An Efficiently Updatable Path Oracle for Terrain Surfaces

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

Abstract—The booming of computer graphics technology 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 efficiently updatable $(1+\epsilon)$-approximate shortest path oracle for a set of *Points-Of-Interests* (*POIs*) on an updated terrain surface. This oracle is capable of improved performance in terms of the 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 POIs stored in the oracle. Our oracle can be easily adapted to answering the shortest path query for any points on the updated terrain surface if POIs are not given as input, and also achieve a good performance. Our empirical study shows that when POIs are given (resp. not given) as input, our oracle is up to 88 times, 12 times and 3 times (resp. 15 times, 50 times and 100 times) better than the best-known existing oracle in terms of the oracle update time, output size and shortest path query.

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

1 Introduction

Calculating shortest paths on terrain surfaces is a topic of widespread interest in both industry and academia [1]. In industry, well-known companies and applications, including Metaverse [2] and Google Earth [3], rely on the ability to find shortest paths on terrain surfaces (e.g., in virtual reality or on Earth) to assist users to reach destinations more quickly. In academia, shortest path querying on terrain surfaces also attracts considerable attention [4], [5], [6], [7], [8], [9], [10], [11]. 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*. Figures 1 (a) and (b) show a real map of Valais, Switzerland [12] with an area of 20km × 20km, and Figures 1 (c) and (d) show Valais terrain surface (consisting of vertices, edges and faces).

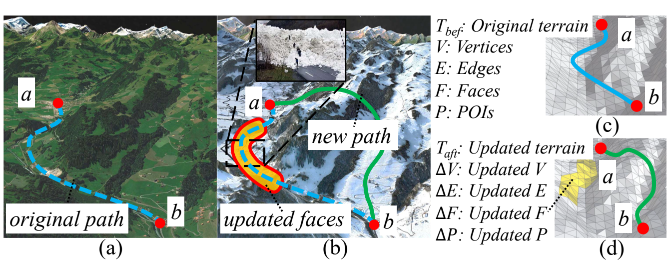

Fig. 1. The real map (a) before and (b) after updates, the terrain surface (c) before and (d) after updates for avalanche in Valais, Switzerland

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

1.1 Motivation

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

(i) **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 Feb 6, 2023 in Turkey and Syria exceeded 40,000 [13], and more than 69,000 died in the 7.9 magnitude earthquake on May 12, 2008 in Sichuan, China [14]. Table 1 lists 6 earthquakes, data from which we use in experiments. A rescue team can save 3 lives every 15 minutes [15], and we expect that the team can arrive at the sites of the quake as early as possible. In practice, (a) satellites or (b) drones can be used to collect the terrain surface after an earthquake, which takes (a) 10s and USD $48.72 [16], and (b) 144s ≈ 2.4 min and USD $100 [17] for a 1km^2 region, respectively, which are time and cost efficient.

(ii) **Avalanche**: Earthquakes may cause avalanches. The 4.1 magnitude earthquake on Oct 24, 2016 in Valais [12] caused an avalanche: Figures 1 (a) and (b) (resp. Figures 1 (c) and (d)) show the original and new shortest paths between a and b on a real map (resp. a terrain surface) before and after terrain surface updates, where a is a village and b is a hotel. We need to efficiently find the shortest rescue paths.

(iii) **Marsquake**: As observed by NASA's InSight lander on May 4, 2022 [18], Mars also experienced a marsquake. In NASA's Mars exploration project [19] (with cost USD 2.5 billion [20]), Mars rovers should find the shortest escape paths quickly and autonomously in regions affected by marsquakes to avoid damage.

2) *P2P and A2A query:* (i) Given a set of *Points-Of-Interest* (*POI*) on a terrain surface, we can calculate the shortest path between *pairs of POIs*, i.e., perform the *POI-to-POI* (*P2P*) *query*. For earthquakes and avalanches, POIs can be villages

TABLE 1
Real earthquake terrain datasets

Name	Magnitude	Date
Tohoku, Japan (TJ) [21]	9.0	Mar 11, 2011
Sichuan, China (SC) [14]	8.0	May 12, 2008
Gujarat, India (GI) [22]	7.6	Jan 26, 2001
Alaska, USA (AU) [23]	7.1	Nov 30, 2018
Leogane, Haiti (LH) [24]	7.0	Jan 12, 2010
Valais, Switzerland (VS) [12]	4.1	Oct 24, 2016

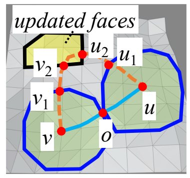

Fig. 2. An unaffected path

waiting for rescuing [25], hospitals and expressway exits. For the Marsquake, POIs can be working stations of Mars rover. (ii) If no POIs are given, we calculate the shortest path between *pairs of arbitrary points* (including the vertices of the terrain surface), i.e., perform the *arbitrary points-to-arbitrary points* (*A2A*) *query*. The A2A query generalizes the P2P query because it allows any points on a terrain surface.

3) ***Oracle:*** Pre-computing shortest paths on a terrain surface using an index, known as an *oracle*, can efficiently reduce the shortest path query time, especially when we need to calculate more than one shortest path with different sources and destinations (where the time taken to pre-compute the oracle is called the *oracle construction time*, the space usage of the output oracle is called the *output size*, and the time taken to return the result is called the *shortest path query time*). We also aim to update the oracle *quickly* when the terrain surface changes (where the time taken to update the oracle is called the *oracle update time*). For example, if we pre-compute shortest paths (among villages and hospitals) using an *oracle* on terrain surfaces prone to earthquakes, and efficiently update the oracle after an earthquake, then we can use it to efficiently return shortest paths with different sources and destinations.

1.2 Challenges

1) ***Inefficiency of on-the-fly algorithms:*** Consider a terrain surface T with N vertices. All existing *exact on-the-fly* shortest path algorithms [26], [27], [28], [29] on a terrain surface are slow when many shortest path queries are involved. The best-known exact algorithm [26] runs in $O(N^2)$ time. Although *approximate* algorithms [6], [7], [8] can reduce the running time, they are still not efficient enough. The best-known approximate algorithm [6] runs in $O((N+N')\log(N+N'))$ time, where N' is the number of Steiner points used for the bound guarantee. Our experiments show that the best-known exact algorithm [26] (resp. approximate algorithm [6]) needs 11,600s $\approx$ 3.2 hours (resp. 8,600s $\approx$ 2.4 hours) to calculate 100 shortest paths on a terrain surface with 0.5M faces.

2) ***Non-existence of oracles on updated terrain surfaces:*** Although existing studies [5], [9], [10] can construct oracles on *static* terrain surfaces, and can then answer P2P or A2A queries efficiently, no existing study can accommodate updated terrains, where the oracle needs to be updated efficiently. When the terrain surface is updated, a straightforward adaptation of the best-known oracle [9], [10] for the P2P query or the best-known oracle [5] for the A2A query is to re-construct the oracles. However, the oracle construction time is $O(nN\log^2 N + c_1 n)$ for the oracle [9], [10], and $O(c_2 N\log^2 N)$ for the oracle [5], where n is the number of POIs and c_1, c_2 are constants depending on T ($c_1 \in [35, 80]$ and $c_2 \in [75, 154]$ on average). In our experiment, the oracle construction time is 35,100s $\approx$ 9.8 hours for the oracle [9], [10] on a terrain surface with 0.5M faces and 250 POIs, and is 35,500s $\approx$ 9.9 hours for the oracle [5] on a terrain surface with 100k faces.

1.3 Path Oracle on Updated Terrain Surfaces

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

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

(i) ***Non-updated terrain shortest path intact* property**: In Figure 2, this property implies that given the light blue path between u and v on T_{bef} (with distance d_1), if the distances from both u and v to the updated faces are large enough (i.e., both larger than $\frac{d_1}{2}$), then the path between u and v on T_{aft} remains the same and does not need to be updated.

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

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

(i) **Earthquake and avalanche**: Given three POIs a, b and c, suppose that a is a damaged village, b and c are unaffected hospitals, the rescue teams need to transport injured citizens to the hospitals, where a is far away from b and c, but b and c are close to each other. We are not interested in using road diggers (on the ruins caused by the earthquake) to dig out two long paths (from a to b and from a to c) and one shortest path (from b to c) for rescuing (since it is time-consuming to dig out a rescue path in the earthquake region [30]). Rather, we only aim to dig out one long path (from a to b) and one short path (from b to c), so we can go from a to c by going via b. That is, given a complete graph G (where the POIs are the vertices of G, and the exact shortest path between POIs are the edges of G), we hope that *UP-Oracle* can efficiently generate a sub-graph of it.

(ii) **Marsquake**: The memory size of NASA's Mars 2020 rover is 256MB [31]. Our experiments show that for a terrain surface with 2.5M faces and 250 POIs, the sub-graph output by *UP-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 in distributed systems for faster network synchronization [32], [33] and in wireless networks for faster signal transmission [34], [35]. The best-known sub-graph generation algorithm [36], [37] runs in $O(n^3 \log n)$ time, which is inefficient. We propose a faster algorithm called *Hierarchy Greedy Spanner* (*HieGreSpan*) that considers several vertices of the complete graph in one group. Our experiments show that when $n = 500$, our algorithm takes 24s, while the best-known algorithm [36], [37] takes 101s.

1.4 Contributions and Organization

Our major contributions are as follows.

(1) We propose the first oracle *UP-Oracle* for solving the updated terrain surfaces problem. It achieves a short oracle update time by satisfying the novel non-updated terrain shortest path intact property, and by utilizing the useful information on T_{bef} (the pairwise P2P exact shortest paths on T_{bef}). We also propose four additional novel techniques to further reduce the oracle update time. An ablation study shows that if any one of the techniques is not satisfied, *UP-Oracle*'s oracle update time will increase very substantially. It is worth mentioning that designing an oracle on an updated terrain surface with a small oracle update time is challenging: there are no existing studies on this, and only limited information about T_{bef} can be re-used. We also develop an efficient algorithm called *HieGreSpan* to reduce the output size. *UP-Oracle* can be easily adapted for answering A2A queries on an updated terrain surface.

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

(3) *UP-Oracle* performs much better than the best-known oracle [9], [10] for the P2P query and the best-known oracle [38] for the A2A query in terms of the oracle update time, output size and shortest path query time. (i) For the P2P query on a terrain surface with 0.5M faces and 250 POIs, the oracle update time and output size of *UP-Oracle* are 400s $\approx$ 6.7 min and 22MB, while the values are 35,100s $\approx$ 9.8 hours and 250MB for the best-known oracle [9], [10], and (ii) the shortest path query time for computing 100 shortest paths with different sources and destinations is 0.1s for *UP-Oracle*, while the time is 8,600s $\approx$ 2.4 hours for the best-known approximate on-the-fly algorithm [6] and 0.3s for the best-known oracle [9], [10] for the P2P query. (iii) For the A2A query on a terrain surface with 20k faces, the oracle update time, output size, and shortest path query time for computing 100 shortest paths of *UP-Oracle* are 480s $\approx$ 7 min, 3MB and 0.05s, while the values are 7,100s $\approx$ 2 hours, 150MB and 5s for the best-known oracle [5].

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

2 PROBLEM DEFINITION

2.1 Notations and Definitions

1) *Terrain surfaces and POIs:* Consider a terrain surface T_{bef} represented as a *Triangulated Irregular Network* (*TIN*) [8], [9], [10], [39]. Let V, E and F be the set of vertices, edges and faces of T_{bef}, respectively. Let L_{max} be the length of the longest edge in E. Let N be the number of vertices. Each vertex $v \in V$ has three coordinates, x_v, y_v and z_v. If the positions of vertices in V are updated, we obtain a new terrain surface, T_{aft}. There is no need to consider the case when N changes. This is because both the original and updated terrain surface have the same 2D grid with $\overline{x} \times \overline{y} = N$ vertices [8], [9], [10]. Figures 1 (c) and (d) show an example of T_{bef} and T_{aft}, respectively. In the P2P query, let P be a set of POIs on T_{bef} and n be the number of POIs. There is no need to consider when n changes, or when $n > N$. When a POI is added, we create an oracle that answers the A2A query, which implies we consider all possible POIs to be added. When a POI is removed, we continue to use the original oracle. When $n > N$, we can still create an oracle that answers the A2A query.

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

3) *Updated and non-updated components:* Given T_{bef}, T_{aft} and P, a set of (i) *updated vertices*, (ii) *updated edges*, (iii) *updated faces* and (iv) *updated POIs* of T_{bef} and T_{aft}, denoted by (i) ΔV, (ii) ΔE, (iii) ΔF and (iv) ΔP, is defined to be a set of (i) vertices $\Delta V = \{v_1, v_2, \ldots, v_{|\Delta V|}\}$, where v_i is a vertex in V with coordinate values differing between T_{bef} and T_{aft}, and $|\Delta V|$ is the number of vertices in ΔV, (ii) edges $\Delta E = \{e_1, e_2, \ldots, e_{|\Delta E|}\}$, where e_i is an edge in E with at least one of its two vertices' coordinate values differing between T_{bef} and T_{aft}, and $|\Delta E|$ is the number of edges in ΔE, (iii) faces $\Delta F = \{f_1, f_2, \ldots, f_{|\Delta F|}\}$, where f_i is a face in F with at least one of its three vertices' coordinate values differing between T_{bef} and T_{aft}, and $|\Delta F|$ is the number of faces in ΔF, and (iv) POIs $\Delta P = \{p_1, p_2, \ldots, p_{|\Delta P|}\}$, where p_i is a POI in P with coordinate values differing between T_{bef} and T_{aft}, and $|\Delta P|$ is the number of POIs in ΔP. It is easy to obtain ΔV, ΔE, ΔF and ΔP by comparing T_{bef}, T_{aft} and P. In Figure 1 (d), the yellow area is ΔF based on T_{bef} and T_{aft}. The vertices, edges and POIs in ΔF are ΔV, ΔE and ΔP. In addition, there is no need to consider the case with two or more *disjoint* non-empty sets of updated faces. If this happens, we can create a larger set of faces that contains these disjoint sets. Thus, the set of updated faces that we consider is connected [40] as shown in Figure 1 (d).

2.2 Updated terrain surfaces problem

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

3 RELATED WORK

3.1 On-the-fly Algorithms on Terrain Surfaces

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

1) *Exact algorithms:* The running time of the exact algorithms [26], [27], [28], [29] are $O(N^2)$, $O(N\log^2 N)$, $O(N^2\log N)$ and $O(N^2\log N)$, respectively. They are *Single-Source All-Destination* (*SSAD*) algorithms, i.e., given a source, they can calculate the shortest path from it to all other vertices *simultaneously*. The best-known exact algorithm *Chen and Han on-the-Fly Algorithm* (*CH-Fly-Algo*) [26] uses a sequence tree for the shortest path query.

2) *Approximate algorithms:* Approximate algorithms [6], [7], [8] aim at reducing the running time. The best-known approximate algorithm *Kaul on-the-Fly Algorithm* (*K-Fly-Algo*) [6] places Steiner points on edges in E, and then constructs a graph using these points and V to calculate a $(1+\epsilon)$-approximate shortest path on a terrain surface. 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.

Drawbacks of the on-the-fly algorithms: All these algorithms are not efficient when multiple shortest path queries are involved. Our experiments show that *CH-Fly-Algo* and *K-Fly-Algo* needs 11,600s $\approx$ 3.2 hours and 8,600s $\approx$ 2.4 hours to compute 100 paths with different sources and destinations on a terrain surface with 0.5M faces.

3.2 Oracle-based Algorithms on Terrain Surfaces

Although *Well-Separated Pair Decomposition Oracle* (*WSPD-Oracle*) [9], [10] (resp. the *Efficiently ARbitrary pints-to-arbitrary points Oracle* (*EAR-Oracle*) [5]) is regarded as the best-known oracle for answering *approximate* P2P (resp. A2A) queries on a terrain surface, no existing oracle can accommodate updated terrain surfaces, where the oracle needs to be updated efficiently.

1) *WSPD-Oracle:* It uses *compressed partition tree*, algorithm *SSAD*, and *well-separated node pair sets* to index the $(1+\epsilon)$-approximate pairwise P2P shortest paths. Its oracle construction time, output size and shortest path query time is $O(\frac{nN\log^2 N}{\epsilon^{2\beta}}+\frac{nh}{\epsilon^{2\beta}}+nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ and $O(h^2)$, respectively, where h is the height of the compressed partition tree and β is the largest capacity dimension [41] ($\beta\in[1.5,2]$ in practice [9], [10]).

Drawback of *WSPD-Oracle*: It *only supports the static terrain surface* and does not address how to update the oracle on an *updated* terrain surface. If we use the straightforward adaptation, i.e., re-construct *WSPD-Oracle* from scratch when the terrain surface is updated. In the P2P query, the oracle update time for *WSPD-Oracle* (resp. *UP-Oracle*) is 35,100s $\approx$ 9.8 hours (resp. 400s $\approx$ 6.7 min) on a terrain dataset with 0.5M faces and 250 POIs.

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

Drawbacks of *WSPD-Oracle-Adapt*: (i) *Not fully utilizing the non-updated terrain shortest path intact property during update*: Since *WSPD-Oracle-Adapt* only stores the pairwise P2P *approximate* shortest paths on T_{bef}, the oracle update time remains large. In the P2P query, the oracle update time for *WSPD-Oracle-Adapt* (resp. *UP-Oracle*) is 8,400s $\approx$ 2.4 hours (resp. 400s $\approx$ 6.7 min) on a terrain dataset with 0.5M faces and 250 POIs. (ii) *Additional information needed during construction*: *WSPD-Oracle-Adapt* needs to calculate the shortest distance between each POI and vertex on T_{bef} when the oracle is constructed to utilize the property in point (i), which increases its oracle construction time, but *UP-Oracle* can calculate this information and the pairwise P2P exact shortest paths on T_{bef} *simultaneously*.

3) *EAR-Oracle:* It uses the same idea as *WSPD-Oracle*, i.e., well-separated pair decomposition. Their differences are that *EAR-Oracle* adapts *WSPD-Oracle* from the P2P query to the A2A query by using Steiner points on the terrain faces and using *highway nodes* (i.e., not POIs in *WSPD-Oracle*) for well-separated pair decomposition. Its oracle construction time, output size and shortest path query time is $O(\lambda\xi mN\log^2(mN)+\frac{nN\log^2 N}{\epsilon^{2\beta}}+\frac{Nh}{\epsilon^{2\beta}}+Nh\log N)$, $O(\frac{\lambda mN}{\xi}+\frac{Nh}{\epsilon^{2\beta}})$ and $O(\lambda\xi\log(\lambda\xi))$, respectively, where λ is the number of highway nodes in one square, ξ is the square root of the number of boxes, and m is the number of Steiner points per face.

Drawback of *EAR-Oracle*: Similar to *WSPD-Oracle*, *EAR-Oracle only supports the static terrain surface*. In the A2A query, the oracle update time for *EAR-Oracle* (resp. *UP-Oracle*) is 7,100s $\approx$ 2 hours (resp. 480s $\approx$ 7 min) on a terrain surface with 20k faces.

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

Drawback of *EAR-Oracle-Adapt*: Similar to *WSPD-Oracle-Adapt*, *EAR-Oracle-Adapt* does not *fully utilize the non-updated terrain shortest path intact property during update*. In the A2A query, the oracle update time for *EAR-Oracle-Adapt* (resp. *UP-Oracle*) is 4,300s $\approx$ 1.2 hours (resp. 480s $\approx$ 7 min) on a terrain surface with 20k faces.

3.3 Sub-graph Generation Algorithm

Algorithm *Greedy Spanner* (*GreSpan*) [36], [37] that runs in $O(n^3\log n)$ time is the best-known sub-graph generation algorithm. But, it is very slow since it does not consider using any simpler structure to approximate the sub-graph when performing Dijkstra's algorithm [42] on the sub-graph.

4 METHODOLOGY

We give an overview and present two main ideas of *UP-Oracle* in Sections 4.1, 4.2 and 4.3. We then cover the implementation details of three phases in Sections 4.4, 4.5 and 4.6. We present the adaption to the A2A query in Section 4.7 and offer a theoretical analysis of *UP-Oracle* in Section 4.8.

4.1 Overview of *UP-Oracle*

We first use an example to illustrate *UP-Oracle*. In Figures 3 (a) to (c), we have T_{bef} and P, and we construct *UP-Oracle*. In Figures 3 (d) to (f), we have T_{aft}, and we efficiently update *UP-Oracle*. In Figure 3 (g), we answer the shortest path between a pair of POIs using *UP-Oracle*. Note that in Figure 3 (e), there is no need to re-calculate all shortest

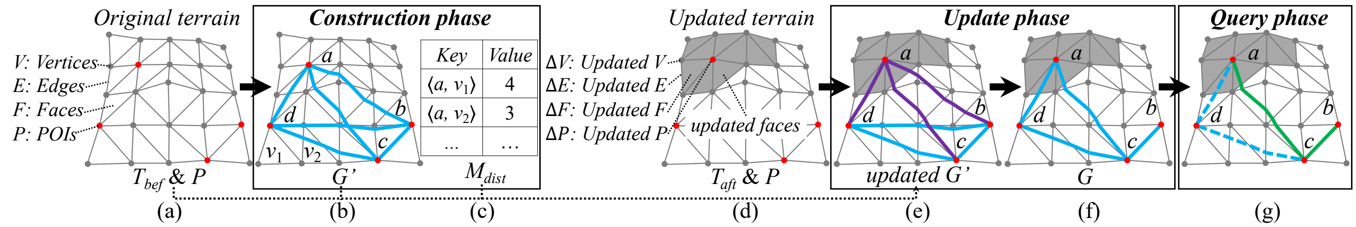

Fig. 3. Framework overview

paths on T_{aft}, and we use Figure 4 for better illustration. In Figures 4 (a) to (c), we use algorithm *SSAD* with a, b and h as sources to update shortest paths on T_{aft}. But, in Figures 4 (d) and (e), due to the *non-updated terrain shortest path intact* property and the stored pairwise P2P exact shortest paths on T_{bef}, we do not need to use algorithm *SSAD* with f, e, c, d and g as sources for path updating. Next, we introduce the four components and three phases of *UP-Oracle*.

1) *Four components of UP-Oracle:* (i) *Temporary complete graph*, (ii) *POI-to-vertex distance mapping table*, (iii) *UP-Oracle output graph* and (iv) *hierarchy graph*.

(i) **The temporary complete graph G**: This is a complete graph that stores the pairwise P2P exact shortest paths. Let $G.V$ and $G.E$ be the sets of vertices and edges of G (where each POI in P is denoted by a vertex in $G.V$). Then an exact shortest path $\Pi(u,v|T)$ between a pair of POIs u and v on T is denoted by a weighted edge $e(u,v|T)$ in $G.E$, and the distance of this path $|\Pi(u,v|T)|$ is denoted by the weight of the edge $|e(u,v|T)|$, where T can be T_{bef} or T_{aft}. Figure 3 (b) shows a complete graph G with 4 vertices and 6 edges. The light blue line between a and c is the exact shortest path $\Pi(a,c|T_{bef})$ on T_{bef}, and also an edge $e(a,c|T_{bef})$ in G.

(ii) **POI-to-vertex distance mapping table M_{dist}**: This is a *hash table* [43] that stores the exact shortest distance from each POI in P to each vertex in V on T_{bef} (calculated when *UP-Oracle* is constructed), used for reducing the oracle udpate time of *UP-Oracle*. A vertex u and a POI v is stored as a key $\langle u,v\rangle$, and their corresponding exact shortest distance $|\Pi(u,v|T_{bef})|$ is stored as a value. M_{dist} needs linear space in terms of the number of distances to be stored. Given $\langle u,v\rangle$, M_{dist} can return $|\Pi(u,v|T_{bef})|$ in $O(1)$ time. In Figure 3 (c), the exact shortest distance between POI a and vertex v_1 is 4.

(iii) **The *UP-Oracle* output graph G'**: This is a graph used for answering pairwise P2P $(1+\epsilon)$-approximate shortest paths. Note 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 $e'(u,v|T_{aft})$ be an edge between two vertices u and v in $G'.E$, and let $|e'(u,v|T_{aft})|$ be the weight of this edge. Given two vertices s and t in $G'.V$, we define $\Pi_{G'}(s,t|T_{aft}) = (v_1, v_2, \ldots, v_l)$ to be a shortest path of *UP-Oracle*, such that the weighted length $\sum_{i=1}^{l-1} |e'(v_i, v_{i+1}|T_{aft})|$ is the minimum, where $v_1 = s$, $v_l = t$, l is the number of vertices in this path, and for each $i \in [1, l-1]$, $(v_i, v_{i+1}) \in G'.E$. Figure 3 (f) shows a G' with 4 POIs, the light blue line between a and c is the exact shortest path $\Pi(a,c|T_{aft})$ on T_{aft} (and also an edge $e'(a,c|T_{aft})$ in G'), and the shortest path $\Pi_{G'}(a,b|T_{aft}) = (a,c,b)$ (from POI a to b via c), which consists of edges $e'(a,c|T_{aft})$ and $e'(c,b|T_{aft})$.

(iv) **The hierarchy graph H**: This is a graph similar to G' (but has a simpler structure than G') used for efficiently generating a G' using G. It 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 $|\Pi_{G'}(u,v|T_{aft})| \leq r$, where $v \in Q_{G'}$. A set of groups $Q^1_{G'}, Q^2_{G'}, \ldots, Q^k_{G'}$ is a group cover of G' if every vertex in $G'.V$ belongs to at least one group, where k is the number of groups. H can form a set of *groups* by regarding several vertices in G' that are close to each other as one vertex. As a result, the shortest distance between u and v on H is an approximation of the shortest distance between u and v on G'. Similar to G', let $H.E$ be the set of edges of H. Given a group $Q^i_{G'}$, we define *intra-edges* to be a set of edges connecting the group center of $Q^i_{G'}$ to all other vertices in $Q^i_{G'}$, and we define *inter-edges* to be a set of edges connecting two group centers. We can use a group cover and add these two types of edges to construct H. Let $e_H(u,v|T_{aft})$ be an (intra- or inter-) edge between two vertices u and v in $H.E$, and let $|e_H(u,v|T_{aft})|$ be the weight of this edge. Given two group centers s and t, we define $\Pi_H(s,t|T_{aft}) = (v_1, v_2, \ldots, v_{l'})$ to be a shortest path of H, such that the weighted length of the inter-edges $\sum_{i=1}^{l'-1} |e'(v_i, v_{i+1}|T_{aft})|$ is minimum, where $v_1 = s$, $v_{l'} = t$, l' is the number of vertices in this path, and for each $i \in [1, l'-1]$, v_i is a group center of H. Figures 5 (a) and (b) show an example of G' and the corresponding H of G', there are three groups with centers c, e and g for H, the light blue lines are intra-edges and the purple lines are inter-edges, and the shortest path of inter-edges $\Pi_H(c,g|T_{aft})$ is (c,e,g) (from POI c to g via e).

2) *Three phases of UP-Oracle:* (i) *Construction phase*, (ii) *update phase* and (iii) *query phase* (see Figure 3).

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

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

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

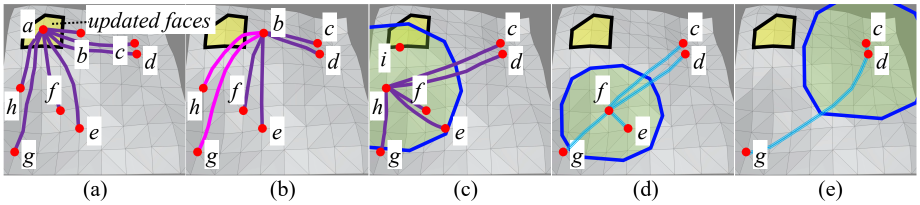

Fig. 4. In the update phase when (a) updating $\Pi(a)$, (b) updating $\Pi(b)$, (c) updating $\Pi(f)$, (d) no need for updating $\Pi(c)$, and (e) no need for updating $\Pi(e)$

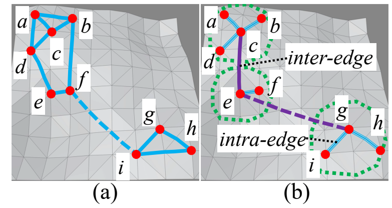

Fig. 5. (a) *UP-Oracle* output graph G' and (b) hierarchy graph H of G'

(iii) **Query phase**: Given two query POIs and G', we answer the path between these two POIs on T_{aft} using G'.

4.2 Reasons for Achieving Short Oracle Update Time

A major contribution of this paper is the short oracle update time of *UP-Oracle*, which is enabled by our design in the pairwise P2P exact shortest path update step of the update phase. In this step, the short oracle update time for *UP-Oracle* is enabled by (1) the *non-updated terrain shortest path intact* property, and (2) the stored pairwise P2P exact shortest paths on T_{bef}. We define the concept of a *disk* first. Given a point p on T_{bef} and a constant $r > 0$, let $D(p, r)$ be a disk centered at p with radius r on T_{bef}, which consists of all points on T_{bef} whose exact shortest distance to p is no more than r. Given a face f_i, if a point q exists on f_i such that the shortest distance between p and q is no more than r, then disk $D(p, r)$ is said to be *intersect with* face f_i. Figure 2 shows two disks centered at u and v with radius $|\Pi(u, o|T_{bef})|$ and $|\Pi(v, o|T_{bef})|$, respectively, and they do not intersect with any updated faces.

Firstly, we give the *non-updated terrain shortest path intact* property in Property 1.

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

Proof Sketch. We show by contradiction that the two paths cannot be different. The detailed proofs of proof sketches in the paper appear in the appendix. □

Secondly, we illustrate the necessity of storing the pairwise P2P exact shortest paths (i.e., G) on T_{bef}. Let $U(A)$ be the *Update ratio* of an oracle A, which is defined to be the number of POIs in P that we need to perform algorithm *SSAD* as source (for path updating on T_{aft}) divided by the total number of POIs. In Figures 4 (a) to (c), we use algorithm *SSAD* with a, b and h as sources to update shortest paths on T_{aft} for *UP-Oracle* and *WSPD-Oracle-Adapt*. In Figure 4 (d), *UP-Oracle* (resp. *WSPD-Oracle-Adapt*) calculates an *exact* (resp. *approximate*) path between c and f on T_{bef} with path distance $|\Pi(c, f|T_{bef}|$ (resp. d_2), where $|\Pi(c, f|T_{bef})| < d_2$. It may happen that the distance from f to ΔF is smaller than $\frac{d_2}{2}$, but larger than $\frac{|\Pi(c,f|T_{bef})|}{2}$. That is, the disk $D(f, \frac{|\Pi(c,f|T_{bef})|}{2})$ does not intersect (resp. $D(f, \frac{|d_2|}{2})$ intersects) with ΔF, and *UP-Oracle* does not need to (resp. *WSPD-Oracle-Adapt* needs to) use algorithm *SSAD* with f as source to update shortest paths on T_{aft}. The case also happens for the paths between c and e. In Figure 4 (e), the case also happens for the path between g and each POI in $\{c, d\}$. Thus, for *UP-Oracle* (resp. *WSPD-Oracle-Adapt*), we need to perform algorithm *SSAD* with 3 POIs a, b, h (resp. 7 POIs a, b, c, d, e, f, g) as a source for path updating on T_{aft}, and there is a total of 8 POIs, so $U(\textit{UP-Oracle}) = \frac{3}{8}$ (resp. $U(\textit{WSPD-Oracle-Adapt}) = \frac{7}{8}$). The oracle update time of *WSPD-Oracle-Adapt* is 2.4 times larger than that of *UP-Oracle*. Given an oracle A, a higher $U(A)$ means that the oracle update time of A is larger. Lemma 1 shows the necessity of storing G.

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

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

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

1) ***Novel path update sequence:*** We propose a novel path update sequence before utilizing Property 1, to minimize the oracle update time. In Figure 4 (a), we need to update shortest paths between a and two POIs in $\{e, g\}$ on T_{aft}. By using algorithm *SSAD*, when we update the paths with a as the source POI, we can update these two paths simultaneously (since e and g are far away from ΔF, we can avoid using algorithm *SSAD* to update the paths with e and g as the source POIs according to Property 1). But, if we first update the paths with e as the source POI, we still need to update the paths with g as the source POI, which increases the oracle update time. We say "a point (either a vertex or a POI) is in ΔF" if it is on a face in ΔF, and "a path passes ΔF" if this path intersects with ΔF. Then, the path update sequences that result in the smallest oracle update time are updating the paths: (i) connecting to the POIs in ΔF, (ii) passing ΔF, and (iii) connecting to the POIs near ΔF. After we update all the paths belonging to one type, we process to the next type. In Figures 4 (a) to (c), (i) a is in ΔF, (ii) one of b's exact shortest path $\Pi(b, h|T_{bef})$ pass ΔF, and (iii) h is near ΔF, so we use a, b and h as source point in algorithm *SSAD* and update the paths on T_{aft} in sequence.

2) ***Novel disk radius selection strategy:*** We design a novel disk radius selection strategy used in Property 1 (i.e., *half* of the distance between a pair of POIs as the disk radius) when updating the paths connecting to the POIs near ΔF to minimize the likelihood of re-calculating shortest paths on T_{aft}. A naive approach is to create two disks centered at u and v with the *full* distance between u and v as radius. It increases the likelihood of re-calculating this path on T_{aft} and increases the oracle update time.

3) ***Novel distance approximation approach:*** We propose a novel distance approximation approach used in Property 1, to avoid performing the expensive shortest path query algorithm on T_{aft}, for determining whether the disk intersects with ΔF on T_{aft} (i.e., whether the minimum distances from the disk center to any point in ΔF on T_{aft} is smaller than the disk radius), by using the POI-to-vertex distance information stored in M_{dist}. In Figure 4 (c), we do not want to perform the shortest path query algorithm between h and i on T_{aft} again, for determining whether the disk centered at h intersects with the updated faces, where i is a point belonging to the updated faces that is the closest point to h (among other points belonging to the updated faces). Instead, in Lemma 2, we show that we can use M_{dist} to obtain the lower bound of the minimum distances from the disk center (i.e., a POI) to any point in ΔF on T_{aft}, to approximate the shortest distance on T_{aft} in $O(1)$ time.

Lemma 2. *The minimum distance from a POI u to any point in ΔF on T_{aft} is no less than $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{bef})| - L_{max}$.*

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

If the lower bound is larger than the disk radius, then the minimum distances from this radius center to any point in ΔF must be larger than the disk radius, i.e., there is no need to update the corresponding paths. In Figure 4 (c), the exact shortest distance between h and i can be calculated in $O(1)$ time. We can calculate G and M_{dist} *simultaneously* when *UP-Oracle* is constructed (by using algorithm *SSAD* with each POI as a source point for n times).

4) ***Novel disk and updated face intersection check approach:*** We design a novel disk and updated face intersection check approach, to minimize the number of intersection checks for each shortest path on T_{aft} when updating the paths connecting to the POIs near ΔF (i.e., when checking whether (i) the disk and (ii) updated face intersects or not in Property 1). In Figure 4 (c), when checking whether we need to re-calculate the paths between h and POI X on T_{aft}, a naive approach is creating 5 disks centered at h (and another 5 disks centered at POI X) with half of the shortest distance between h and POI X as radius, and check whether these 10 disks intersect with ΔF, where $X \in \{c, d, e, f, g\}$. Since there are total $O(n^2)$ paths, it needs to create $O(n^2)$ disks. But, we just need to create *one* disk centered at h with half of the longest distance of the paths between h and each POI in $\{c, d, e, f, g\}$ as radius, and check whether this disk intersects with ΔF. Since there are total $O(n)$ POIs, we just need to create $O(n)$ disks. Specifically, for each POI not in ΔF and not the endpoint of the paths pass ΔF, we sort them from near to far based on their minimum distance to any vertex in ΔV on T_{bef}. We then determine whether there is a need to update shortest paths using Lemma 3.

Lemma 3. *If the disk centered at u with half of the longest distance of all non-updated paths adjacent to u as radius, intersects with ΔF, we use algorithm SSAD to update all the non-updated paths adjacent to u. Otherwise, there is no need to update shortest paths adjacent to u.*

Proof Sketch. If the disk with the largest radius intersects with ΔF, we just need to update the paths and there is no need to check other disks. If the disk with the largest radius and with the center closest to ΔF does not intersect with ΔF, then other disks cannot intersect with ΔF, so there is no need to update the paths. □

(i) In Figure 4 (c), the sorted POIs are h, f, e, d, c, g. We create one disk $D(h, \frac{|\Pi(c,h|T_{bef})|}{2})$, since it *intersects* with ΔF, we use algorithm *SSAD* to update all shortest paths adjacent to h that have not been updated. We do not need to create 10 disks, i.e., 5 disks $D(h, \frac{|\Pi(X,h|T_{bef})|}{2})$ and 5 disks $D(X, \frac{|\Pi(X,h|T_{bef})|}{2})$, where $X \in \{c, d, e, f, g\}$.

(ii) In Figure 4 (d), the sorted POIs are f, e, d, c, g. We create one disk $D(f, \frac{|\Pi(c,f|T_{bef})|}{2})$, since it *does not intersect* with ΔF, there is no need to update shortest paths adjacent to f. We do not need to create 8 disks, i.e., 4 disks $D(f, \frac{|\Pi(X,f|T_{bef})|}{2})$ and 4 disks $D(X, \frac{|\Pi(X,f|T_{bef})|}{2})$, where $X \in \{c, d, e, g\}$.

4.3 Reason for Efficiently Achieving Small Output Size

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

We first illustrate algorithm *GreSpan*. Given a complete graph G, it first sorts the edge in G based on the weight of each edge from minimum to maximum, and initializes a sub-graph G' to be empty. Then, for each sorted edge $e(u, v) \in G$, if $|e(u, v)|$ is longer than $(1+\epsilon)$ times the distance between u and v on G' (calculated using Dijkstra's algorithm on G'), then $e(u, v)$ is added into G' (in Figure 5 (a), we add the dashed light blue line $e(f, i)$ into G'). It iterates until all the paths have been processed, and returns G' as output.

We then illustrate algorithm *HieGreSpan*. The main difference between *HieGreSpan* and *GreSpan* is the usage of H. To construct H, we first sort the edges in $G.E$ in increasing order, and then divide them into $\log n$ intervals, where each interval contains edges with weights in $(\frac{2^{i-1}D}{n}, \frac{2^i D}{n}]$ 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 $\delta \frac{2^i D}{n}$, where $\delta \in (0, \frac{1}{2})$ is a small constant depending on ϵ), such that the shortest distance between the vertices in the same group is very small (and can be regarded as 0) compared with the current processing interval edges' weights. Thus, when checking whether $|e(u, v)|$ is longer than $(1 + \epsilon)$ times the distance between u and v on G', we use Dijkstra's algorithm between the group centers of u and v on H, to approximate the distance between u and v on G'. This takes $O(1)$ time on H, but takes $O(n \log n)$ time on G' in algorithm *GreSpan*. When we need to process the next interval of edges with larger weight, we update H such that the radius of each group of vertices will also increase, and H is a valid approximated graph of G'. In Figure 5 (a), we add the dashed light blue line between f and i into G'. In Figure 5 (b), f belongs to e and i belongs to g, so we add the dashed purple edge between e and g into

H. Our experiments show that when $n = 500$, algorithm *HieGreSpan* needs 24s, but algorithm *GreSpan* needs 101s. Due to algorithm *HieGreSpan*, the output size of *UP-Oracle* is only 22MB on a terrain surface with 0.5M faces and 250 POIs, but the value is 260MB for *WSPD-Oracle*.

4.4 Implementation Details of the Construction Phase

We give the implementation details of the construction phase. Given T_{bef} and P, by regarding each POI $p_i \in P$ as a source point, we use algorithm *SSAD* to (1) calculate the exact shortest paths between p_i and other POIs in P on T_{bef}, and then store them in G, (2) calculate the exact shortest distance between p_i and each vertex in V on T_{bef}, and then store them in M_{dist}. Figure 3 (a) shows T_{bef} and P. In Figures 3 (b) and (c), we first take a as a source point, and use algorithm *SSAD* to calculate the exact shortest paths between a and $\{b, c, d\}$ (the light blue lines), and the exact shortest distance between a and all vertices. Next, we take b as a source point, and calculate the exact shortest paths between b and $\{c, d\}$, and the exact shortest distance between b and all vertices.

4.5 Implementation Details of the Update Phase

We present the implementation details of the update phase for (1) the pairwise P2P exact shortest path update step and (2) the sub-graph generation step. Figure 3 (d) shows T_{aft}, Figures 3 (e) and (f) show the output of the two steps, and Figures 4 and 5 show the detailed examples of the two steps.

1) *Pairwise P2P exact shortest path update:* Given two POIs u and v in P, after we have updated an exact shortest path $\Pi(u, v|T_{bef})$ (stored in G) between u and v on T_{bef}, the updated exact shortest path between u and v on T_{aft} is denoted as $\Pi(u, v|T_{aft})$. Let $P_{remain} = \{p'_1, p'_2, \dots, p'_{|P_{remain}|}\}$ be a set of remaining POIs in P on T_{aft} that we have not processed, where $|P_{remain}|$ is the number of POIs in P_{remain}. P_{remain} is initialized to be P. In each update iteration, whenever we have processed a POI, we remove this POI from P_{remain}. In Figures 4 (a) and (b), $P_{remain} = \{b, c, d, e, f, g, h\}$ and $P_{remain} = \{c, d, e, f, g, h\}$, respectively. This set is different from ΔP (which stores a set of POIs with different coordinate values between T_{bef} and T_{aft}). For ΔP, we use algorithm *SSAD* to update shortest paths on T_{aft} with the POIs in ΔP as sources. For P_{remain}, when processing a POI u, among all the POIs in P_{remain}, we use the POI v in P_{remain} such that $|\Pi(u, v|T_{bef})|$ is the longest, and we use $|\Pi(u, v|T_{bef})|$ as the disk radius in our novel disk and updated face intersection check approach as mentioned in Section 4.2. Given a POI $u \in P_{remain}$, we let $\Pi(u) = \{\Pi(u, v_1|T_{bef}), \Pi(u, v_2|T_{bef}), \dots, \Pi(u, v_{|\Pi(u)|}|T_{bef})\}$ be a set of the exact shortest paths stored in G on T_{bef} with u as an endpoint and $v_i \in P_{remain} \setminus \{u\}$, $i \in \{1, l_u\}$ 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, where $|\Pi(u)|$ is the number of paths in $\Pi(u)$. In Figures 4 (a) to (c), the purple and pink lines denote $\Pi(a)$, $\Pi(b)$ and $\Pi(h)$, respectively.

Detail and example: Algorithm 1 and 2 show this step. For Algorithm 1, in Figure 4 (a), we compute *Update* $(a, T_{aft}, G, P_{remain} = \{b, c, d, e, f, g, h\})$. The following illustrates Algorithm 2 with an example.

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

Input: a POI u, T_{aft}, temporary complete graph G and P_{remain}
Output: updated G and updated P_{remain}

```
use u as source point in algorithm SSAD to calculate Π(u, v|T_aft) for
    each POI v ∈ P_remain simultaneously
for each POI v ∈ P_remain do
    G.E ← G.E − {Π(u, v|T_bef)} ∪ {Π(u, v|T_aft)}
    Π(v) ← Π(v) − {Π(u, v|T_bef)}
P_remain ← P_remain − {u}
return updated G and P_remain
```

Algorithm 2 *PairwiseP2PUpdate* $(G, P, M_{dist}, \Delta F, \Delta P)$

Input: G, a set of POIs P, M_{dist}, ΔF and ΔP
Output: updated G

```
1:  P_remain ← P
2:  for each POI u ∈ P_remain do
3:      Π(u) ← all the exact shortest paths in G with u as an endpoint
4:  for each POI u ∈ P_remain do
5:      if u ∈ ΔP then
6:          Update (u, T_aft, G, P_remain)
7:  for each POI u ∈ P_remain do
8:      if u ∉ ΔP but there exists an exact shortest path in Π(u) passes
        ΔF then
9:          Update (u, T_aft, G, P_remain)
10: sort each POI in P_remain from near to far based on their minimum
    distance to any vertex in ΔV on T_bef using M_dist
11: for each sorted POI u ∈ P_remain do
12:     v ← a POI in P_remain such that Π(u, v|T_bef) has the longest
        distance among all Π(u)
13:     if disk D(u, |Π(u, v|T_bef)/2|) intersects with ΔF then
14:         Update (u, T_aft, G, P_remain)
15:     else
16:         P_remain ← P_remain − {u}
17: return updated G
```

(i) *Path update for POI in updated face*: Lines 4-6. In Figure 4 (a), $a \in \Delta P$, we update the paths in purple on T_{aft}.

(ii) *Path update for path passing updated face*: Lines 7-9. In Figure 4 (b), $b \notin \Delta P$ but one exact shortest path $\Pi(b, h|T_{bef}) \in \Pi(u)$ passes ΔF (the black circle), so we update the paths in purple and pink on T_{aft}.

(iii) *Path update for POI near updated face*: Lines 10-16. In lines 13-14 and Figure 4 (c), the sorted POIs are h, f, e, d, c, g, the path with the longest distance is $\Pi(c, h|T_{bef})$. Since the disk with blue circle intersects with ΔF, we update the paths in purple on T_{aft}. In lines 15-16 and Figure 4 (d), the sorted POIs are f, e, d, c, g, the paths with the longest distance is $\Pi(c, f|T_{bef})$. Since the disk with blue circle does not intersect with ΔF, we do not need to update the paths.

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

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

(i) *Edge sort, interval split, and G' initialization*: Lines 2-6.

(ii) *G' maintenance*: Lines 7-25, and Figures 5 (a) and (b). In lines 9-15, *group construction and H intra-edge insertion*: Based on $G'.V$, we have three groups with group center c, e and g in H. We add light blue edges $e'(a, c|T_{aft})$, ..., $e'(g, i|T_{aft})$ in H. In lines 16-19, *H first type inter-edge insertion*: We add purple edge $e(c, e|T_{aft})$ in H. In lines 20-22, *H edge examine*: We need to examine edge $e(f, i|T_{aft})$ on G', the corresponding shortest path on H is $\Pi_H(e, g|T_{aft})$ and $|\Pi_H(e, g|T_{aft})| = \infty > (1 + \epsilon)|e(f, i|T_{aft})|$. In lines 24, *$G'$ edge insertion*: We add $e(f, i|T_{aft})$ into G'. In line 25, *H second*

Algorithm 3 *HieGreSpan* (G, ϵ)

Input: temporary complete graph G and error parameter ϵ
Output: *UP-Oracle* output graph G' (a sub-graph of G)
$D \leftarrow$ the weight of the longest edge in $G.E$
for each edge $e(u, v|T_{aft}) \in G.E$ **do**
  sort edge weights in increasing order
  create intervals $I_0 = (0, \frac{D}{N}]$, $I_i = (\frac{2^{i-1}D}{n}, \frac{2^i D}{n}]$ for $i \in [1, \log n]$
  $G.E^i \leftarrow$ sorted edges of $G.E$ with weight in I_i
$G'.E \leftarrow G.E^0$
for $i \leftarrow 1$ to $\log n$ **do**
  $H.E \leftarrow \emptyset$
  for each $u_j \in G.V$ that has not been visited **do**
    perform Dijkstra's algorithm on G', such that the algorithm never visits vertices further than $\delta \frac{2^i D}{n}$ from u_j
    create a group $Q^j_{G'} \leftarrow \{u_j\}$ with group center u_j, $u_j \leftarrow$ *visited*
    for each $v \in G.V$ such that $|\Pi_{G'}(u_j, v|T_{aft})| \leq \delta \frac{2^i D}{n}$ **do**
      $Q^j_{G'} \leftarrow \{v\}$, $v \leftarrow$ *visited*
      H intra-edges $\leftarrow H.E \cup \{e_H(u_j, v|T_{aft})\}$, where $|e_H(u_j, v|T_{aft})| = |\Pi_{G'}(u_j, v|T_{aft})|$
    $j \leftarrow j + 1$
  for each group center u_j **do**
    perform Dijkstra's algorithm on G', such that the algorithm never visits vertices further than $\frac{2^i D}{n} + 2\delta \frac{2^i D}{n}$ from u_j
    H inter-edges $\leftarrow H.E \cup \{e_H(u_j, u|T_{aft})\}$, where u is other group centers and $|e_H(u_j, u|T_{aft})| = |\Pi_{G'}(u_j, u|T_{aft})|$
    $j \leftarrow j + 1$
  for each edge $e(u, v|T_{aft}) \in G.E^i$ **do**
    $w \leftarrow$ group center of u, $x \leftarrow$ group center of v
    $\Pi_H(w, x|T_{aft}) \leftarrow$ the shortest path between w and x calculated using Dijkstra's algorithm on H
    if $|\Pi_H(w, x|T_{aft})| > (1 + \epsilon)|e(u, v|T_{aft})|$ **then**
      $G'.E \leftarrow G'.E \cup \{e(u, v|T_{aft})\}$
      H inter-edge $\leftarrow H.E \cup \{e_H(w, x|T_{aft})\}$, where $|e_H(w, x|T_{aft})| = |e_H(w, u|T_{aft})| + |e(u, v|T_{aft})| + |e_H(v, x|T_{aft})|$
return G'

type inter-edge insertion: We add $e_H(e, g|T_{aft})$ with weight $|e_H(e, f|T_{aft})| + |e(e, g|T_{aft})| + |e_H(g, i|T_{aft})|$ into H.

4.6 Implementation Details of the Query Phase

We then give the implementation details of the query phase. Given G', and two query POIs s and t in P (i.e., two query vertices s and t in $G'.V$), we use Dijkstra's algorithm [42] to find the shortest path between s and t on G', i.e., $\Pi_{G'}(s, t|T_{aft})$, which is a $(1 + \epsilon)$-approximate path of $\Pi(s, t|T_{aft})$. In Figure 3 (g), given two query POIs a and b, we calculate $\Pi_{G'}(a, b|T_{aft})$, which consists of two green lines, i.e., $\Pi(a, c|T_{aft})$ and $\Pi(c, b|T_{aft})$.

4.7 Adaption to the A2A Query

We can adapt *UP-Oracle* to answer the A2A query. We first place Steiner points on T_{bef} using the method in study [44], then use these them as input (not the POIs) to construct *UP-Oracle*. When T_{bef} changes to T_{aft}, the positions of Steiner points (based on T_{aft}) also change, we update *UP-Oracle* using these Steiner points accordingly. For the query phase, given arbitrary point s (resp. t) on face f_s (resp. f_t), we let $\mathcal{S}(s)$ (resp. $\mathcal{S}(t)$) be a set of Steiner points on f_s (resp. f_t) and its adjacent faces [44]. Then, we return $\Pi_{G'}(s, t|T_{aft})$ such that $|\Pi_{G'}(s, t|T_{aft})| = \min_{p \in \mathcal{S}(s), q \in \mathcal{S}(t)}[|\Pi(s, p|T_{aft})| + |\Pi_{G'}(p, q|T_{aft})| + |\Pi(q, t|T_{aft})|]$, where $|\Pi(s, p|T_{aft})|$ and $|\Pi(q, t|T_{aft})|$ can be calculated in $O(1)$ time using algorithm *SSAD* and $|\Pi_{G'}(p, q|T_{aft})|$ is distance of the path between p and q returned by *UP-Oracle*.

4.8 Theoretical Analysis

Theorems 1 and 2 show the analysis of algorithm *HieGreSpan* and *UP-Oracle*, respectively.

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

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

Theorem 2. *The oracle construction time, oracle update time, output size and shortest path query time of UP-Oracle for (1) the P2P query are $O(nN \log^2 N)$, $O(N \log^2 N + n \log^2 n)$, $O(n)$ and $O(\log n)$, and (2) the A2A 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 $O(\frac{1}{\sin\theta\cdot\epsilon} \log(\frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon}))$. UP-Oracle satisfies $|\Pi_{G'}(s, t|T)| \leq (1 + \epsilon)|\Pi(s, t|T)|$ for all pairs of (1) POIs s and t in P for the P2P query, and (2) arbitrary points s and t on T for the A2A query.*

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

We then discuss the A2A query. Since there are total $\frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon}$ Steiner points [44], we use this value to substitute n in the P2P query to obtain the *oracle construction time*, *oracle update time* and *output size* for the A2A query. The *shortest path query time* is due to the $O(\log(\frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon}))$ query time for any pair of Steiner points p and q, and the total $|\mathcal{S}(s)| \cdot |\mathcal{S}(t)| = \frac{1}{\sin\theta\cdot\epsilon}$ combinations of having p and q for arbitrary pair of points s and t [44]. The *error bound* is due to the error bound of *UP-Oracle* in the P2P query. □

5 EMPIRICAL STUDY

5.1 Experimental Setup

We conduct experiments on a Linux machine with a 2.20 GHz CPU and 512GB memory. All algorithms are imple-

TABLE 2
Comparison of algorithms

Algorithm	Oracle construction time	Oracle update time	Output size	Shortest path query time
Oracle-based algorithm				
WSPD-Oracle [9], [10]	Large	Large	Large	Small
WSPD-Oracle-Adapt [9], [10]	Large	Large	Small	Small
EAR-Oracle [5]	Large	Large	Large	Medium
EAR-Oracle-Adapt [5]	Large	Large	Small	Small
***UP-Oracle* (ours)**	**Small**	**Small**	**Small**	**Small**
On-the-fly algorithm				
CH-Fly-Algo [26]	N/A	N/A	N/A	Large
K-Fly-Algo [6]	N/A	N/A	N/A	Large

mented in C++. Our experimental setup generally follows the setups in the literature [6], [7], [8], [9], [10].

1) ***Datasets:*** We conduct our experiment on 30 real before and after earthquake terrain datasets listed in Table 1 with 0.5M faces. We obtain the earthquake terrain satellite maps with a 5km $\times$ 5km region from Google Earth [3] with a resolution of 10m [8], [9], [10], [39], and then we use Blender [45] to generate the terrain model. To study the scalability, we follow an existing multi-resolution terrain dataset generation procedure [8], [9], [10] to obtain different resolutions of these datasets with 1M, 1.5M, 2M, 2.5M faces. This procedure appears in the appendix. We extract 500 POIs using OpenStreetMap [9], [10].

2) ***Algorithms:*** We include the the best-known exact on-the-fly algorithm *CH-Fly-Algo* [26], the best-known approximate on-the-fly algorithm *K-Fly-Algo* [6], the best-known oracle *WSPD-Oracle* [9], [10] for the P2P query, its adaption *WSPD-Oracle-Adapt*, the best-known oracle *EAR-Oracle* [5] for the A2A query, and its adaption *EAR-Oracle-Adapt* as baselines. In Table 2, we compare these algorithms. The comparisons of all algorithms (with big-O notations) can be found in the appendix.

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

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

5.2 Experimental Results

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

1) ***Ablation study for the P2P query:*** We consider 6 variations of *UP-Oracle*, i.e., (i) we use a random path update sequence, instead of using our novel path update sequence, (ii) we use the full shortest distance of a shortest path as the disk radius, instead of using our novel disk radius selection strategy, (iii) we do not store the POI-to-vertex distance information and re-calculate the shortest path on T_{aft} for determining whether the disk intersects with ΔF, instead of using our novel distance approximation approach, (iv) we create two disks for each path when checking whether we need to re-calculate the shortest path between a pair of POIs, instead of using our novel disk and updated face intersection check approach, (v) we remove the sub-graph generation step, i.e., algorithm *HieGreSpan* in the update phase and use a hash table to store the pairwise P2P exact shortest paths on T_{aft} in G, and (vi) we use algorithm *GreSpan* [36], [37], instead of using algorithm *HieGreSpan* in the sub-graph generation step of the update phase. We use *UP-Oracle-$\boldsymbol{X}$* where $\boldsymbol{X} \in \{$*RanUpdSeq, FullRad, NoDistAppr, NoEffIntChe, NoEdgPru, NoEffEdgPru*$\}$ to denote these variations. The first four oracles correspond to the four techniques in Section 4.2. The last two oracles correspond to the idea covered in Section 4.3.

In Figure 6 (resp. Figure 7), we tested the 5 values of n in {50, 100, 150, 200, 250} on *TJ* (resp. {500, 1000, 1500, 2000, 2500} on *SC*) dataset while fixing ϵ at 0.1 and *DS* at 0.5M (resp. ϵ to 0.25 and *DS* to 0.5M) for the ablation study involving 6 variations (resp. the last 3 variations, since the first 3 variations have excessive oracle update times with 500 POIs) and *UP-Oracle*. The oracle update time for *UP-Oracle-$\boldsymbol{X}$*, where $\boldsymbol{X} \in \{$*RanUpdSeq, FullRad, NoDistAppr, NoEffIntChe, NoEffEdgPru*$\}$ exceeds that of *UP-Oracle* due to the four techniques from Section 4.2 and the use of algorithm *HieGreSpan* from Section 4.3. Although the oracle update time and the shortest path query time of *UP-Oracle-NoEdgPru* is slightly smaller than that of *UP-Oracle*, the output size for *UP-Oracle-NoEdgPru* is 10^4 times due to the usage of algorithm *HieGreSpan*. Thus, *UP-Oracle* is the best oracle among the variations.

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

Effect of ϵ: In Figure 8, we tested the 6 values of ϵ in {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *GI* dataset with fewer POIs while fixing n at 50 and *DS* at 0.5M. Although all algorithms have errors close to 0%, *UP-Oracle* offers superior performance over *WSPD-Oracle*, *WSPD-Oracle-Adapt*, *EAR-Oracle*, *EAR-Oracle-Adapt* and *CH-Fly-Algo* in terms of the oracle construction time, oracle update time, output size and shortest path query time due to the *non-updated terrain shortest path intact* property, the stored pairwise P2P exact shortest paths on T_{bef}, and the usage of algorithm *HieGreSpan* in *UP-Oracle*. Although the oracle size of *UP-Oracle* is slightly larger than that of *WSPD-Oracle* and *WSPD-Oracle-Adapt*, the oracle update time of *UP-Oracle* is 88 times and 21 times smaller than that of *WSPD-Oracle* and *WSPD-Oracle-Adapt*. Varying ϵ has (i) no impact on the oracle construction time of *UP-Oracle* since it is independent of ϵ, (ii) a small impact on the oracle update time of *UP-Oracle*, since when n is small, the pairwise P2P exact shortest path update step dominates the sub-graph generation step, and the former step is independent of ϵ, and (iii) a small impact on the oracle construction time and oracle update time of other oracles since their early termination criteria of using algo-

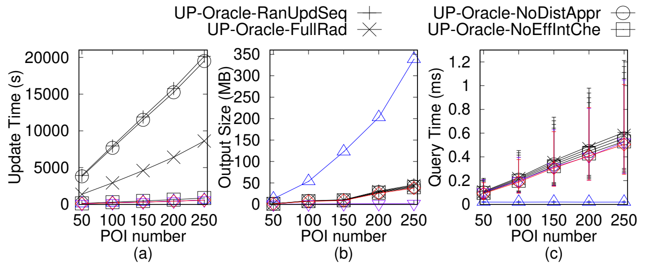

Fig. 6. Ablation study on *TJ* dataset with fewer POIs for the P2P query

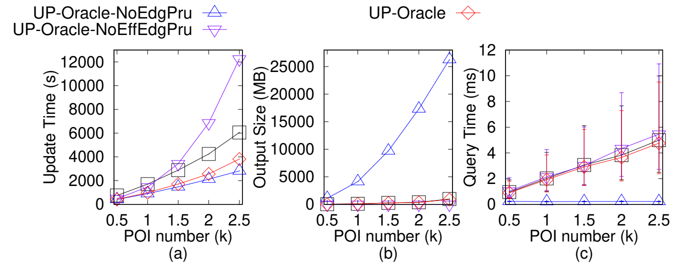

Fig. 7. Ablation study on *SC* dataset with more POIs for the P2P query

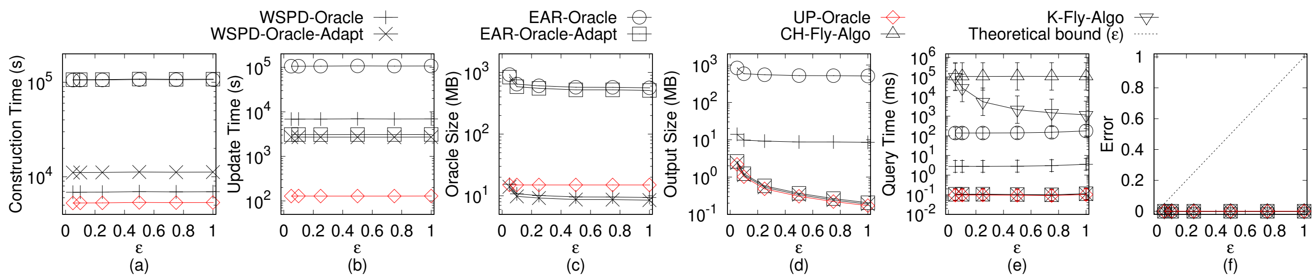

Fig. 8. Baseline comparisons (effect of ϵ on *GI* dataset with fewer POIs) for the P2P query

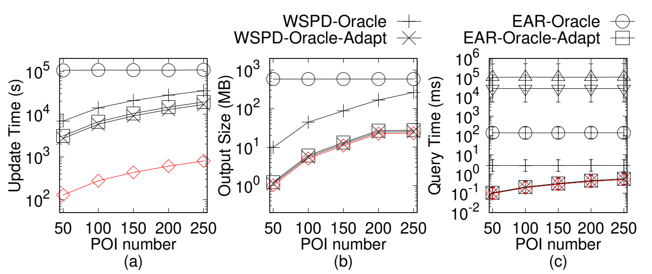

Fig. 9. Baseline comparisons (effect of n on *AU* dataset with fewer POIs) for the P2P query

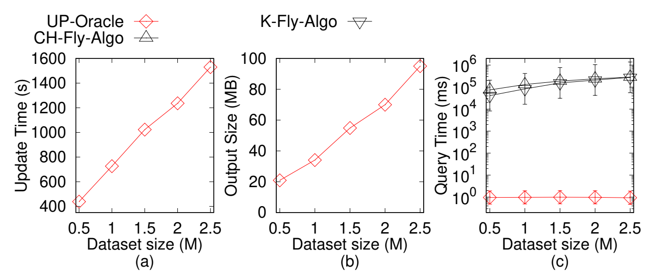

Fig. 10. Scalability test (effect of *DS* on *LH* dataset with more POIs) for the P2P query

rithm *SSAD* are loose (i.e., they need to use algorithm *SSAD* to cover most of the POIs or highway nodes as destinations even when ϵ is large).

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

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

4) *A2A query:* We compared *UP-Oracle* with other baselines for the A2A query. The result can be found in the appendix. The oracle update time of *UP-Oracle* is 15 times better than the best-known oracle *EAR-Oracle* [5].

5) *Case study:* We conducted a case study on the 4.1 magnitude earthquake (which caused an avalanche) in Valais as mentioned in Section 1.1. In this case study, on a terrain surface with 0.5M faces and 250 POIs, *UP-Oracle* just needs 400s $\approx$ 6.7 min to update the oracle, showing the usefulness of *UP-Oracle* in real-life applications.

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

6 CONCLUSION

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

References

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

[2] "Metaverse," 2023. [Online]. Available: https://about.facebook.com/meta

[3] "Google earth," 2023. [Online]. Available: https://earth.google.com/web

[4] K. Deng, H. T. Shen, K. Xu, and X. Lin, "Surface k-nn query processing," in *Proceedings of the International Conference on Data Engineering*. IEEE, 2006, pp. 78–78.

[5] B. Huang, V. J. Wei, R. C.-W. Wong, and B. Tang, "Ear-oracle: on efficient indexing for distance queries between arbitrary points on terrain surface," *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[6] M. Kaul, R. C.-W. Wong, and C. S. Jensen, "New lower and upper bounds for shortest distance queries on terrains," *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 168–179, 2015.

[7] M. Kaul, R. C.-W. Wong, B. Yang, and C. S. Jensen, "Finding shortest paths on terrains by killing two birds with one stone," *Proceedings of the VLDB Endowment*, vol. 7, no. 1, pp. 73–84, 2013.

[8] L. Liu and R. C.-W. Wong, "Finding shortest path on land surface," in *Proceedings of the ACM SIGMOD International Conference on Management of data*, 2011, pp. 433–444.

[9] V. J. Wei, R. C.-W. Wong, C. Long, D. Mount, and H. Samet, "Proximity queries on terrain surface," *ACM Transactions on Database Systems*, vol. 47, no. 4, pp. 1–59, 2022.

[10] V. J. Wei, R. C.-W. Wong, C. Long, and D. M. Mount, "Distance oracle on terrain surface," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1211–1226.

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

[12] "Moderate mag. 4.1 earthquake - 6.3 km northeast of sierre, valais, switzerland, on monday, october 24, 2016 at 16:44 gmt," 2023. [Online]. Available: https://www.volcanodiscovery.com/earthquakes/quake-info/1451397/mag4quake-Oct-24-2016-Leukerbad-VS.html

[13] "Turkey–syria earthquakes 2023," 2023. [Online]. Available: https://www.bbc.com/news/topics/cq0zxdd0y39t

[14] K. Pletcher and J. P. Rafferty, "Sichuan earthquake of 2008," 2023. [Online]. Available: https://www.britannica.com/event/Sichuan-earthquake-of-2008

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

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

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

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

[19] "Nasa mars exploration," 2023. [Online]. Available: https://mars.nasa.gov

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

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

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

[23] "2018 anchorage earthquake," 2023. [Online]. Available: https://www.usgs.gov/news/featured-story/2018-anchorage-earthquake

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

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

[26] J. Chen and Y. Han, "Shortest paths on a polyhedron," in *Proceedings of the sixth annual symposium on Computational geometry*, New York, NY, USA, 1990, p. 360–369.

[27] S. Kapoor, "Efficient computation of geodesic shortest paths," in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, 1999, pp. 770–779.

[28] J. S. Mitchell, D. M. Mount, and C. H. Papadimitriou, "The discrete geodesic problem," *SIAM Journal on Computing*, vol. 16, no. 4, pp. 647–668, 1987.

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

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

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

[32] B. Awerbuch, "Communication-time trade-offs in network synchronization," in *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, 1985, pp. 272–276.

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

[34] P. Von Rickenbach and R. Wattenhofer, "Gathering correlated data in sensor networks," in *Proceedings of the joint workshop on Foundations of mobile computing*, 2004, pp. 60–66.

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

[36] I. Althöfer, G. Das, D. Dobkin, and D. Joseph, "Generating sparse spanners for weighted graphs," in *Proceedings of the Scandinavian Workshop on Algorithm Theory*, 1990, pp. 26–37.

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

[38] G. Hou, Q. Guo, F. Zhang, S. Wang, and Z. Wei, "Personalized pagerank on evolving graphs with an incremental index-update scheme," *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[39] C. Shahabi, L.-A. Tang, and S. Xing, "Indexing land surface for efficient knn query," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1020–1031, 2008.

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

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

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

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

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

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

[46] G. Das and G. Narasimhan, "A fast algorithm for constructing sparse euclidean spanners," in *Proceedings of the tenth annual symposium on Computational geometry*, 1994, pp. 132–139.

Appendix A
Summary of Frequent Used Notations

Table 3 shows a summary of frequent used notations.

TABLE 3
Summary of frequent used notations

Notation	Meaning
T_{bef}/T_{aft}	The terrain surface before / after updates
$V/E/F$	The set of vertices / edges / faces of terrain surface
L_{max}	The length of the longest edge in E of T_{bef}
N	The number of vertices of T
ΔV	The updated vertices of T_{bef} and T_{aft}
ΔE	The updated edges edges of T_{bef} and T_{aft}
ΔF	The updated faces of T_{bef} and T_{aft}
P	The set of POI
n	The number of vertices of P
ΔP	The updated POIs on T_{bef} and T_{aft}
$\Pi(s,t\|T)$	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 *UP-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_{bef} with u as an endpoint and $v_i \in P_{remain} \setminus u$, $i \in \{1,l\}$ as the other endpoint, such that all these paths has not been updated
P_{remain}	A set of remaining POIs of P on T_{aft} that we have not processed
D	The longest edge's weight in $G.E$
$Q_{G'}$	A group of vertices in G' on H
$e_H(u,v\|T)$	An edge between u and v in H
$\Pi_H(s,t\|T)$	The shortest path of inter-edges between s and t in H

Appendix B
Comparison of All Algorithms

Table 4 shows a comparison of all algorithms in terms of the oracle construction time, oracle update time, output size and shortest path query time.

Appendix C
Empirical Studies

C.1 Experimental Results

(1) Figure 11 and Figure 12 show the ablation study on *SC* dataset with fewer and more POIs, respectively. (2) Figure 13, Figure 14, and Figure 15 show the result on the *TJ* dataset (with fewer POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (3) Figure 16, Figure 17, and Figure 18 show the result on the *SC* dataset (with fewer POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (4) Figure 8, Figure 19, and Figure 20 show the result on the *GI* dataset (with fewer POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (5) Figure 21, Figure 22, and Figure 23 show the result on the *AU* dataset (with fewer POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (6) Figure 24, Figure 25, and Figure 26 show the result on the *LH* dataset (with fewer POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (7) Figure 27, Figure 28, and Figure 29 show the result on the *VS* dataset (with fewer POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (8) Figure 30, Figure 31, and Figure 32 show the result on the *TJ* dataset (with more POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (9) Figure 33, Figure 34, and Figure 35 show the result on the *SC* dataset (with more POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (10) Figure 36, Figure 37, and Figure 38 show the result on the *GI* dataset (with more POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (11) Figure 39, Figure 40, and Figure 41 show the result on the *AU* dataset (with more POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (12) Figure 42, Figure 43, and Figure 44 show the result on the *LH* dataset (with more POIs) when varying n, ϵ, and *DS* (for scalability test), respectively. (13) Figure 45, Figure 46, and Figure 47 show the result on the *VS* dataset (with more POIs) when varying n, ϵ, and *DS* (for scalability test), respectively.

1) ***Ablation study for the P2P query:*** In Figure 11 and Figure 12, we tested the 5 values of n in {50, 100, 150, 200, 250} on *SC* dataset while fixing ϵ at 0.1 and *DS* at 0.5M for ablation study involving 6 variations and *UP-Oracle*, and the 5 values of n in {500, 1000, 1500, 2000, 2500} on *SC* dataset while fixing ϵ at 0.25 and *DS* at 0.5M for ablation study involving *UP-Oracle-**X***, where ***X*** = {*NoEffIntChe, NoEdgPru, NoEffEdgPru*} and *UP-Oracle*, respectively. The oracle update time for *UP-Oracle-**X***, where ***X*** = {*RanUpdSeq, FullRad, NoDistAppr, NoEffIntChe, NoEffEdgPru*} exceed that of *UP-Oracle*. Although the oracle update time and the shortest path query time of *UP-Oracle-NoEdgPru* is slightly smaller than that of *UP-Oracle*, the output size for *UP-Oracle-NoEdgPru* is 10^4 times larger than *UP-Oracle*. Thus, *UP-Oracle* is the best oracle among these variations.

2) ***Baseline comparisons for the P2P query:*** Starting from this subsection, we compare different baselines with *UP-Oracle*.

Effect of ϵ. In Figure 13, Figure 16, Figure 8, Figure 21, Figure 24 and Figure 27, we tested the 6 values of ϵ in {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *TJ*, *SC*, *GI*, *AU*, *LH* and *VS* datasets (with fewer POIs) while fixing n at 50 and *DS* at 0.5M. In Figure 30, Figure 33, Figure 36, Figure 39, Figure 42 and Figure 45, we tested the 6 values of ϵ in {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *TJ*, *SC*, *GI*, *AU*, *LH* and *VS* datasets (with fewer POIs) while fixing n at 500 and *DS* at 0.5M. Although *UP-Oracle* and other baselines have the similar small error (close to 0%) which are much smaller than the theoretical bound, *UP-Oracle* offers superior performance over *WSPD-Oracle*, *WSPD-Oracle-Adapt*, *EAR-Oracle*, *EAR-Oracle-Adapt*, *CH-Fly-Algo*, and *K-Fly-Algo* in terms of the oracle construction time, oracle update time, output size and shortest path query time. Although the oracle size of *UP-Oracle* is slightly larger than that of *WSPD-Oracle* and *WSPD-Oracle-Adapt*, the oracle update time of *UP-Oracle* is 88 times and 21 times smaller than that of *WSPD-Oracle* and *WSPD-Oracle-Adapt*. Varying ϵ has a small impact on the oracle update time, since when n is small, the pairwise P2P exact shortest path update step dominates the sub-graph generation step, and the former step is independent of ϵ.

TABLE 4
Comparison of algorithms with details

Algorithm	Oracle construction time		Oracle update time		Output size		Shortest path query time	
Oracle-based algorithm								
WSPD-Oracle [9], [10]	$O(\frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Large	$O(h^2)$	Small
WSPD-Oracle-Adapt [9], [10]	$O(\frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\mu_1 N\log^2 N + n\log^2 n)$	Large	$O(n)$	Small	$O(\log n)$	Small
EAR-Oracle [5]	$O(\lambda\xi mN\log^2(mN) + \frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\lambda\xi mN\log^2(mN) + \frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi\log(\lambda\xi))$	Medium
EAR-Oracle-Adapt [5]	$O(\lambda\xi mN\log^2(mN) + \frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\mu_2 N\log^2 N + n\log^2 n)$	Large	$O(n)$	Small	$O(\log n)$	Small
UP-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
UP-Oracle-FullRad	$O(nN\log^2 N)$	Small	$O(\mu_3 N\log^2 N + n\log^2 n)$	Medium	$O(n)$	Small	$O(\log n)$	Small
UP-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
UP-Oracle-NoEffIntChe	$O(nN\log^2 N)$	Small	$O(N\log^2 N + n^2)$	Medium	$O(n)$	Small	$O(\log n)$	Small
UP-Oracle-NoEdgPru	$O(nN\log^2 N + n^2)$	Small	$O(N\log^2 N + n)$	Small	$O(n^2)$	Large	$O(1)$	Small
UP-Oracle-NoEffEdgPru [36], [37]	$O(nN\log^2 N)$	Small	$O(N\log^2 N + n^3\log n)$	Medium	$O(n)$	Small	$O(\log n)$	Small
***UP-Oracle* (ours)**	$\boldsymbol{O(nN\log^2 N)}$	**Small**	$\boldsymbol{O(N\log^2 N + n\log^2 n)}$	**Small**	$\boldsymbol{O(n)}$	**Small**	$\boldsymbol{O(\log n)}$	**Small**
On-the-fly algorithm								
CH-Fly-Algo [26]	-	N/A	-	N/A	-	N/A	$O(N^2)$	Large
K-Fly-Algo [6]	-	N/A	-	N/A	-	N/A	$O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$	Large

Remark: $n < N$, h is the height of the compressed partition tree, β is the largest capacity dimension [41], μ_1, μ_2 and μ_3 are data-dependent variables, $\mu_1 \in [5, 20]$, $\mu_2 \in [12, 45]$ and $\mu_3 \in [5, 10]$ in our experiment.

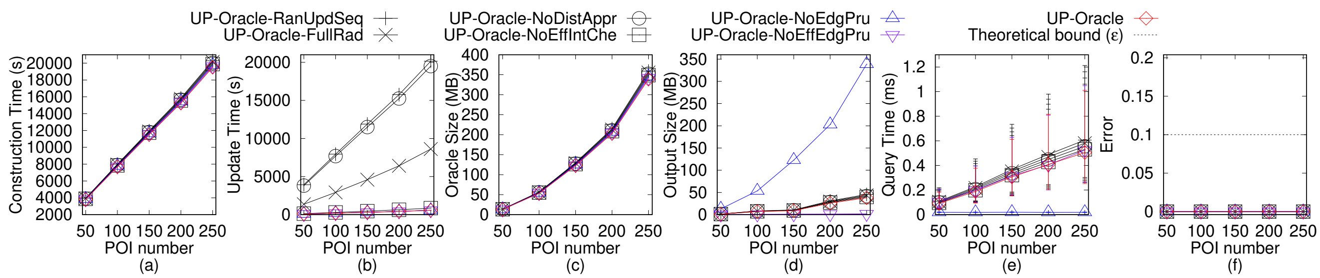

Fig. 11. Ablation study on *SC* dataset with fewer POIs for the P2P query

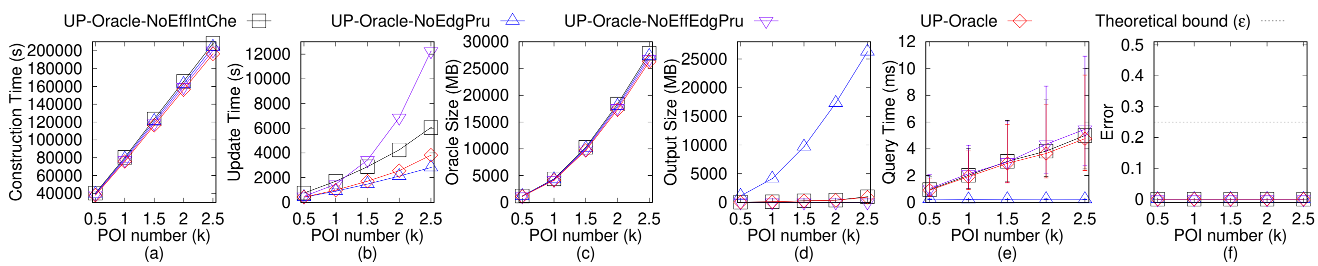

Fig. 12. Ablation study on *SC* dataset with more POIs for the P2P query

Effect of n. In Figure 14, Figure 17, Figure 19, Figure 22, Figure 25 and Figure 28, 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 31, Figure 34, Figure 37, Figure 40, Figure 43 and Figure 46, 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. The oracle update time, output size and shortest path query time of *UP-Oracle* remain better than those of the baselines. Specifically, the oracle update time of *UP-Oracle* is 21 times and 23 times smaller than that of *WSPD-Oracle-Adapt* and *EAR-Oracle-Adapt*.

3) ***Scalability test for the P2P query*** **(effect of *DS*):** In Figure 15, Figure 18, Figure 20, Figure 23, Figure 26 and Figure 29, 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 32, Figure 35, Figure 38, Figure 41, Figure 44 and Figure 47, we tested 5 values of *DS* in {0.5M, 1M, 1.5M, 2M, 2.5M} on *TJ*, *SC*, *GI*, *AU*, *LH* and *VS* datasets (with more POIs) while fixing ϵ at 0.25 and n at 500. Varying *DS* has a small impact on the shortest path query time of *UP-Oracle*, but has a large impact on that of *CH-Fly-Algo* and *K-Fly-Algo*. The shortest path query time of *UP-Oracle* is 10^5 times smaller than that of *K-Fly-Algo*. The oracle update time and output size of *UP-Oracle* is scalable when *DS* is large.

4) ***A2A query:*** In Figure 48, we tested the A2A 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 *UP-Oracle* superior performance of *WSPD-Oracle*, *WSPD-Oracle-*

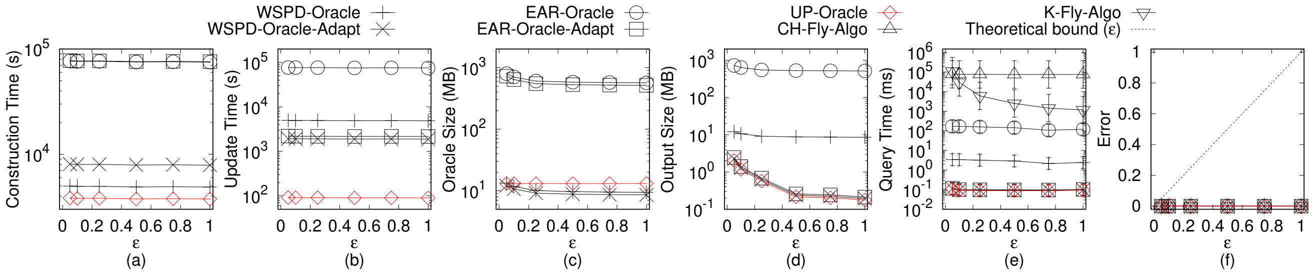

Fig. 13. Baseline comparisons (effect of ϵ on *TJ* dataset with fewer POIs) for the P2P query

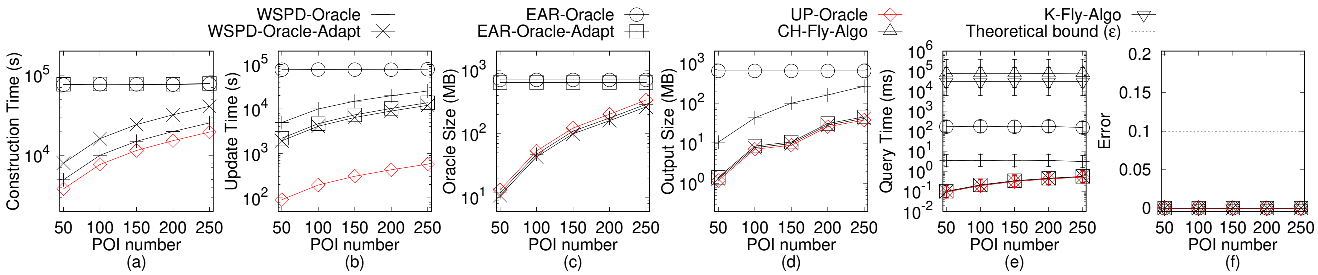

Fig. 14. Baseline comparisons (effect of n on *TJ* dataset with fewer POIs) for the P2P query

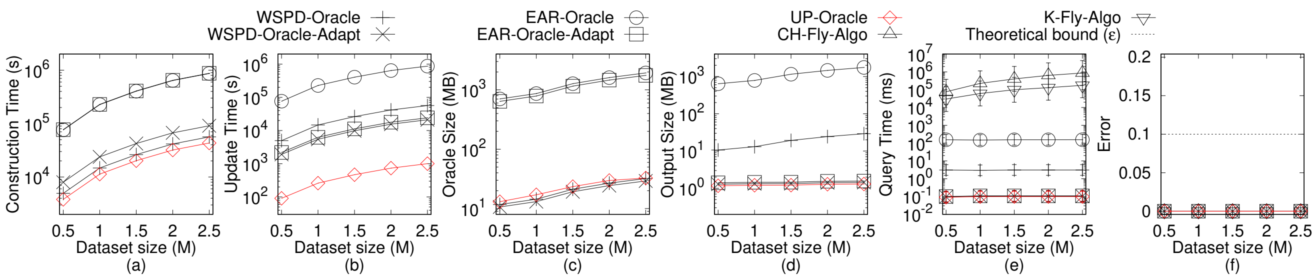

Fig. 15. Scalability test (effect of *DS* on *TJ* dataset with fewer POIs) for the P2P query

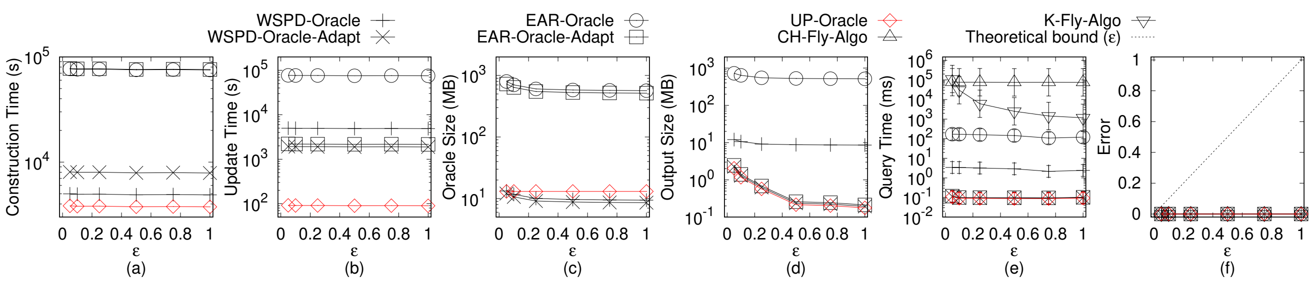

Fig. 16. Baseline comparisons (effect of ϵ on *SC* dataset with fewer POIs) for the P2P query

Adapt, *EAR-Oracle*, *EAR-Oracle-Adapt*, *CH-Fly-Algo*, and *K-Fly-Algo* in terms of the oracle construction time, oracle update time, output size and shortest path query time. For *EAR-Oracle* and *EAR-Oracle-Adapt*, although their oracle construction time is slightly smaller than *UP-Oracle*, their oracle update time is still large. Thus, *UP-Oracle* is still the best oracle for the A2A query.

C.2 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 [8], [9], [10]. Let $T_t = (V_t, E_t, F_t)$ be our target terrain surface that we want to generate with ex_t edges along x-coordinate, ey_t edges along y-coordinate and dataset size of DS_t, where $DS_t = 2 \cdot ex_t \cdot ey_t$. Let $T_o = (V_o, E_o, F_o)$ be the original terrain surface that we currently have with ex_o edges along x-coordinate, ey_o edges along y-coordinate and dataset size of DS_o, where $DS_o = 2 \cdot ex_o \cdot ey_o$. We then generate $(ex_t+1)\cdot(ey_t+1)$ 2D points (x, y) based on a Normal distribution $N(\mu_N, \sigma_N^2)$, where $\mu_N = (\overline{x} = \frac{\sum_{v_o \in V_o} x_{v_o}}{(ex_o+1)\cdot(ey_o+1)}, \overline{y} = \frac{\sum_{v_o \in V_o} y_{v_o}}{(ex_o+1)\cdot(ey_o+1)})$ and $\sigma_N^2 = (\frac{\sum_{v_o \in V_o}(x_{v_o}-\overline{x})^2}{(ex_o+1)\cdot(ey_o+1)}, \frac{\sum_{v_o \in V_o}(y_{v_o}-\overline{y})^2}{(ex_o+1)\cdot(ey_o+1)})$. In the end, we project each generated point (x, y) to the surface of T_o and take the projected point (also add edges between neighbours of two points to form edges and faces) as the newly generate T_t.

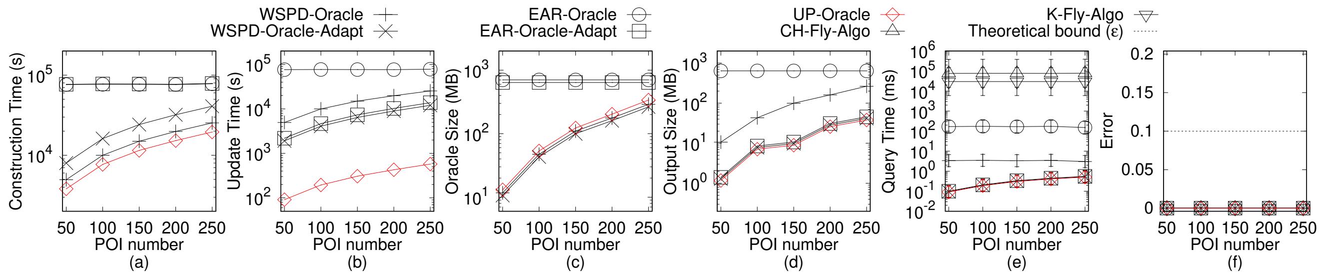

Fig. 17. Baseline comparisons (effect of n on *SC* dataset with fewer POIs) for the P2P query

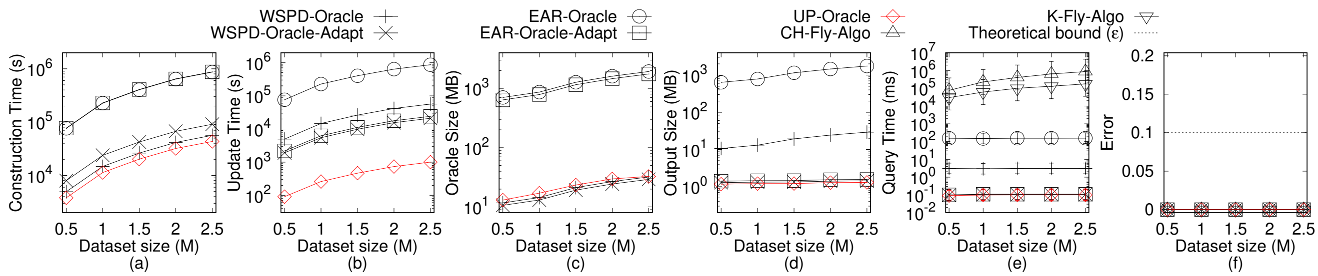

Fig. 18. Scalability test (effect of *DS* on *SC* dataset with fewer POIs) for the P2P query

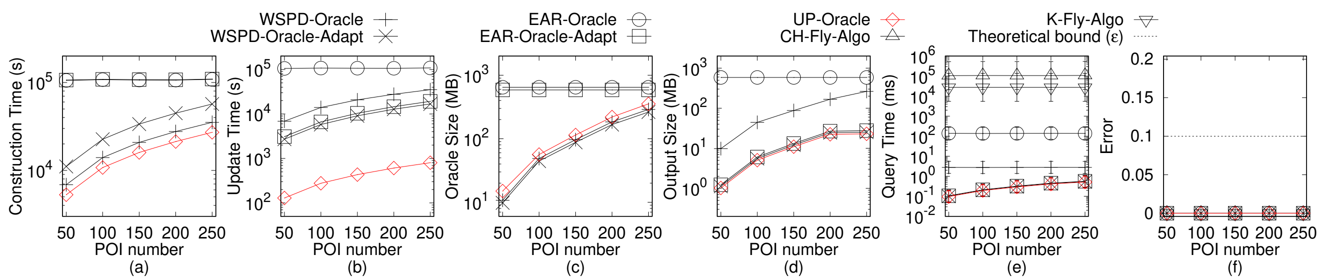

Fig. 19. Baseline comparisons (effect of n on *GI* dataset with fewer POIs) for the P2P query

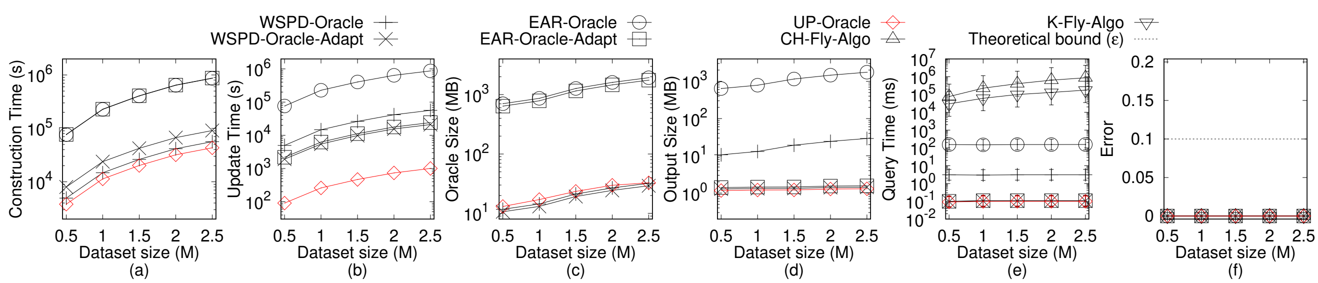

Fig. 20. Scalability test (effect of *DS* on *GI* dataset with fewer POIs) for the P2P query

APPENDIX D
PROOF

Proof of Property 1. We prove by contradiction. Suppose that two disks $D(u, \frac{|\Pi(u,v|T_{bef})|}{2})$ and $D(v, \frac{|\Pi(u,v|T_{bef})|}{2})$ do not intersect with ΔF, but $\Pi(u,v|T_{aft})$ is different from $\Pi(u,v|T_{bef})$, and we need to update $\Pi(u,v|T_{bef})$ to $\Pi(u,v|T_{aft})$ due to the smaller distance of $\Pi(u,v|T_{aft})$, i.e., $|\Pi(u,v|T_{aft})| < |\Pi(u,v|T_{bef})|$. This case will only happen when $\Pi(u,v|T_{aft})$ passes ΔF. We let u_1 (resp. v_1) be the point on $\Pi(u,v|T_{aft})$ that the exact shortest distance $\Pi(u,u_1|T)$ (resp. $\Pi(v,v_1|T)$) on T is the same as $|\frac{\Pi(u,v|T_{bef})}{2}|$. We let u_2 (resp. v_2) be the point on $\Pi(u,v|T_{aft})$ 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 $\Pi(u,v|T_{aft})$ and ΔF, such that the exact shortest distance $\Pi(u,u_2|T)$ (resp. $\Pi(v,v_2|T)$) on T is the minimum one. Note that a point is said to be in ΔF if this point is on a face in ΔF. We let o be the midpoint on $\Pi(u,v|T_{bef})$, clearly we have $|\Pi(u,o|T)| = |\Pi(n,v|T)| = |\frac{\Pi(u,v|T_{bef})}{2}|$. We also know that $|\Pi(u,u_1|T)| = |\Pi(u,o|T)| = |\Pi(v,v_1|T)| = |\Pi(v,o|T)| = |\frac{\Pi(u,v|T_{bef})}{2}|$. Figure 2 shows an example of these notations. The light blue line is $\Pi(u,v|T_{bef})$ and the yellow line is $\Pi(u,v|T_{aft})$. Since the minimum distance from both u and v to the updated faces ΔF is no smaller than $|\frac{\Pi(u,v|T_{bef})}{2}|$, 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 $\Pi(u,v|T_{aft})$ passes ΔF, $|\Pi(u_2,v_2|T_{aft})| \geq 0$. Thus, we have $|\Pi(u,u_2|T)| + |\Pi(v,v_2|T)| + |\Pi(u_2,v_2|T_{aft})| = |\Pi(u,v|T_{aft})| \geq |\Pi(u,v|T_{bef})| = |\Pi(u,o|T)| + |\Pi(v,o|T)|$, which is a contradiction of our assumption $|\Pi(u,v|T_{aft})| < |\Pi(u,v|T_{bef})|$. Thus, we finish the

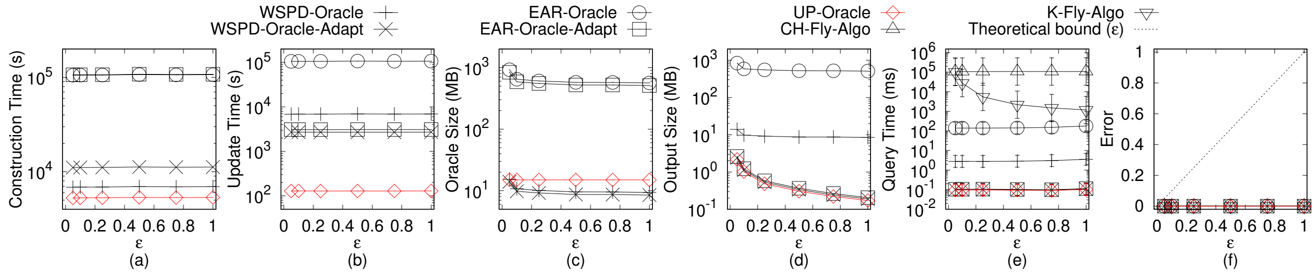

Fig. 21. Baseline comparisons (effect of ϵ on *AU* dataset with fewer POIs) for the P2P query

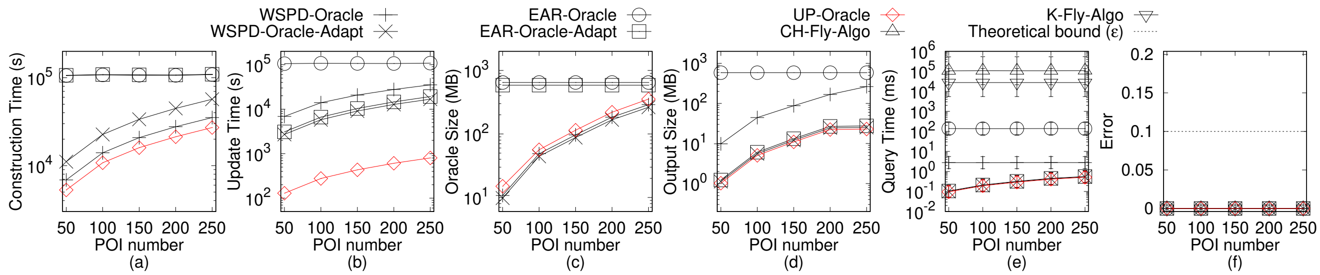

Fig. 22. Baseline comparisons (effect of n on *AU* dataset with fewer POIs) for the P2P query

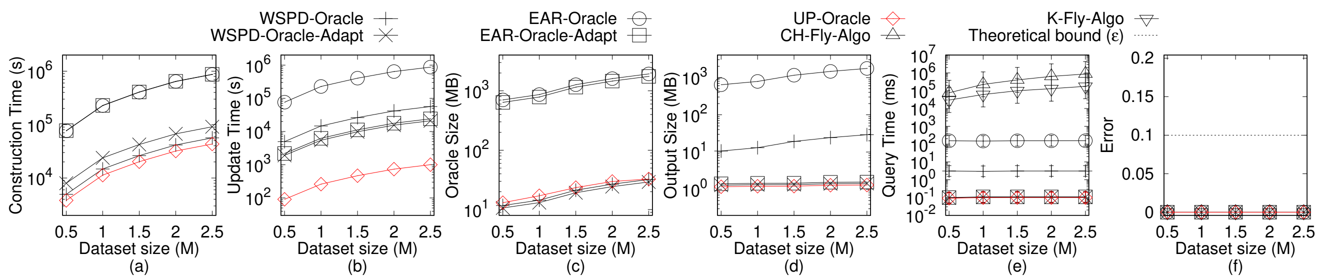

Fig. 23. Scalability test (effect of *DS* on *AU* dataset with fewer POIs) for the P2P query

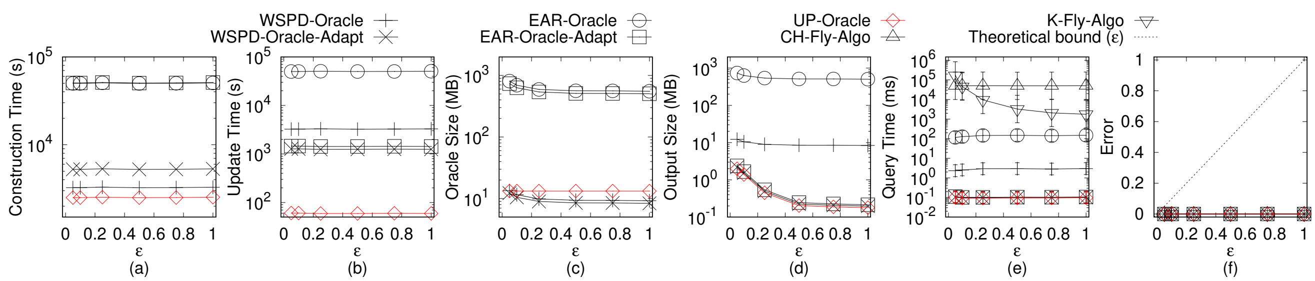

Fig. 24. Baseline comparisons (effect of ϵ on *LH* dataset with fewer POIs) for the P2P query

proof. □

Proof of Lemma 2. According to [9], [10], since the exact shortest distance on a terrain surface is a metric, and therefore it satisfies the triangle inequality. Given an edge e which belongs to a face in ΔF with two endpoints u_1 and u_2, suppose that the exact shortest path from u to ΔF intersects with any point on e for the first time. There are two cases:

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

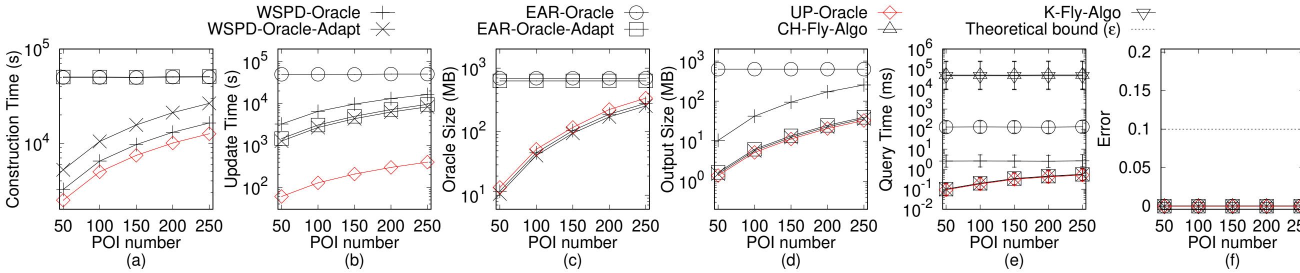

Fig. 25. Baseline comparisons (effect of n on *LH* dataset with fewer POIs) for the P2P query

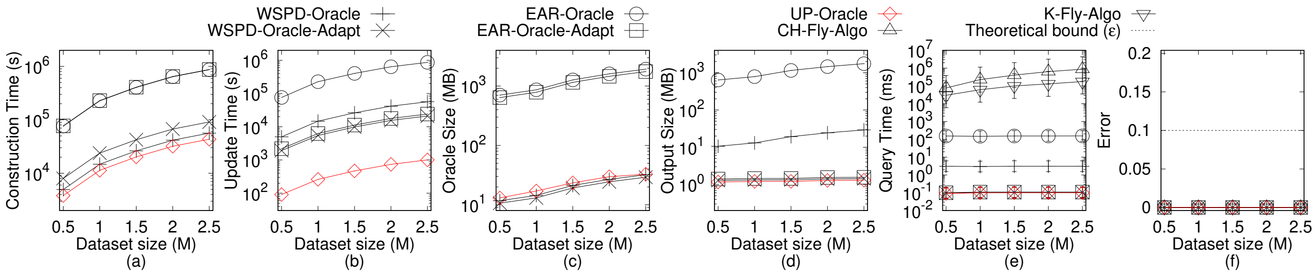

Fig. 26. Scalability test (effect of *DS* on *LH* dataset with fewer POIs) for the P2P query

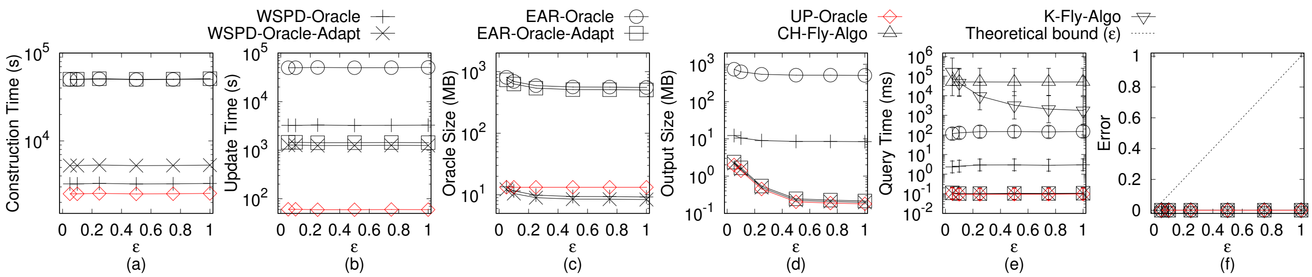

Fig. 27. Baseline comparisons (effect of ϵ on *VS* dataset with fewer POIs) for the P2P query

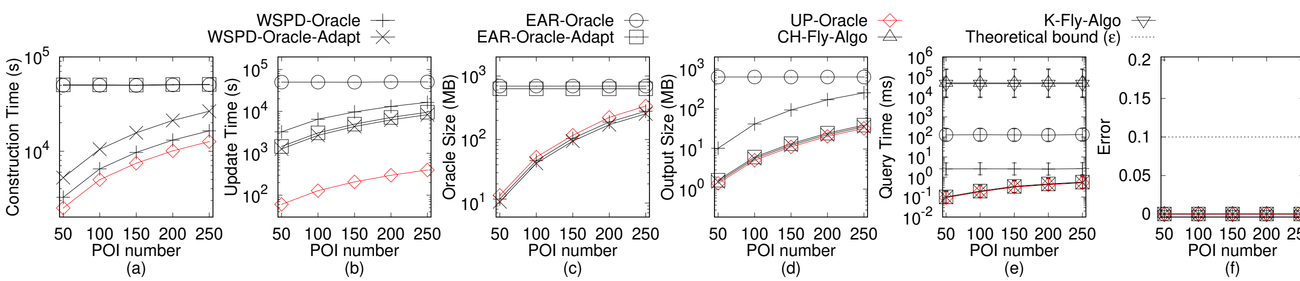

Fig. 28. Baseline comparisons (effect of n on *VS* dataset with fewer POIs) for the P2P query

is at least the minimum distance from u to any vertex in ΔV on T_{bef} and $|u_1u_3|$ is at most L_{max}, we obtain that the minimum distance from u to ΔF in non-updated faces of T_{aft} is at least the minimum distance from u to any vertex in ΔV on T_{bef} minus L_{max}.

□

Proof of Lemma 3. If the disk with the largest radius intersects with ΔF, we just need to update the paths and there is no need to check other disks. In Figure 4 (c), the sorted POIs are h, f, e, d, c, g. We create one disk $D(h, \frac{|\Pi(c,h|T_{bef})|}{2})$, since it *intersects* with ΔF, we use algorithm *SSAD* to update all shortest paths adjacent to h that have not been updated. We do not need to create ten disks, i.e., five disks $D(h, \frac{|\Pi(X,h|T_{bef})|}{2})$ and five disks $D(X, \frac{|\Pi(X,h|T_{bef})|}{2})$, where $X = \{c, d, e, f, g, h\}$. Since the disk $D(h, \frac{|\Pi(c,h|T_{bef})|}{2})$ with the largest radius already intersects with ΔF, so there is no need to check other disks.

If the disk with the largest radius and with the center closest to ΔF does not intersect with ΔF, then other disks cannot intersect with ΔF, so there is no need to update the paths. In Figure 4 (d), the sorted POIs are f, e, d, c, g. We create one disk $D(f, \frac{|\Pi(c,f|T_{bef})|}{2})$, since it *does not intersect* with ΔF, there is no need to update shortest paths adjacent to f. We do not need to create eight disks, i.e., four disks $D(f, \frac{|\Pi(X,f|T_{bef})|}{2})$ and four disks $D(X, \frac{|\Pi(X,f|T_{bef})|}{2})$, where $X = \{c, d, e, f, g\}$. Since the disk $D(f, \frac{|\Pi(c,f|T_{bef})|}{2})$ with the largest radius does not intersect with ΔF, so the disks $D(f, \frac{|\Pi(X,f|T_{bef})|}{2})$ with smaller radius and the disks $D(X, \frac{|\Pi(X,f|T_{bef})|}{2})$ with centers further away from ΔF

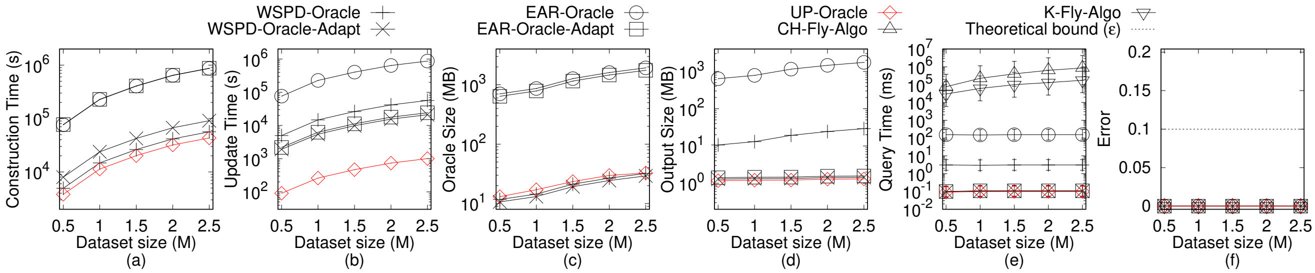

Fig. 29. Scalability test (effect of *DS* on *VS* dataset with fewer POIs) for the P2P query

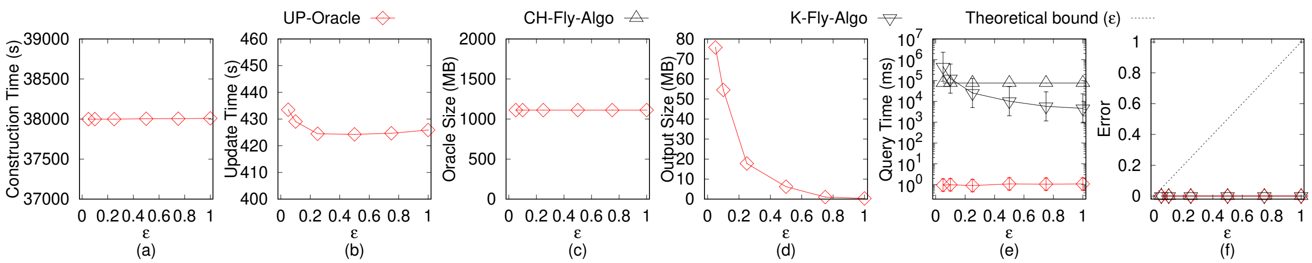

Fig. 30. Baseline comparisons (effect of ϵ on *TJ* dataset with more POIs) for the P2P query

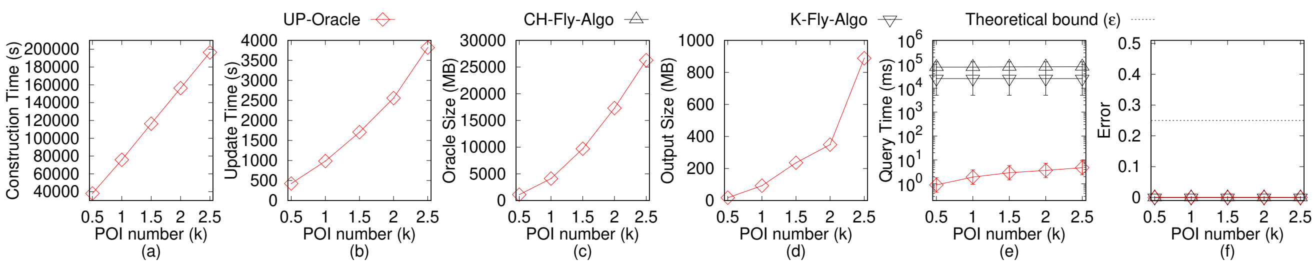

Fig. 31. Baseline comparisons (effect of n on *TJ* dataset with more POIs) for the P2P query

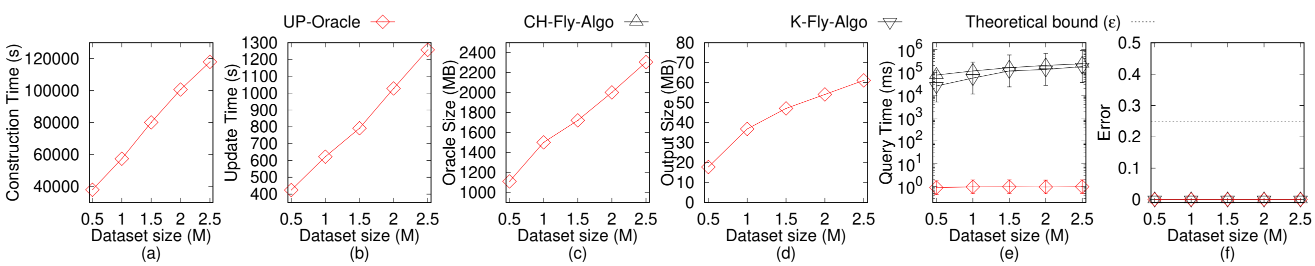

Fig. 32. Scalability test (effect of *DS* on *TJ* dataset with more POIs) for the P2P query

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

Lemma 4. *After the pairwise P2P exact shortest path update step in the update phase of UP-Oracle, G stores the correct exact shortest path between all pairs of POIs in P on T_{aft}.*

Proof of Lemma 4. After the pairwise P2P exact shortest path update 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_{aft}, and (2) the non-updated exact shortest paths calculated on T_{bef}. Due to Property 1, we know that the non-updated exact shortest paths calculated on T_{bef} is exactly the same as the exact shortest path on T_{aft}. Thus, after the pairwise P2P exact shortest path update step in the update phase of *UP-Oracle*, G stores the correct exact shortest path between all pairs of POIs in P on T_{aft}. $\square$

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

- In the edge sort, interval split, and G' initialization step, it needs $O(n)$ time. Since we perform algorithm *SSAD* for each POI to generate G, so given a POI, the distances between this POI and other POIs have already been

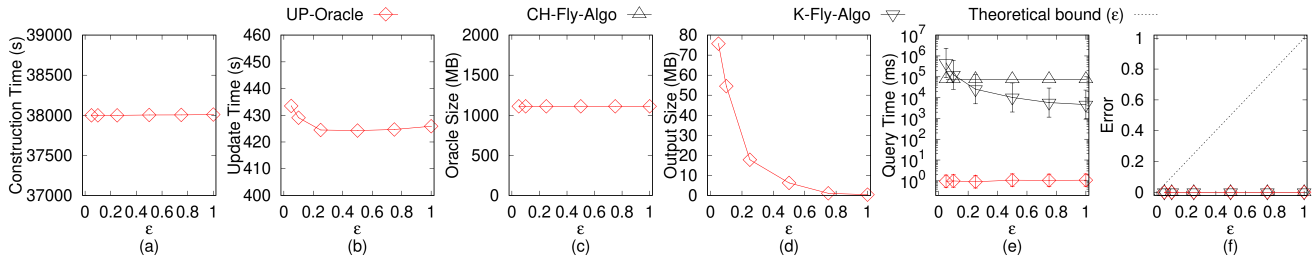

Fig. 33. Baseline comparisons (effect of ϵ on *SC* dataset with more POIs) for the P2P query

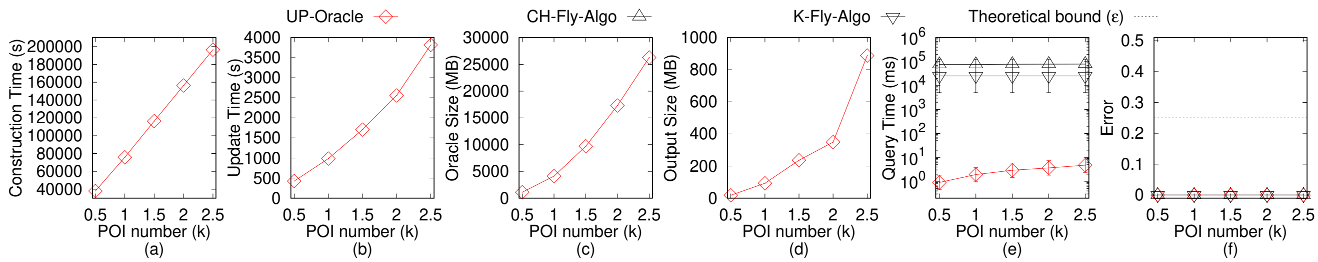

Fig. 34. Baseline comparisons (effect of n on *SC* dataset with more POIs) for the P2P query

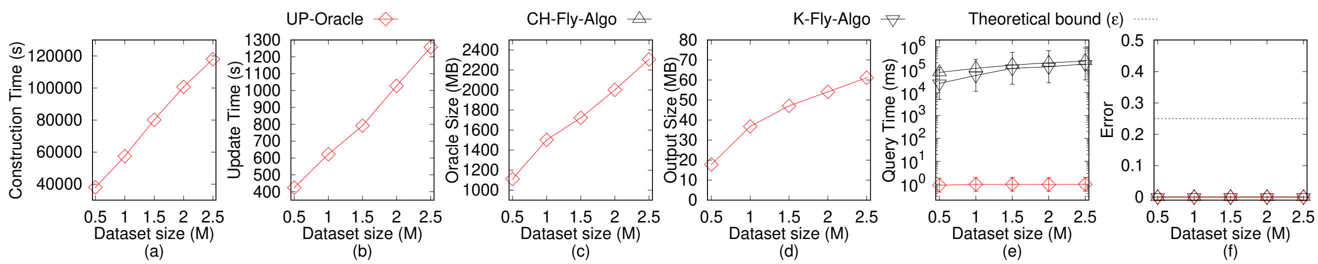

Fig. 35. Scalability test (effect of *DS* on *SC* dataset with more POIs) for the P2P query

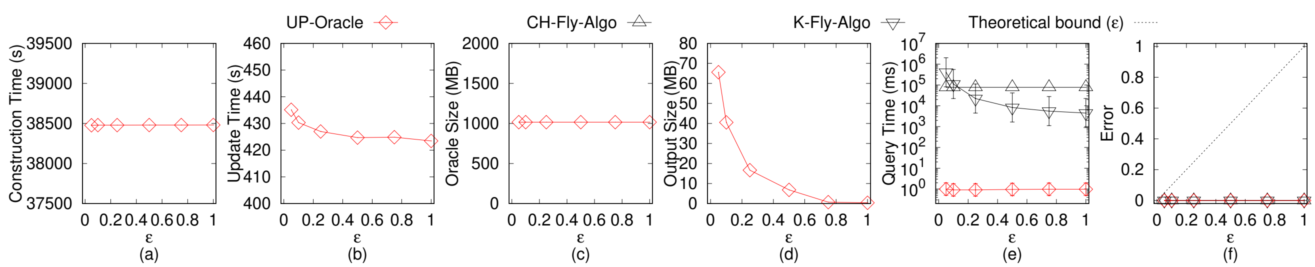

Fig. 36. Baseline comparisons (effect of ϵ on *GI* dataset with more POIs) for the P2P query

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 group construction and H intra-edge insertion step, it needs $O(n \log n)$ time. This is because according to Lemma 6 in [46], 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 H first type inter-edge insertion 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 H edge examine step, it needs $O(n)$ time. According to [46], 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 [46], 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 H edge examine 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

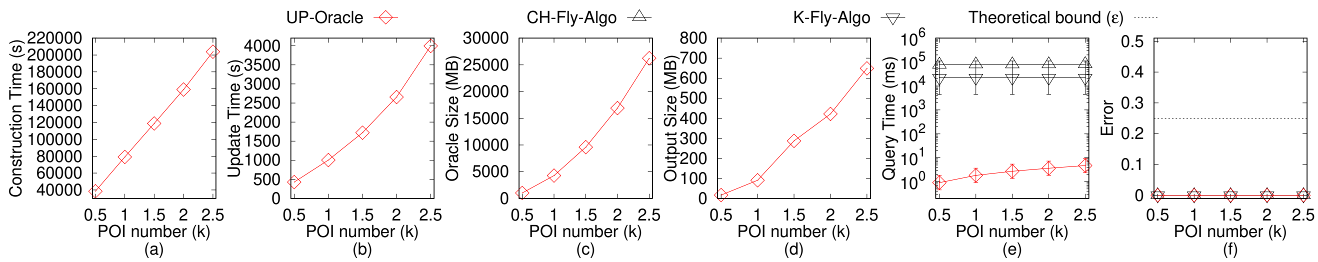

Fig. 37. Baseline comparisons (effect of n on *GI* dataset with more POIs) for the P2P query

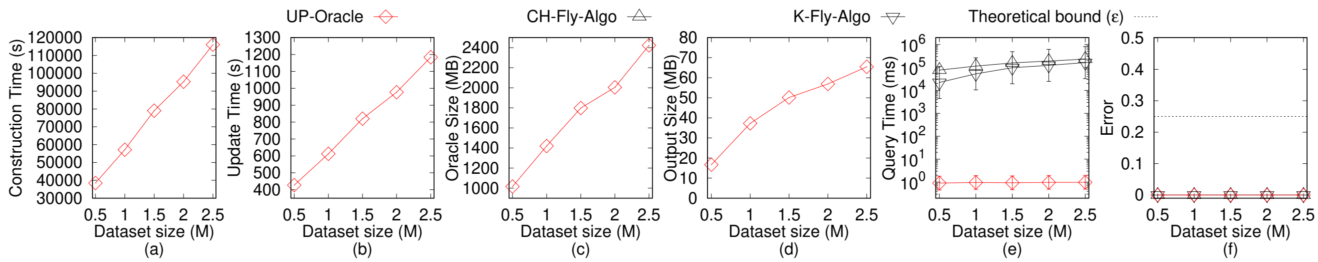

Fig. 38. Scalability test (effect of *DS* on *GI* dataset with more POIs) for the P2P query

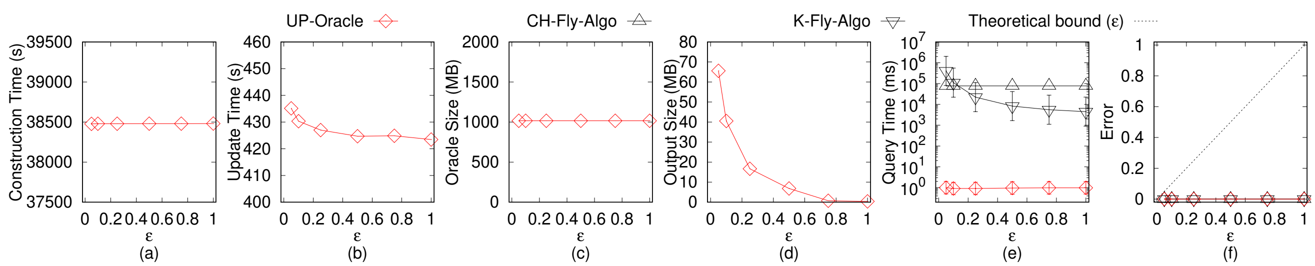

Fig. 39. Baseline comparisons (effect of ϵ on *AU* dataset with more POIs) for the P2P query

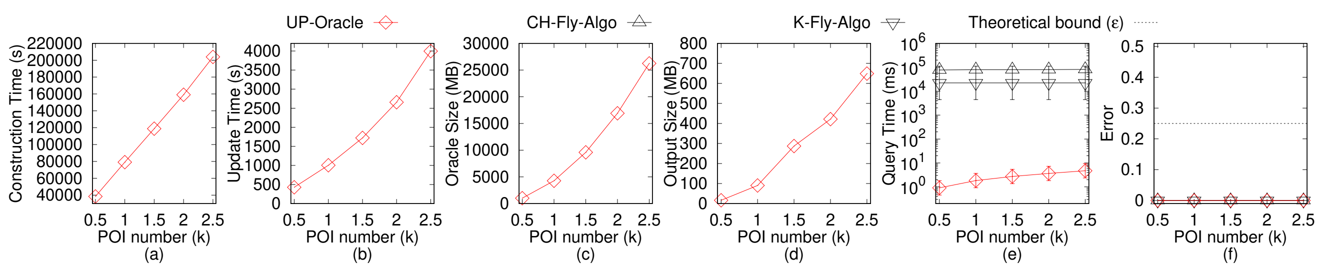

Fig. 40. Baseline comparisons (effect of n on *AU* dataset with more POIs) for the P2P query

$|\Pi_H(w,x|T)| > (1+\epsilon)|e(u,v|T)|$, where $\Pi_H(w,x|T)$ is the shortest path of group centers calculated using Dijkstra's algorithm on H, w and x are two group centers, such that, u is in w's group, and v is in x's group, $\Pi_H(w,x|T)$ is a valid approximation of $\Pi_{G'}(u,v|T)$. In other words, we are actually checking whether $|\Pi_{G'}(u,v|T)| > (1+\epsilon)|e(u,v|T)|$ or not. Consider any edge $e(u,v|T) \in G'.E$ between two vertices u and v which is not added to G' by algorithm *HieGreSpan*. Since $e(u,v|T)$ is discarded, it implies that $|\Pi_{G'}(u,v|T)| \leq (1+\epsilon)|e(u,v|T)|$. Since $|e(u,v|T)| = |\Pi(u,v|T)|$, so on the output graph of algorithm *HieGreSpan*, i.e., G', we always have $|\Pi_{G'}(u,v|T)| \leq (1+\epsilon)|\Pi(u,v|T)|$ for all pairs of vertices u and v in $G'.V$. We finish the proof. □

Proof of Theorem 2. We give the proof for the P2P query as follows.

Firstly, we prove the *oracle construction time* of *UP-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_{bef}. So the oracle construction time of *UP-Oracle* is $O(nN\log^2 N)$.

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

- In the terrain surface and POI update detection step, it needs $O(N+n)$ time. Since we just need to iterate each face in T_{aft} and T_{bef}, and iterate each POI in P. Since the number of faces in T_{aft} and T_{bef} 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 path update 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_{aft}, and each algorithm *SSAD* needs $O(N\log^2 N)$ time, so it

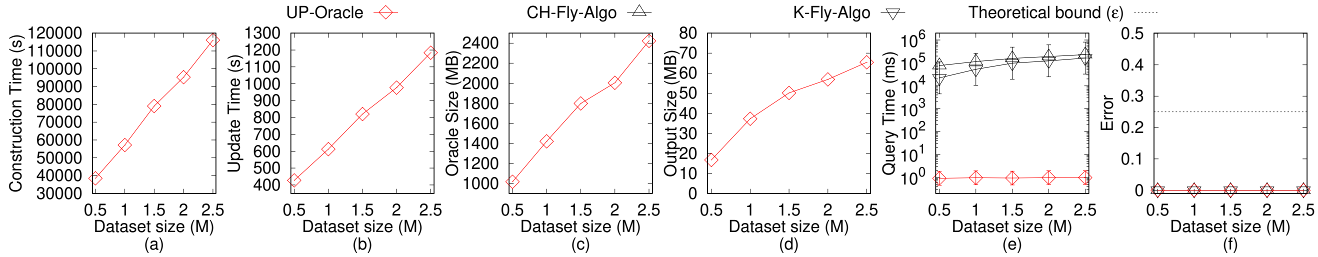

Fig. 41. Scalability test (effect of *DS* on *AU* dataset with more POIs) for the P2P query

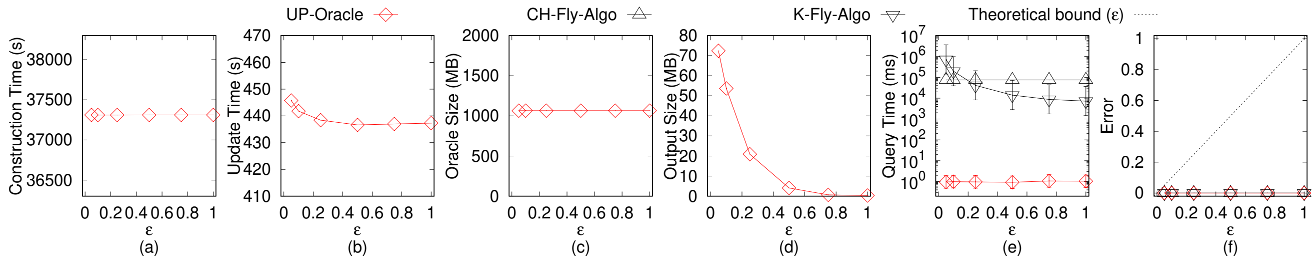

Fig. 42. Baseline comparisons (effect of ϵ on *LH* dataset with more POIs) for the P2P query

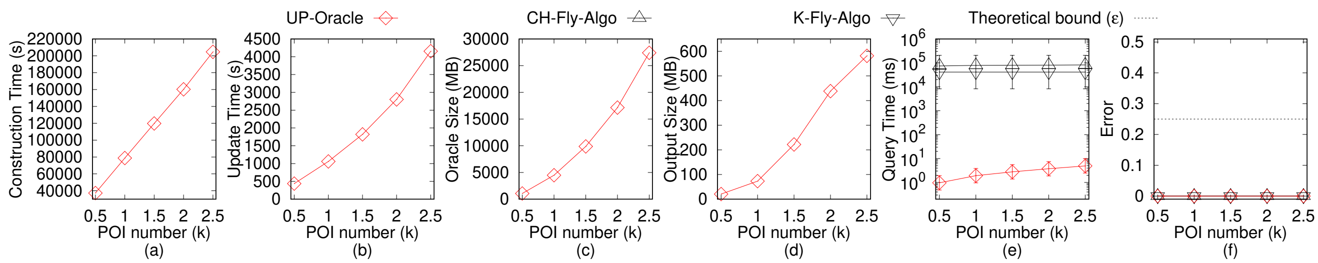

Fig. 43. Baseline comparisons (effect of n on *LH* dataset with more POIs) for the P2P query

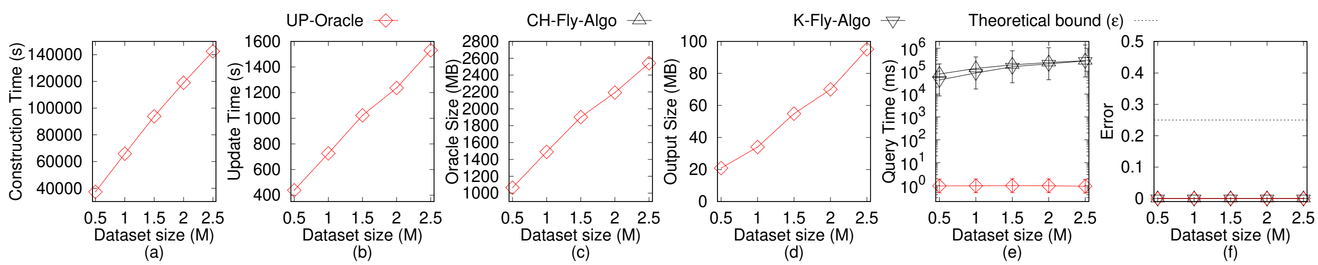

Fig. 44. Scalability test (effect of *DS* on *LH* dataset with more POIs) for the P2P query

needs $O(N \log^2 N)$ time in total.

- In the sub-graph generation step, it needs $O(n \log^2 n)$ time. Since this step is using algorithm *HieGreSpan*, and algorithm *HieGreSpan* runs in $O(n \log^2 n)$ time as stated in Theorem 1.

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

Thirdly, we prove the *output size* of *UP-Oracle*. According to [46], we know that the output graph of algorithm *HieGreSpan*, i.e., G', has $O(n)$ edges. So, the output size of *UP-Oracle* is $O(n)$.

Fourthly, we prove the *shortest path query time* of *UP-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 *UP-Oracle* is $O(\log n)$.

Fifthly, we prove the *error bound* of *UP-Oracle*. The error bound of *UP-Oracle* is due to the error bound of algorithm *HieGreSpan*. As stated in Theorem 1, on the output graph of algorithm *HieGreSpan*, i.e., G', we always have $|\Pi_{G'}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ for all pairs of vertices u and v in $G'.V$. Thus, we have the error bound of *UP-Oracle*, i.e., *UP-Oracle* satisfies $|\Pi_{G'}(u, v|T)| \leq (1+\epsilon)|\Pi(u, v|T)|$ for all pairs of POIs u and v in P.

We give the proof for the A2A query as follows. For the *oracle construction time*, *oracle update time* and *output size*, since we place total $\frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon}$ Steiner points [44] on the faces of the terrain surface, so we use this value to substitute n in the oracle construction time, oracle update time and output size of *UP-Oracle* for the P2P query to obtain the results for the A2A query. For the *shortest path query time*, we first use $\frac{N}{\sin\theta\sqrt{\epsilon}} \log \frac{1}{\epsilon}$ to substitute n in the shortest path query time of *UP-Oracle* for the P2P query,

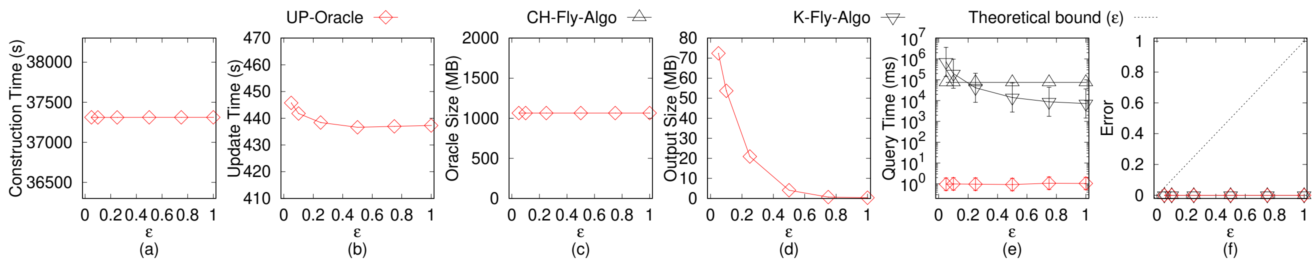

Fig. 45. Baseline comparisons (effect of ϵ on *VS* dataset with more POIs) for the P2P query

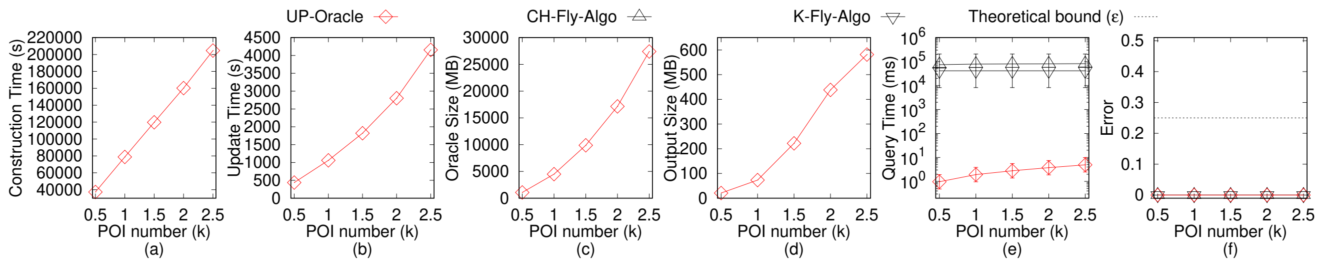

Fig. 46. Baseline comparisons (effect of n on *VS* dataset with more POIs) for the P2P query

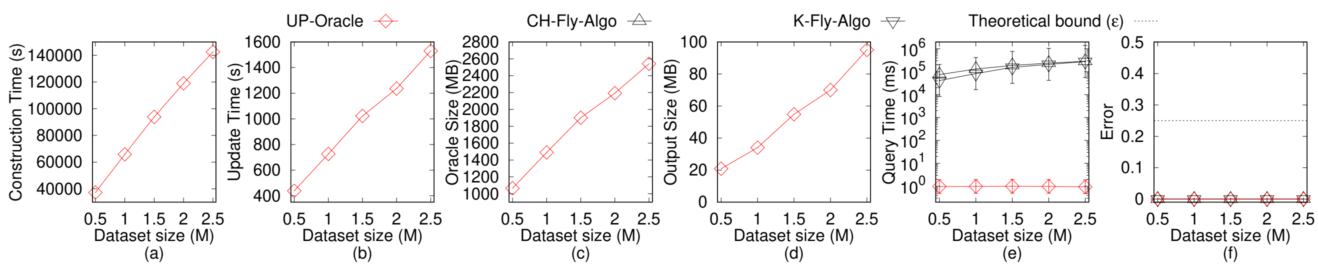

Fig. 47. Scalability test (effect of *DS* on *VS* dataset with more POIs) for the P2P query

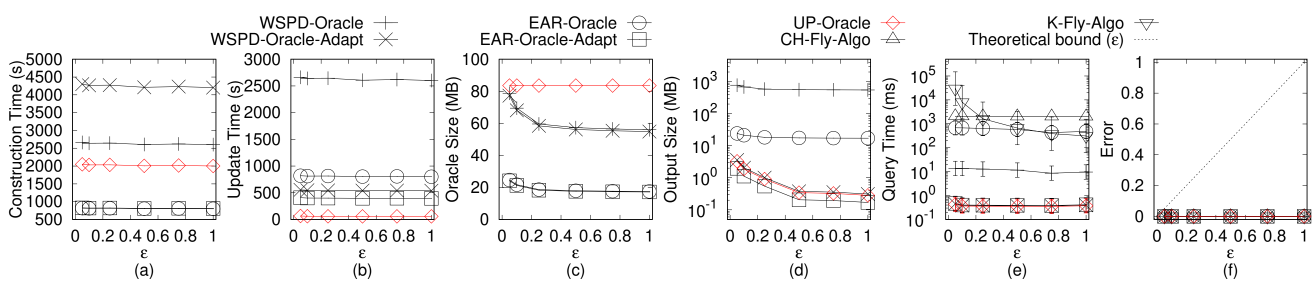

Fig. 48. A2A query on *SC* dataset

to obtain the $O(\log(\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$ query time for finding the shortest path between any two Steiner points p and q. Since $|\mathcal{S}(s)|\cdot|\mathcal{S}(t)| = \frac{1}{\sin\theta\cdot\epsilon}$ [44], the total shortest path query time for finding the shortest path between two arbitrary points s and t is $O(\frac{1}{\sin\theta\cdot\epsilon}\log(\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$. For the *error bound*, we first know that $|\Pi_{G'}(p,q|T)| \leq (1+\epsilon)|\Pi(p,q|T)|$ for any two Steiner points p and q due to the error bound of *UP-Oracle* for the P2P query. According to study [44], if $|\Pi_{G'}(p,q|T)| \leq (1+\epsilon)|\Pi(p,q|T)|$, then $|\Pi_{G'}(s,t|T)| \leq (1+\epsilon)|\Pi(s,t|T)|$ for two arbitrary points s and t.

In general, we finish the proof of the oracle construction time, oracle update time, output size, shortest path query time and error bound of *UP-Oracle* for both P2P and A2A queries. □

Theorem 3. *The oracle construction time, oracle update time, output size and shortest path query time of WSPD-Oracle [9], [10] are* $O(\frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ *and* $O(h^2)$, *respectively. WSPD-Oracle has* $(1-\epsilon)|\Pi(s,t|T)| \leq |\Pi_{\textit{WSPD-Oracle}}(s,t|T)| \leq (1+\epsilon)|\Pi(s,t|T)|$ *for all pairs of POIs* s *and* t *in* P, *where* $\Pi_{\textit{WSPD-Oracle}}(s,t|T)$ *is the shortest path of WSPD-Oracle between* s *and* t.

Proof. The proof of the oracle construction time, output size and error bound of *WSPD-Oracle* is in [9], [10].

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 [9], [10]. Thus, the shortest path query time of *WSPD-Oracle* should be $O(3h^2) = O(h^2)$. □

Theorem 4. *The oracle construction time, oracle update time, output size and shortest path query time of WSPD-Oracle-Adapt [9], [10] are* $O(\frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\mu_1 N \log^2 N + n\log^2 n)$, $O(n)$ *and* $O(\log n)$, *respectively, where* μ_1 *is a data-dependent variable, and* $\mu_1 \in [5, 20]$ *in our experiment. WSPD-Oracle-Adapt satisfies* $|\Pi_{\textit{WSPD-Oracle-Adapt}}(s, t|T)| \leq (1+\epsilon)|\Pi(s, t|T)|$ *for all pairs of POIs* s *and* t *in* P, *where* $\Pi_{\textit{WSPD-Oracle-Adapt}}(s, t|T)$ *is the shortest path of WSPD-Oracle-Adapt between* s *and* t.

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

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

For the *oracle update time*, since *WSPD-Oracle-Adapt* uses the update phase of *UP-Oracle*, so it first needs $O(N+n)$ time for terrain surface and POI 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 $T_{\textit{aft}}$, where each algorithm *SSAD* needs $O(N\log^2 N)$ time, and then needs $O(n\log^2 n)$ time for sub-graph generation. So the oracle update time of *WSPD-Oracle-Adapt* is $O(\mu_1 N\log^2 N + n\log^2 n)$. □

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

Proof. The proof of the oracle construction time, output size, shortest path query time and error bound of *EAR-Oracle* is in [5].

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

Theorem 6. *The oracle construction time, oracle update time, output size and shortest path query time of EAR-Oracle-Adapt [5] are* $O(\lambda\xi m N\log^2(mN) + \frac{nN\log^2 N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\mu_2 N\log^2 N + n\log^2 n)$, $O(n)$ *and* $O(\log n)$, *respectively, where* μ_2 *is a data-dependent variable, and* $\mu_1 \in [12, 45]$ *in our experiment. EAR-Oracle-Adapt satisfies* $|\Pi_{\textit{EAR-Oracle-Adapt}}(s, t|T)| \leq (1+\epsilon)|\Pi(s, t|T)|$ *for all pairs of POIs* s *and* t *in* P, *where* $\Pi_{\textit{EAR-Oracle-Adapt}}(s, t|T)$ *is the shortest path of EAR-Oracle-Adapt between* s *and* t.

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

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

Theorem 7. *The oracle construction time, oracle update time, output size and shortest path query time of UP-Oracle-RanUpdSeq for the P2P query are* $O(nN\log^2 N)$, $O(nN\log^2 N + n\log^2 n)$, $O(n)$ *and* $O(\log n)$, *respectively. UP-Oracle-RanUpdSeq satisfies* $|\Pi_{\textit{UP-Oracle-NoEffEdgPru}}(s, t|T)| \leq (1+\epsilon)|\Pi(s, t|T)|$ *for all pairs of POIs* s *and* t *in* P *for the P2P query, where* $\Pi_{\textit{UP-Oracle-RanUpdSeq}}(s, t|T)$ *is the shortest path of UP-Oracle-RanUpdSeq between* s *and* t.

Proof. The proof of the oracle construction time, output size, shortest path query time and error bound of *UP-Oracle-RanUpdSeq* for the P2P query is similar in *UP-Oracle* for the P2P query.

For the *oracle update time*, the only difference between *UP-Oracle* for the P2P query and *UP-Oracle-RanUpdSeq* for the P2P query 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 path update step, it needs to use algorithm *SSAD* for all POIs for n times. The other oracle update time is the same as the *UP-Oracle* for the P2P query. So the oracle update time of *UP-Oracle-RanUpdSeq* for the P2P query is $O(nN\log^2 N + n\log^2 n)$. □

Theorem 8. *The oracle construction time, oracle update time, output size and shortest path query time of UP-Oracle-FullRad for the P2P query are* $O(nN\log^2 N)$, $O(\mu_3 N\log^2 N + n\log^2 n)$, $O(n)$ *and* $O(\log n)$, *respectively, where* μ_3 *is a data-dependent variable, and* $\mu_3 \in [5, 10]$ *in our experiment. UP-Oracle-FullRad satisfies* $|\Pi_{\textit{UP-Oracle-NoEffEdgPru}}(s, t|T)| \leq (1+\epsilon)|\Pi(s, t|T)|$ *for all pairs of POIs* s *and* t *in* P *for the P2P query, where* $\Pi_{\textit{UP-Oracle-FullRad}}(s, t|T)$ *is the shortest path of UP-Oracle-FullRad between* s *and* t.

Proof. The proof of the oracle construction time, output size, shortest path query time and error bound of *UP-Oracle-FullRad* for the P2P query is similar in *UP-Oracle* for the P2P query.

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

Theorem 9. *The oracle construction time, oracle update time, output size and shortest path query time of UP-Oracle-NoDistAppr for the P2P query are $O(nN\log^2 N)$, $O(nN\log^2 N + n\log^2 n)$, $O(n)$ and $O(\log n)$, respectively. UP-Oracle-NoDistAppr satisfies $|\Pi_{\textit{UP-Oracle-NoEffEdgPru}}(s,t|T)| \leq (1+\epsilon)|\Pi(s,t|T)|$ for all pairs of POIs s and t in P for the P2P query, where $\Pi_{\textit{UP-Oracle-NoDistAppr}}(s,t|T)$ is the shortest path of UP-Oracle-NoDistAppr between s and t.*

Proof. The proof of the oracle construction time, output size, shortest path query time and error bound of *UP-Oracle-NoDistAppr* for the P2P query is similar in *UP-Oracle* for the P2P query.

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

Theorem 10. *The oracle construction time, oracle update time, output size and shortest path query time of UP-Oracle-NoEffIntChe for the P2P query are $O(nN\log^2 N)$, $O(nN\log^2 N + n\log^2 n)$, $O(n)$ and $O(\log n)$, respectively. UP-Oracle-NoEffIntChe satisfies $|\Pi_{\textit{UP-Oracle-NoEffEdgPru}}(s,t|T)| \leq (1+\epsilon)|\Pi(s,t|T)|$ for all pairs of POIs s and t in P for the P2P query, where $\Pi_{\textit{UP-Oracle-NoEffIntChe}}(s,t|T)$ is the shortest path of UP-Oracle-NoEffIntChe between s and t.*

Proof. The proof of the oracle construction time, output size, shortest path query time and error bound of *UP-Oracle-NoEffIntChe* for the P2P query is similar in *UP-Oracle* for the P2P query.

For the *oracle update time*, the only difference between *UP-Oracle* for the P2P query and *UP-Oracle-NoEffIntChe* for the P2P query 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 path update step, since there are total $O(n^2)$ pairwise P2P exact shortest paths, it needs to create $O(n^2)$ disks. The other oracle update time is the same as the *UP-Oracle* for the P2P query. So the oracle update time of *UP-Oracle-NoEffIntChe* for the P2P query is $O(nN\log^2 N + n\log^2 n)$. ☐

Theorem 11. *The oracle construction time, oracle update time, output size and shortest path query time of UP-Oracle-NoEdgPru for the P2P query are $O(nN\log^2 N + n^2)$, $O(N\log^2 N + n)$, $O(n^2)$ and $O(1)$, respectively. UP-Oracle-NoEdgPru satisfies $|\Pi_{\textit{UP-Oracle-NoEdgPru}}(s,t|T)| = |\Pi(s,t|T)|$ for all pairs of POIs s and t in P for the P2P query, where $\Pi_{\textit{UP-Oracle-NoEdgPru}}(s,t|T)$ is the shortest path of UP-Oracle-NoEdgPru between s and t.*

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

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

Thirdly, we prove the *output size* of *UP-Oracle-NoEdgPru* for the P2P query. Since there are $O(n^2)$ edges in *UP-Oracle-NoEdgPru* for the P2P query, so the output size of *UP-Oracle-NoEdgPru* for the P2P query is $O(n)$.

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

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

In general, we finish the proof of the oracle construction time, oracle update time, output size, shortest path query time and error bound of *UP-Oracle-NoEdgPru* for the P2P query. ☐

Theorem 12. *The oracle construction time, oracle update time, output size and shortest path query time of UP-Oracle-NoEffEdgPru for the P2P query are $O(nN\log^2 N)$, $O(N\log^2 N + n^3\log n)$, $O(n)$ and $O(\log n)$, respectively. UP-Oracle-NoEffEdgPru satisfies $|\Pi_{\textit{UP-Oracle-NoEffEdgPru}}(s,t|T)| \leq (1+\epsilon)|\Pi(s,t|T)|$ for all pairs of POIs s and t in P for the P2P query, where $\Pi_{\textit{UP-Oracle-NoEffEdgPru}}(s,t|T)$ is the shortest path of UP-Oracle-NoEffEdgPru between s and t.*

Proof. Firstly, we prove the *oracle construction time* of *UP-Oracle-NoEffEdgPru* for the P2P query. The oracle construction of *UP-Oracle-NoEffEdgPru* for the P2P query is similar in *UP-Oracle* for the P2P query. So the oracle construction time of *UP-Oracle-NoEffEdgPru* for the P2P query is $O(nN\log^2 N)$.

Secondly, we prove the *oracle update time* of *UP-Oracle-NoEffEdgPru* for the P2P query. For the oracle update time, the only difference between *UP-Oracle* for the P2P query and *UP-Oracle-NoEffEdgPru* for the P2P query is that the latter one uses algorithm *GreSpan* for the sub-graph generation step. In the sub-graph generation step, since there are n vertices in *UP-Oracle-NoEffEdgPru*, so answering the shortest path query using Dijkstra's algorithm on *UP-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 *UP-Oracle-NoEffEdgPru* for the P2P query is $O(N \log^2 N + n^3 \log n)$.

Thirdly, we prove the *output size* of *UP-Oracle-NoEffEdgPru* for the P2P query. According to [46], we know that the output graph of algorithm *GreSpan*, i.e., *UP-Oracle-NoEffEdgPru*, has $O(n)$ edges. So, the output size of *UP-Oracle-NoEffEdgPru* for the P2P query is $O(n)$.

Fourthly, we prove the *shortest path query time* of *UP-Oracle-NoEffEdgPru* for the P2P query. 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 *UP-Oracle-NoEffEdgPru* for the P2P query is $O(\log n)$.

Fifthly, we prove the *error bound* of *UP-Oracle-NoEffEdgPru* for the P2P query. The error bound of *UP-Oracle-NoEffEdgPru* is due to the error bound of algorithm *GreSpan*. Let $V_{\mathit{UP\text{-}Oracle\text{-}NoEffEdgPru}}$ and $E_{\mathit{UP\text{-}Oracle\text{-}NoEffEdgPru}}$ be the set of vertices and edges of *UP-Oracle-NoEffEdgPru*. In algorithm *GreSpan*, consider any edge $e_{\mathit{UP\text{-}Oracle\text{-}NoEffEdgPru}_t}(s,t|T) \in G'.E$ between two vertices s and t which is not added to *UP-Oracle-NoEffEdgPru*. Since $e_{\mathit{UP\text{-}Oracle\text{-}NoEffEdgPru}_t}(s,t|T)$ is discarded, it implies that $|\Pi_{\mathit{UP\text{-}Oracle\text{-}NoEffEdgPru}}(s,t|T)| \leq (1+\epsilon)|\Pi(s,t|T)|$. Since $|\Pi(s,t|T)| = |\Pi(s,t|T)|$, so on the output graph of algorithm *GreSpan*, i.e., *UP-Oracle-NoEffEdgPru*, we always have $|\Pi_{\mathit{UP\text{-}Oracle\text{-}NoEffEdgPru}}(s,t|T)| \leq (1+\epsilon)|\Pi(s,t|T)|$ for all pairs of vertices s and t in $V_{\mathit{UP\text{-}Oracle\text{-}NoEffEdgPru}}$. Thus, we have the error bound of *UP-Oracle-NoEffEdgPru*, i.e., *UP-Oracle-NoEffEdgPru* satisfies $|\Pi_{\mathit{UP\text{-}Oracle\text{-}NoEffEdgPru}}(s,t|T)| \leq (1+\epsilon)|\Pi(s,t|T)|$ for all pairs of POIs s and t in P for the P2P query.

In general, we finish the proof of the oracle construction time, oracle update time, output size, shortest path query time and error bound of *UP-Oracle-NoEffEdgPru* for the P2P query. □

Theorem 13. *The shortest path query time of CH-Fly-Algo [26] is $O(N^2)$. CH-Fly-Algo returns the exact shortest path for all pairs of POIs s and t in P.*

Proof. The proof can be found in the work [26]. □

Theorem 14. *The shortest path query time of K-Fly-Algo [6] 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 $|\Pi_{\mathit{K\text{-}Fly\text{-}Algo}}(s,t|T)| \leq (1+\epsilon)|\Pi(s,t|T)|$ for all pairs of POIs s and t in P, where $\Pi_{\mathit{K\text{-}Fly\text{-}Algo}}(s,t|T)$ is the shortest path of K-Fly-Algo between s and t.*

Proof. The proof of the *shortest path query time* and *error bound* of *K-Fly-Algo* is in [6]. Note that in Section 4.2 of [6], 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 [6], 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 $O(\frac{l_{max} N}{\epsilon l_{min} \sqrt{1-\cos\theta}} \log(\frac{l_{max} N}{\epsilon l_{min} \sqrt{1-\cos\theta}}))$. □