Fast Update Path Oracle on Updated Terrain Surface

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 114 times and 4,100 times, and of improvements in output size and shortest path query time of up to 12 times and 3 times.

PVLDB Reference Format:
Yinzhao Yan, Raymond Chi-Wing Wong, and Christian S. Jensen. Fast Update Path Oracle on Updated Terrain Surface. PVLDB, 17(1): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yanyinzhao/DynamicStructureTerrainCode.

1 INTRODUCTION

The need for finding the shortest paths on a terrain surface is increasingly important in real-world settings [62]. Thus, well-known companies and applications, including Metaverse [6], Google Earth [5], and Cyberpunk 2077 (a renowned 3D computer game) [4], all rely on the ability to find the shortest paths on a terrain surface (e.g., Earth or in virtual reality) to assist users to reach destinations more quickly. In academia, shortest path querying on a terrain surface also attracts considerable attention from researchers [26, 36, 37, 41, 59, 60, 63, 64]. 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.

Proceedings of the VLDB Endowment, Vol. 17, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1.1 Motivation

1.1.1 ***Updated terrain surface and oracle***. Computing the shortest path on an *updated* terrain surface *quickly* is critical.

(1) **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 [11], and more than 69,000 died in the 7.9 magnitude earthquake on May 12, 2008 in Sichuan, China [1]. Annually, there are 15 earthquakes with magnitude 7 and one with 8 or higher on average [56]. The rescue team can save 3 lives every 15 minutes [40], so early arrival at the sites of the quake was of essence. If we can pre-compute the shortest paths by means of indexing (called an *oracle*) on terrain surfaces prone to earthquakes (e.g., around the edges of the Pacific Ocean [55]), and efficiently update these after an earthquake, then we can use the oracle to efficiently return the shortest paths. In the earthquake of Sichuan, after the terrain surface is updated, our oracle can be updated in just 400s $\approx$ 7 min (and then returns shortest paths in 0.1ms) on a terrain surface with 0.5M faces, which shows the usefulness of our oracle in real-life application. In contrast, the best-known oracle [59, 60] needs 3,075,000s $\approx$ 35.5 days to update the oracle, which implies that it is not applicable in the realistic setting.

(2) **Avalanche**: An earthquake may also cause an avalanche or glacier collapse. The 4.1 magnitude earthquake on October 24, 2016 in Valais, Switzerland [7] 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, 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 also need to efficiently calculate the new shortest paths for recusing.

(3) **Glacier collapse**: Due to global warming, a glacier collapse happened in July 2022 in Marmolada glacier, Italy, killed 11 people [57]. We also aim to efficiently find the new shortest recuse paths on the deeply cracked glacier.

(3) **Marsquake**: As observed by NASA's InSight lander on March 7, 2021 [19], Mars also experiences marsquake. For NASA's Mars exploration project [51] (with cost USD 2.5 billion [43]), China National Space Administration's Mars mission project [12] (with an annual budget of USD 8.9 billion [48]), and the SpaceX Mars mission project [15] (with cost USD 67 million per launch [14]), it is essential to find the shortest escape paths quickly for Mars rovers in regions affected by marsquakes to avoid damages (a Mars rover costs USD 2.5 billion [16] and cannot be repaired remotely [33]). The round trip signal delay between Earth and Mars is 40 minutes

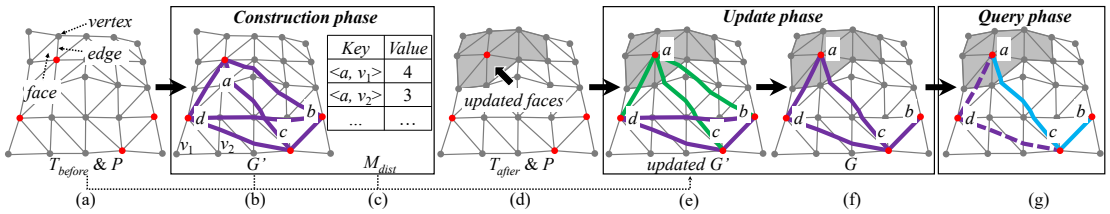

Figure 1: Framework overview

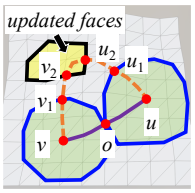

Figure 2: An unaffected path

[50], 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. It is essential that Mars rovers can calculate the shortest paths quickly and autonomously after a marsquake.

1.1.2 ***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. In the earthquake and avalanche scenario, POIs can be villages waiting for rescuing [46], hospitals, and expressway exits. In the Marsquake scenario, POIs can be Mars rover's working stations. In other applications, the POIs can be reference points when calculating similarities between two 3D objects [39, 53], and can be feeding or breeding destinations when studying migration patterns of animals [28, 42].

1.2 Challenges

1.2.1 ***Inefficiency for on-the-fly algorithm.*** Consider a terrain surface T with N vertices. All existing *exact on-the-fly* shortest path algorithms [23, 35, 44, 61] on a terrain surface are very slow. The best-known exact algorithm [23] runs in $O(N^2)$, and it takes more than 300s on a terrain surface with 200k vertices [36]. Although some *approximate* algorithms [36, 37, 41] were proposed for reducing the running time, they are still not efficient enough. The best-known approximate algorithm [36] runs in $O(N \log N)$ time, but our experiments show that it needs 7,200s $\approx$ 2 hours to calculate the shortest path on a terrain surface with 0.5M faces, which is not acceptable.

1.2.2 ***Non-existence of oracles on updated terrain surface.*** Although existing studies [59, 60] can construct oracles on a *static* terrain surface, and can then return shortest path query results efficiently (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 oracle is called the *output size*, and the time taken to return the result is called the *shortest path query time*), no existing study can accommodate dynamic terrains, where the oracle needs to be updated efficiently. A straightforward adaptation of the best-known oracle [59, 60] is to re-construct the oracle when the terrain surface is updated. However, its oracle construction time is $O(cnN \log^2 N)$, where n is the number of POIs on T and c is a constant whose value is close to n. In our experiments, its oracle construction time is 3,075,000s $\approx$ 35.5 days for a terrain dataset with 0.5M faces and 250 POIs, which is very slow.

1.3 Path Oracle on Updated Terrain Surface

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 [59, 60]) 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*.

1.3.1 ***Key idea for achieving a short oracle update time.*** We first define the concept of a *disk*. 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. The key means 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 pairwise P2P exact shortest paths on T_{before} stored when constructing *FU-Oracle*.

(1) ***Non-updated terrain shortest path intact* property**: Consider two terrain surfaces before and after updates, i.e., T_{before} and T_{after}, respectively. This property implies that given a shortest path on T_{before}, if the two endpoints of this path are far away from the updated faces, the shortest path between the two endpoints on T_{after} is the same. Figure 2 shows two disks centered at u and v with radius equal to *half* of the shortest distance between u and v on T_{before}. If they do not intersect with the updated faces, then the shortest path between u and v (i.e., the purple line) on T_{after} is the same as the one on T_{before}. Otherwise, a shorter path between u and v may exist that intersects with the updated faces, and we need to re-calculate the shortest path between u and v on T_{after}.

(2) **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 constructing *FU-Oracle*. This is because the *exact* shortest distances are no larger than the *approximate* shortest distances. With the exact (resp. approximate) shortest paths, we can (resp. cannot) minimize the radii of the disks centered at each POI, reducing (resp. increasing) the overlaps of these disks with the updated faces, and reducing (resp. increasing) the calculations

of the shortest paths on T_{after} are smaller (resp. higher). The best-known oracle [59, 60] only stores the pairwise P2P *approximate* shortest paths on T_{before}. However, although we can adapt *non-updated terrain shortest path intact* property to it, such that there is no need to re-construct the oracle when the terrain surface is updated, its oracle update time remains large. Our experiments show that the adapted oracle update time of work [59, 60] is 21 times larger than that of *FU-Oracle*.

1.3.2 ***Key idea for achieving a small output size.*** After the terrain surface is updated, we are not interested in returning the *pairwise* P2P exact shortest paths on T_{after} as the oracle output.

(1) **Earthquake, avalanche, and glacier collapse**: 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. Since it is hard and time-consuming to dig out a rescue path in the earthquake region [32], we are not interested in digging out a path from a to b, and from a to c. Rather, we aim at digging out only one path from a to c, and to re-use this path to go to b ($a \rightarrow c \rightarrow b$), i.e., we aim at using fewer paths for connecting a, b, and c. By taking all POIs into consideration, we hope that *FU-Oracle* can output *fewer* paths among these POIs. 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.

(2) **Marsquake**: The memory size of NASA's Mars 2020 rover is 256MB [49]. Our experimental result shows 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, 47], in wireless and sensor networks for faster signal transmission [54, 58], etc. The best-known algorithm [17, 18] 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 result shows that when $n = 500$, our algorithm takes 24s, while the best-known algorithm [17, 18] takes 101s.

1.4 Contribution and Organization

We summarize our major contributions as follows.

(1) We propose a novel oracle, called *FU-Oracle*, which is the first oracle that answers shortest path queries efficiently on an updated terrain surface to the best of our knowledge. Specifically, by satisfying the novel non-updated terrain shortest path intact property, and utilizing the pairwise P2P exact shortest paths on T_{before}, the oracle achieves a short oracle update time. We also propose four additional novel techniques to further reduce the oracle update time.

(2) We develop algorithm *HieGreSpan* for efficiently generating a sub-graph from a complete graph, for reducing the output size.

(3) 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*.

(4) *FU-Oracle* performs much better than the best-known oracle [59, 60] 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-life 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 3,075,000s $\approx$ 35.5 days; and (2) the shortest path query time of *FU-Oracle* is 0.1ms, while the best-known on-the-fly algorithm needs 7,200s $\approx$ 2 hours and the best-known oracle needs 0.3ms.

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

2 PROBLEM DEFINITION

2.1 Notation and Definition

2.1.1 ***Terrain surfaces.*** 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, 52, 59, 60] (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 (i.e., $N = |V|$). Each vertex $v \in V$ has three coordinates, denoted by 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, 59, 60]. In Figure 1 (a), the terrain surface is constructed by mapping the 2D grid into a 3D model. 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), they have the same 2D grid (i.e., the x- and y-coordinates of each vertex in V in T_{before} and T_{after} are same). In practice, (1) satellites or (2) drones can be used to collect T_{before} and T_{after}, which takes (1) 144s $\approx$ 2.4 min and USD 48.72 [45], and (2) 2.16×10^5s $\approx$ 2.5 days and USD 1000 [20] for a 1km^2 region, respectively.

2.1.2 ***POIs.*** Let P be a set of POIs on the surface of the terrain and n be the size of P (i.e., $n = |P|$). We focus on the case when $n \leq N$. According to recently used earthquake rescue models [34, 46], n is at most 50, while other studies [27, 41, 52, 59, 60] suggest a magnitude of N in benchmark real terrain datasets in the millions, i.e., 10^6. We discuss how we handle the case when $n > N$ in the appendix.

2.1.3 ***Updated and non-updated components.*** Given T_{before}, T_{after}, and P, a set of (1) *updated vertices*, (2) *updated edges*, (3) *updated faces*, and (4) *updated POIs* of T_{before} and T_{after}, denoted by (1) ΔV, (2) ΔE, (3) ΔF, and (4) ΔP, is defined to be a set of (1) vertices $\Delta V = \{v_1, v_2, \dots\}$ where v_i is a vertex in V which has coordinate values differ between T_{before} and T_{after}, (2) edges $\Delta E = \{e_1, e_2, \dots\}$ where e_i is an edge in E which has any one of its two verities' coordinate values differ between T_{before} and T_{after}, (3) faces $\Delta F = \{f_1, f_2, \dots\}$ where f_i is a face in F which has any one of its three verities' coordinate values differ between T_{before}

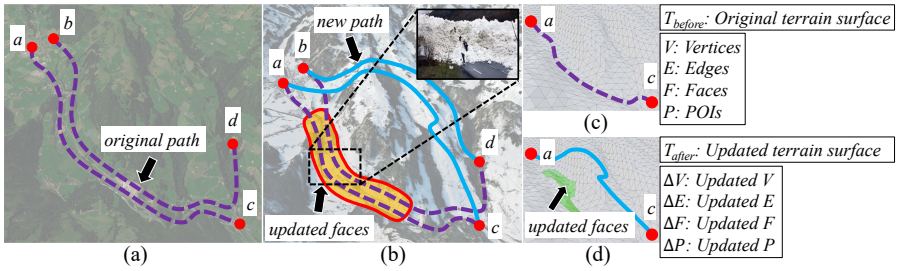

Figure 3: The (a) real map before updates, (b) real map after updates, (c) terrain model before updates, and (d) terrain model after updates for the avalanche in Switzerland

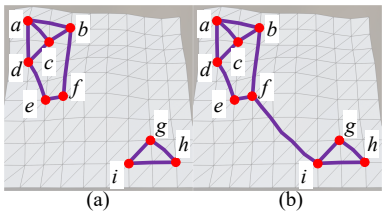

Figure 4: *FU-Oracle* output graph G

and T_{after}, (4) and POIs $\Delta P = \{p_1, p_2, \dots\}$ where p_i is a POI in P which has coordinate values 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 considered is connected [13] (in Figure 1 (d), the set of updated faces is connected).

2.1.4 ***Three types of queries.*** We study three types of queries, (1) *POI-to-POI (P2P) path query*, (2) *vertex-to-vertex (V2V) path query*, and (3) *arbitrary point-to-arbitrary point (A2A) path query*, i.e., returning the shortest path between pairs of (1) POIs, (2) vertices, and (3) 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 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. For clarity, 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 the size of P changes. When a POI is added, we can create an oracle to answer the A2A path query, which implies we have considered all possible POIs to be added. When a POI is removed, we can still use the original oracle after removing the POI.

2.1.5 ***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 we define $|\cdot|$ to be the distance of a path (e.g., $|\Pi(s, t|T)|$ is the exact distance of $\Pi(s, t|T)$ on T), where T can be T_{before} or T_{after}. A notation table appears in the appendix.

2.2 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.

3 RELATED WORK

3.1 On-the-fly Algorithm

There are two types of algorithms for computing the shortest path on a terrain surface *on-the-fly*: (1) *exact* [23, 35, 44, 61] and (2) *approximate* [36, 37, 41] algorithms.

- **Exact algorithm**: The time complexities of the exact algorithms [23, 35, 44, 61] are $O(N^2)$, $O(N \log^2 N)$, $O(N^2 \log N)$ and $O(N^2 \log N)$, respectively, which are very slow even on moderate terrain data. According to several studies [36, 37, 52, 62], the algorithm by Chen and Han [23] is recognized as the best-known exact algorithm, but it takes more than 300s on a terrain surface with 200k vertices [36].
- **Approximate algorithm**: Approximate algorithms [36, 37, 41] aim at reducing the running time. The *Kaul on-the-Fly Algorithm* (*K-Fly-Algo*) in [36] can return a $(1+\epsilon)$-approximate shortest path on a terrain surface, and it is recognized as the best-known approximate algorithm [59, 60]. 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. Experiment results [36] show that *K-Fly-Algo* needs more than 300s for even a loose error parameter $\epsilon = 0.25$.

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.

3.2 Oracle

Due to the expensive shortest path query time of on-the-fly algorithms, the only existing and best-known study, i.e., the *Well-Separated Pair Decomposition Oracle* (*WSPD-Oracle*) [59, 60], aiming at using an *oracle* to pre-compute the shortest paths on a terrain surface, and answer the *approximate* shortest path queries using the oracle.

***WSPD-Oracle*:** *WSPD-Oracle* uses the *well-separated pair decomposition* idea [22] to build its oracle. Specifically, it first builds a *compressed partition tree* in which each node corresponds to a disk containing a set of POIs. Then, it constructs a set of the pairs of nodes from the compressed partition tree, called the *well-separated node pair set*. Finally, it uses this set to index the $(1+\epsilon)$-approximation shortest path between any pair of POIs. The oracle

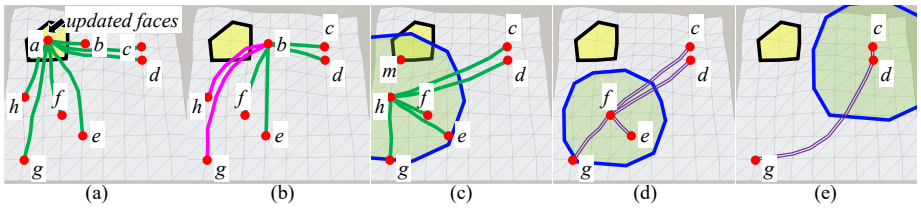

Figure 5: In update phase when (a) updating $\Pi(a)$, (b) updating $\Pi(b)$, (c) updating $\Pi(f)$, (d) no need for updating $\Pi(c)$, and (e) no need for updating $\Pi(e)$

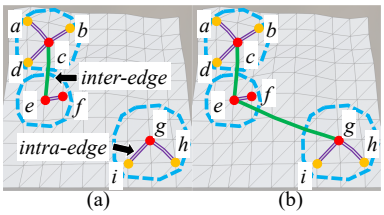

Figure 6: Hierarchy graph H

construction time, output size, and shortest path query time [59, 60] are $O(\frac{nhN\log^2 N}{\epsilon^{2\beta}})$, $O(\frac{nh}{\epsilon^{2\beta}})$, and $O(h^2)$ respectively, where h is the height of the compressed partition tree and β is the largest capacity dimension [31, 38] ($\beta \in [1.5, 2]$ in practice [59, 60]).

Drawbacks of the existing oracles: *WSPD-Oracle* only considers constructing an oracle on a *static* terrain surface, and it does not consider updating the oracle on an *updated* terrain surface. (1) If we use the straightforward adaptation, i.e., re-construct *WSPD-Oracle* from scratch when the terrain surface is updated, an update takes 3,075,000s $\approx$ 35.5 days on a terrain dataset with 0.5M faces and 250 POIs. (2) If we use the smart adaption, i.e., utilize the *non-updated terrain shortest path intact* property in *WSPD-Oracle* (we denote it as *WSPD-Oracle-Adapt*), since *WSPD-Oracle-Adapt* does not store the pairwise P2P *exact* shortest paths on T_{before} when constructing their oracle, the oracle update time remains large, as discussed in Section 1.3.1 (the 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*). In addition, to be discussed later, in order to fully utilize this property, *WSPD-Oracle-Adapt* also needs to calculate the shortest distance between each POI and vertex on T_{before} when constructing the oracle, which increases its oracle construction time (but *FU-Oracle* can calculate this information and the pairwise P2P exact shortest paths on T_{before} *simultaneously*). The oracle construction time of *WSPD-Oracle-Adapt* is 3,102,000s $\approx$ 35.9 days for a terrain dataset with 0.5M faces and 250 POIs, while *FU-Oracle* takes 27,000s $\approx$ 7.5 hours.

4 METHODOLOGY

4.1 Overview of *FU-Oracle*

4.1.1 ***Components of FU-Oracle.*** *FU-Oracle* has three components, i.e., (1) the *FU-Oracle output graph*, (2) the *temporary complete graph*, and (3) the *POI-to-vertex distance mapping table*.

The *FU-Oracle* output graph G: This is a graph that can be 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 $|e(u, v|T_{after})|$. There are n vertices in $G.V$. Given two vertices s and t in $G.V$, we define a shortest path $\Pi_G(s, t|T_{after}) = (v_1, v_2, \ldots, v_l)$ 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 $\Pi_G(s, t|T_{after})$. Given a user-defined error parameter $\epsilon > 0$, *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 $\Pi(a, c|T_{after})$ between a and c on T_{after}, and also an edge $e(a, c|T_{after})$ between a and c in G. The shortest path $\Pi_G(a, b|T_{after})$ between a and b in G is (a, b, c), which consists of edges $e(a, c|T_{after})$ and $e(c, b|T_{after})$.

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.

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}, used for further 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.

4.1.2 ***Phases of FU-Oracle.*** *FU-Oracle* has three phases, i.e., *construction phase*, *update phase* and *query phase* (see Figure 1). (1) In the construction phase, given T_{before} and P, we calculate the pairwise P2P exact shortest paths on T_{before} (stored in G') and the POI-to-vertex distance information (store in M_{dist}). (2) In the update phase, given T_{before}, T_{after} and P, we efficiently update the pairwise P2P exact shortest paths on T_{after} in G' and produce G (a sub-graph of G'). (3) In the query phase, given two query POIs and G, we compute the path between these two POIs on T_{after} using G efficiently.

4.1.3 ***Key ideas of short oracle update time of FU-Oracle.*** In the update phase, the key reasons for the short oracle update time for *FU-Oracle* is due to (1) the *non-updated terrain shortest path intact* property, and (2) the pairwise P2P exact shortest paths on T_{before} stored when constructing *FU-Oracle*. We formally present the *non-updated terrain shortest path intact* property in Property 1.

Property 1 (Non-updated Terrain Shortest Path Intact Property). *Given T_{before}, T_{after}, and $\Pi(u, v|T_{before})$, when the terrain surface is updated from T_{before} to T_{after}, 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 $\Pi(u, v|T_{after})$ cannot be different from $\Pi(u, v|T_{before})$ in our setting. The detailed proof appears in the appendix. □

We have discussed these two ideas in Section 1.3.1. In order to fully utilize this property, 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), suppose that we need to update the shortest paths between a and two POIs in $\{e, g\}$ on T_{after}. By using *Single-Source All-Destination* (*SSAD*) algorithm [23, 35, 44, 61], i.e., a Dijkstra-based exact shortest path query algorithm [29] on a terrain surface, 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 the updated faces, we can avoid using algorithm *SSAD* for updating the paths with e and g as the source POIs according to the non-updated terrain shortest path intact property). But, if we use a different path update sequence, e.g., 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.

We give some notations first before we introduce our path update sequence. A point (either a vertex or a POI) is said to be in ΔF if it is on a face in ΔF. A path on a terrain surface is said to pass ΔF if this path intersects with ΔF. In Figure 5 (a) and (b), a is said to be in ΔF, the shortest path $\Pi(a, h|T_{before})$ and $\Pi(b, h|T_{before})$ are said to pass ΔF. We then introduce three types of path update sequences, (1) updating the shortest paths for POI in updated faces, (2) updating the shortest paths for path passing updated faces, and (3) updating the shortest path for POI near updated faces. 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 $\Pi(b, h|T_{before})$ 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 shortest path for POI near updated faces 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 will increase the chance of re-calculating this shortest path on T_{after} and increase 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 additional information calculated when constructing *FU-Oracle*. 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, we can use the POI-to-vertex distance information stored in M_{dist}, which is calculated on T_{before} when constructing *FU-Oracle*, 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} in Lemma 4.1, to approximate the shortest distance on T_{after} in $O(1)$ time.

Lemma 4.1. *The minimum distance from a POI u to any point in ΔF on T_{after} is no less than $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{before})| - L_{max}$.*

Proof Sketch. We use triangle inequality to show that minimum distances from a POI to any point in ΔF on T_{after} plus L_{max} is no less than $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{before})|$. The detailed proof appears in the appendix. □

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, which implies that 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. Note that calculating the POI-to-vertex distance information will not increase the oracle construction time. When constructing *FU-Oracle*, 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 shortest path for POI near updated faces. In Figure 5 (d), when checking whether we need to re-calculate the shortest paths between f and each POI in $\{c, d, e, g\}$ on T_{after}, a naive approach is creating disks centered at f and each POI in $\{c, d, e, g\}$ with radius equal to half of the shortest distance between f and each POI in $\{c, d, e, g\}$, and check whether these eight disks intersect with ΔF. Since there are total $O(n^2)$ paths, it needs to create $O(n^2)$ disks. But, our approach just needs to create only one disk centered at f with radius equal to half of the longest distance of the shortest paths between f and each POI in $\{c, d, e, g\}$, and check whether this only one 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 passes ΔF, we sort them from near to far according to their minimum distance to any vertex in ΔV on T_{before}. For each sorted POI u, we create *one* disk centered at u with radius equal to half of the longest distance of all the shortest paths (that have not been checked) adjacent to u, and check whether this disk intersects with ΔF. There are two cases.

(4a) In Figure 5 (c), the sorted POIs are h, f, e, d, c, g. We create one disk $D(h, \frac{|\Pi(c,h|T_{before})|}{2})$, 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.

(4b) In Figure 5 (d), the sorted POIs are f, e, d, c, g. We create one disk $D(f, \frac{|\Pi(c,f|T_{before})|}{2})$, 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 cannot intersect with ΔF, and the disks $D(X, \frac{|\Pi(X,f|T_{before})|}{2})$ with centers further away from ΔF 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}, so if $D(f, \frac{|\Pi(c,f|T_{before})|}{2})$ does not intersect with ΔF, this means that $\min_{\forall v \in \Delta V} |\Pi(c, v|T_{before})| - L_{max} > \frac{|\Pi(c,f|T_{before})|}{2}$, 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 $X = \{c, d, e, f, g\}$.

Limitation of the best-known oracle after adaption: Even if we can adapt the *non-updated terrain shortest path intact* property to the best-known oracle *WSPD-Oracle* and obtain *WSPD-Oracle-Adapt*, *WSPD-Oracle-Adapt* still has two limitations.

(1) Its oracle update time is still large because it only stores the pairwise P2P *approximate* shortest path on T_{before}. In Figure 5 (d), suppose that *WSPD-Oracle-Adapt* calculates an approximate path between c and f on T_{before}, whose distance is longer than the exact shortest distance between c and f on T_{before}. So the disk centered at f with radius equal to half of the approximate shortest distance between c and f on T_{before} may intersect with ΔF, and 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 $\{c, d\}$. These cases will highly increase the oracle update time. Thus, this shows the necessity of storing the pairwise P2P exact shortest paths on T_{before}. In our experiment, the oracle update time of *WSPD-Oracle-Adapt* is 8,400s $\approx$ 2.4 hours with $\epsilon = 0.1$ for a terrain dataset with 0.5M faces and 250 POIs, but the value is only 400s $\approx$ 7 min for *FU-Oracle*.

(2) Its oracle construction time is increased compared with *WSPD-Oracle*. Because *WSPD-Oracle-Adapt* needs to calculate the POI-to-vertex distance information on T_{before} using algorithm *SSAD* for each POI additionally. The oracle construction time is 3,102,000s $\approx$ 35.9 days for *WSPD-Oracle-Adapt* on a terrain dataset with 0.5M faces and 250 POIs, but is 27,000s $\approx$ 7.5 hours for *FU-Oracle*.

4.1.4 *Key ideas of short oracle construction time of FU-Oracle*.

In the construction phase of *FU-Oracle*, each POI is given as a source point, then we can just use algorithm *SSAD* for total n times to calculate the pairwise P2P exact shortest paths on T_{before}.

Limitation of the best-known oracle before and after adaption: But, in *WSPD-Oracle* and *WSPD-Oracle-Adapt*, they do not utilize the idea of algorithm *SSAD*. Given two POIs a and b which are the centers of two disks O and O' in the well-separated node pair set, it only calculates the exact shortest path between a and b on the terrain surface. They hope to use this exact shortest path to approximate the shortest path between the centers of children nodes of O and O' on the terrain surface, for reducing the total number of the P2P shortest path. But, our experiments show that the number of pairs in the well-separated node pair set is more than n, which implies that *WSPD-Oracle* and *WSPD-Oracle-Adapt* needs to perform algorithm *SSAD* more than n times.

If they pre-compute the pairwise P2P exact shortest paths using algorithm *SSAD* for n times for time-saving, then there is no need to use the well-separated pair decomposition idea at all, since the pairwise P2P exact shortest paths is exactly the information that we want to generate during the construction phase of *FU-Oracle*. Furthermore, no matter whether we pre-compute the pairwise P2P exact shortest paths using algorithm *SSAD* when constructing *WSPD-Oracle* and *WSPD-Oracle-Adapt*, they are always constructed based on two additional data structures, i.e., the compressed partition tree and the well-separated node pair set, where the construction of these two data structures are also time-consuming. The oracle construction time of *WSPD-Oracle* is 3,075,000s $\approx$ 35.5 days for a terrain dataset with 0.5M faces and 250 POIs, but the value is 27,000s $\approx$ 7.5 hours for *FU-Oracle*.

4.1.5 *Key ideas of small output size of FU-Oracle*.

In the update phase of *FU-Oracle*, after updating of the pairwise P2P exact shortest paths on T_{after} and storing them in G', we generate a sub-graph of G', i.e., G, such that the shortest distance $|\Pi_G(s, t|T_{after})|$ (stored in G) between any pair of POIs s and t in P is at most $(1 + \epsilon)$ times the exact shortest distance of $\Pi(s, t|T_{after})$, i.e., $|\Pi_G(s, t|T_{after})| \leq (1 + \epsilon)|\Pi(s, t|T_{after})|$.

Naive method: Algorithm *Greedy Spanner* (*GreSpan*) [17, 18] first sorts the pairwise P2P exact shortest paths on the terrain surface based on their distance from minimum to maximum, and initializes G to be empty. Then, for each sorted exact shortest path $\Pi(u, v|T)$ between two POIs u and v in P, it checks whether $|\Pi_G(u, v|T)|$ is longer than $(1+\epsilon)|\Pi(u, v|T)|$ or not. If this checking is positive, then $\Pi(u, v|T)$ is added as an edge $e(u, v|T)$ into G (see Figure 4). It iterates until all the paths have been processed, and returns G as output. It is very time-consuming since it needs to use Dijkstra's algorithm [29] on *all* edges of G, to perform the shortest path query of $|\Pi_G(u, v|T)|$. Its time complexity is $O(n^3 \log n)$.

Efficient method: Our algorithm *HieGreSpan* with running time $O(n \log^2 n)$ runs faster and can be used in *FU-Oracle*. Algorithm *HieGreSpan* involves one more index, called *hierarchy graph* H, which has a simpler structure compared with G, because 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, and calculating the shortest path between u and v on H takes $O(1)$ time (but it takes $O(n \log n)$ time on G). Our experimental result shows 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*. In Figure 1 (f), the graph in purple line is the output of algorithm *HieGreSpan*, i.e., G.

4.2 Construction Phase

In the construction phase of *FU-Oracle*, given T_{before} and P, we aim to calculate the pairwise P2P exact shortest paths and the POI-to-vertex distance information. Specifically, 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', and (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.

4.3 Update Phase

In the update phase of *FU-Oracle*, given T_{before}, T_{after}, P, G', and M_{dist}, we aim to efficiently update the pairwise P2P exact shortest paths on T_{after} in G', and generate G, i.e., a sub-graph of G'. Three steps are involved:

- **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., the output graph G, for reducing the output size of *FU-Oracle*, 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}.

Notation: 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 $P_{remain} = \{p_1, p_2, \dots\}$ 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 $\Pi(u) = \{\Pi(u, v_1|T_{before}), \Pi(u, v_2|T_{before}), \dots, \Pi(u, v_l|T_{before})\}$ be a set of the exact shortest paths stored in G' on T_{before} with u as one 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: The terrain surface and POIs update detection step is easy to understand, and we discuss the sub-graph generating step in Section 4.5. We focus on the pairwise P2P exact shortest paths updating step. Before we discuss this step, we need one algorithm called *UpdatePath*, in Algorithm 1. In Figure 5 (c), we compute *UpdatePath* $(h, T_{after}, G', P_{remain} = \{c, d, e, f, g\})$. Algorithm 2 shows the pairwise P2P exact shortest paths updating step.

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

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: $\quad G'.E \leftarrow G'.E - \{\Pi(u, v|T_{before})\} \cup \{\Pi(u, v|T_{after})\}$
4: $\quad \Pi(v) \leftarrow \Pi(v) - \{\Pi(u, v|T_{before})\}$
5: $P_{remain} \leftarrow P_{remain} - \{u\}$
6: **return** updated G' and P_{remain}

Algorithm 2 *PairwiseP2PUpdatePath* $(G', P, M_{dist}, \Delta F, \Delta P)$

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

Example: The following shows an example.

(1) *Exact shortest path updating for POIs in updated faces*: Line 4-6. In Figure 5 (a), $a \in \Delta P$, and we need to update the exact shortest paths in green on T_{after}.

(2) *Exact shortest path updating for paths passing updated faces*: Line 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 need to update the exact shortest paths in green and pink on T_{after}.

(3) *Exact shortest path updating for POIs near updated faces*: Line 10-16. (3a) *Disk intersects*: Line 13-15. In Figure 5 (c), the sorted POIs are h, f, e, d, c, g, the selected path with the longest distance is $\Pi(c, h|T_{before})$, and the disk is the blue circle. Since it intersects with ΔF, we need to update the paths in green on T_{after}. (3b) *Disk does not intersect*: Line 15-16. In Figure 5 (d), the sorted POIs are f, e, d, c, g, the selected path with the longest distance is $\Pi(c, f|T_{before})$, and the disk is in the blue circle. Since it does not intersect with ΔF, we do not need to update the paths.

4.4 Query Phase

In the query phase of *FU-Oracle*, 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 [29] to find the shortest path between s and t on G, i.e., $\Pi_G(s, t|T_{after})$, which is a $(1+\epsilon)$-approximat exact shortest path of $\Pi(s, t|T_{after})$ on 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 $\Pi(c, b|T_{after})$.

4.5 Implementation Details of Algorithm *HieGreSpan*

Given G' and ϵ, by using *HieGreSpan*, we generate a sub-graph of G', i.e., the *FU-Oracle* output graph G with small output size efficiently, such that for all pairs of vertices u and v in $G.V$, we have $|\Pi_G(u, v|T_{after})| \leq (1+\epsilon)|\Pi(u, v|T_{after})|$.

4.5.1 ***Overview of HieGreSpan.*** In *HieGreSpan*, apart from G' and G, we need one more index, called *hierarchy graph* H, which is maintained simultaneously with 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. With the concept of *group*, we give an overview of *HieGreSpan*. Intuitively, 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 $\delta = \frac{1}{2}(\frac{\sqrt{\epsilon+1}-1}{\sqrt{\epsilon+1}+3}) \in (0, \frac{1}{2})$ is a small constant, since $\epsilon \in (0, \infty)$) in H, 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, we can answer the shortest path query using Dijkstra's algorithm on H efficiently (since H has fewer vertices and edges). 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.

4.5.2 ***Notation.*** Similar to G, let $H.E$ be the set of edges of H. Given a group Q_G, we define *intra-edges* to be a set of edges connecting the group center of Q_G to all other vertices in Q_G, and we define *inter-edges* to be a set of edges connecting two group centers. The hierarchy graph 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, with endpoints u and v, the weight of this edge is denoted as $|e_H(u, v|T_{after})|$. Given two group centers s and t, we define a shortest path of inter-edges $\Pi_H(s, t|T_{after}) = (v_1, v_2, \ldots, v_l)$ 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).

4.5.3 ***Detail and example.*** Algorithm 3 shows *HieGreSpan*, and the following illustrates the algorithm with an example.

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

(2) *G maintenance*: Line 7-25.

(2a) *Groups construction and intra-edges adding for H*: Line 9-15. In Figure 6 (a), we have three groups with group center c, e, and g. We then add purple lines $e(a, c|T_{after})$, $e(b, c|T_{after})$, ... as intra-edges in H.

(2b) *First type inter-edges adding for H*: Line 16-19. In Figure 6 (a), we add green lines $e'(c, e|T_{after})$ as inter-edges in H.

Algorithm 3 *HieGreSpan* (G', ϵ)

Input: temporary complete graph G' and error parameter ϵ
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 weight in increasing order
4: create intervals $I_0 = (0, \frac{D}{N}]$, $I_i = (\frac{2^{i-1}D}{n}, \frac{2^iD}{n}]$ for $i \in [1, \log n]$
5: $G'.E^i \leftarrow$ sorted edges of $G'.E$ with 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 $u_j \in G'.V$ such that has not been visited before **do**
10: perform Dijkstra's algorithm on G, such that the algorithm never visits vertices further than $\delta\frac{2^iD}{n}$ from u_j
11: create 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 $|\Pi_G(u_j, v|T_{after})| \leq \delta\frac{2^iD}{n}$ **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 $\frac{2^iD}{n} + 2\delta\frac{2^iD}{n}$ from u_j
18: H inter-edges $\leftarrow H.E \cup \{e_H(u_j, u|T_{after})\}$, where u is other group centers, $|\{e_H(u_j, U|T_{after})\}| = |\Pi_G(u_j, u|T_{after})| \leq \frac{2^iD}{n} + 2\delta\frac{2^iD}{n}$
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

(2c) *Edges examining on H*: Line 20-25. In Figure 4 (a), supposing 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 $|\Pi_H(e, g|T_{after})| = \infty > (1+\epsilon)|e'(f, i|T_{after})|$. (2c-i) *Edges adding for G*: Line 24. In Figure 4 (b), we add $e'(f, i|T_{after})$ into G. (2c-ii) *Second type inter-edges adding for H*: Line 25. In Figure 6 (b), 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.

4.5.4 ***Analysis.*** Theorem 4.2 shows the theoretical analysis.

Theorem 4.2. *The running time of HieGreSpan is* $O(n \log^2 n)$. *The output of HieGreSpan, i.e., G, satisfies* $|\Pi_G(u, v|T_{after})| \leq (1+\epsilon)|\Pi(u, v|T_{after})|$ *for all pairs of vertices u and v in G.V.*

Proof Sketch. The running time includes the edge sorting and interval splitting time $O(n)$ and G maintenance time $O(n \log^2 n)$. The error bound is due to the correctness of *GreSpan*. The detailed proof appears in the appendix. □

Another study [25] also builds an additional graph to approximate the shortest path on the result graph, but it is totally different from ours, and can yield incorrect results. It is claimed that the additional graph built in each weight interval has similar weights, so that one can simply use a brute-force search to find the shortest path on this graph. However, our experimental study finds that the weight of each edge in the additional graph in one weight interval can differ a lot, meaning that using a brute-force search can cause wrong shortest path results. Instead, in our case, if two vertices

Algorithm	Oracle construction time		Oracle update time		Output size		Shortest path query time	
WSPD-Oracle [59, 60]	$O(\frac{nhN\log^2 N}{\epsilon^{2\beta}})$	Large	$O(\frac{nhN\log^2 N}{\epsilon^{2\beta}})$	Gigantic	$O(\frac{nh}{\epsilon^{2\beta}})$	Large	$O(h^2)$	Small
WSPD-Oracle-Adapt	$O(\frac{nhN\log^2 N}{\epsilon^{2\beta}} + nN\log^2 N)$	Large	$O(\mu_1 N\log^2 N + n\log^2 n)$	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	$O(\mu_2 N\log^2 N + n\log^2 n)$	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	$O(nN\log^2 N)$	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**
K-Fly-Algo [36]	-	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

Table 1: Comparison of algorithms

Remark: $n < N$, h is the height of the compressed partition tree, β is the largest capacity dimension [31, 38], μ_1 and μ_2 are two data-dependent variables, $\mu_1 \in [5, 20]$ and $\mu_2 \in [5, 10]$ in our experiment.

are far away, we use the shortest path of inter-edges in H to approximate the real shortest path in G. The correctness is given by Theorem 4.2, and our experimental result is also consistent with it.

4.6 Necessity of Storing G'

4.6.1 ***Notation and corollary.*** 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 updating the exact shortest paths 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 updating the exact shortest paths on T_{after}, and there is a total of 8 POIs, so $UR(\textit{FU-Oracle}) = \frac{3}{8}$. Given an oracle A, a higher $UR(A)$ means the oracle update time of A is larger. Corollary 4.3 shows the necessity of storing G'.

Corollary 4.3. *Given T_{before}, T_{after}, and P, FU-Oracle has the smallest UR(FU-Oracle) compared with all other oracles that do not store the pairwise P2P exact shortest paths on T_{before}, because FU-Oracle stores the pairwise P2P exact shortest paths on T_{before} in G'.*

4.6.2 ***Limitation of the best-known oracle after adaption.*** Given an oracle A that does not store the pairwise P2P exact shortest paths on T_{before}, we show that $UR(\textit{FU-Oracle}) \leq UR(A)$. Since *WSPD-Oracle-Adapt* is an instance of A, we use it as an example. In Figure 5, for *WSPD-Oracle-Adapt*, recall that the disk centered at each POI has a larger radius, and that 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 updating the exact shortest paths on T_{after}, and $UR(\textit{FU-Oracle}) = \frac{7}{8}$. In other words, 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 recuse paths faster, so it is necessary to store the pairwise P2P exact shortest paths on T_{before} in G'.

4.7 Theoretical Analysis

Theorem 4.4 provides a theoretical analysis of *FU-Oracle*.

Theorem 4.4. *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 $O(nN\log^2 N)$. The oracle update time includes terrain surface and POIs update detection time $O(N + n)$, the pairwise P2P exact shortest paths update time $O(N\log^2 N)$, and the sub-graph generating time $O(n\log^2 n)$. The output size is $O(n)$, owing 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*. The detailed proof appears in the appendix. □

4.8 Baselines and Comparisons

4.8.1 ***Baseline on-the-fly algorithm.*** We include the best-known on-the-fly algorithm *K-Fly-Algo* [36] as a baseline.

4.8.2 ***Baseline oracle.*** We include the best-known oracle, *WSPD-Oracle* [59, 60], and its adaption *WSPD-Oracle-Adapt* as baselines. They 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 constructing the oracle.

Furthermore, in *FU-Oracle*, we (1) use a random path update sequence instead of our novel path update sequence, (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 the updated faces on T_{after} 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* 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.

There is no need to remove the non-updated terrain shortest path intact property in *FU-Oracle*, such that when the terrain surface is updated, we re-construct *FU-Oracle* from scratch, for ablation study. In this case, the oracle update time of this adaption is even larger than the oracle construction time of *FU-Oracle*, since we still need the sub-graph generating step in the update phase of this adaption. In addition, there is no need to not store the pairwise P2P exact shortest paths on T_{before} when constructing *FU-Oracle*, for ablation study. This is because when constructing *FU-Oracle*, it can already store such information.

4.8.3 ***Comparisons.*** We compare 10 algorithms (9 baselines and our oracle *FU-Oracle*) in terms of oracle construction time, oracle update time, output size, and shortest path query time in Table 1. The detailed theoretical analysis with proofs for the 9 baselines appear in the appendix. *FU-Oracle* is the best in terms of all performance metrics.

5 EMPIRICAL STUDY

5.1 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++. For the following experiment setup, we mainly follow the experiment setups used in the literature [36, 37, 41, 59, 60].

Name	Magnitude	Date	$\|F\|$
Tohoky, Japan (TJ) [10]	9.0	Mar 11, 2011	**0.5M**, 1M, 1.5M, 2M, 2.5M
Sichuan, China (SC) [1]	8.0	May 12, 2008	**0.5M**, 1M, 1.5M, 2M, 2.5M
Gujarat, India (GI) [8]	7.6	Jan 26, 2001	**0.5M**, 1M, 1.5M, 2M, 2.5M
Alaska, USA (AU) [2]	7.1	Nov 30, 2018	**0.5M**, 1M, 1.5M, 2M, 2.5M
Leogane, Haiti (LH) [9]	7.0	Jan 12, 2010	**0.5M**, 1M, 1.5M, 2M, 2.5M
Valais, Switzerland (VS) [7]	4.1	Oct 24, 2016	**0.5M**, 1M, 1.5M, 2M, 2.5M

Table 2: Real earthquake terrain datasets

Datasets: We conduct our experiment based on 30 real before and after earthquake terrain datasets listed in Table 2 with 0.5M faces by default. We first obtain the before and after earthquake terrain satellite maps with a 5km × 5km covered region from Google Earth [5] with a resolution of 10m [27, 41, 52, 59, 60], and then we use Blender [3] to generate the terrain model. In order to study the scalability, we follow an existing generation procedure for multi-resolution terrain datasets [41, 59, 60] 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 [59, 60].

Algorithms: Our oracle *FU-Oracle*, and the baselines, i.e., *K-Fly-Algo* [36], *WSPD-Oracle* [59, 60], and *WSPD-Oracle-Adapt*, as well as *FU-Oracle-**X*** where ***X*** = {*RanUpdSeq, FullRad, NoDistAppr, NoEffIntChe, NoEdgPru, NoEffEdgPru*}, are studied. Since *WSPD-Oracle* and *WSPD-Oracle-Adapt* are not feasible with 500 POIs 500 due to their expensive oracle construction time, and since *FU-Oracle-**X*** where ***X*** = {*RanUpdSeq, FullRad, NoDistAppr*} have excessive oracle update times with 500 POIs, we (1) compare these 10 algorithms on 30 datasets with fewer POIs (50 by default), and (2) compare *FU-Oracle-**X*** where ***X*** = {*NoEffIntChe, NoEdgPru, NoEffEdgPru*}, *FU-Oracle*, and *K-Fly-Algo* on 30 datasets with more POIs (500 by default).

Query generation: We randomly choose pairs of POIs in P on T_{after}, as source and destination. For each measurement, the average, minimum, and maximum results of 100 queries are reported.

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). In addition, we consider six performance metrics, namely (1) *oracle construction time*, (2) *oracle update time*, (3) *oracle size* (i.e., the space consumption of the oracle in the oracle construction phase), (4) *output size*, (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).

5.2 Experimental Results

Figure 7, Figure 8, and Figure 9 show the P2P path query result on *GI*, *LH*, and *TJ* datasets (with fewer POIs) when varying n, ϵ, and *DS*, respectively. Figure 10 and Figure 11 show the result on *AU* and *VS* datasets (with more POIs) when varying 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.

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). *FU-Oracle* offers superior performance over *WSPD-Oracle*, *WSPD-Oracle-Adapt*, 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 difference between *FU-Oracle-NoEffEdgPru* and *FU-Oracle* is significant when n is large. In Figure 7 (d) and Figure 10 (b), the output size for *FU-Oracle-NoEdgPru* is larger than *FU-Oracle*.

Effect of ϵ for the P2P path query. In Figure 8, 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 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. Although the output size of *FU-Oracle* is slightly larger than that of *FU-Oracle-NoEffEdgPru*, the latter oracle's update time is larger (see Figure 10 (a)).

Effect of *DS* (scalability test) for the P2P path query. In Figure 9 (resp. Figure 11), we tested the 5 values of *DS* in {0.5M, 1M, 1.5M, 2M, 2.5M} on *TJ* dataset with fewer POIs (resp. *VS* dataset with more POIs) while fixing ϵ at 0.1 and n at 50 (ϵ to 0.25 and n to 500) to study scalability. The shortest path query time of *FU-Oracle* is 10^7 to 10^8 times smaller than that of *K-Fly-Algo*.

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 multi-resolution of *SC* dataset. The result appears in the appendix. The results are similar to those for the P2P path query in the case $n \leq N$. *FU-Oracle* still performs

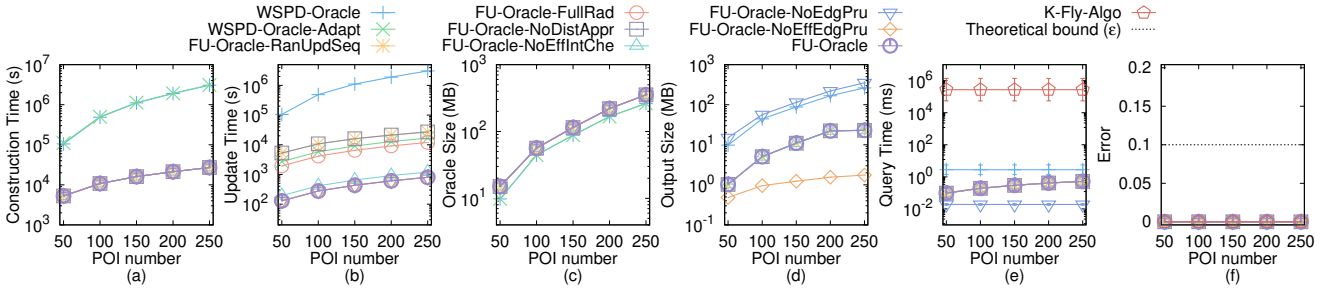

Figure 7: Effect of n on *GI* dataset (fewer POIs)

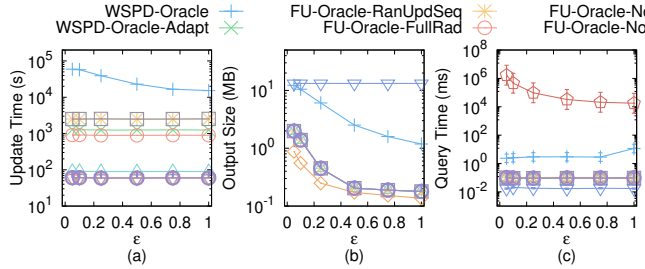

Figure 8: Effect of ϵ on *LH* dataset (fewer POIs)

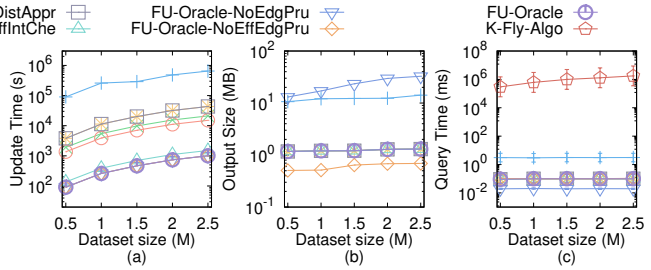

Figure 9: Effect of DS on *TJ* dataset (fewer POIs)

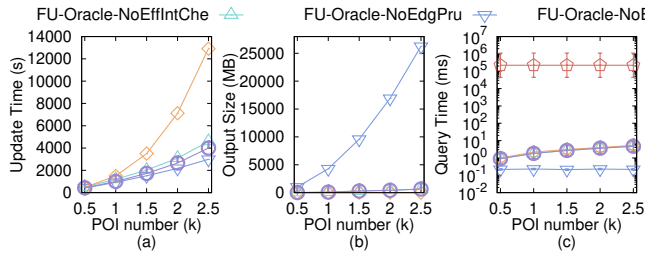

Figure 10: Effect of n on *AU* dataset (more POIs)

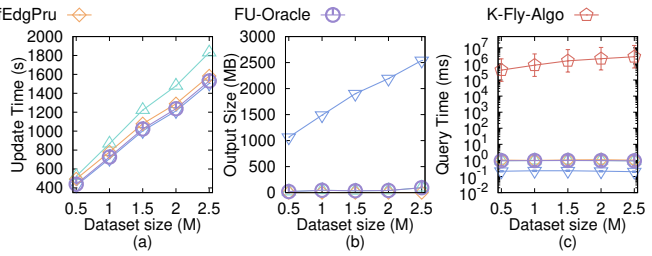

Figure 11: Effect of DS on *VS* dataset (more POIs)

better than *WSPD-Oracle* and *WSPD-Oracle-Adapt* in terms of all metrics.

5.3 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 [7], 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 $\approx$ 7 min to update the oracle, while the best-known oracle *WSPD-Oracle* needs 3,075,000s $\approx$ 35.5 days.

5.4 Experimental Results Summary

FU-Oracle consistently outperforms all existing oracles, i.e., *WSPD-Oracle* and *WSPD-Oracle-Adapt* in terms of all performance metrics (i.e., oracle construction time, oracle update time, output size, and shortest path query time). Specifically, *FU-Oracle* is up to 114 times, 4,100 times, 12 times, and 3 times better than the best-known oracle i.e., *WSPD-Oracle*, in terms of oracle construction time, oracle update time, output size, and shortest path query time. For a terrain dataset with 0.5M faces and 250 POIs, *FU-Oracle*'s oracle update time is 400s $\approx$ 7 min, while *WSPD-Oracle* needs 3,075,000s $\approx$ 35.5 days, and *WSPD-Oracle-Adapt* needs 8,400s $\approx$ 2.4 hours. When dataset size is 0.5M and with $\epsilon = 0.05$, *FU-Oracle*'s shortest path query time is 0.1ms, while algorithm *K-Fly-Algo* needs 7,200s $\approx$ 2 hours, and *WSPD-Oracle* and *WSPD-Oracle-Adapt* need 0.3ms.

6 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. In future work, it is of interest to study whether additional pruning steps in *FU-Oracle* can be invented that can further reduce the oracle update time.

REFERENCES

[1] 2022. *2008 Sichuan earthquake.* https://en.wikipedia.org/wiki/2008_Sichuan_earthquake

[2] 2022. *2018 Anchorage earthquake.* https://en.wikipedia.org/wiki/2018_Anchorage_earthquake

[3] 2022. *Blender.* https://www.blender.org

[4] 2022. *Cyberpunk 2077.* https://www.cyberpunk.net

[5] 2022. *Google Earth.* https://earth.google.com/web

[6] 2022. *Metaverse.* https://about.facebook.com/meta

[7] 2022. *Moderate mag. 4.1 earthquake - 6.3 km northeast of Sierre, Valais, Switzerland, on Monday, October 24, 2016 at 16:44 GMT.* https://www.volcanodiscovery.com/earthquakes/quake-info/1451397/mag4quake-Oct-24-2016-Leukerbad-VS.html

[8] 2023. *2001 Gujarat earthquake.* https://en.wikipedia.org/wiki/2001_Gujarat_earthquake

[9] 2023. *2010 Haiti earthquake.* https://en.wikipedia.org/wiki/2010_Haiti_earthquake

[10] 2023. *2011 Tohoku earthquake and tsunami.* https://en.wikipedia.org/wiki/2011_Tohoku_earthquake_and_tsunami

[11] 2023. *2023 Turkey–Syria earthquake.* https://en.wikipedia.org/wiki/2023_Turkey-Syria_earthquake

[12] 2023. *China National Space Administration.* https://en.wikipedia.org/wiki/China_National_Space_Administration

[13] 2023. *Connected space.* https://en.wikipedia.org/wiki/Connected_space

[14] 2023. *Falcon 9.* https://en.wikipedia.org/wiki/Falcon_9

[15] 2023. *SpaceX.* https://en.wikipedia.org/wiki/SpaceX

[16] Chang Alicia. 2011. *Cost of NASA's next Mars rover soars to $2.5 billion.* https://www.nbcnews.com/id/wbna41374241

[17] Ingo Althöfer, Gautam Das, David Dobkin, and Deborah Joseph. 1990. Generating sparse spanners for weighted graphs. In *Proceedings of the Scandinavian Workshop on Algorithm Theory*. 26–37.

[18] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. 1993. On sparse spanners of weighted graphs. *Discrete & Computational Geometry* 9, 1 (1993), 81–100.

[19] Good Andrew, Fox Karen, and Alana Johnson. 2021. *NASA's InSight Detects Two Sizable Quakes on Mars.* https://www.nasa.gov/feature/jpl/nasa-s-insight-detects-two-sizable-quakes-on-mars

[20] Antonio Annis, Fernando Nardi, Andrea Petroselli, Ciro Apollonio, Ettore Arcangeletti, Flavia Tauro, Claudio Belli, Roberto Bianconi, and Salvatore Grimaldi. 2020. UAV-DEMs for small-scale flood hazard mapping. *Water* 12, 6 (2020), 1717.

[21] Baruch Awerbuch. 1985. Communication-time trade-offs in network synchronization. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*. 272–276.

[22] Paul B Callahan and S Rao Kosaraju. 1995. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM* 42, 1 (1995), 67–90.

[23] Jindong Chen and Yijie Han. 1990. Shortest Paths on a Polyhedron. In *Proceedings of the sixth annual symposium on Computational geometry*. New York, NY, USA, 360–369.

[24] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms.* MIT press.

[25] Gautam Das and Giri Narasimhan. 1994. A fast algorithm for constructing sparse Euclidean spanners. In *Proceedings of the tenth annual symposium on Computational geometry*. 132–139.

[26] Ke Deng, Heng Tao Shen, Kai Xu, and Xuemin Lin. 2006. Surface k-NN query processing. In *Proceedings of the International Conference on Data Engineering*. IEEE, 78–78.

[27] Ke Deng and Xiaofang Zhou. 2004. Expansion-based algorithms for finding single pair shortest path on surface. In *Proceedings of the International Workshop on Web and Wireless Geographical Information Systems*. 151–166.

[28] Brett G Dickson and P Beier. 2007. Quantifying the influence of topographic position on cougar (Puma concolor) movement in southern California, USA. *Journal of Zoology* 271, 3 (2007), 270–277.

[29] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[30] Hristo N Djidjev and Christian Sommer. 2011. Approximate distance queries for weighted polyhedral surfaces. In *Proceedings of the European Symposium on Algorithms*. 579–590.

[31] Mingyu Fan, Hong Qiao, and Bo Zhang. 2009. Intrinsic dimension estimation of manifolds by incising balls. *Pattern Recognition* 42, 5 (2009), 780–787.

[32] Yu Hong and Jun Liang. 2022. Excavators used to dig out rescue path on cliff in earthquake-hit Luding of SW China's Sichuan. *People's Daily Online* (2022). http://en.people.cn/n3/2022/0909/c90000-10145381.html

[33] Berrio Mardo Jaime. 2021. *Is NASA able to remotely repair the Mars rover?* https://www.quora.com/Is-NASA-able-to-remotely-repair-the-Mars-rover

[34] Xiang-Yu Jiang, Nai-Yuan Pa, Wen-Chang Wang, Tian-Tian Yang, and Wen-Tsao Pan. 2020. Site Selection and Layout of Earthquake Rescue Center Based on K-Means Clustering and Fruit Fly Optimization Algorithm. In *Proceedings of the IEEE International Conference on Artificial Intelligence and Computer Applications*. IEEE, 1381–1389.

[35] Sanjiv Kapoor. 1999. Efficient computation of geodesic shortest paths. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*. 770–779.

[36] Manohar Kaul, Raymond Chi-Wing Wong, and Christian S Jensen. 2015. New lower and upper bounds for shortest distance queries on terrains. *Proceedings of the VLDB Endowment* 9, 3 (2015), 168–179.

[37] Manohar Kaul, Raymond Chi-Wing Wong, Bin Yang, and Christian S Jensen. 2013. Finding shortest paths on terrains by killing two birds with one stone. *Proceedings of the VLDB Endowment* 7, 1 (2013), 73–84.

[38] Balázs Kégl. 2002. Intrinsic dimension estimation using packing numbers. *Advances in neural information processing systems* 15 (2002).

[39] Marcel Körtgen, Gil-Joo Park, Marcin Novotni, and Reinhard Klein. 2003. 3D shape matching with 3D shape contexts. In *Proceedings of the 7th central European seminar on computer graphics*, Vol. 3. 5–17.

[40] Hongyang Li and Zhiling Huang. 2022. *82 die in Sichuan quake, rescuers race against time to save lives.* https://www.chinadailyhk.com/article/289413#82-die-in-Sichuan-quake-rescuers-race-against-time-to-save-lives

[41] Lian Liu and Raymond Chi-Wing Wong. 2011. Finding shortest path on land surface. In *Proceedings of the ACM SIGMOD International Conference on Management of data*. 433–444.

[42] Anders Mårell, John P Ball, and Annika Hofgaard. 2002. Foraging and movement paths of female reindeer: insights from fractal analysis, correlated random walks, and Lévy flights. *Canadian Journal of Zoology* 80, 5 (2002), 854–865.

[43] Niall McCarthy. 2021. *Exploring the red planet is a costly undertaking.* https://www.statista.com/chart/24232/life-cycle-costs-of-mars-missions/

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

[45] Janet E Nichol, Ahmed Shaker, and Man-Sing Wong. 2006. Application of high-resolution stereo satellite images to detailed landslide hazard assessment. *Geomorphology* 76, 1-2 (2006), 68–75.

[46] Shenrun Pan and Manzhi Li. 2020. Construction of earthquake rescue model based on hierarchical voronoi diagram. *Mathematical Problems in Engineering* 2020 (2020), 1–13.

[47] David Peleg and Jeffrey D Ullman. 1987. An optimal synchronizer for the hypercube. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 77–85.

[48] China Power. 2023. *What's Driving China's Race to Build a Space Station?* https://chinapower.csis.org/chinese-space-station/

[49] NASA Science. 2022. *Mars 2020 mission perseverance rover brains.* https://mars.nasa.gov/mars2020/spacecraft/rover/brains/

[50] NASA Science. 2022. *Mars 2020 mission perseverance rover communications.* https://www.statista.com/chart/24232/life-cycle-costs-of-mars-missions/

[51] NASA Science. 2023. *NASA Mars Exploration.* https://mars.nasa.gov

[52] Cyrus Shahabi, Lu-An Tang, and Songhua Xing. 2008. Indexing land surface for efficient knn query. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1020–1031.

[53] Jamie Shotton, John Winn, Carsten Rother, and Antonio Criminisi. 2006. Textonboost: Joint appearance, shape and context modeling for multi-class object recognition and segmentation. In *Proceedings of the European conference on computer vision*. 1–15.

[54] Hanan Shpungin and Michael Segal. 2010. Near-optimal multicriteria spanner constructions in wireless ad hoc networks. *IEEE/ACM Transactions on Networking* 18, 6 (2010), 1963–1976.

[55] British Geological Survey. 2023. *Where do earthquakes occur?* https://www.bgs.ac.uk/discovering-geology/earth-hazards/earthquakes/where-do-earthquakes-occur/

[56] United States Geological Survey. 2023. *Why are we having so many earthquakes?* https://www.usgs.gov/faqs/why-are-we-having-so-many-earthquakes-has-naturally-occurring-earthquake-activity-been

[57] Kennedy Tristan. 2023. *A Deadly Glacier Collapse Sends a Dire Climate Warning.* https://www.wired.co.uk/article/marmolada-glacier-collapse

[58] Pascal Von Rickenbach and Rogert Wattenhofer. 2004. Gathering correlated data in sensor networks. In *Proceedings of the joint workshop on Foundations of mobile computing*. 60–66.

[59] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, David Mount, and Hanan Samet. 2022. Proximity Queries on Terrain Surface. *ACM Transactions on Database Systems* 47, 4 (2022), 1–59.

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

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

[62] Songhua Xing, Cyrus Shahabi, and Bei Pan. 2009. Continuous monitoring of nearest neighbors on land surface. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1114–1125.

[63] Da Yan, Zhou Zhao, and Wilfred Ng. 2012. Monochromatic and bichromatic reverse nearest neighbor queries on land surfaces. In *Proceedings of the 21st ACM*

international conference on Information and knowledge management. 942–951.
[64] Yinzhao Yan and Raymond Chi-Wing Wong. 2021. Path Advisor: a multi-functional campus map tool for shortest path. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2683–2686.

A SUMMARY OF FREQUENT USED NOTATIONS

Table 3 shows a 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 one 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

Table 3: Summary of frequent used notations

B 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 4, 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.

C 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 4, 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 [30]. Specifically, [30] 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 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 [30]. 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 [30], $|\mathcal{N}(u)| \cdot |\mathcal{N}(v)| = \frac{1}{\sin\theta\epsilon}$, 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.

D 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. 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.

E EMPIRICAL STUDIES

E.1 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*

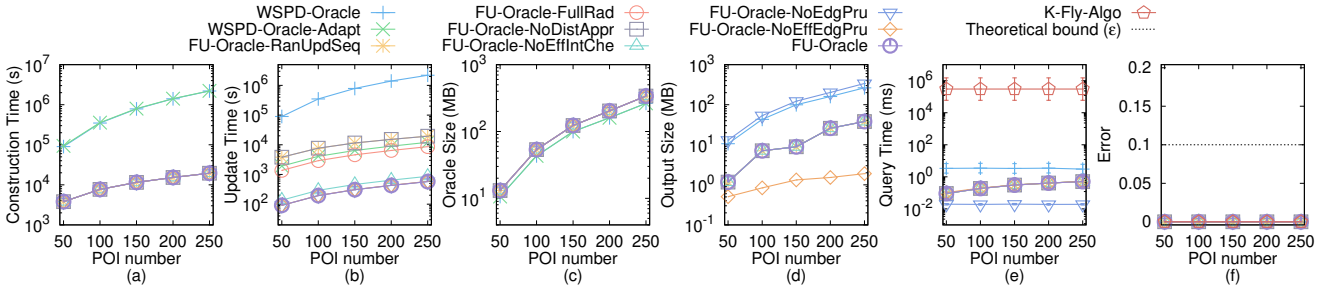

Figure 12: Effect of n on *TJ* dataset (fewer POIs) for the P2P path query

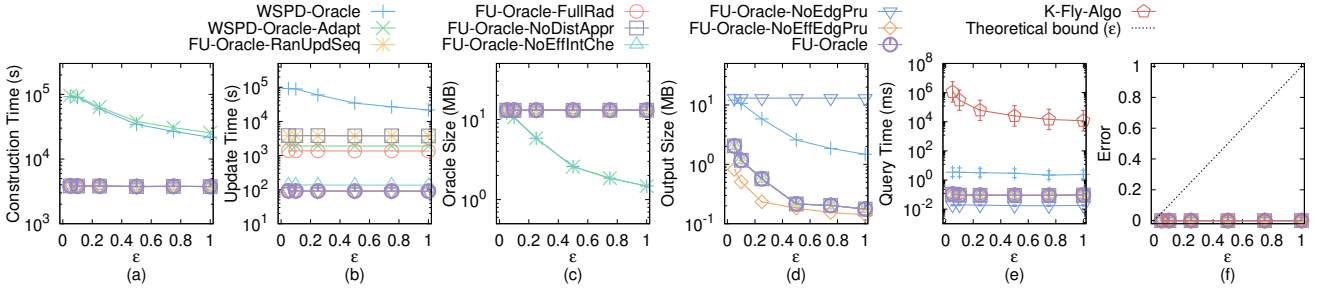

Figure 13: Effect of ϵ on *TJ* dataset (fewer POIs) for the P2P path query

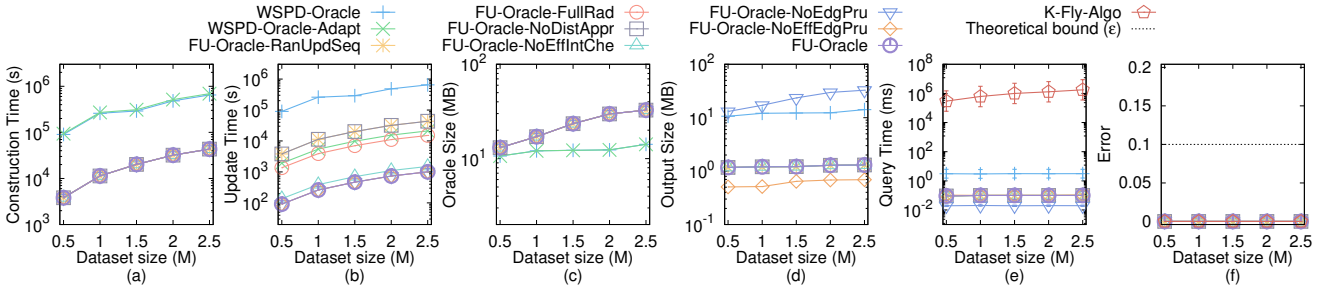

Figure 14: Effect of DS on *TJ* dataset (fewer POIs) for the P2P path query

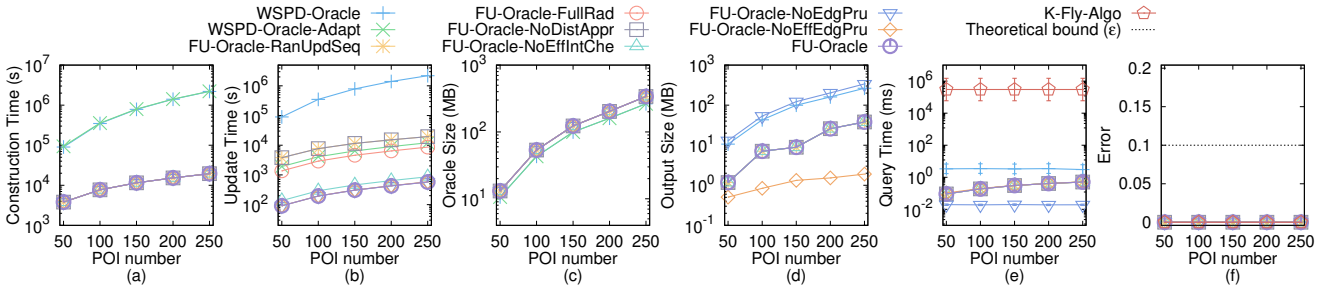

Figure 15: Effect of n on *SC* dataset (fewer POIs) for the P2P path query

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

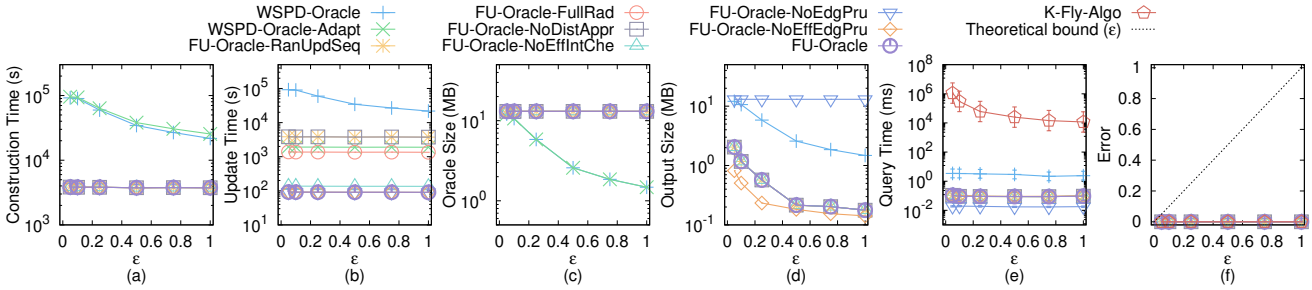

Figure 16: Effect of ϵ on *SC* dataset (fewer POIs) for the P2P path query

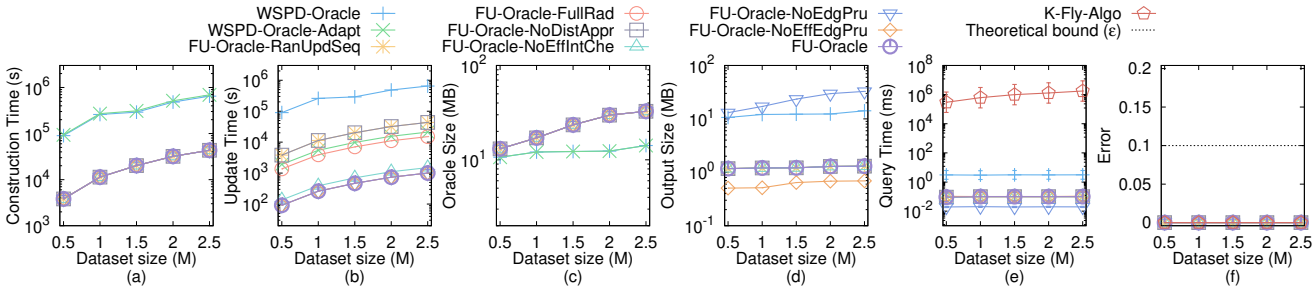

Figure 17: Effect of *DS* on *SC* dataset (fewer POIs) for the P2P path query

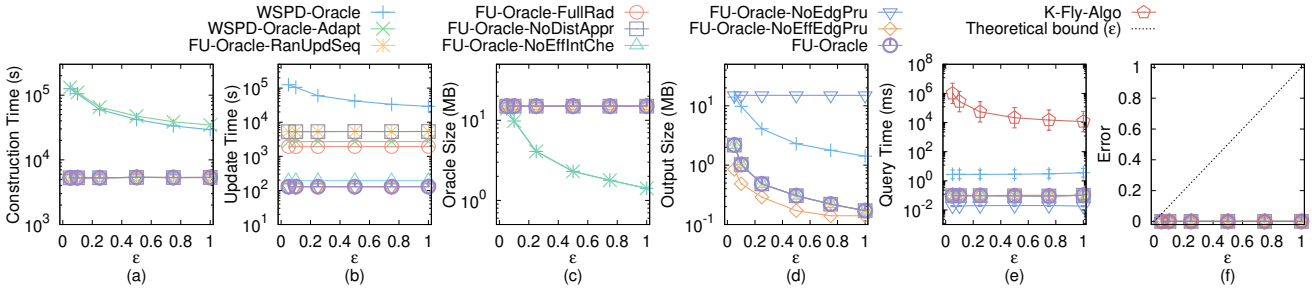

Figure 18: Effect of ϵ on *GI* dataset (fewer POIs) for the P2P path query

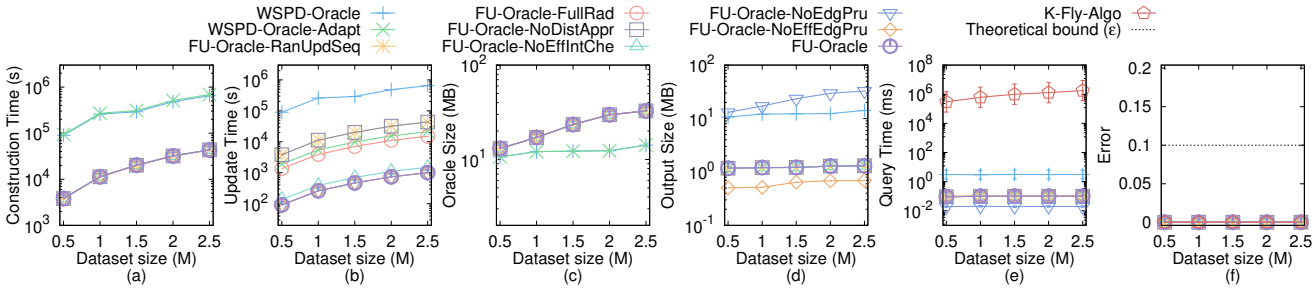

Figure 19: Effect of *DS* on *GI* dataset (fewer POIs) for the P2P path query

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*,

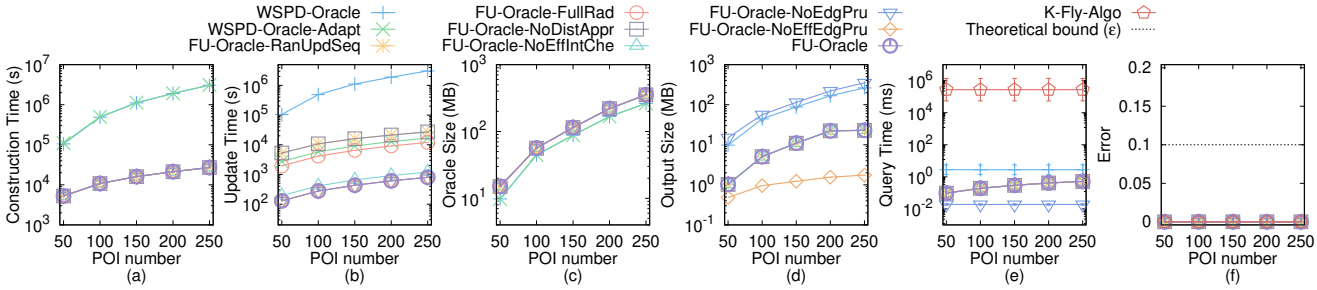

Figure 20: Effect of n on *AU* dataset (fewer POIs)

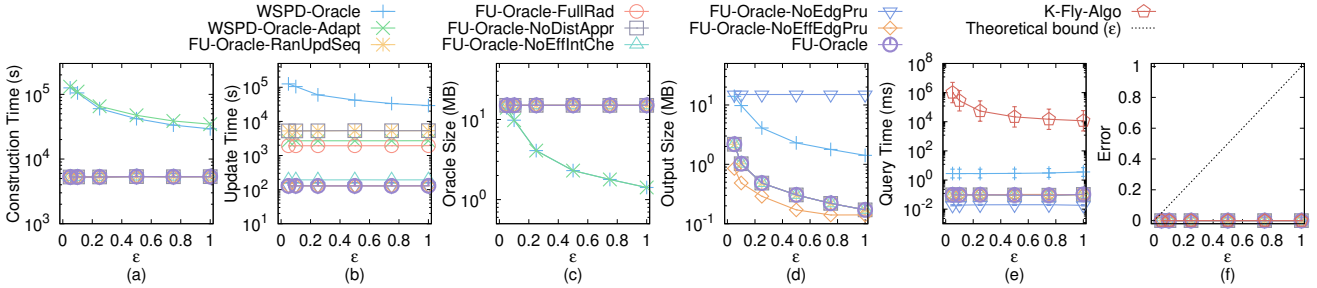

Figure 21: Effect of ϵ on *AU* dataset (fewer POIs) for the P2P path query

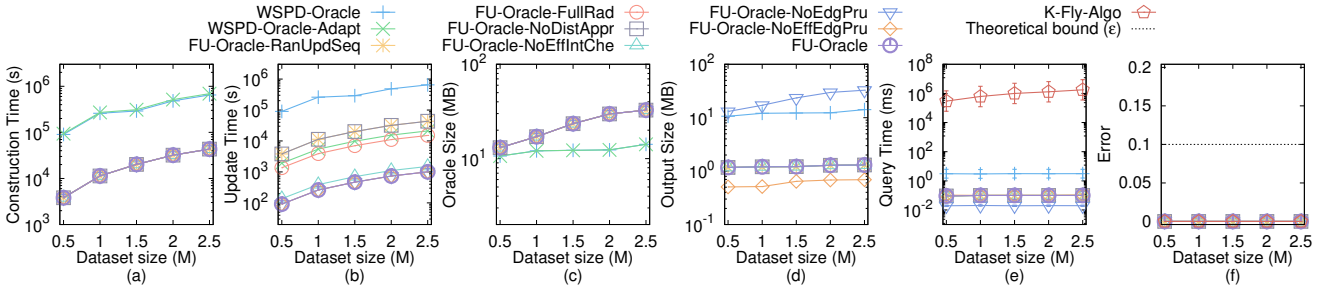

Figure 22: Effect of DS on *AU* dataset (fewer POIs) for the P2P path query

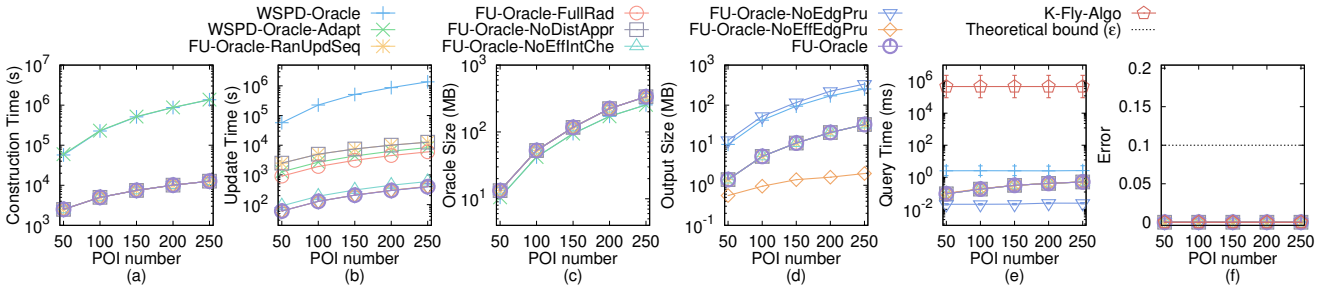

Figure 23: Effect of n on *LH* dataset (fewer POIs) for the P2P path query

WSPD-Oracle-Adapt, 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 difference between *FU-Oracle-NoEffEdgPru* and *FU-Oracle* is significant when n is large. The output size for *FU-Oracle-NoEdgPru* is larger than *FU-Oracle*. Thus, these show the superior performance of *FU-Oracle* in all performance metrics.

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,

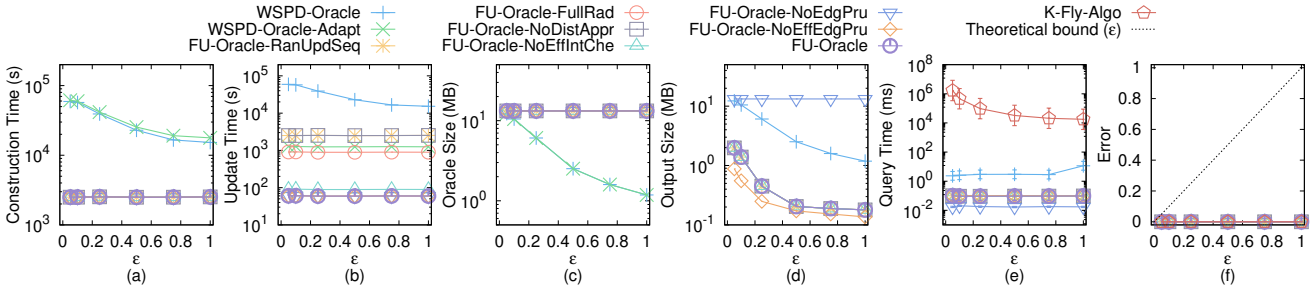

Figure 24: Effect of ϵ on *LH* dataset (fewer POIs) for the P2P path query

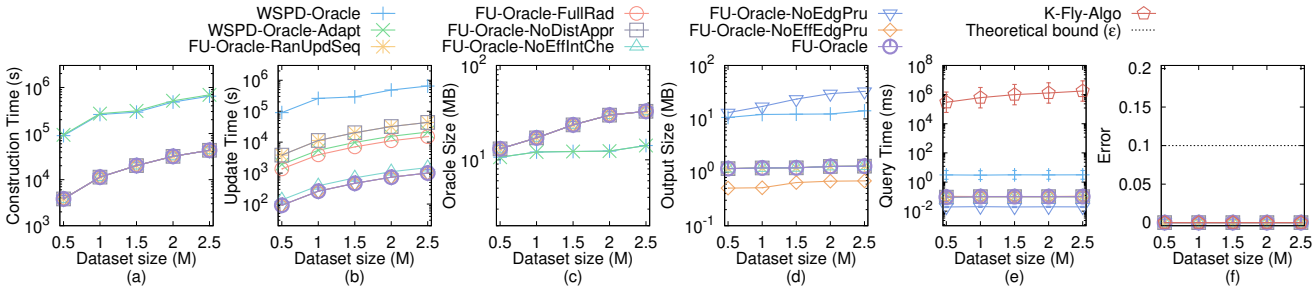

Figure 25: Effect of DS on *LH* dataset (fewer POIs) for the P2P path query

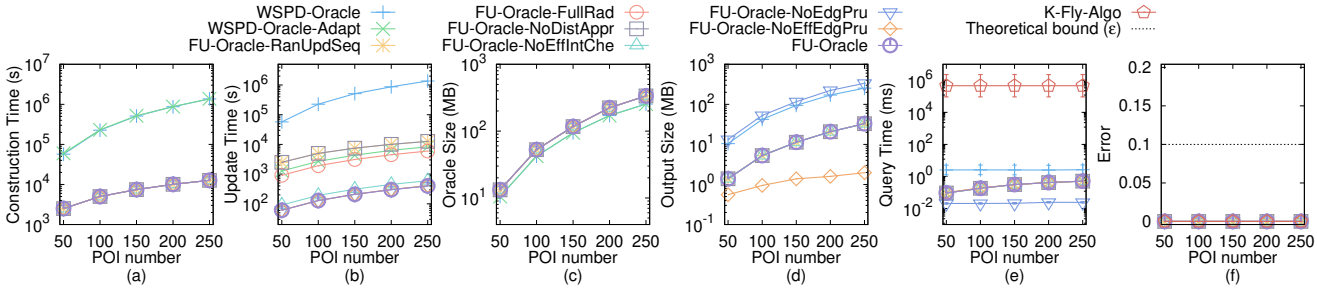

Figure 26: Effect of n on *VS* dataset (fewer POIs) for the P2P path query

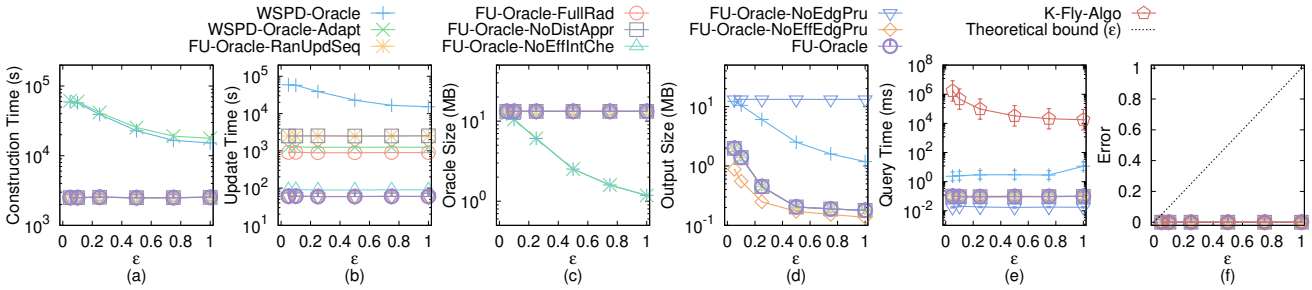

Figure 27: Effect of ϵ on *VS* dataset (fewer POIs) for the P2P path query

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 update time, output size, and shortest path query time of *FU-Oracle* still perform better than other baselines. 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

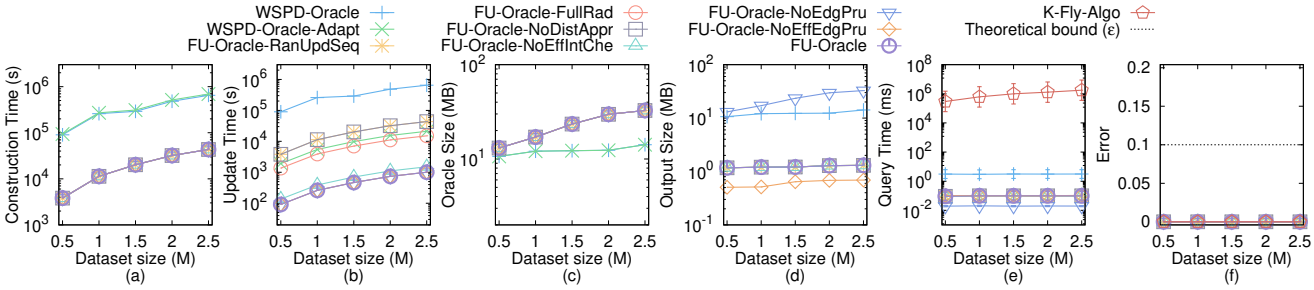

Figure 28: Effect of *DS* on *VS* dataset (fewer POIs) for the P2P path query

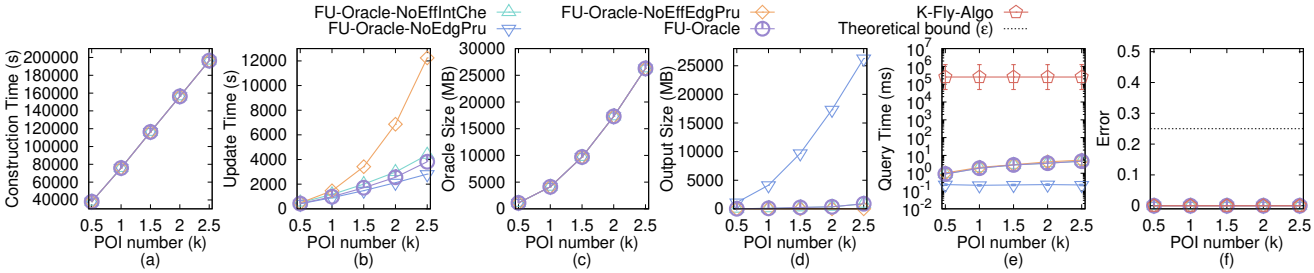

Figure 29: Effect of *n* on *TJ* dataset (more POIs) for the P2P path query

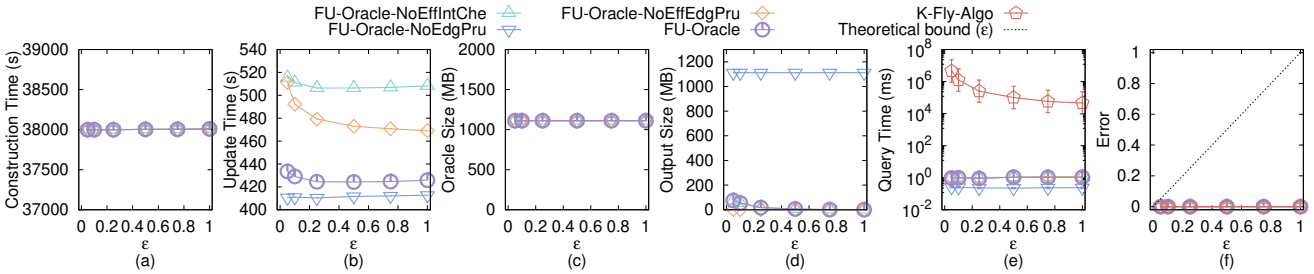

Figure 30: Effect of ϵ on *TJ* dataset (more POIs) for the P2P path query

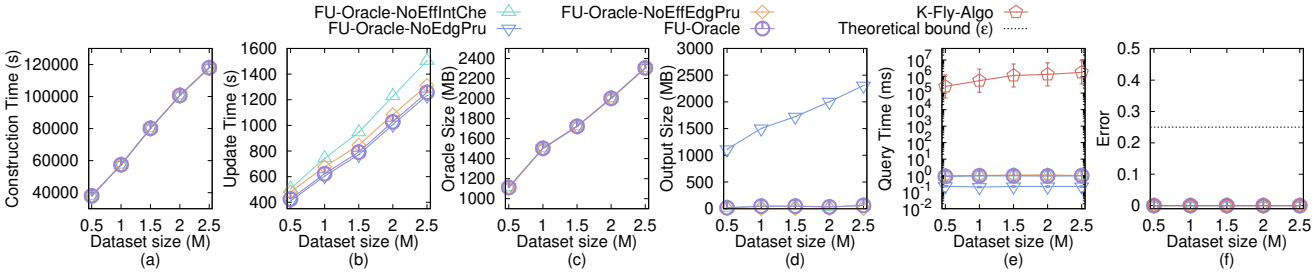

Figure 31: Effect of *DS* on *TJ* dataset (more POIs) for the P2P path query

n at 500. The shortest path query time of *FU-Oracle* is 10^7 to 10^8 smaller than *K-Fly-Algo*.

E.2 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*, and *K-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 ***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-NoEffEdgPru*. Furthermore, the output size for *FU-Oracle-NoEdgPru* is larger than *FU-Oracle*.

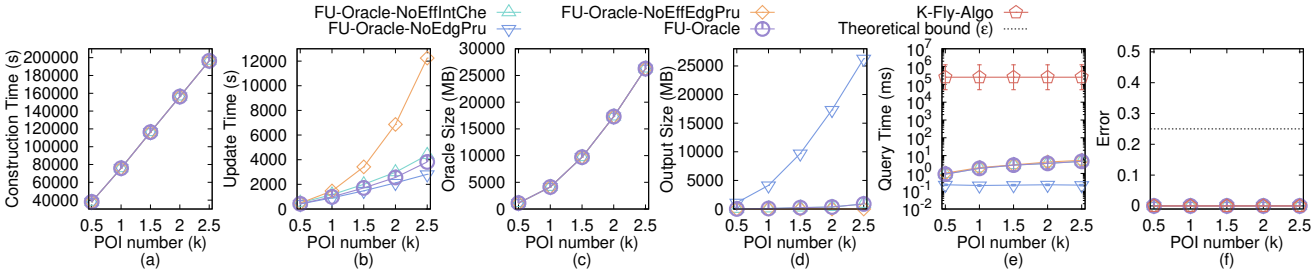

Figure 32: Effect of n on *SC* dataset (more POIs) for the P2P path query

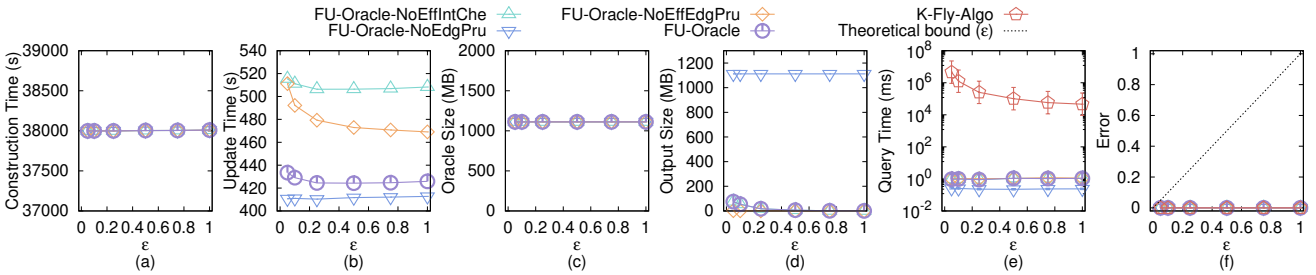

Figure 33: Effect of ϵ on *SC* dataset (more POIs) for the P2P path query

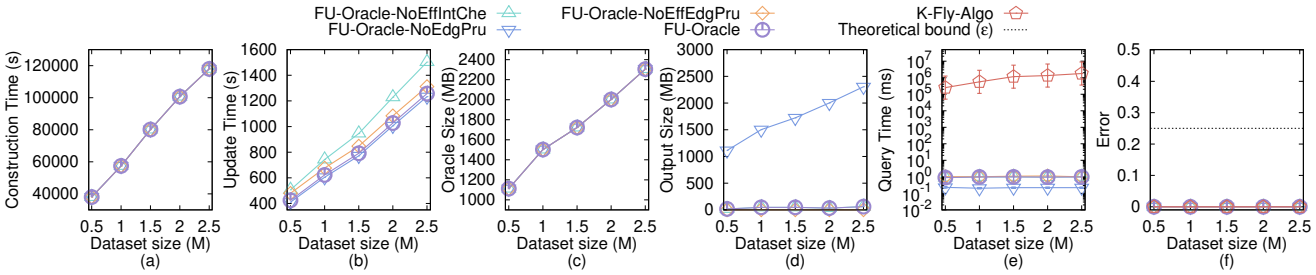

Figure 34: Effect of DS on *SC* dataset (more POIs) for the P2P path query

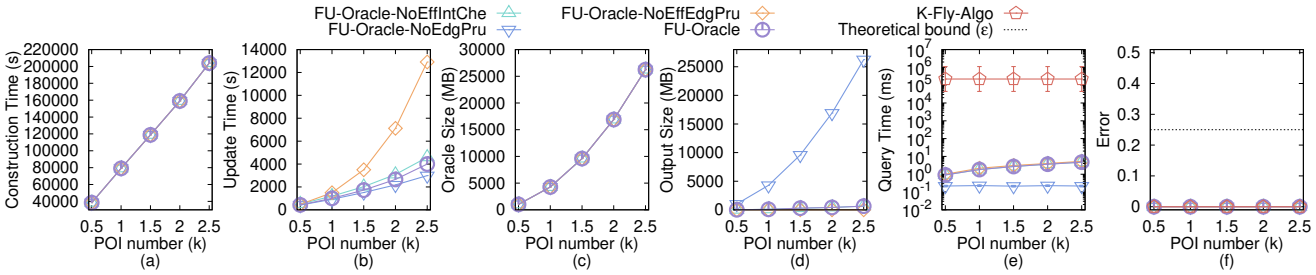

Figure 35: Effect of n on *GI* dataset (more POIs) for the P2P path query

E.3 Experimental Results on the P2P Path Query in the Case $n > N$ 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*, and *K-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 ***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-NoEffEdgPru*. Furthermore, the output size for *FU-Oracle-NoEdgPru* is larger than *FU-Oracle*.

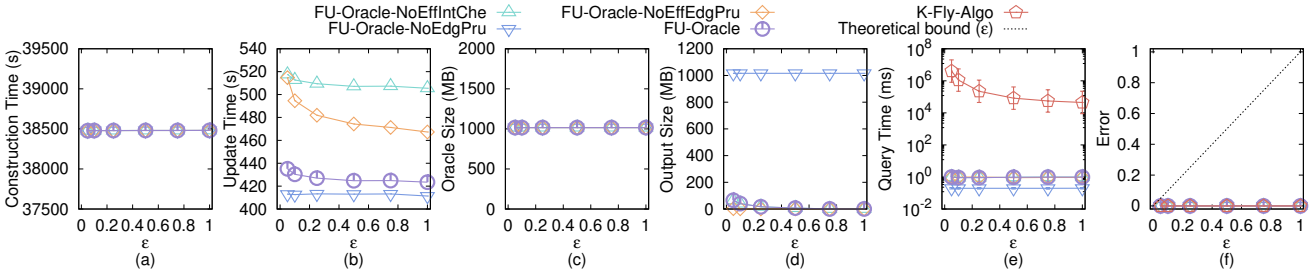

Figure 36: Effect of ϵ on *GI* dataset (more POIs) for the P2P path query

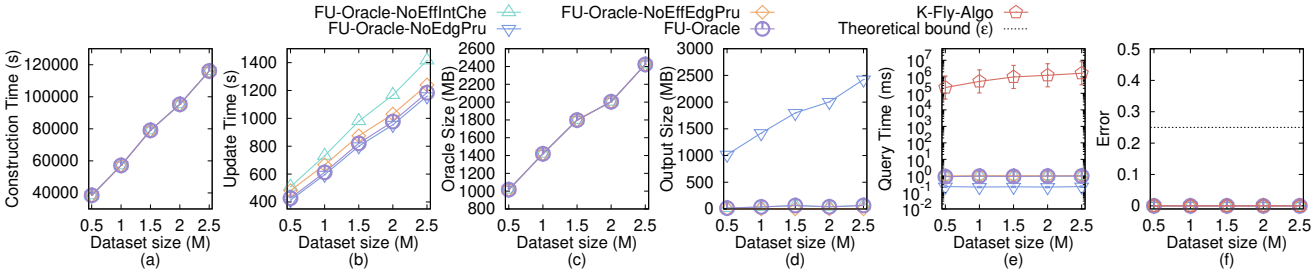

Figure 37: Effect of DS on *GI* dataset (more POIs) for the P2P path query

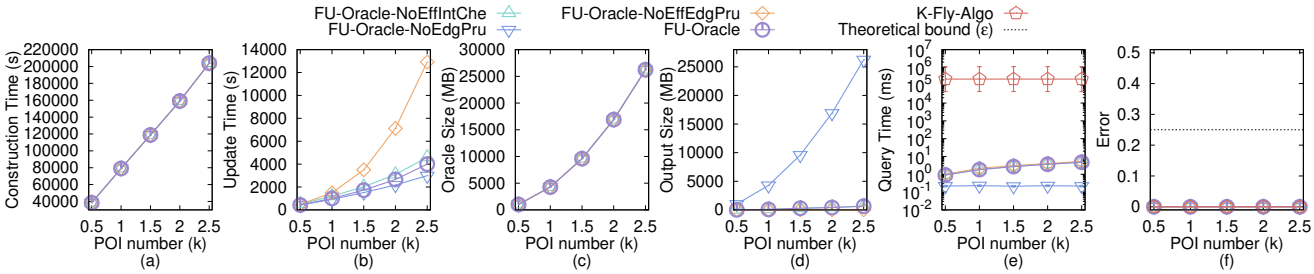

Figure 38: Effect of n on *AU* dataset (more POIs) for the P2P path query

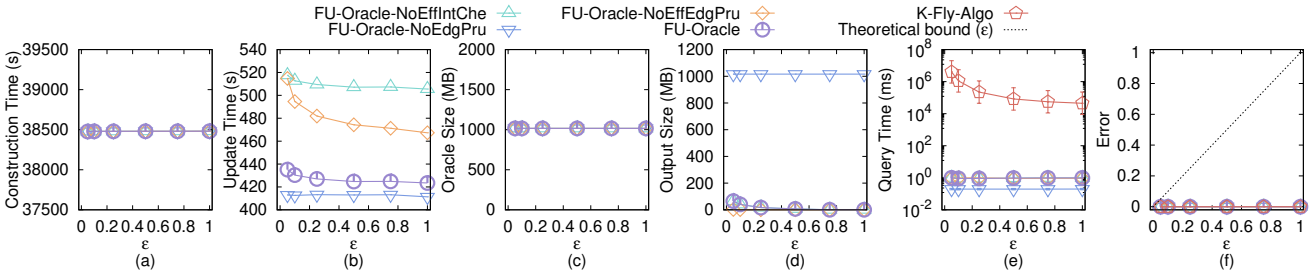

Figure 39: Effect of ϵ on *AU* dataset (more POIs) for the P2P path query

E.4 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, 59, 60]. 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

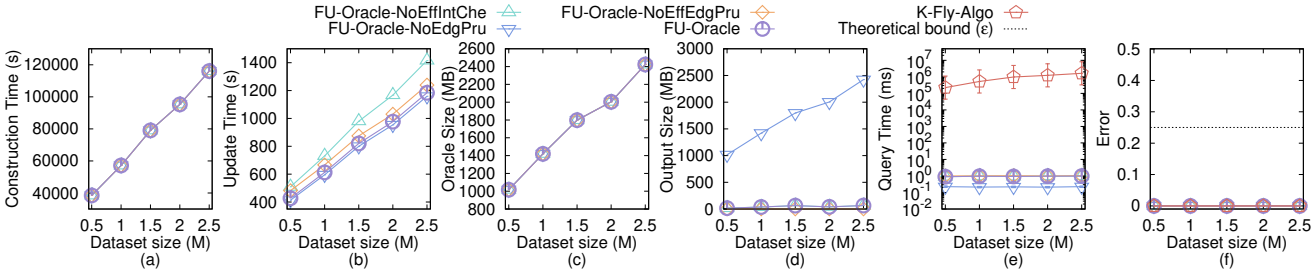

Figure 40: Effect of DS on AU dataset (more POIs) for the P2P path query

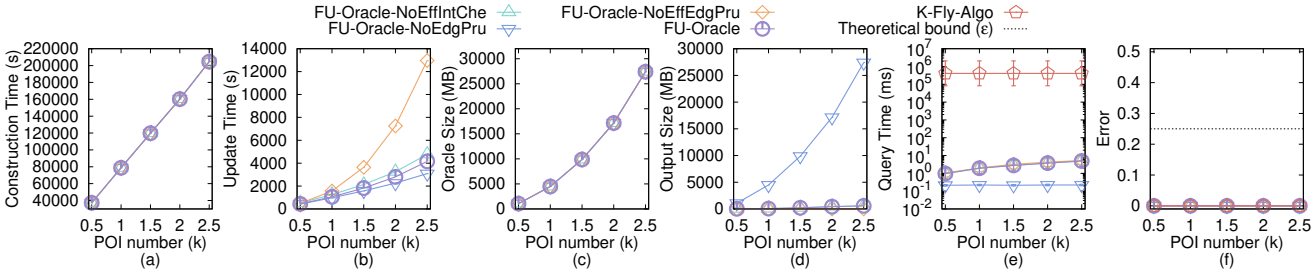

Figure 41: Effect of n on LH dataset (more POIs) for the P2P path query

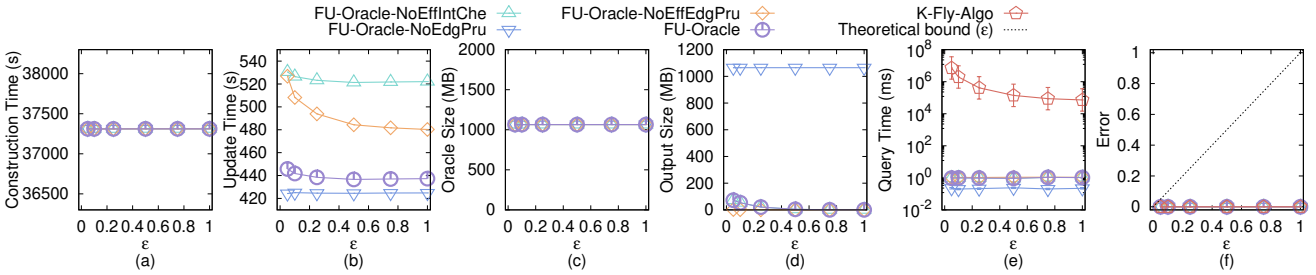

Figure 42: Effect of ϵ on LH dataset (more POIs) for the P2P path query

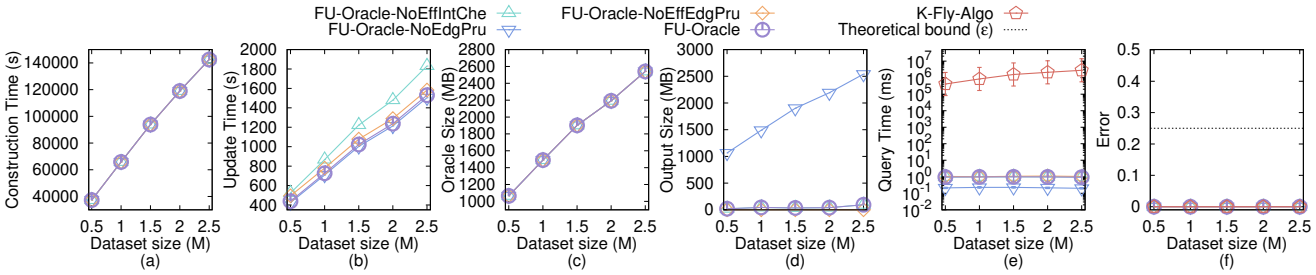

Figure 43: Effect of DS on LH dataset (more POIs) for the P2P path query

and take the projected point (also add edges between neighbours of two points to form edges and faces) as the newly generate T_t.

F 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 $\Pi(u, v|T_{before})$, 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

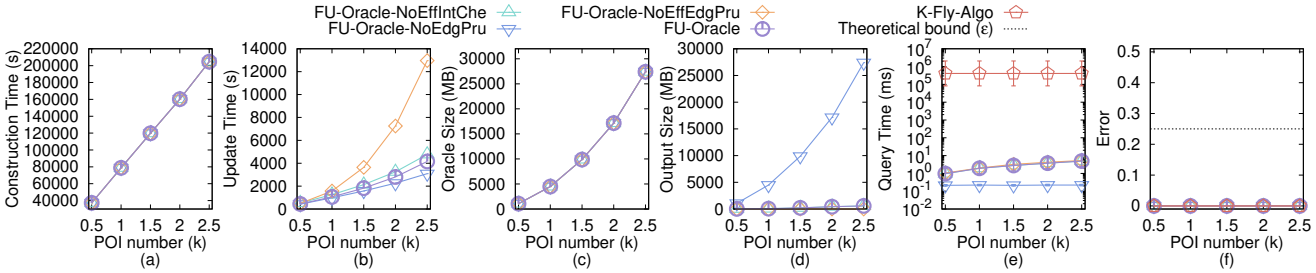

Figure 44: Effect of n on *VS* dataset (more POIs) for the P2P path query

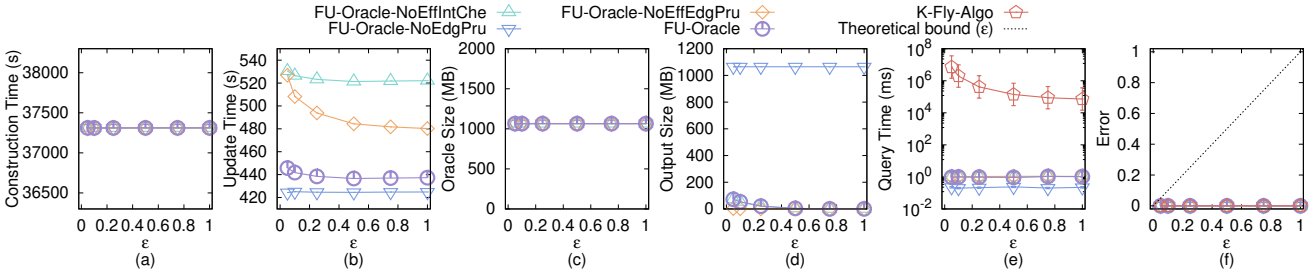

Figure 45: Effect of ϵ on *VS* dataset (more POIs) for the P2P path query

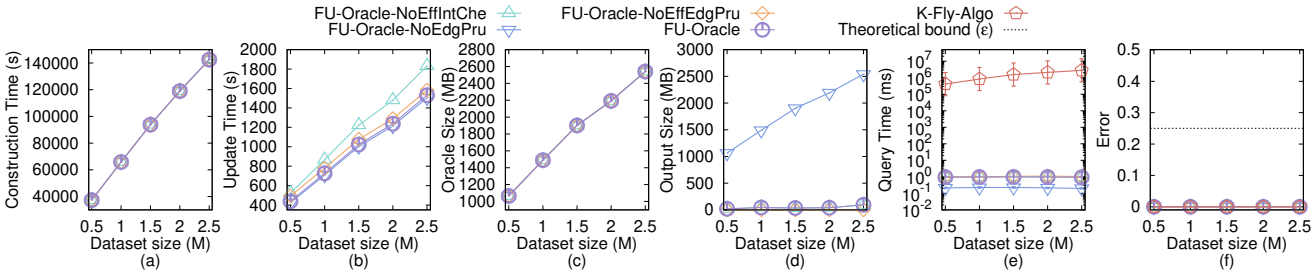

Figure 46: Effect of DS on *VS* dataset (more POIs) for the P2P path query

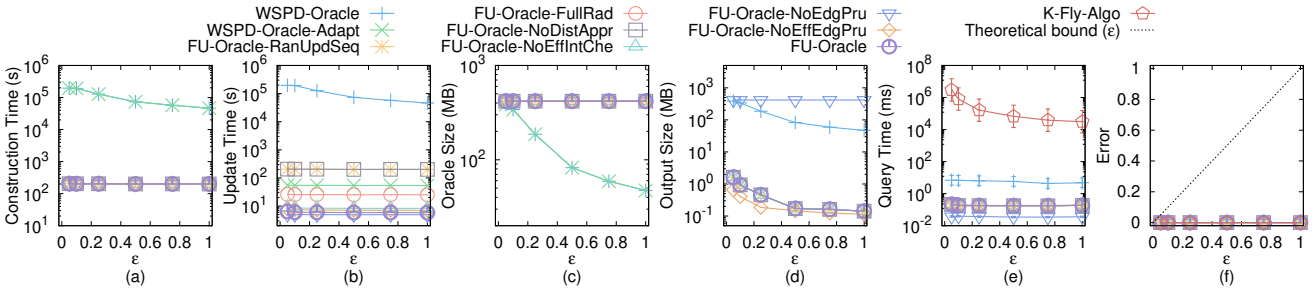

Figure 47: V2V path query on *SC* dataset

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

Proof of Lemma 4.1. According to [59, 60], 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

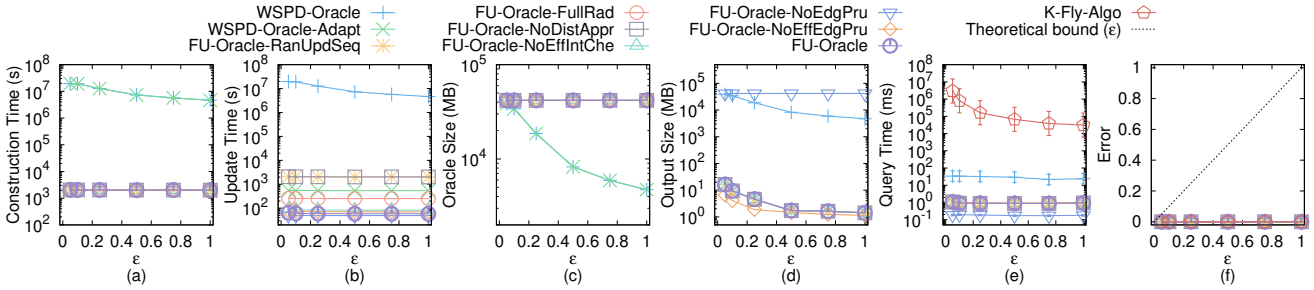

Figure 48: P2P path query in the case $n > N$ and A2A path query on *SC* dataset

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_2. Without loss of generality, suppose that the exact shortest distance from u to u_1 on T_{before} minus $|u_1u_2|$ is smaller than the exact shortest distance from u to u_2 on T_{before} minus $|u_2u_2|$, where $|u_1u_2|$ (resp. $|u_2u_2|$) is the length of the segment between u_1 and u_2 (resp. between u_2 and u_2) 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 $|u_1u_2|$. 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 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 $|u_1u_2|$ 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}.

□

Lemma F.1. *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 F.1. 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 4.2. Firstly, we prove the running time of algorithm *HieGreSpan*.

- In the edge sorting and interval splitting 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.

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 4.4. 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*.

- 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 4.2.

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 4.2, 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 F.2. *The oracle construction time, oracle update time, output size, and shortest path query time of WSPD-Oracle [59, 60] are* $O(\frac{nhN \log^2 N}{\epsilon^{2\beta}})$, $O(\frac{nhN \log^2 N}{\epsilon^{2\beta}})$, $O(\frac{nh}{\epsilon^{2\beta}})$, *and* $O(h^2)$, *respectively. WSPD-Oracle has* $(1-\epsilon)|\Pi(u, v|T)| \leq |\Pi_{WSPD\text{-}Oracle}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{WSPD\text{-}Oracle}(u, v|T)$ *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 [59, 60].

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 [59, 60]. Thus, the shortest path query time of *WSPD-Oracle* should be $O(3h^2) = O(h^2)$. □

Theorem F.3. *The oracle construction time, oracle update time, output size, and shortest path query time of WSPD-Oracle-Adapt [59, 60] are* $O(\frac{nhN \log^2 N}{\epsilon^{2\beta}} + nN \log^2 N)$, $O(\mu_1 N \log^2 N + n \log^2 n)$, $O(n)$, *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* $|\Pi_{WSPD\text{-}Oracle\text{-}Adapt}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{WSPD\text{-}Oracle\text{-}Adapt}(u, v|T)$ *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 $O(\frac{nhN \log^2 N}{\epsilon^{2\beta}})$ for constructing the oracle, 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 $O(\frac{nhN \log^2 N}{\epsilon^{2\beta}} + nN \log^2 N)$.

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 F.4. *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* $|\Pi_{FU\text{-}Oracle\text{-}NoEffEdgPru}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{FU\text{-}Oracle\text{-}RanUpdSeq}(u, v|T)$ *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 F.5. *The oracle construction time, oracle update time, output size, and shortest path query time of FU-Oracle-FullRad are* $O(nN \log^2 N)$, $O(\mu_2 N \log^2 N + n \log^2 n)$, $O(n)$, *and* $O(\log n)$, *respectively, where* μ_2 *is a data-dependent variable, and* $\mu_2 \in [5, 10]$ *in our experiment. FU-Oracle-FullRad satisfies* $|\Pi_{FU\text{-}Oracle\text{-}NoEffEdgPru}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{FU\text{-}Oracle\text{-}FullRad}(u, v|T)$ *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_2 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 F.6. *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* $|\Pi_{FU\text{-}Oracle\text{-}NoEffEdgPru}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{FU\text{-}Oracle\text{-}NoDistAppr}(u, v|T)$ *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_{after} again for determining whether the disk intersects with the updated faces on T_{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_{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 F.7. *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* $|\Pi_{FU\text{-}Oracle\text{-}NoEffEdgPru}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{FU\text{-}Oracle\text{-}NoEffIntChe}(u, v|T)$ *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 F.8. *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* $|\Pi_{FU\text{-}Oracle\text{-}NoEdgPru}(u, v|T)| = |\Pi(u, v|T)|$ *for all pairs of POIs* u *and* v *in* P, *where* $\Pi_{FU\text{-}Oracle\text{-}NoEdgPru}(u, v|T)$ *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_{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_{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 $|\Pi_{FU\text{-}Oracle\text{-}NoEdgPru}(u, v|T)| = |\Pi(u, v|T)|$ 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 F.9. *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* $|\Pi_{FU\text{-}Oracle\text{-}NoEffEdgPru}(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ *for all pairs of POIs u and v in P, where* $\Pi_{FU\text{-}Oracle\text{-}NoEffEdgPru}(u,v|T)$ *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 $V_{FU\text{-}Oracle\text{-}NoEffEdgPru}$ and $E_{FU\text{-}Oracle\text{-}NoEffEdgPru}$ be the set of vertices and edges of *FU-Oracle-NoEffEdgPru*. In algorithm *GreSpan*, consider any edge $e_{FU\text{-}Oracle\text{-}NoEffEdgPru_t}(u,v|T) \in G.E$ between two vertices u and v which is not added to *FU-Oracle-NoEffEdgPru*. Since $e_{FU\text{-}Oracle\text{-}NoEffEdgPru_t}(u,v|T)$ is discarded, it implies that $|\Pi_{FU\text{-}Oracle\text{-}NoEffEdgPru}(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$. 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 $|\Pi_{FU\text{-}Oracle\text{-}NoEffEdgPru}(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ for all pairs of vertices u and v in $V_{FU\text{-}Oracle\text{-}NoEffEdgPru}$. Thus, we have the error bound of *FU-Oracle-NoEffEdgPru*, i.e., *FU-Oracle-NoEffEdgPru* satisfies $|\Pi_{FU\text{-}Oracle\text{-}NoEffEdgPru}(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ 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 F.10. *The shortest path query time of K-Fly-Algo [36] 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* $|\Pi_{K\text{-}Fly\text{-}Algo}(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ *for all pairs of POIs u and v in P, where* $\Pi_{K\text{-}Fly\text{-}Algo}(u,v|T)$ *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 [36]. Note that in Section 4.2 of [36], 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 [36], 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 $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$. □