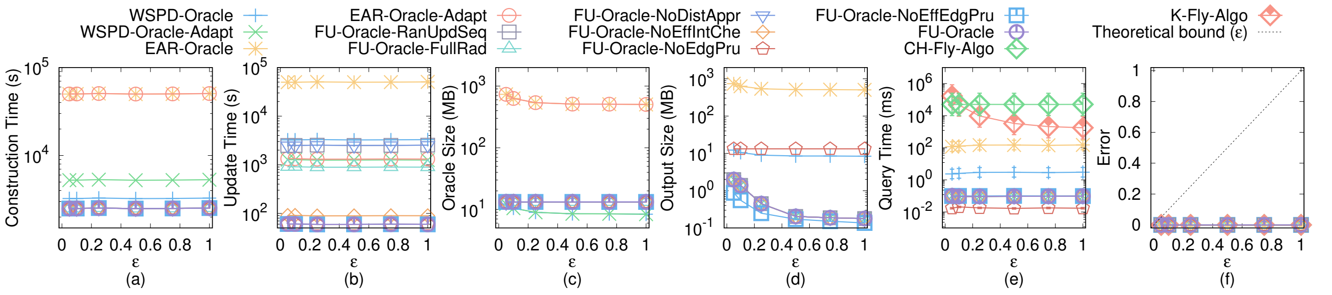
Fig. 27. Effect of ϵ on *VS* dataset (fewer POIs) for the P2P path query

$\Pi(u, v|T_{before})$, and we need to update $\Pi(u, v|T_{before})$ to $\Pi(u, v|T_{after})$ due to the smaller distance of $\Pi(u, v|T_{after})$, i.e., $|\Pi(u, v|T_{after})| < |\Pi(u, v|T_{before})|$. This case will only happen when $\Pi(u, v|T_{after})$ passes ΔF. We let u_1 (resp. v_1) be the point on $\Pi(u, v|T_{after})$ that the exact shortest distance $\Pi(u, u_1|T)$ (resp. $\Pi(v, v_1|T)$) on T is the same as $|\frac{\Pi(u,v|T_{before})}{2}|$. We let u_2 (resp. v_2) be the point on $\Pi(u, v|T_{after})$ that u_2 (resp. v_2) is a point in ΔF and the exact shortest distance $\Pi(u, u_2|T)$ (resp. $\Pi(v, v_2|T)$) on T is the minimum one. Clearly, u_2 (resp. v_2) is the intersection point between $\Pi(u, v|T_{after})$ and ΔF, such that the exact shortest distance $\Pi(u, u_2|T)$ (resp. $\Pi(v, v_2|T)$) on T is the minimum one. Note that a point is said to be in ΔF if this point is on a face in ΔF. We let m be the midpoint on $\Pi(u, v|T_{before})$, clearly we have $|\Pi(u, m|T)| = |\Pi(n, v|T)| = |\frac{\Pi(u,v|T_{before})}{2}|$. We also know that $|\Pi(u, u_1|T)| = |\Pi(u, m|T)| = |\Pi(v, v_1|T)| = |\Pi(v, m|T)| = |\frac{\Pi(u,v|T_{before})}{2}|$. Figure 2 shows an example of these notations. The purple line is $\Pi(u, v|T_{before})$ and the yellow line is $\Pi(u, v|T_{after})$. Since the minimum distance from both u and v to the updated faces ΔF is no smaller than $|\frac{\Pi(u,v|T_{before})}{2}|$, we know $|\Pi(u, o|T)| = |\Pi(u, u_1|T)| \leq |\Pi(u, u_2|T)|$ and $|\Pi(v, o|T)| = |\Pi(v, v_1|T)| \leq |\Pi(v, v_2|T)|$. Since $\Pi(u, v|T_{after})$ passes ΔF, $|\Pi(u_2, v_2|T_{after})| \geq 0$. Thus, we have $|\Pi(u, u_2|T)| + |\Pi(v, v_2|T)| + |\Pi(u_2, v_2|T_{after})| = |\Pi(u, v|T_{after})| \geq |\Pi(u, v|T_{before})| = |\Pi(u, o|T)| + |\Pi(v, o|T)|$, which is a contradiction of our assumption $|\Pi(u, v|T_{after})| <$

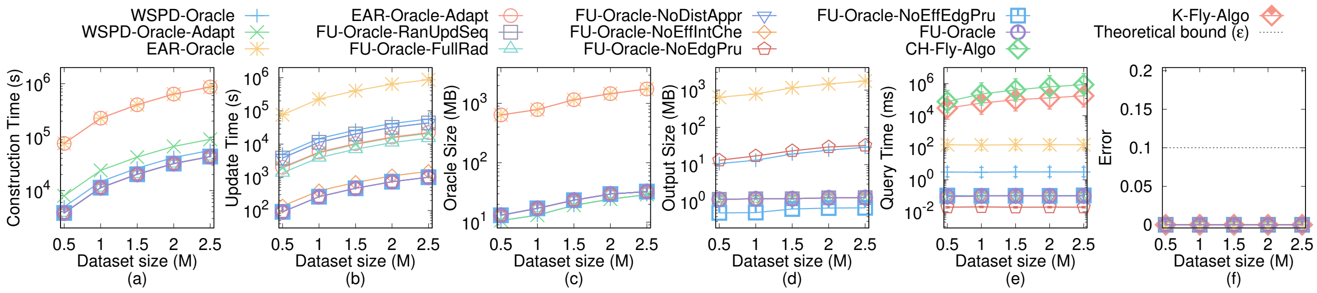

Fig. 28. Effect of *DS* on *VS* dataset (fewer POIs) for the P2P path query

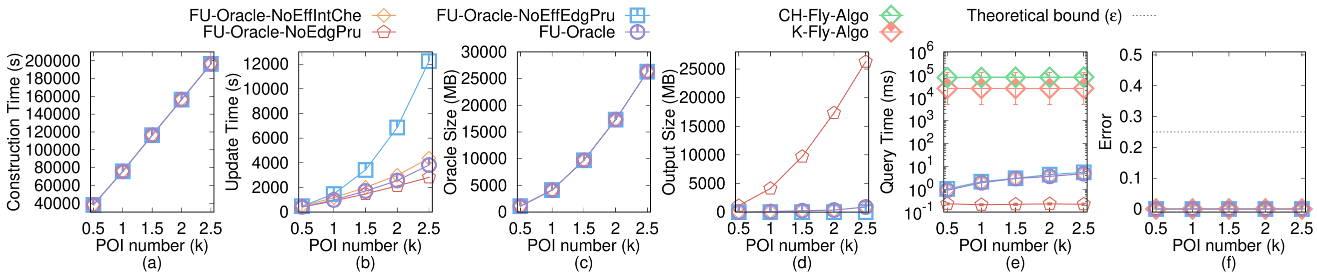

Fig. 29. Effect of n on *TJ* dataset (more POIs) for the P2P path query

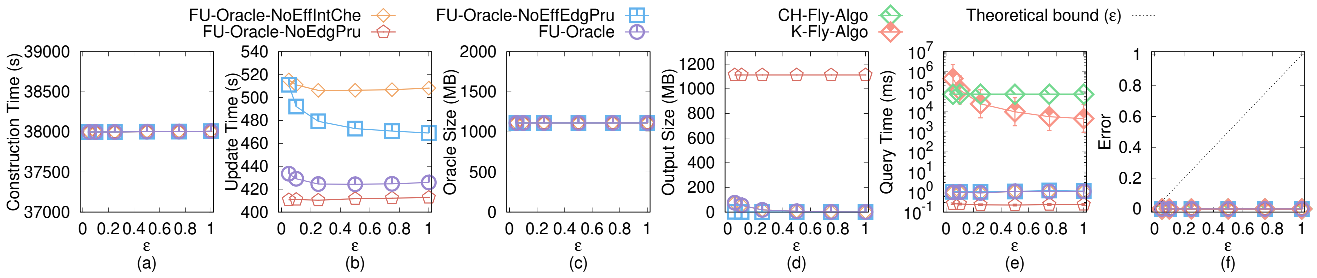

Fig. 30. Effect of ϵ on *TJ* dataset (more POIs) for the P2P path query

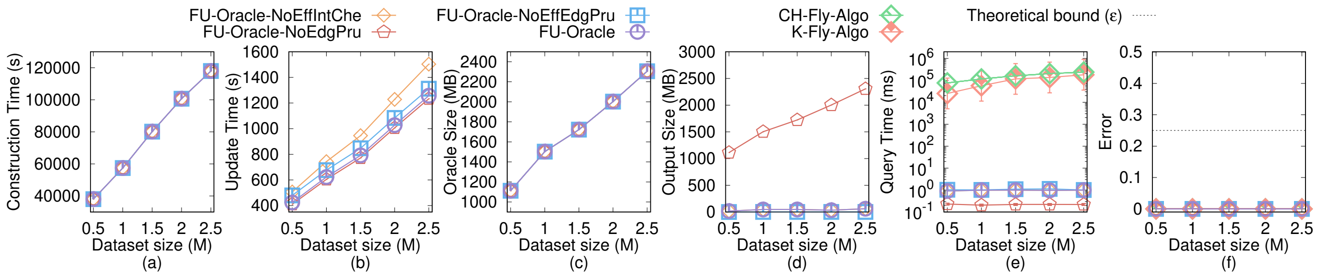

Fig. 31. Effect of *DS* on *TJ* dataset (more POIs) for the P2P path query

$|\Pi(u, v|T_{before})|$. Thus, we finish the proof. □

Proof of Lemma 1. According to [54], [55], since the exact shortest distance on a terrain surface is a metric, and therefore it satisfies the triangle inequality. Given an edge e which belongs to a face in ΔF with two endpoints u_1 and u_2, suppose that the exact shortest path from u to ΔF intersects with any point on e for the first time. There are two cases:

- If the intersection point is one of the two endpoints of e (e.g., u_1 without loss of generality), since u_1 is a vertex of a face in ΔF, so the minimum distance from u to ΔF in non-updated faces of T_{after} is the same as the exact shortest distance from u to u_1 on T_{before}. Since the exact shortest distance from u to u_1 on T_{before} is at least the minimum distance from u to any vertex in ΔV on T_{before}, we obtain that the minimum distance from u to ΔF in non-updated faces of T_{after} is at least the minimum distance from u to any vertex in ΔV on T_{before}.
- If the intersection point is on e, we denote this intersection point as u_3. Without loss of generality, suppose that the exact shortest distance from u to u_1 on T_{before} minus $|u_1u_3|$ is smaller than the exact shortest distance from u to u_2 on T_{before} minus $|u_2u_3|$, where $|u_1u_3|$ (resp. $|u_2u_3|$) is the length of the segment between u_1 and u_3 (resp. between u_2 and u_3) on edge e. According to triangle inequality, the minimum distance from u to ΔF in non-updated faces of T_{after} is at least the exact shortest distance from u to u_1 on T_{after} minus $|u_1u_3|$. Since we only care about the minimum distance, so the exact shortest distance from u to u_1 on T_{after} is the same as the exact shortest distance from u to

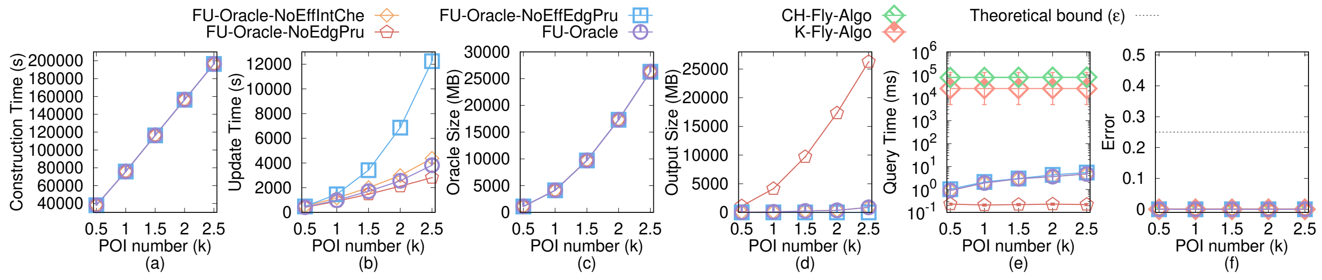

Fig. 32. Effect of n on *SC* dataset (more POIs) for the P2P path query

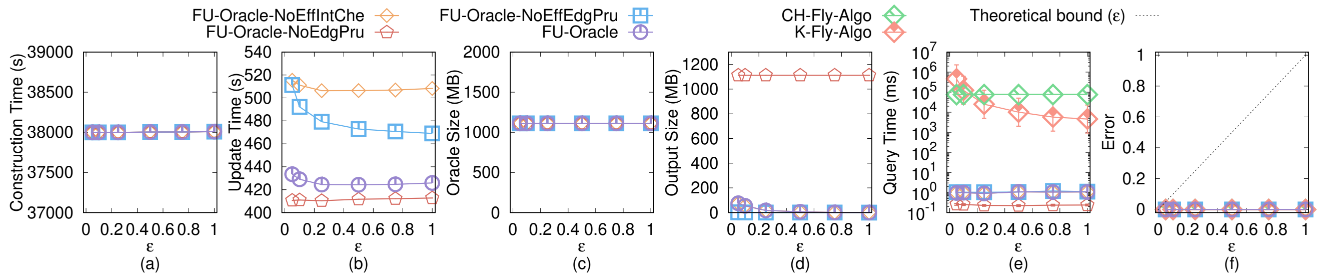

Fig. 33. Effect of ϵ on *SC* dataset (more POIs) for the P2P path query

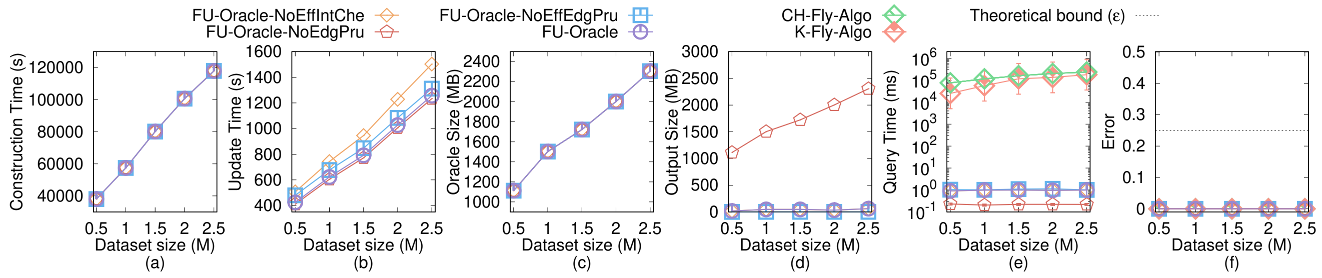

Fig. 34. Effect of *DS* on *SC* dataset (more POIs) for the P2P path query

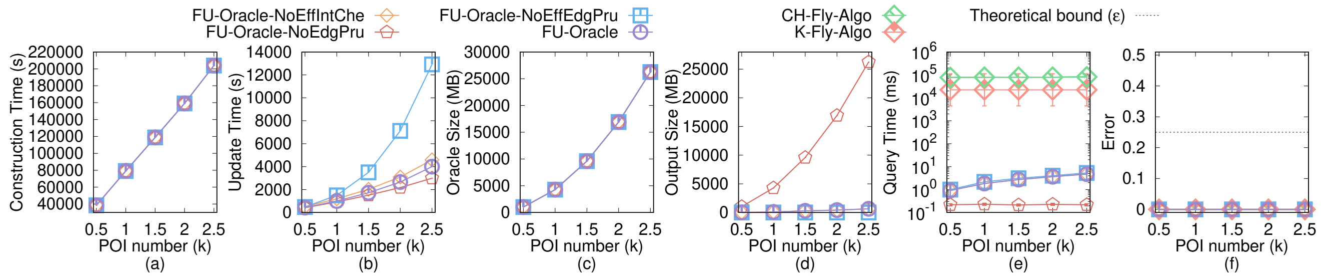

Fig. 35. Effect of n on *GI* dataset (more POIs) for the P2P path query

u_1 on T_{before}. Since the exact shortest distance from u to u_1 on T_{before} is at least the minimum distance from u to any vertex in ΔV on T_{before}, and $|u_1 u_3|$ is at most L_{max}, we obtain that the minimum distance from u to ΔF in non-updated faces of T_{after} is at least the minimum distance from u to any vertex in ΔV on T_{before} minus L_{max}.

□

Proof of Lemma 2. If the disk with the largest radius intersects with ΔF, we just need to update the paths and there is no need to check other disks. In Figure 5 (c), the sorted POIs are h, f, e, d, c, g. We create one disk $D(h, \frac{|\Pi(c,h|T_{before})|}{2})$, since it *intersects* with ΔF, we use algorithm *SSAD* to update all the shortest paths adjacent to h that have not been updated. We do not need to create ten disks, i.e., five disks $D(h, \frac{|\Pi(X,h|T_{before})|}{2})$ and five disks $D(X, \frac{|\Pi(X,h|T_{before})|}{2})$, where $X = \{c, d, e, f, g, h\}$. Since the disk $D(h, \frac{|\Pi(c,h|T_{before})|}{2})$ with the largest radius already intersects with ΔF, so 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, so there is no need to update the paths. In Figure 5 (d), the sorted POIs are f, e, d, c, g. We create one disk $D(f, \frac{|\Pi(c,f|T_{before})|}{2})$, since it *does not intersect* with ΔF, there is no need to update the shortest paths adjacent to f. We do not need to create eight disks, i.e., four disks $D(f, \frac{|\Pi(X,f|T_{before})|}{2})$ and four disks $D(X, \frac{|\Pi(X,f|T_{before})|}{2})$, where $X = \{c, d, e, f, g\}$. Since the disk $D(f, \frac{|\Pi(c,f|T_{before})|}{2})$ with the largest radius does not intersect with ΔF, so the disks $D(f, \frac{|\Pi(X,f|T_{before})|}{2})$ with smaller radius and the disks $D(X, \frac{|\Pi(X,f|T_{before})|}{2})$ with centers further away from ΔF

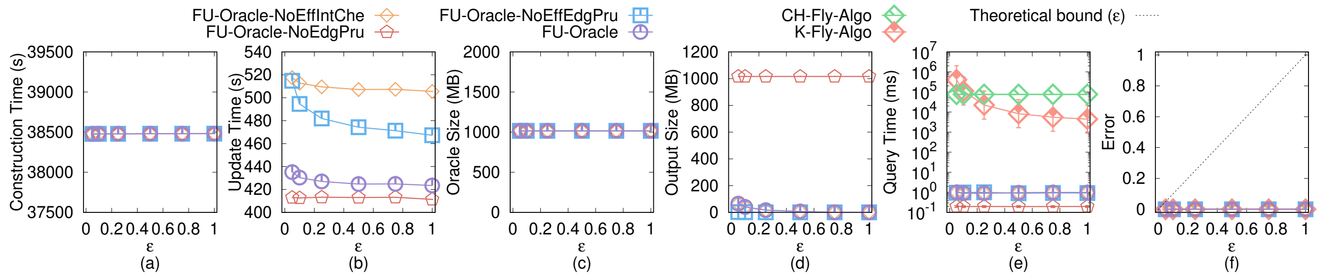

Fig. 36. Effect of ϵ on *GI* dataset (more POIs) for the P2P path query

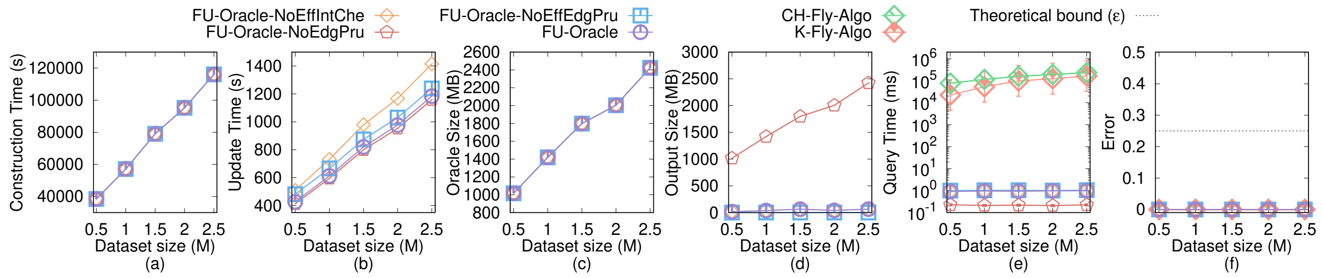

Fig. 37. Effect of *DS* on *GI* dataset (more POIs) for the P2P path query

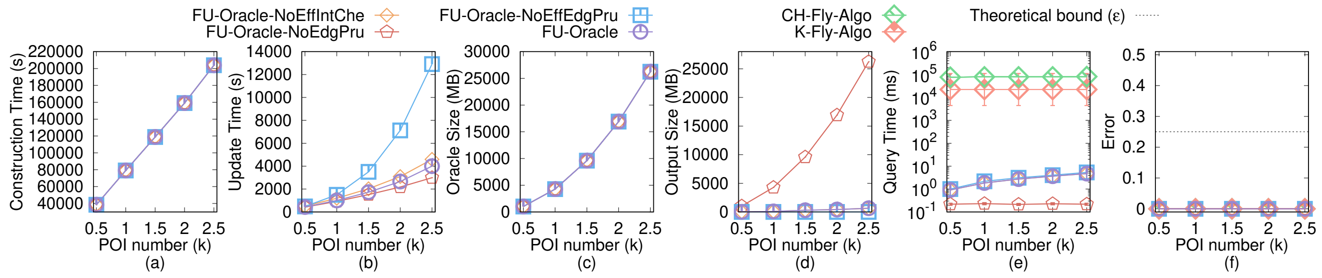

Fig. 38. Effect of n on *AU* dataset (more POIs) for the P2P path query

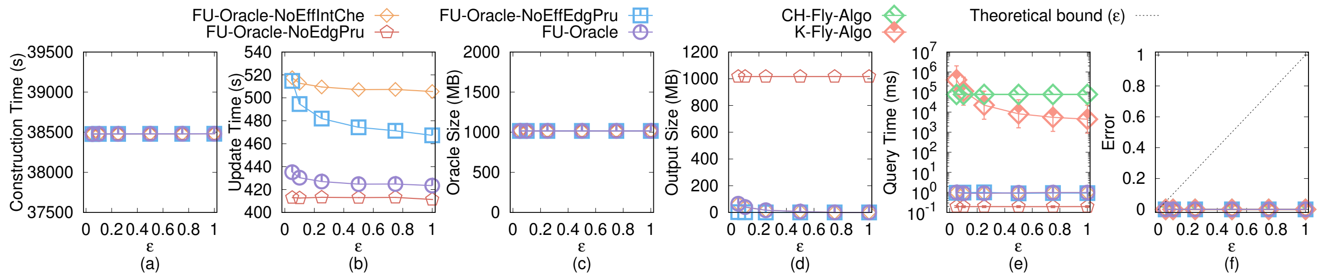

Fig. 39. Effect of ϵ on *AU* dataset (more POIs) for the P2P path query

compared with f cannot intersect with ΔF. Recall that given a POI u, we use $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{before})| - L_{max}$ as the lower bound of the minimum distance from u to any point in ΔF on T_{after}. If $D(f, \frac{|\Pi(c,f|T_{before})|}{2})$ does not intersect with ΔF, then $\min_{\forall v \in \Delta V} |\Pi(c, v|T_{before})| - L_{max} > \frac{|\Pi(c,f|T_{before})|}{2}$, then $\min_{\forall v \in \Delta V} |\Pi(X, v|T_{before})| - L_{max} > \frac{|\Pi(c,f|T_{before})|}{2}$ (since we sort X from near to far according to their minimum distance to any vertex in ΔV on T_{before}), and then $\min_{\forall v \in \Delta V} |\Pi(X, v|T_{before})| - L_{max} > \frac{|\Pi(X,f|T_{before})|}{2}$ (since $|\Pi(c, f|T_{before})| \geq |\Pi(X, f|T_{before})|$), i.e., the disks $D(X, \frac{|\Pi(X,f|T_{before})|}{2})$ cannot intersect with ΔF, where $X = \{c, d, e, f, g\}$. □

Lemma 3. *After the pairwise P2P exact shortest paths updating step in the update phase of FU-Oracle, G' stores the correct exact shortest path between all pairs of POIs in P on T_{after}.*

Proof of Lemma 3. After the pairwise P2P exact shortest paths updating step, there are two types of pairwise P2P exact shortest paths stored in G', i.e., (1) the updated exact shortest paths calculated on T_{after}, and (2) the non-updated exact shortest paths calculated on T_{before}. Due to Property 1, we know that the non-updated exact shortest paths calculated on T_{before} is exactly the same as the exact shortest path on T_{after}. Thus, after the pairwise P2P exact shortest paths updating step in the update phase of *FU-Oracle*, G' stores the correct exact shortest path between all pairs of POIs in P on T_{after}. □

Proof of Theorem 1. Firstly, we prove the *running time* of algorithm *HieGreSpan*.

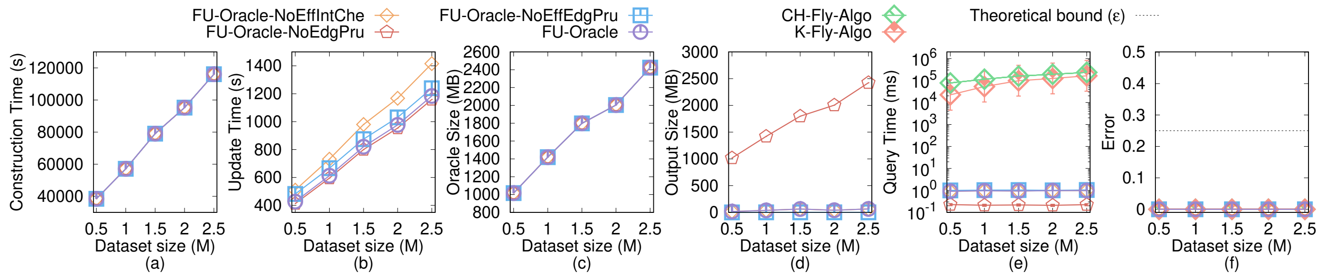

Fig. 40. Effect of *DS* on *AU* dataset (more POIs) for the P2P path query

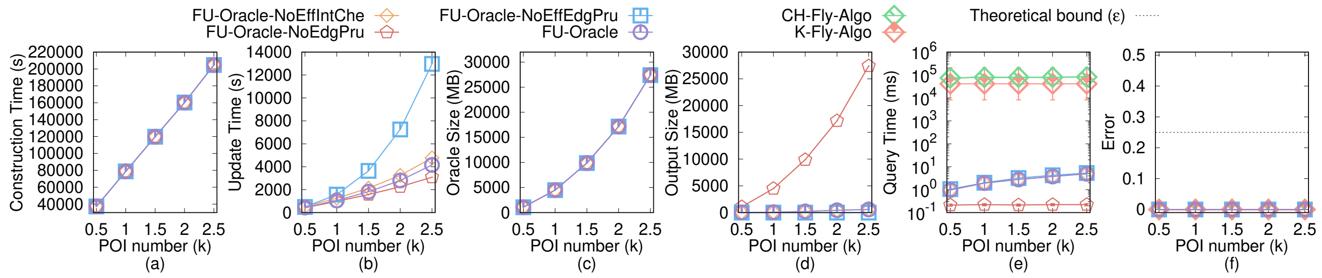

Fig. 41. Effect of n on *LH* dataset (more POIs) for the P2P path query

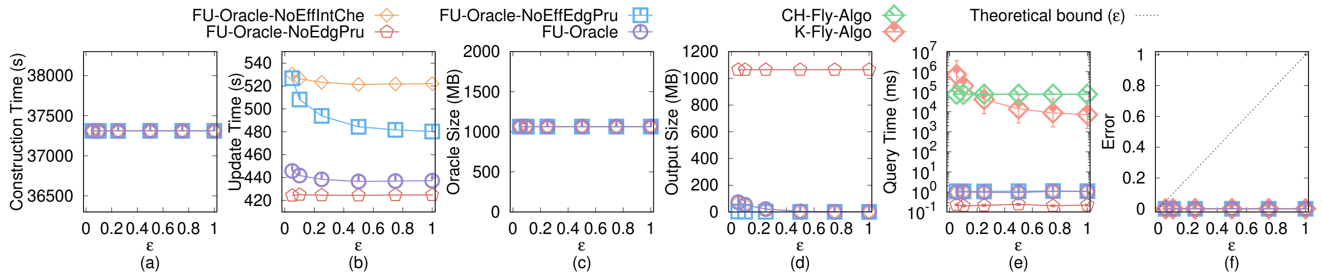

Fig. 42. Effect of ϵ on *LH* dataset (more POIs) for the P2P path query

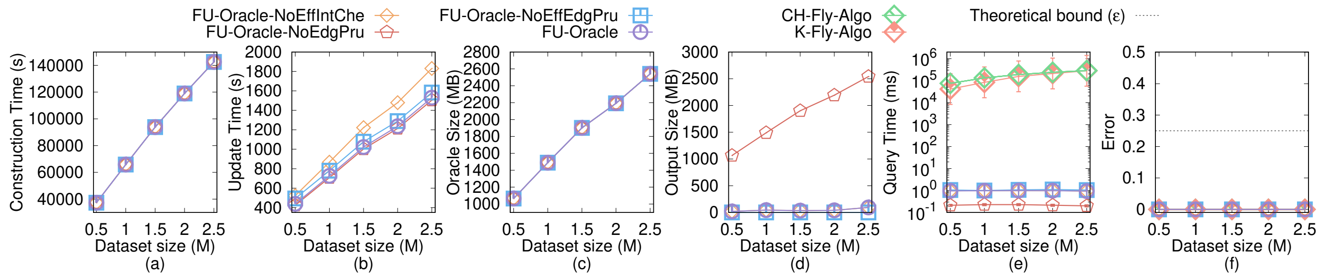

Fig. 43. Effect of *DS* on *LH* dataset (more POIs) for the P2P path query

- In the edge sorting, interval splitting, and G initialization step, it needs $O(n)$ time. Since we perform algorithm *SSAD* for each POI to generate G', so given a POI, the distances between this POI and other POIs have already been sorted. Since there are n vertices in G', so this step needs $O(n)$ time.
- In the G maintenance step, for each edge interval, it needs $O(n \log n + n) = O(n \log n)$ time (shown as follows). Since there are total $\log n$ intervals, it needs $O(n \log^2 n)$ time.
  - In the groups construction and intra-edges adding for H step, it needs $O(n \log n)$ time. This is because according to Lemma 6 in [25], we know that a vertex in H belongs to at most $O(1)$ groups (i.e., there are at most $O(1)$ group centers in H), so we just need to run $O(n \log n)$ Dijkstra's algorithm on G for $O(1)$ times in order to calculate intra-edges for H.
  - In the first type inter-edges adding for H step, it needs $O(n \log n)$ time. This is still because there are at most $O(1)$ group centers in H, so we just need to run $O(n \log n)$ Dijkstra's algorithm on G for $O(1)$ times in order to calculate inter-edges for H.
  - In the edges examining on H step, it needs $O(n)$ time. According to [25], there are $O(n)$ edges in each interval. Since there are at most $O(1)$ group centers in H, so answering the shortest path query using Dijkstra's algorithm on H needs $O(1)$ time. So, in order to examine $O(n)$ edges, this step needs $O(1)$ Dijkstra's algorithm on H for $O(n)$ times, and the total running time is $O(n)$.

In general, the running time for algorithm *HieGreSpan* is $O(n) + O(n \log^2 n) = O(n \log^2 n)$, and we finish the proof.

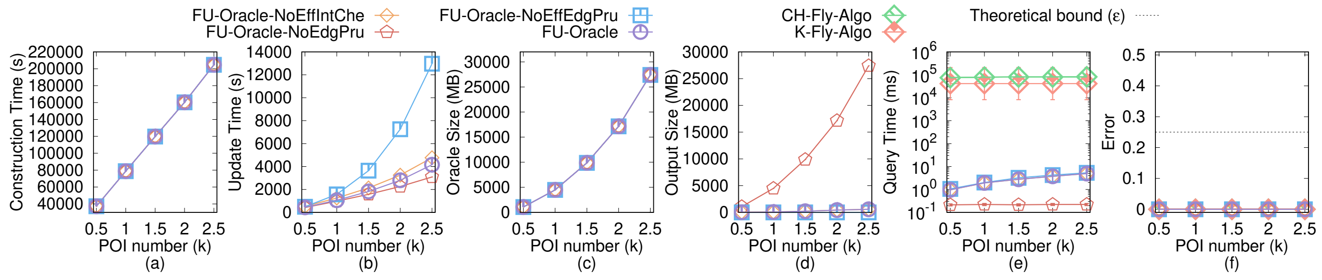

Fig. 44. Effect of n on *VS* dataset (more POIs) for the P2P path query

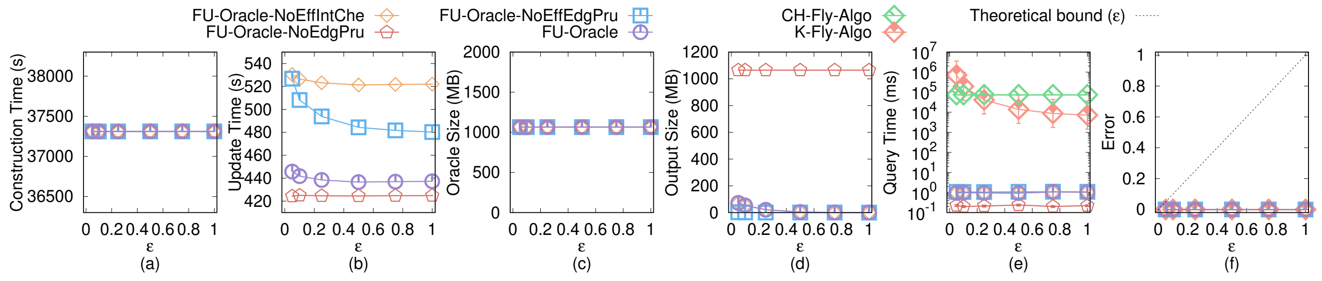

Fig. 45. Effect of ϵ on *VS* dataset (more POIs) for the P2P path query

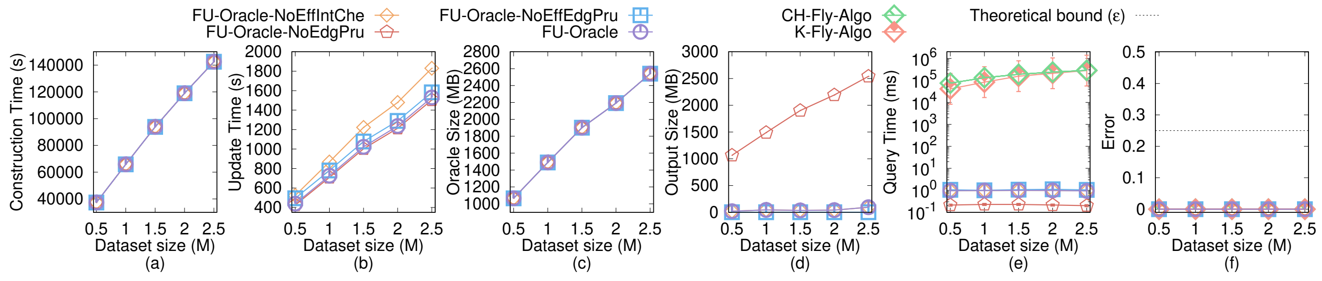

Fig. 46. Effect of *DS* on *VS* dataset (more POIs) for the P2P path query

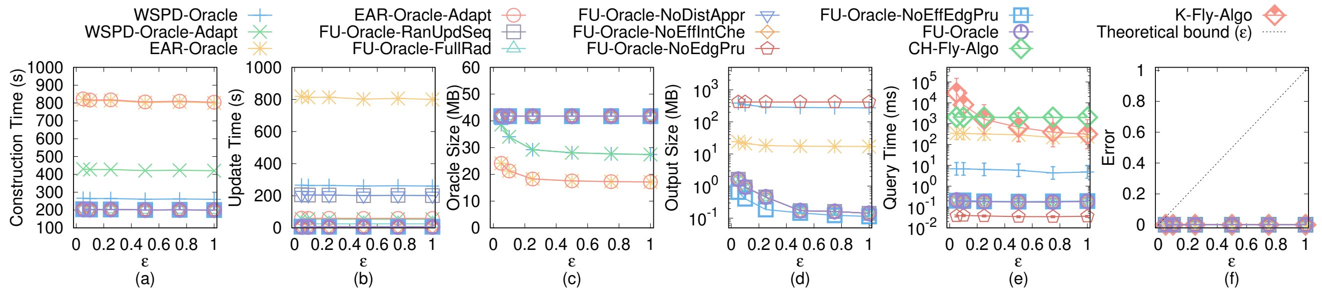

Fig. 47. V2V path query on *SC* dataset

Secondly, we prove the *error bound* of algorithm *HieGreSpan*. According to Lemma 8 in [25], we know that in algorithm *HieGreSpan*, during the processing of any group of edges $G.E^i$, the hierarchy graph H is always a valid approximation of G. Thus, in the edges examining on H step of algorithm *HieGreSpan*, for each edge $e'(u, v|T) \in G.E^i$ between two vertices u and v, when we need to check whether $|\Pi_H(w, x|T)| > (1 + \epsilon)|e'(u, v|T)|$, where $\Pi_H(w, x|T)$ is the shortest path of group centers calculated using Dijkstra's algorithm on H, w and x are two group centers, such that, u is in w's group, and v is in x's group, $\Pi_H(w, x|T)$ is a valid approximation of $\Pi_G(u, v|T)$. In other words, we are actually checking whether $|\Pi_G(u, v|T)| > (1 + \epsilon)|e'(u, v|T)|$ or not. Consider any edge $e'(u, v|T) \in G.E$ between two vertices u and v which is not added to G by algorithm *HieGreSpan*. Since $e'(u, v|T)$ is discarded, it implies that $|\Pi_G(u, v|T)| \leq (1 + \epsilon)|e'(u, v|T)|$. Since $|e'(u, v|T)| = |\Pi(u, v|T)|$, so on the output graph of algorithm *HieGreSpan*, i.e., G, we always have $|\Pi_G(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ for all pairs of vertices u and v in $G.V$. We finish the proof.

□

Proof of Theorem 2. Firstly, we prove the *oracle construction time* of *FU-Oracle*. When calculating the pairwise P2P exact shortest paths, it needs $O(nN \log^2 N)$ time, since there are n POIs, and each POI needs $O(N \log^2 N)$ time using algorithm *SSAD* for calculating the exact shortest path from this POI to other POI on T_{before}. So the oracle construction time of *FU-Oracle* is $O(nN \log^2 N)$.

Secondly, we prove the *oracle update time* of *FU-Oracle*.

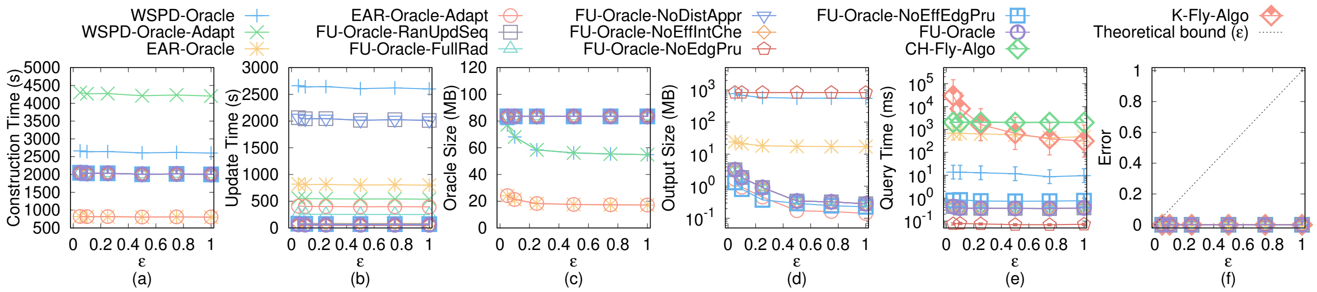

Fig. 48. P2P path query in the case $n > N$ and A2A path query on *SC* dataset

- In the terrain surface and POIs update detection step, it needs $O(N + n)$ time. Since we just need to iterate each face in T_{after} and T_{before}, and iterate each POI in P. Since the number of faces in T_{after} and T_{before} is $O(N)$, and the number of POIs in P is n, so it needs $O(N + n)$ time.
- In the pairwise P2P exact shortest paths updating step, it needs $O(N \log^2 N)$ time. Since we just need to update a constant number of POIs (which is shown by our experimental result) using algorithm *SSAD* for calculating the exact shortest path from this POI to other POI on T_{after}, and each algorithm *SSAD* needs $O(N \log^2 N)$ time, so it needs $O(N \log^2 N)$ time in total.
- In the sub-graph generating step, it needs $O(n \log^2 n)$ time. Since this step is using algorithm *HieGreSpan*, and algorithm *HieGreSpan* runs in $O(n \log^2 n)$ time as stated in Theorem 1.

In general, the oracle update time of *FU-Oracle* is $O(N \log^2 N + n \log^2 n)$.

Thirdly, we prove the *output size* of *FU-Oracle*. According to [25], we know that the output graph of algorithm *HieGreSpan*, i.e., G, has $O(n)$ edges. So, the output size of *FU-Oracle* is $O(n)$.

Fourthly, we prove the *shortest path query time* of *FU-Oracle*. Since we need to perform Dijkstra's algorithm on G, and in our experiment, G has a constant number of edges and n vertices, so using a Fibonacci heap in Dijkstra's algorithm, the shortest path query time of *FU-Oracle* is $O(\log n)$.

Fifthly, we prove the *error bound* of *FU-Oracle*. The error bound of *FU-Oracle* is due to the error bound of algorithm *HieGreSpan*. As stated in Theorem 1, on the output graph of algorithm *HieGreSpan*, i.e., G, we always have $|\Pi_G(u, v|T)| \leq (1 + \epsilon)|\Pi(u, v|T)|$ for all pairs of vertices u and v in $G.V$. Thus, we have the error bound of *FU-Oracle*, i.e., *FU-Oracle* satisfies $|\Pi_G(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ for all pairs of POIs u and v in P.

In general, we finish the proof of the oracle construction time, oracle update time, output size, shortest path query time, and error bound of *FU-Oracle*. □

Theorem 3. *The oracle construction time, oracle update time, output size, and shortest path query time of WSPD-Oracle [54], [55] are $O(\frac{nN \log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, $O(\frac{nN \log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$, respectively. WSPD-Oracle has $(1 - \epsilon)|\Pi(u, v|T)| \leq |\Pi_{\textit{WSPD-Oracle}}(u, v|T)| \leq (1 + \epsilon)|\Pi(u, v|T)|$ for all pairs of POIs u and v in P, where $\Pi_{\textit{WSPD-Oracle}}(u, v|T)$ is the shortest path of WSPD-Oracle between u and v.*

Proof. The proof of the oracle construction time, output size, and error bound of *WSPD-Oracle* is in [54], [55].

For the *oracle update time*, since *WSPD-Oracle* does not support the updated terrain surface setting, so the oracle update time is the same as the oracle construction time.

For the *shortest path query time*, suppose that we need to query the shortest path between two POIs a and b, a belongs to a disk with c as center, b belongs to a disk with d as center, and *WSPD-Oracle* stores the exact shortest path between c and d. In order to find the shortest path between a and b, we also need to find the shortest path between a and c, d and b, then connect the shortest path between a and c, c and d, d and b, to form the shortest path between a and b. It takes $O(h^2)$ time to query the shortest path between a and c, c and d, d and b, respectively, since the shortest path query time of *WSPD-Oracle* is $O(h^2)$ in [54], [55]. Thus, the shortest path query time of *WSPD-Oracle* should be $O(3h^2) = O(h^2)$. □

Theorem 4. *The oracle construction time, oracle update time, output size, and shortest path query time of WSPD-Oracle-Adapt [54], [55] are $O(\frac{nN \log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, $O(\mu_1 N \log^2 N + n \log^2 n)$, $O(n)$, and $O(\log n)$, respectively, where μ_1 is a data-dependent variable, and $\mu_1 \in [5, 20]$ in our experiment. WSPD-Oracle-Adapt satisfies $|\Pi_{\textit{WSPD-Oracle-Adapt}}(u, v|T)| \leq (1 + \epsilon)|\Pi(u, v|T)|$ for all pairs of POIs u and v in P, where $\Pi_{\textit{WSPD-Oracle-Adapt}}(u, v|T)$ is the shortest path of WSPD-Oracle-Adapt between u and v.*

Proof. The proof of the output size, shortest path query time, and error bound of *WSPD-Oracle-Adapt* is similar in *FU-Oracle*.

For the *oracle construction time*, *WSPD-Oracle-Adapt* first needs $O(\frac{nN \log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$ for oracle construction, which is the same as *WSPD-Oracle*. It then needs $O(nN \log^2 N)$ for computing the distance from each POI to each vertex in V on T_{before}. So the oracle construction time is $O(\frac{nN \log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$.

For the *oracle update time*, since *WSPD-Oracle-Adapt* uses the update phase of *FU-Oracle*, so it first needs $O(N+n)$ time for terrain surface and POIs update detection, then needs to update μ_1 number of POIs (which is shown by our experimental result) using algorithm *SSAD* for calculating the exact shortest path from this POI to other POI on T_{after}, where each algorithm

SSAD needs $O(N \log^2 N)$ time, and then needs $O(n \log^2 n)$ time for sub-graph generating. So the oracle update time of *WSPD-Oracle-Adapt* is $O(\mu_1 N \log^2 N + n \log^2 n)$. ☐

Theorem 5. *The oracle construction time, oracle update time, output size, and shortest path query time of EAR-Oracle [32] are* $O(\lambda\xi mN \log^2(mN) + \frac{nN \log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$, $O(\lambda\xi mN \log^2(mN) + \frac{nN \log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$, *and* $O(\lambda\xi \log(\lambda\xi))$, *respectively. EAR-Oracle has* $|\Pi_{\textit{EAR-Oracle}}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)| + 2\delta|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{\textit{EAR-Oracle}}(u, v|T)$ *is the shortest path of EAR-Oracle between* u *and* v, *and* δ *is an error parameter [32].*

Proof. The proof of the oracle construction time, output size, shortest path query time, and error bound of *EAR-Oracle* is in [32].

For the *oracle update time*, since *EAR-Oracle* does not support the updated terrain surface setting, so the oracle update time is the same as the oracle construction time. ☐

Theorem 6. *The oracle construction time, oracle update time, output size, and shortest path query time of EAR-Oracle-Adapt [32] are* $O(\lambda\xi mN \log^2(mN) + \frac{nN \log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$, $O(\mu_2 N \log^2 N + n \log^2 n)$, $O(n)$, *and* $O(\log n)$, *respectively, where* μ_2 *is a data-dependent variable, and* $\mu_1 \in [12, 45]$ *in our experiment. EAR-Oracle-Adapt satisfies* $|\Pi_{\textit{EAR-Oracle-Adapt}}(u, v|T)| \leq (1 + \epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{\textit{EAR-Oracle-Adapt}}(u, v|T)$ *is the shortest path of EAR-Oracle-Adapt between* u *and* v.

Proof. The proof of the oracle construction time of *EAR-Oracle-Adapt* is similar in *EAR-Oracle*. The proof of the output size, shortest path query time, and error bound of *EAR-Oracle-Adapt* is similar in *FU-Oracle*.

For the *oracle update time*, since *EAR-Oracle-Adapt* uses the update phase of *FU-Oracle*, so it first needs $O(N+n)$ time for terrain surface and POIs update detection, then needs to update μ_2 number of POIs (which is shown by our experimental result) using algorithm *SSAD* for calculating the exact shortest path from this POI to other POI on $T_{\textit{after}}$, where each algorithm *SSAD* needs $O(N \log^2 N)$ time, and then needs $O(n \log^2 n)$ time for sub-graph generating. So the oracle update time of *EAR-Oracle-Adapt* is $O(\mu_2 N \log^2 N + n \log^2 n)$. ☐

Theorem 7. *The oracle construction time, oracle update time, output size, and shortest path query time of FU-Oracle-RanUpdSeq are* $O(nN \log^2 N)$, $O(nN \log^2 N + n \log^2 n)$, $O(n)$, *and* $O(\log n)$, *respectively. FU-Oracle-RanUpdSeq satisfies* $|\Pi_{\textit{FU-Oracle-NoEffEdgPru}}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{\textit{FU-Oracle-RanUpdSeq}}(u, v|T)$ *is the shortest path of FU-Oracle-RanUpdSeq between* u *and* v.

Proof. The proof of the oracle construction time, output size, shortest path query time, and error bound of *FU-Oracle-RanUpdSeq* is similar in *FU-Oracle*.

For the *oracle update time*, the only difference between *FU-Oracle* and *FU-Oracle-RanUpdSeq* is that the latter one uses the random path update sequence before utilizing the non-updated terrain shortest path intact property, so it cannot fully utilize this property, and in the pairwise P2P exact shortest paths updating step, it needs to use algorithm *SSAD* for all POIs for n times. The other oracle update time is the same as the *FU-Oracle*. So the oracle update time of *FU-Oracle-RanUpdSeq* is $O(nN \log^2 N + n \log^2 n)$. ☐

Theorem 8. *The oracle construction time, oracle update time, output size, and shortest path query time of FU-Oracle-FullRad are* $O(nN \log^2 N)$, $O(\mu_3 N \log^2 N + n \log^2 n)$, $O(n)$, *and* $O(\log n)$, *respectively, where* μ_3 *is a data-dependent variable, and* $\mu_3 \in [5, 10]$ *in our experiment. FU-Oracle-FullRad satisfies* $|\Pi_{\textit{FU-Oracle-NoEffEdgPru}}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{\textit{FU-Oracle-FullRad}}(u, v|T)$ *is the shortest path of FU-Oracle-FullRad between* u *and* v.

Proof. The proof of the oracle construction time, output size, shortest path query time, and error bound of *FU-Oracle-FullRad* is similar in *FU-Oracle*.

For the *oracle update time*, the only difference between *FU-Oracle* and *FU-Oracle-FullRad* is that the latter one uses the full shortest distance of a shortest path as the disk radius. In the pairwise P2P exact shortest paths updating step, it needs to use algorithm *SSAD* for μ_3 number of POIs (which is shown by our experimental result). The other oracle update time is the same as the *FU-Oracle*. So the oracle update time of *FU-Oracle-FullRad* is $O(mu_2 N \log^2 N + n \log^2 n)$. ☐

Theorem 9. *The oracle construction time, oracle update time, output size, and shortest path query time of FU-Oracle-NoDistAppr are* $O(nN \log^2 N)$, $O(nN \log^2 N + n \log^2 n)$, $O(n)$, *and* $O(\log n)$, *respectively. FU-Oracle-NoDistAppr satisfies* $|\Pi_{\textit{FU-Oracle-NoEffEdgPru}}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{\textit{FU-Oracle-NoDistAppr}}(u, v|T)$ *is the shortest path of FU-Oracle-NoDistAppr between* u *and* v.

Proof. The proof of the oracle construction time, output size, shortest path query time, and error bound of *FU-Oracle-NoDistAppr* is similar in *FU-Oracle*.

For the *oracle update time*, the only difference between *FU-Oracle* and *FU-Oracle-NoDistAppr* is that the latter one does not store the POI-to-vertex distance information and needs to calculate the shortest path on $T_{\textit{after}}$ again for determining whether the disk intersects with the updated faces on $T_{\textit{after}}$. It needs to perform such shortest path queries for each POI, so we can regard it re-calculate the pairwise P2P exact shortest paths $T_{\textit{after}}$, that is, it needs to use algorithm *SSAD* for all POIs for n times. The other oracle update time is the same as the *FU-Oracle*. So the oracle update time of *FU-Oracle-NoDistAppr* is $O(nN \log^2 N + n \log^2 n)$. ☐

Theorem 10. *The oracle construction time, oracle update time, output size, and shortest path query time of FU-Oracle-NoEffIntChe are* $O(nN \log^2 N)$, $O(nN \log^2 N + n \log^2 n)$,

$O(n)$, and $O(\log n)$, respectively. FU-Oracle-NoEffIntChe satisfies $|\Pi_{\textit{FU-Oracle-NoEffEdgPru}}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ for all pairs of POIs u and v in P, where $\Pi_{\textit{FU-Oracle-NoEffIntChe}}(u, v|T)$ is the shortest path of FU-Oracle-NoEffIntChe between u and v.

Proof. The proof of the oracle construction time, output size, shortest path query time, and error bound of *FU-Oracle-NoEffIntChe* is similar in *FU-Oracle*.

For the *oracle update time*, the only difference between *FU-Oracle* and *FU-Oracle-NoEffIntChe* is that the latter one creates two disks for each path when checking whether we need to re-calculate the shortest path between a pair of POIs. In the pairwise P2P exact shortest paths updating step, since there are total $O(n^2)$ pairwise P2P exact shortest paths, so it needs to create $O(n^2)$ disks. The other oracle update time is the same as the *FU-Oracle*. So the oracle update time of *FU-Oracle-NoEffIntChe* is $O(nN \log^2 N + n \log^2 n)$. □

Theorem 11. *The oracle construction time, oracle update time, output size, and shortest path query time of FU-Oracle-NoEdgPru are $O(nN \log^2 N + n^2)$, $O(N \log^2 N + n)$, $O(n^2)$, and $O(1)$, respectively. FU-Oracle-NoEdgPru satisfies $|\Pi_{\textit{FU-Oracle-NoEdgPru}}(u, v|T)| = |\Pi(u, v|T)|$ for all pairs of POIs u and v in P, where $\Pi_{\textit{FU-Oracle-NoEdgPru}}(u, v|T)$ is the shortest path of FU-Oracle-NoEdgPru between u and v.*

Proof. Firstly, we prove the *oracle construction time* of *FU-Oracle-NoEdgPru*. The oracle construction of *FU-Oracle-NoEdgPru* is similar in *FU-Oracle*. But, it also needs to store the pairwise P2P exact shortest paths on $T_{\textit{before}}$ into a hash table in $O(n^2)$ time. So the oracle construction time of *FU-Oracle-NoEdgPru* is $O(nN \log^2 N + n^2)$.

Secondly, we prove the *oracle update time* of *FU-Oracle-NoEdgPru*. For the oracle update time, the only difference between *FU-Oracle* and *FU-Oracle-NoEdgPru* is that the latter one does not use any sub-graph generating algorithm to prune out the edges. So there is no sub-graph generating step. But after The pairwise P2P exact shortest paths updating step, it needs to update a constant number of POIs using algorithm *SSAD* for calculating the exact shortest path from this POI to other n POI on $T_{\textit{after}}$, and update them in the hash table takes $O(n)$ time. So the oracle update time of *FU-Oracle-NoEdgPru* is $O(N \log^2 N + n)$.

Thirdly, we prove the *output size* of *FU-Oracle-NoEdgPru*. Since there are $O(n^2)$ edges in *FU-Oracle-NoEdgPru*, so the output size of *FU-Oracle-NoEdgPru* is $O(n)$.

Fourthly, we prove the *shortest path query time* of *FU-Oracle-NoEdgPru*. Since we have a hash table to store the pairwise P2P exact shortest paths of *FU-Oracle-NoEdgPru*, and the hash table technique needs $O(1)$ time to return the value with the given key, the shortest path query time of *FU-Oracle-NoEdgPru* is $O(1)$.

Fifthly, we prove the *error bound* of *FU-Oracle-NoEdgPru*. Since *FU-Oracle-NoEdgPru* stores the pairwise P2P exact shortest paths, so there is no error in *FU-Oracle-NoEdgPru*, i.e., *FU-Oracle-NoEdgPru* satisfies $|\Pi_{\textit{FU-Oracle-NoEdgPru}}(u, v|T)| = |\Pi(u, v|T)|$ for all pairs of POIs u and v in P.

In general, we finish the proof of the oracle construction time, oracle update time, output size, shortest path query time, and error bound of *FU-Oracle-NoEdgPru*. □

Theorem 12. *The oracle construction time, oracle update time, output size, and shortest path query time of FU-Oracle-NoEffEdgPru are $O(nN \log^2 N)$, $O(N \log^2 N + n^3 \log n)$, $O(n)$, and $O(\log n)$, respectively. FU-Oracle-NoEffEdgPru satisfies $|\Pi_{\textit{FU-Oracle-NoEffEdgPru}}(u, v|T)| \leq (1 + \epsilon)|\Pi(u, v|T)|$ for all pairs of POIs u and v in P, where $\Pi_{\textit{FU-Oracle-NoEffEdgPru}}(u, v|T)$ is the shortest path of FU-Oracle-NoEffEdgPru between u and v.*

Proof. Firstly, we prove the *oracle construction time* of *FU-Oracle-NoEffEdgPru*. The oracle construction of *FU-Oracle-NoEffEdgPru* is similar in *FU-Oracle*. So the oracle construction time of *FU-Oracle-NoEffEdgPru* is $O(nN \log^2 N)$.

Secondly, we prove the *oracle update time* of *FU-Oracle-NoEffEdgPru*. For the oracle update time, the only difference between *FU-Oracle* and *FU-Oracle-NoEffEdgPru* is that the latter one uses algorithm *GreSpan* for the sub-graph generating step. In the sub-graph generating step, since there are n vertices in *FU-Oracle-NoEffEdgPru*, so answering the shortest path query using Dijkstra's algorithm on *FU-Oracle-NoEffEdgPru* needs $O(n \log n)$ time. Since we need to examine total $O(n^2)$ edges in G', so the total running time of algorithm *GreSpan* is $O(n^3 \log n)$. So the oracle update time of *FU-Oracle-NoEffEdgPru* is $O(N \log^2 N + n^3 \log n)$.

Thirdly, we prove the *output size* of *FU-Oracle-NoEffEdgPru*. According to [25], we know that the output graph of algorithm *GreSpan*, i.e., *FU-Oracle-NoEffEdgPru*, has $O(n)$ edges. So, the output size of *FU-Oracle-NoEffEdgPru* is $O(n)$.

Fourthly, we prove the *shortest path query time* of *FU-Oracle-NoEffEdgPru*. Since we need to perform Dijkstra's algorithm on G, and in our experiment, G has a constant number of edges and n vertices, so using a Fibonacci heap in Dijkstra's algorithm, the shortest path query time of *FU-Oracle-NoEffEdgPru* is $O(\log n)$.

Fifthly, we prove the *error bound* of *FU-Oracle-NoEffEdgPru*. The error bound of *FU-Oracle-NoEffEdgPru* is due to the error bound of algorithm *GreSpan*. Let $V_{\textit{FU-Oracle-NoEffEdgPru}}$ and $E_{\textit{FU-Oracle-NoEffEdgPru}}$ be the set of vertices and edges of *FU-Oracle-NoEffEdgPru*. In algorithm *GreSpan*, consider any edge $e_{\textit{FU-Oracle-NoEffEdgPru}_t}(u, v|T) \in G.E$ between two vertices u and v which is not added to *FU-Oracle-NoEffEdgPru*. Since $e_{\textit{FU-Oracle-NoEffEdgPru}_t}(u, v|T)$ is discarded, it implies that $|\Pi_{\textit{FU-Oracle-NoEffEdgPru}}(u, v|T)| \leq (1 + \epsilon)|\Pi(u, v|T)|$. Since $|\Pi(u, v|T)| = |\Pi(u, v|T)|$, so on the output graph of algorithm *GreSpan*, i.e., *FU-Oracle-NoEffEdgPru*, we always have $|\Pi_{\textit{FU-Oracle-NoEffEdgPru}}(u, v|T)| \leq (1 + \epsilon)|\Pi(u, v|T)|$ for all pairs of vertices u and v in $V_{\textit{FU-Oracle-NoEffEdgPru}}$. Thus, we have the error bound of *FU-Oracle-NoEffEdgPru*, i.e., *FU-Oracle-NoEffEdgPru* satisfies

$|\Pi_{\textit{FU-Oracle-NoEffEdgPru}}(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ for all pairs of POIs u and v in P.

In general, we finish the proof of the oracle construction time, oracle update time, output size, shortest path query time, and error bound of *FU-Oracle-NoEffEdgPru*. □

Theorem 13. *The shortest path query time of CH-Fly-Algo [23] is $O(N^2)$. K-Fly-Algo returns the exact shortest path for all pairs of POIs u and v in P.*

Proof. The proof can be found in the work [23]. □

Theorem 14. *The shortest path query time of K-Fly-Algo [35] is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$. K-Fly-Algo satisfies $|\Pi_{\textit{K-Fly-Algo}}(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ for all pairs of POIs u and v in P, where $\Pi_{\textit{K-Fly-Algo}}(u,v|T)$ is the shortest path of K-Fly-Algo between u and v.*

Proof. The proof of the *shortest path query time* and *error bound* of *K-Fly-Algo* is in [35]. Note that in Section 4.2 of [35], the shortest path query time of *K-Fly-Algo* is $O((N+N')(\log(N+N')+(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}})^2))$, where $N' = O(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}}N)$ and K is a parameter which is a positive number at least 1. By Theorem 1 of [35], we obtain that its error bound ϵ is equal to $\frac{1}{K-1}$. Thus, we can derive that the shortest path query time of *K-Fly-Algo* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}})+\frac{l_{max}^2}{(\epsilon l_{min}\sqrt{1-\cos\theta})^2})$. Since for N, the first term is larger than the second term, so we obtain the shortest path query time of *K-Fly-Algo* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$. □