Efficiently Finding Shortest Path on 3D Weighted Terrain Surface

Anonymous
Anonymous
Anonymous

Anonymous
Anonymous
Anonymous

ABSTRACT

Nowadays, the rapid development of computer graphics technology and geo-spatial positioning technology promotes the growth of using the digital terrain data. Studying the shortest path query on terrain data has aroused widespread concern in industry and academia. In this paper, we propose an efficient method for the *weighted region problem* on a three-dimensional (3D) weighted terrain surface. Specifically, the weighted region problem aims to find the shortest path between two points passing different regions on the terrain surface and different regions are assigned different weights. Since it has been proved that, even in a two-dimensional (2D) environment, there is no exact algorithm for finding the exact solution of the weighted region problem efficiently when the number of faces in the terrain is greater than two, we propose a $(1+\epsilon)$-approximate efficient method to solve it on the terrain surface. In both the theoretical and practical analysis, our algorithm gives a shorter running time and less memory usage compared with the best-known algorithm.

ACM Reference Format:
Anonymous and Anonymous. 2023. Efficiently Finding Shortest Path on 3D Weighted Terrain Surface. In *Proceedings of 2024 International Conference on Management of Data (SIGMOD '24).* ACM, New York, NY, USA, 26 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In recent years, the digital terrain data becomes increasingly widespread in industry and academia [40]. In industry, many existing commercial companies/applications, such as Metaverse [6, 31, 32], Cyberpunk 2077 (a popular three-dimensional (3D) computer game) [2] and Google Earth [5], are using terrain data of objects such as mountains, valleys, and hills with different features (e.g., water and grassland) to help users reach the destination faster. In academia, researchers paid considerable attention to studying shortest path queries on terrain datasets [18, 25, 26, 34, 37, 38, 41, 42]. A terrain surface is represented by a set of *faces* each of which corresponds to a triangle. Each face (or triangle) has three line segments called *edges* connected with each other at three *vertices*. Figure 1 shows an example of a terrain surface. The *weighted shortest path* on a terrain refers to the shortest path between a source point s and a destination point t that passes the face on the terrain where each face is assigned with a *weight*, and the *unweighted shortest path* refers to the shortest path between s and t where each face weight is set to a fixed value (e.g., 1). In Figure 1, the sequence of consecutive blue (resp. pruple dashed) line segments denotes the weighted (resp. unweighted) shortest path from s to t on this terrain surface.

SIGMOD '24, June 11–16, 2024, Santiago, Chile

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1.1 Motivation

Given a source point s and a destination point t, computing the weighted shortest path on the terrain surface between s and t with different meanings of the face weights is involved in numerous applications, including autonomous vehicles' obstacle avoidance path planning, human's overland route-recommendation systems and laying pipelines or electrical cables [14, 23, 24, 28, 42, 43]. In Figure 1, a robot wants to move on a 3D terrain surface from s to t which consists of water (the faces with a blue color) and grassland (the faces with a green color), and avoid passing through the water. We could set the terrain faces corresponding to water (resp. grassland) with a larger (resp. smaller) weight. So, the weighted length of the path that passes water is larger, and the robot will choose the path that does not pass water (i.e., the weighted length is smaller). In addition, in a real-life example for placement of undersea optical fiber cable on the seabed, i.e., a terrain, we aim to minimize the weighted length of the cable for cost saving (over 1.35×10^5km of undersea cables have been constructed nowadays [27]). For the regions with a deeper sea level, the hydraulic pressure is higher, and the cable's lifespan is reduced, so it is more expensive to repair and maintain the cable [15]. We set the terrain faces for this type of regions with a larger weight. So, we could avoid placing the cable on these regions, and reduce the cost. The motivation study in our experiment shows that the total estimated cost of the cable for following the weighted shortest path and the unweighted shortest path are USD $366B and $438B, respectively, which shows the usefulness of the weighted shortest path. Motivated by these, we aim to find the shortest path on a 3D terrain surface between two points passing different regions on the terrain surface and different regions are assigned with different weights, and this problem is called the *weighted region problem*. The weight on the terrain surface is usually set according to the problem.

1.2 Challenges

Consider a terrain T with n vertices. Let V, E, and F be the set of vertices, edges, and faces of the terrain, respectively. Solving the 3D weighted region problem is very challenging due to four reasons.

1.2.1 ***Different from unweighted case.*** Solving the 3D weighted region problem is very different from calculating the unweighted shortest path in 3D. When calculating the *exact* unweighted shortest path in 3D, the state-of-art solution is to unfold the 3D terrain surface into a 2D terrain, and connect the source point s and destination point t using a line segment on this 2D terrain [16]. But, this does not apply to the weighted case. Even in 2D

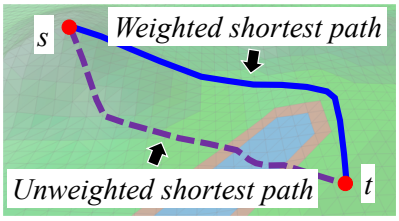

Figure 1: A terrain surface, unweighted and weighted shortest path

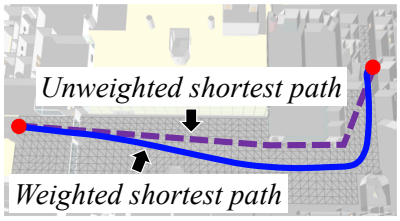

Figure 2: An example of paths in Path Advisor

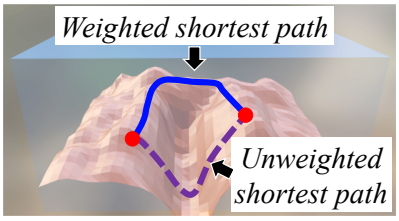

Figure 3: An example of paths on seabed

(instead of 3D), consider a light ray passing the boundary of two different media (e.g., air and water), due to the fact that the light seeks the path with minimum time (i.e., *Fermat's Principle* [9]), it will bend at the boundary of two different media, and the angles of incidence and refraction for this light satisfy one widely known fact from physics, i.e., Snell's law [11]. Thus, even in 2D, when calculating the *exact* weighted shortest path, if the path passes the boundary between two faces with different weights, it will not result in a straight line, and will bend when it crosses an edge in E every time [35]. The blue line in Figure 4 is the exact weighted shortest path that satisfies Snell's law from s to t that passes edges $e_1, \cdots e_l$ in order. So, we cannot use a straight line segment to connect s and t in the unfolded 2D terrain (i.e., the idea in the unweighted case) for solving the weighted region problem.

1.2.2 ***No exact solution.*** To the best of our knowledge, there is no *exact* solution for solving the weighted region problem when the number of faces in the terrain is larger than two, and most (if not all) existing algorithms aim to calculate the weighted shortest path on the weighted region problem *approximately* [17]. In Figure 4, given two points s and t, on the first edge e_1 opposite to s, if we could find a point c on e_1, such that there is a path starting from s, and then passing c, and then following Snell's law when it crosses an edge, and finally going through t, with the minimum distance, then we can find the *exact* weighted shortest path, where this c is called *optimal point*. One may assume that we can set the position of c to be unknown, then solve it using a polynomial equation. But, [17] shows that even if the exact weighted shortest path needs to cross only three faces, the equation will contain unknown with a degree of six, which cannot be solved using Algebraic Computation Model over the Rational Numbers (ACM$\mathbb{Q}$).

1.2.3 ***Limitations in existing work.*** All existing algorithms [25, 28, 30, 35, 39] for computing the *approximate* weighted shortest path on a terrain surface are either very slow (even when the terrain dataset has a moderate size), or cannot guarantee on the quality of the result path returned (in terms of the length of the path) with a given time limit and a given maximum memory. There are three categories of existing algorithms for solving the weighted region problem approximately: (1) *Continuous wavefront* approach [35], (2) *heuristics* approach [28, 39], and (3) *Steiner point* approach [25, 30].

(1) The *Continuous wavefront* approach exploits Snell's law and continuous Dijkstra algorithm to calculate an approximate weighted shortest path. But, its running time is $O(n^8 \log c_1)$, which is very large, where c_1 is a constant depending on the error and some geometry information of T. This is because when we calculate a path that hits a vertex in V, if we still want to utilize Snell's law for the path after it exits this vertex, we no longer have complete information about where it goes next. [35] uses a continuous Dijkstra algorithm to check all the combinations of the cases when a path hits a vertex that may happen on T, and thus, their running time is very large. As we will discuss later, if we could find a sequence of edges S that the optimal weighted shortest path passes, then uses Snell's law on S to find the result path, then the running time could be reduced (this is the technique used in our algorithm).

(2) The *heuristics* approach uses simulated annealing [28] and genetic algorithm [39] to calculate the weighted shortest path, but it cannot guarantee on the quality of the result path returned (in terms of the length of the path) with a given time limit and a given maximum memory.

(3) The *Steiner point* approach places discrete points (i.e., Steiner points) on edges in E, and then uses Dijkstra algorithm on a weighted graph constructed using these Steiner points and V, to calculate the weighted shortest path. It runs in $O(n^3 \log n)$, and [25, 30] are regarded as the best-known existing works for solving the weighted region problem. But, [25, 30] (3a) do not utilize any geometry information on T *between two adjacent faces in* F *that share one edge*, i.e., Snell's law, so they need to place tremendous Steiner points (and each two Steiner points on the same edge are very close to each other) on edges in E, and (3b) do not utilize any geometry information on T *for each face in* F, e.g., the weight of a face, the internal angle of a face, the length of an edge, etc., so they always place the same numbers of Steiner points on each edge in E (i.e., for the edges with a longer length and with a shorter length, they place the same number of Steiner points on them, but there is no need to place too many Steiner points on the edges with a shorter length), so their running time and memory usage are still very large. Our experimental result shows that for a terrain with 50k faces, our algorithm just needs to place about 10 Steiner points per edge to find a rough path in 71s (≈ 1.2 min), then we can use Snell's law on the edge sequence passed by the rough path to calculate a refined path in 2s, but the best-known algorithm [25, 30] needs to place more than 600 Steiner points per edge to find the result path in 119,000s (≈ 1.5 days) under the same error.

1.2.4 ***Inefficiency for constrained programming.*** It is also inefficient to find the *approximate* weighted shortest path even using constrained programming. In Figure 4, by setting the position of c to be unknown, then using a polynomial equation to denote the length of the path starting from s, passing c, and following Snell's law, we can use constrained programming to minimize the length

of this path. Although we can use Newton's method [10] to find the approximate solution, the running time could be large since the degree of the unknown is large, and it will increase when the number of faces crossed by the path increases (e.g., the equation will contain unknown with a degree of six even if the weighted shortest path needs to cross only three faces).

1.3 Contribution & Organization

Motivated by these, we propose an efficient two-step algorithm to calculate the weighted shortest path in the 3D weighted region problem using algorithm *Roug-Ref*, such that for a given source point s and destination point t on T, our algorithm returns a path with $(1+\epsilon)$-approximation of the weighted shortest distance between s and t without unfolding any face in the given terrain surface, where ϵ is a non-negative real user parameter for controlling the error ratio, called the *error parameter*. Our algorithm does not belong to any categories of approach as mentioned in Section 1.2. Specifically, (1) in algorithm *Roug*, given a terrain surface T, s and t on T, and error parameter ϵ, we aim to efficiently find a rough path between s and t with error guarantee $(1+\eta\epsilon)$, where $\eta > 1$ is a constant, and it is *reversed* calculated based on T and ϵ. The reason why we first find a rough path is that we would like to quickly reduce the weighted shortest path searching region from the whole T to some faces, edges, or vertices on T. (2) In algorithm *Ref*, given the rough path, we aim to efficiently refine this path to be a $(1+\epsilon)$-approximated weighted shortest path. The reason why we can efficiently refine the rough path is that we consider one geometry information of T between two adjacent faces in F that share one edge, i.e., Snell's law, such that the best-known existing works [25, 30] do not consider. We then summarize our major **contributions**.

(1) To the best of our knowledge, we are the first to propose the novel and efficient algorithm *Roug-Ref* that calculates a $(1+\epsilon)$-approximated weighted shortest path, since there is no existing work that uses the rough-refine idea for calculating the weighted shortest path within the error bound. It is worth mentioning that in algorithm *Ref*, even though we can use Snell's law to efficiently refine a rough path to be a $(1+\epsilon)$-approximated weighted shortest path in more than 99% cases, it may still happen that Snell's law fails to refine a rough path within the error bound. For the 1% cases, an additional step is required for error bound guarantee. So it is challenging to design algorithm *Ref*, such that even for the 1% cases, the total running time of algorithm *Roug-Ref* is still similar to the adapted best-known algorithm (note that the best-known algorithm [25, 30] is still very slow, we use a different but efficient Steiner point placement scheme to adapt it, i.e., we consider the geometry information on T for each face in F, e.g., the weight of a face, the internal angle of a face, the length of an edge, etc., that [25, 30] does not consider). Furthermore, we also have different novel techniques to efficiently reduce the algorithm running time and memory usage, i.e., (1) by further reducing the searching region in algorithm *Ref*, in addition to algorithm *Roug*, using *progressive* idea, and (2) by considering one more geometry information in algorithm *Ref*, called *effective weight*, for pruning.

(2) We provide a thorough theoretical analysis on the running time, memory usage, and error bound of our algorithm.

(3) Our algorithm performs much better than the best-known existing work [25, 30] in terms of running time and memory usage, and our algorithm is suitable for real-life applications that require real-time responses. Our experimental results show that our algorithm runs up to 1630 times faster than the best-known algorithm [25, 30] on benchmark real datasets with the same error ratio. In addition, for a terrain with 50k faces with $\epsilon = 0.1$, our algorithm's total query time is 73s ($\approx$ 1.2 min), and total memory usage is 43MB, but the best-known existing work's [25, 30] total query time is 119,000s ($\approx$ 1.5 days), and total memory usage is 2.9GB. For a real-time map application in our user study in Section 5.3.1, our algorithm just needs 0.1s to calculate the result.

The remainder of the paper is organized as follows. Section 2 provides the preliminary. Section 3 presents our algorithm. Section 4 shows the related work and baseline algorithms. Section 5 presents the experimental results and Section 6 concludes the paper.

2 PRELIMINARY

2.1 Problem Definition

Consider a terrain T. Let V, E, and F be the set of vertices, edges, and faces of the terrain, respectively. Let n be the number of vertices of T (i.e., $n = |V|$). Each vertex $v \in V$ has three coordinate values, denoted by x_v, y_v and z_v. If two faces share a common edge, they are said to be adjacent. Each face $f_i \in F$ is assigned a weight w_i, which is a positive real number, and the weight of an edge is equal to the smaller weight of the face that contains that edge. Given a face f_i, and two points p and q on f_i, we define $d(p,q)$ to be the Euclidean distance between point p and q on f_i, and $D(p,q) = w_i \cdot d(p,q)$ to be the weighted surface distance from p to q on f_i. Given two points s and t, the weighted region problem aims to find the optimal weighted shortest path $\Pi^*(s,t) = (s, r_1, \ldots, r_l, t)$, with $l \geq 0$, on the surface of T such that the weighted distance $\sum_{i=0}^{l} D(r_i, r_{i+1})$ is minimum, where $r_0 = s$, $r_{l+1} = t$, each r_i for $i \in \{1, \ldots, l\}$ is named as a *intersection point* in $\Pi^*(s,t)$, and it is a point on an edge in E. The blue line in Figure 4 shows an example of $\Pi^*(s,t)$ on a terrain surface. We define $|\cdot|$ to be the weighted distance of a path (e.g., $|\Pi^*(s,t)|$ is the weighted distance of $\Pi^*(s,t)$). Given a face f_i, and two points p and q on f_i, we define $\overline{pq}$ to be a line segment on f_i.

Depending on whether s and t are in V, there are two types of queries, (1) *vertex-to-vertex (V2V) path query*, i.e., both s and t are in V, and (2) *arbitrary point-to-arbitrary point (A2A) path query*, i.e., both s and t are two arbitrary points on the surface of T. Answering the V2V path query is more general than answering the A2A path query. This is because if s or t is not in V, we could simply make them as vertices by adding new triangles between them and their neighbour vertices in T, and then the A2A path query could be regarded as one form of the V2V path query. Thus, for clarity, in the main body of this paper, we focus on the V2V path query. We study the A2A path query in the appendix. A notation table could be found in the appendix of Table 2.

2.2 Snell's Law

In Figure 4, let $S = ((v_1, v_1'), \ldots, (v_l, v_l')) = (e_1, \ldots, e_l)$ be a sequence of edges that $\Pi^*(s,t)$ connects from s to t in order based on T, and S is said to be *passed by* $\Pi^*(s,t)$. Let $F(S) = (f_0, f_1, \ldots, f_{l-1}, f_l)$ be a sequence of adjacent faces with respect

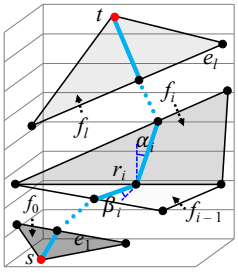

Figure 4: An example of $\Pi^*(s,t)$

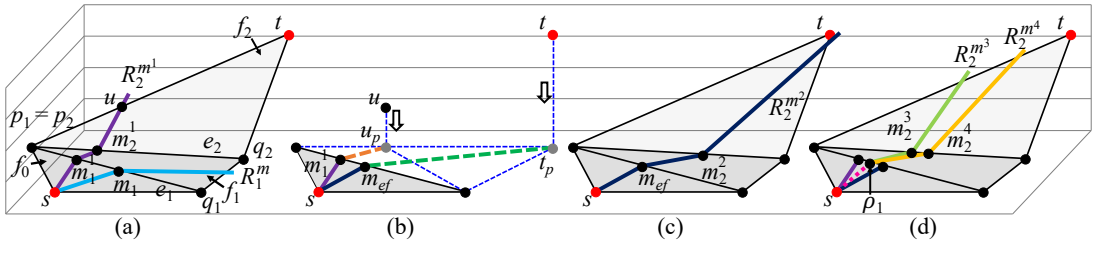

Figure 5: Snell's law path refinement step in *Ref* (a) with initial ray for calculating effective weight on the effective face $\triangle u_p p_1 q_1$, (b) for calculating m_{ef} using the weight of f_0 and the effective weight of $\triangle u_p p_1 q_1$, (c) with final ray passing through m_{ef}, and (d) processing on the remaining edges

to S such that for every f_i with $i \in \{1, \dots l-1\}$, f_i is the face containing e_i and e_{i+1} in S, while f_0 is the adjacent face of f_1 at e_1 and f_l is the adjacent face of f_{l-1} at e_l. Note that s and t are two vertices of f_0 and f_l. Let $W(S) = (w_0, w_1, \dots, w_{l-1}, w_l)$ be a weight list with respect to $F(S)$ such that for every w_i with $i \in \{0, \dots l\}$, w_i is the face weight of f_i in $F(S)$. We define α_i and β_i to be the angles of incidence and refraction of $\Pi^*(s,t)$ on e_i for $i \in \{1, \dots l\}$, respectively. Proposition 2.1 states Snell's law with $i \in \{1, \dots, l\}$.

Proposition 2.1. *$\Pi^*(s,t)$ has $w_i \cdot \sin\alpha_i = w_{i-1} \cdot \sin\beta_i$.*

3 METHODOLOGY

3.1 Overview

In our two-step algorithm, given a terrain $T = (V, E, F)$, two vertices s and t in V, and error parameter ϵ, we (1) first use algorithm *Roug* to calculate a rough path, the orange dashed line in Figure 6 (c), between s and t with error guarantee $(1+\eta\epsilon)$, i.e., the **rough path calculation** step, where the error $\eta\epsilon$ is *reversely* calculated using T and ϵ, and (2) then use algorithm *Ref* to efficiently refine this path and calculate a $(1+\epsilon)$-approximated weighted shortest path, the green line in Figure 6 (e), using Snell's law, i.e., the **Snell's law path refinement** step. The reason why we first find a rough path is that we need to obtain an edge sequence S passed by the rough path based on T, then we can apply Snell's law on S. If we do not know S, we need to use Snell's law on different combinations of edges in E, which is time-consuming. Then, by using Snell's law on S, we can obtain a refined path that is very close to the optimal weighted shortest path $\Pi^*(s,t)$, i.e., the distance between the intersection point of the refined path on each edge of S and the intersection point of $\Pi^*(s,t)$ on each edge of S is smaller than a user parameter δ, where δ is used for controlling the error and proportional depending on ϵ, then there is no need to place tremendous Steiner points on edges in E to calculate the result path as in the best-known algorithm [25, 30]. In Figure 6 (e), we can use Snell's law on the edge sequence S (edges in red) passed by the rough path, to obtain a refined path (the green line) on S. Next, we focus on two cases that whether the refined path is within the $(1+\epsilon)$ error bound.

Cases for error bound guarantee: In algorithm *Ref*, see Figure 6 (f), if the distance of the refined path after utilizing Snell's law is smaller than $\frac{(1+\epsilon)}{(1+\eta\epsilon)}$ times the distance of the rough path, then the refined path is guaranteed to be a $(1+\epsilon)$-approximated weighted shortest path (because we could show that the rough path is guaranteed to be a $(1+\eta\epsilon)$-approximated weighted shortest path), so we can directly return the refined path and terminate our algorithm (this happens 99% theoretically, and happens 100% in our experiment). If not (this only happens when the edge sequence S passed by the rough path is not the same as the edge sequence passed by the optimal weighted shortest path), we need one more step, i.e., **error guaranteed path refinement** step in algorithm *Ref*, to guarantee the error bound, see Figure 6 (g). Even with this step, we hope that the total running time of algorithm *Roug-Ref* is still similar to the best-known algorithm [25, 30]. We achieve this by considering one more information, called *state information*, calculated in the rough path calculation step in algorithm *Roug*.

Major idea in the rough path calculation step in algorithm *Roug* and the error guaranteed path refinement step in algorithm *Ref*: The idea for these two steps (see Figure 6 (c) and (g)) are similar, i.e., we place Steiner point on each edge in E, and apply Dijkstra algorithm [20] on a weighted graph constructed by these Steiner points and V to calculate the corresponding path. The only difference is that the input error of the error guaranteed path refinement step in algorithm *Ref* is ϵ, but the input error of the rough path calculation step in algorithm *Roug* is $\eta\epsilon$.

(1) In the **rough path calculation** step in algorithm *Roug*, we first need to calculate $\eta\epsilon$ as input for this step. So, in algorithm *Roug*, in Figure 6 (b), we calculate η by first placing Steiner points on each edge in E with input error ϵ, then from two endpoints of each edge to the center of this edge, whenever we encounter a Steiner point, we keep one and remove the next one iteratively. With the remaining Steiner points, we calculate $\eta\epsilon$ *reversely*. As such, the remaining Steiner points used in the rough path calculation step in algorithm *Roug* is a subset of the Steiner points used in the error guaranteed path refinement step in algorithm *Ref*. So, when using Dijkstra algorithm to calculate the rough path from s to t in the rough path calculation step in algorithm *Roug*, we could store the shortest distance from s to these Steiner points, and each Steiner point's previous point along the shortest path starting from s, which are the *state information*. In Figure 7, the blue points are the remaining Steiner points in algorithm *Roug*, the shortest distance from s to c is $s \to a \to c$ with distance $5+2=7$ and the previous point of c along this shortest path is a, i.e., these are the state information (where the number next to a and b are the shortest distance from s to a and b, respectively).

(2) In the **error guaranteed path refinement** step in algorithm *Ref*, in Figure 7, we add back the removed Steiner points, so the blue

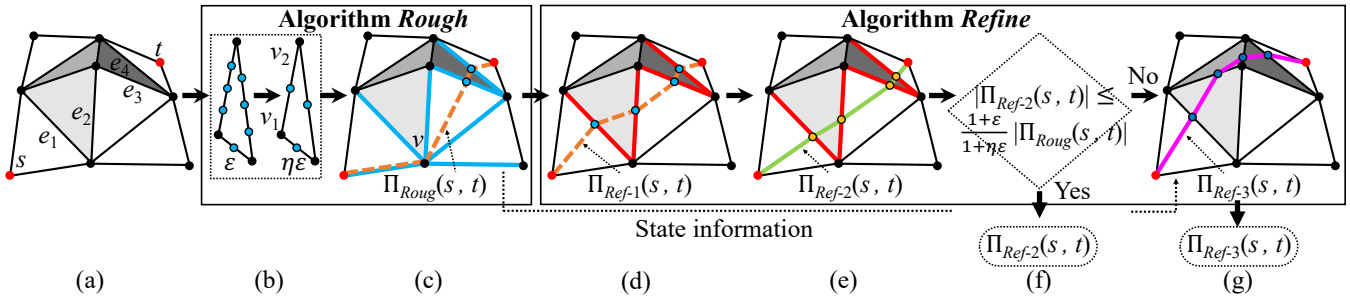

Figure 6: Framework overview

and green points are the original Steiner points. Suppose that in the priority queue of Dijkstra algorithm, we are ready to dequeue the edge contains b, since the shortest distance from s to b is 4.7, which is smaller than a with the shortest distance 5. When we follow the path $s \to b \to c$, the distance from s to c is $4.7 + 3 = 7.7 > 7$. So, we know that the shortest path from s to c will not pass b, and we do not need to insert c and $\overline{bc}$ with distance 3 into the queue, which saves the inqueue and dequeue time. It is easy to verify that the case that we do not insert a Steiner point into the queue in the error guaranteed path refinement step in algorithm *Ref*, is exactly the same as the case that we perform Dijkstra algorithm on the remaining Steiner points in the rough path calculation step in algorithm *Roug*. Thus, the total running time of the rough path calculation step in algorithm *Roug* and the error guaranteed path refinement step in algorithm *Ref*, is exactly the same as the running time that we perform Dijkstra algorithm on the weighted graph constructed by the original Steiner points and V.

It is worth mentioning that even if our experimental result shows that in 100% cases, there is no need to use the error guaranteed path refinement step, it does not mean the useless of the efficient technique in this step. During the calculation of η in algorithm *Roug*, whenever we encounter a Steiner point, we can keep one and remove the next k Steiner points iteratively. The value of k is called *removing value*. When the removing value is larger, then more Steiner points are removed, and η is larger, so the chance that the checking result is false becomes larger, which means we need the error guaranteed path refinement step for error guarantee (our experimental result verifies that the optimal removing value is 2, since it will minimize the algorithm's total running time). By changing the removing value from 2 to a value larger than 2, our experimental result shows that the chance of using this error guaranteed path refinement step will increase from 0% to 100%.

3.1.1 ***Algorithm Roug overview.*** In algorithm *Roug*, given a terrain T, two vertices s and t in V, and ϵ, we aim to find a rough path between s and t with error guarantee $(1 + \eta\epsilon)$.

Details: We (1) first calculate η, i.e., η **calculation** in Figure 6 (b), and (2) then use an efficient Steiner point placement scheme to place Steiner points (instead of the one used in the best-known existing work [25, 30]), construct a weighted graph and apply Dijkstra algorithm to calculate a rough path between s and t with error guarantee $(1 + \eta\epsilon)$, i.e., **rough path calculation** in Figure 6 (c). The reason why our Steiner point placement scheme is efficient is that we consider the additional geometry information of T, e.g., the weight of a face, the internal angle of a face, the length of an edge, etc., so that we can place *different* numbers of Steiner points on different edges of E, and the interval between two adjacent Steiner points on the same edge could be *different*. This allows us to minimize the number of Steiner points per edge (and could also guarantee the error bound). Figure 9 (b) shows an example of our placement of Steiner points, where there are 8, 7, and 7 Steiner points on edge edge of f_i.

In contrast, the Steiner point placement scheme used in the best-known existing work [25, 30], called algorithm *Fixed Steiner Point placement scheme* (*FixSP*), will result in a large number of Steiner points per edge, and make their algorithm very slow. Because algorithm *FixSP* just places some evenly distributed Steiner points on every edge e_i in E, it does not consider any additional geometry information of T, so in order to bound the error, no matter how T looks like, it always needs to place $O(n^2)$ Steiner points per edge to bound the error [30]. But our Steiner point placement scheme only needs $O(n \log c_2)$ Steiner points per edge, where c_2 is a constant depending on the error and the geometry information of T. Figure 10 shows an example of the placement of Steiner points in algorithm *FixSP* (with the same error ratio as of Figure 9 (b)). Our experiment shows that applying Dijkstra algorithm on the weighted graph constructed using our Steiner point placement scheme just needs 219s ($\approx$ 3.6 min) on terrain with 50k faces with $\epsilon = 0.1$, but algorithm *FixSP* needs 119,000s ($\approx$ 1.5 days) under the same setting.

3.1.2 ***Algorithm Ref overview.*** In algorithm *Ref*, given the rough path, we aim to efficiently refine this path and calculate a $(1+\epsilon)$-approximated weighted shortest path. Before we introduce the detailed steps, we need one more concept. Recall that we use Snell's law on the edge sequence S passed by this rough path, for efficient refinement. But, as shown in Figure 6 (c), if the rough path passes a vertex v, when using Snell's law on the edge sequence S passed by this rough path, apart from the edges with lengths larger than 0 that passed by the rough path, we also need to add *all* the edges with v as one endpoint in S (edges in blue) for error guarantees. But, the fact is that we just need *some* edges with v as one endpoint and the edges with lengths larger than 0 when applying Snell's law, so we need to try different combinations of edge sequences that we obtained, which will significantly increase the running time. To handle this, we need one more step before the Snell's law path refinement step in algorithm *Ref*, i.e., we need

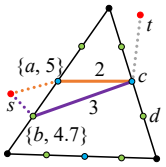

Figure 7: Pruned Dijkstra algorithm

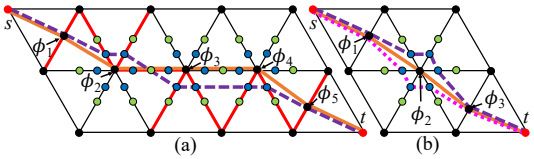

Figure 8: (a) Successive endpoint case, and (b) single endpoint case

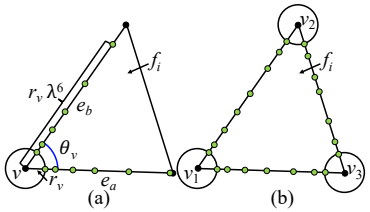

Figure 9: Steiner points on (a) e_a and e_b, and (b) f_i

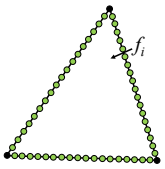

Figure 10: Steiner points on f_i in *FixSP*

to efficiently modify the rough path, such that we could directly obtain some edges with v as one endpoint and other edges with lengths larger than 0, i.e., all the edges passed by this modified rough path have lengths larger than 0, as shown in Figure 6 (d). Given an edge sequence $S = ((v_1, v_1'), \ldots, (v_l, v_l')) = (e_1, \ldots, e_l)$, S is said to be a *full edge sequence* if the length of each edge in S is larger than 0. In Figure 8 (a), given the path denoted as the sequence of consecutive purple dashed line segments between s and t, the edge sequence S_a (edges in red) passed by this path is an example of a full edge sequence since the length of each edge in S_a is larger than 0. Similarly, given an edge sequence S, S is said to be a *non-full edge sequence* if there exists at least one edge whose length is 0. In Figure 8 (a), given the path denoted as the sequence of consecutive orange line segments between s and t, the edge sequence S_b passed by this path is an example of a non-full edge sequence since the edge length at (ϕ_2, ϕ_2), (ϕ_3, ϕ_3) and (ϕ_4, ϕ_4) are 0. Then, we introduce the detailed steps.

Details: We (1) first modify the rough path to convert its corresponding edge sequence from a non-full edge sequence to a full edge sequence progressively to further reduce the number of edges in the edge sequence passed by the rough path based on T, i.e., **full edge sequence conversion** in Figure 6 (d), (2) then use Snell's law to efficiently refine the modified rough path using its corresponding edge sequence, i.e., **Snell's law path refinement** in Figure 6 (e), (3) next check whether the refined path satisfy error requirement, i.e., **path checking** in Figure 6 (f), if satisfy, we return the refined path as output, (4) if not satisfy, we finally add back the removed Steiner point, use Dijkstra algorithm and state information on a new weighted graph constructed by these Steiner points and V to efficiently calculate a $(1+\epsilon)$-approximated weighted shortest path, i.e., **error guaranteed path refinement** in Figure 6 (g). We have illustrated the major idea of the reason why the error guaranteed path refinement step is efficient. We then give the major idea for the full edge sequence conversion and Snell's law path refinement step, which involves another two novel techniques.

(1) In the **full edge sequence conversion** step, in Figure 8 (a), given a rough path $(s, \phi_1, \phi_2, \phi_3, \phi_4, \phi_5, t)$ (i.e., the sequence of consecutive orange line segments between s and t) whose corresponding edge sequence is not a full edge sequence, we would like to obtain a modified rough path (i.e., the sequence of consecutive purple dashed line segments between s and t) whose corresponding edge sequence (edges in red) is a full edge sequence. We use the *progressive* idea to handle it. Specifically, we divide $(s, \phi_1, \phi_2, \phi_3, \phi_4, \phi_5, t)$ into a smaller segment $(\phi_1, \phi_2, \phi_3, \phi_4, \phi_5)$ such that all the edge sequence passed by this segment has edge length equal to 0, i.e., at (ϕ_2, ϕ_2), (ϕ_3, ϕ_3) and $(\phi_4, h\phi_4)$. We then add more Steiner points and use this step progressively to find a new path segment, i.e., the sequence of consecutive purple dashed line segments between ϕ_1 and ϕ_5, until the edge sequence (edges in red) passes by this path segment is a full edge sequence. If the distance of the new path segment is smaller than the distance of the original path segment, we replace the new one with the original one, and obtain the modified rough path. Since we do not want the rough path to pass the vertex, so it seems that in algorithm *Roug*, we can use Dijkstra algorithm on the weighted graph constructed only using the Steiner points (i.e., not using the Steiner points and V to construct the weight graph). But, in this case, we cannot guarantee the rough path to be a $(1+\eta\epsilon)$-approximated weighted shortest path.

(2) In the **Snell's law path refinement** step, in Figure 6 (e), given the modified rough path edge sequence S (edges in red), we aim to efficiently calculate a refined path using S. The basic idea is to use binary search to find the optimal point of the weighted shortest path by utilizing Snell's law on each edge in S, then connect these points to form the refined path. But, we introduce an efficient pruning step, such that by considering one useful information on T, called *effective weight*, we could reduce the total number of iterations in the binary search, and reduce the algorithm's running time. Specifically, in Figure 5 (a), for the first edge e_1 in S that is opposite to s, we select the midpoint m_1 on e_1, and trace a blue light ray that follows Snell's law from s to m_1, then this light ray will follow S, and bend at each edge in S. We check whether t is on the left or right of this ray, and modify the position of m_1 to be left or right, accordingly, and iterate this procedure once the light ray passes the whole S. Now, we have the purple light ray s and passes m_1^1 passes the S for the first time, then we can use effective weight to prune out unnecessary checking. In Figure 5 (a) and (b), we regard all the faces except the first face (i.e., f_1 and f_2) in $F(S)$ as one *effective face* (i.e., $\triangle u_p p_1 q_1$), and using the ray in the purple line for calculating effective weight of $\triangle u_p p_1 q_1$. Then, we can calculate the position of effective point m_{ef} on the first edge in S in one simple quartic equation using the weight of f_0 and the effective weight of $\triangle u_p p_1 q_1$. In Figure 5 (c), we find the ray starts from s and passes m_{ef} (i.e., the dark blue line), which is very close to t, implies that m_{ef} is very close to the optimal point. We iterate the midpoint selection step until (1) the light ray hits t or (2) the distance between new m_1 and previous m_1 is smaller than δ. In Figure 5 (d), after the processing on e_1, we continue on other edges in S, and we have the ray starts from ρ_1 and passes m_2^3 (resp. m_2^4) (i.e., the green (resp. yellow) line), until all the edges in S have been processed.

3.2 Algorithm *Roug*

In algorithm *Roug*, given a terrain T, two vertices s and t in V, and ϵ, we aim to find a rough path between s and t with error guarantee $(1+\eta\epsilon)$. There are two steps involved (see Figure 6 (b) - (c)):

- η **calculation**: Placing Steiner points using ϵ, removing some Steiner points, and then calculating $\eta\epsilon$ reversely.
- **Roug path calculation**: Using the remaining Steiner points and V to construct a weighted graph, and then applying Dijkstra algorithm to calculate a $(1+\eta\epsilon)$-approximated rough path.

Notations: Given two points s and t in V, we define $\Pi_{Roug}(s,t)$ to be the rough path, i.e., the orange dashed line in Figure 6 (c), calculated using algorithm *Roug*. We define ϵ' to be the another error parameter for algorithm *Roug*, where $(2+\frac{2W}{(1-2\epsilon')\cdot w})\epsilon' = \epsilon$, and W and w are the maximum and minimum weight of all faces in F, respectively. Let L be the length of the longest edge of T, and N be the smallest integer value that is larger than or equal to the coordinate value of any vertex in V. Given a vertex v in V, we define h_v to be the minimum distance from v to the boundary of one of its incident faces. Given a vertex v in V, we define a polygonal cap around v, denoted as C_v, to be a *sphere* with center at v. Let $r_v = \epsilon' h_v$ be the radius of C_v with $0 < \epsilon' < \frac{1}{2}$. Let θ_v be the angle between any two edges of T that are incident to v. Let h, r and θ be the minimum h_v, r_v and θ_v for all $v \in V$, respectively. Figure 9 (a) shows an example of the polygonal cap C_v around v, with radius r_v, and the angle θ_v of v. Furthermore, we define SP_A to be a set of Steiner points on each edge of E constructed based on error ϵ *before* or *after* removing certain Steiner points, which will be used for Dijkstra algorithm in algorithm *Ref* or *Roug*, where A is a placeholder and could be *Roug* or *Ref*. In Figure 7, SP_{Roug} is denoted as three blue points, and SP_{Ref} is denoted as three blue points and six green points. Given a set of points $G_A.V = SP_A \bigcup V$ where A could be *Roug* or *Ref*, we construct a connected weighted graph $G_A = (G_A.V, G_A.E)$, such that for each point p and q in $G_A.V$, if p and q lie on the same face in F, then there is a weight edge $\overline{pq}$ connecting them, and a set of weight edges forms $G_A.E$. The weight for edge $\overline{pq}$ is $w_{pq} \cdot d(p,q)$, where w_{pq} means the weight of the face or edge that both p and q lie on. In Figure 7, $G_{Ref}.V$ is denoted as three blue points, six green points and three black points. The purple line is a weight edge $\overline{bc}$ with weight 3 in $G_{Ref}.E$.

Details: We introduce how to place Steiner points based on ϵ. Let $\lambda = (1+\epsilon'\cdot\sin\theta_v)$ if $\theta_v < \frac{\pi}{2}$, and $\lambda = (1+\epsilon')$ otherwise. For each vertex v of face f_i, let e_a and e_b be the edges of f_i incident to v. We place Steiner points $a_1, a_2, \ldots, a_{\tau_a-1}$ (resp. $b_1, b_2, \ldots b_{\tau_b-1}$) along e_a (resp. e_b) such that $|\overline{va_j}| = r_v\lambda^{j-1}$ (resp. $|\overline{vb_k}| = r_v\lambda^{k-1}$) where $\tau_a = \log_\lambda \frac{|e_a|}{r_v}$ (resp. $\tau_b = \log_\lambda \frac{|e_b|}{r_v}$) for every integer $2 \le j \le \tau_a - 1$ (resp. $2 \le k \le \tau_b - 1$). We repeat it on each edge of f_i. Figure 9 (a) and Figure 9 (b) show an example of Steiner points on e_a and e_b, and f_i based on ϵ, respectively. Knowing this, in Figure 6 (b), if we move from two endpoints v_1 and v_2 on each edge, whenever we encounter a Steiner point, we keep one and remove the next k iteratively, and we can derive $\eta\epsilon$ reversely using the remaining Steiner points. Then, we can use Dijkstra algorithm on G_{Roug} (constructed using the remaining Steiner points and V) to calculate $\Pi_{Roug}(s,t)$.

3.3 Algorithm *Ref*

In algorithm *Ref*, given the rough path, we aim to efficiently refine this path and calculate a $(1+\epsilon)$-approximated weighted shortest path. There are four steps involved (see Figure 6 (d) - (g)):

- **Full edge sequence conversion**: Modifying the rough path to convert its corresponding edge sequence from a non-full edge sequence to a full edge sequence S progressively.
- **Snell's law path refinement**: Using Snell's law to efficiently refine the modified rough path using S.
- **Path checking**: Returning the refined path as output if it satisfies error requirement.
- **Error guaranteed path refinement**: Otherwise, using SP_{Ref} and V, to construct a weighted graph, and then applying Dijkstra algorithm to calculate a $(1+\epsilon)$-approximated weighted shortest path with guarantee.

Notations: Given two points s and t in V, we define (1) $\Pi_{Ref\text{-}1}(s,t)$ to be the modified rough path, i.e., the orange dashed line in Figure 6 (d), which corresponding edge sequence is converted from a non-full edge sequence to a full edge sequence $S = (e_1, e_2, e_3, e_4)$ (edges in red), calculated using the full edge sequence conversion step of algorithm *Ref*, (2) $\Pi_{Ref\text{-}2}(s,t) = (s, \rho_1, \ldots, \rho_l, t)$ to be the refined path on S, i.e., the green line in Figure 6 (e), calculated using the Snell's law path refinement step of algorithm *Ref*, where each ρ_i for $i \in \{1, \ldots, l\}$ is an intersection point in $\Pi_{Ref\text{-}2}(s,t)$, and it is a point on an edge in E, and (3) $\Pi_{Ref\text{-}3}(s,t)$ to be the refined path, i.e., the purple line in Figure 6 (g), calculated using the error guaranteed path refinement step of algorithm *Ref*.

3.3.1 ***Full edge sequence conversion.*** Recall that the corresponding edge sequence of $\Pi_{Roug}(s,t)$ may be a non-full edge sequence, so we need the full edge sequence conversion step. The path $(s, \phi_1, \phi_2, \phi_3, \phi_4, \phi_5, t)$ in Figure 8 (a) (resp. path $(s, \phi_1, \phi_2, \phi_3, t)$ in Figure 8 (b)) is an example of $\Pi_{Roug}(s,t)$ that the corresponding edge sequence is non-full edge sequences since the edge length at (ϕ_2, ϕ_2), (ϕ_3, ϕ_3) and $(\phi_4, \phi_)$ (resp. (ϕ_2, ϕ_2)) is 0. The path denoted as the sequence of consecutive purple dashed line segments between s and t in Figure 8 (a) and (b) are two examples of $\Pi_{Ref\text{-}1}(s,t)$ since the corresponding edge sequence has been converted to a full edge sequence. We then introduce the process in this step.

Along the rough path $\Pi_{Roug}(s,t)$ from s to t, we check the path vertex by vertex, and let the current, next, and previous checking point in $\Pi_{Roug}(s,t)$ be v_c, v_n and v_p, respectively. Given a checking point v, v is on the edge means that the corresponding point in $\Pi_{Roug}(s,t)$ lies in the internal of an edge in E, v is on the original vertex in V means that the corresponding point in $\Pi_{Roug}(s,t)$ lies on the vertex in V. Depending on the type of v_c, there are two cases:

- If v_c is on the edge, we will not process it (e.g., $v_c = \phi_1$ in Figure 8 (a) or 8 (b)).
- If v_c is on the original vertex in V, there are two more cases:
  - **Successive endpoint** (see Figure 8 (a)): (1) If v_n is on the vertex and v_p is on the edge (e.g., $v_c = \phi_2$, $v_n = \phi_3$ and $v_p = \phi_1$), it means that there are at least two successive points in $\Pi_{Roug}(s,t)$ that is on the vertex. We store v_p as v_s (e.g., $v_s = \phi_1$), i.e., the start vertex of successive endpoint case. (2) If both v_n and v_p (e.g., $v_c = \phi_3$, $v_n = \phi_4$ and $v_p = \phi_2$) are on the vertex, we do nothing. (3) If v_n is on the edge and v_p is on the vertex (e.g., $v_c = \phi_4$, $v_n = \phi_5$ and $v_p = \phi_3$), it means we have finished

finding the successive endpoints. We store v_n as v_e (e.g., $v_n = \phi_5$), i.e., the end vertex of successive endpoint case. Then, from v_s to v_e, we double the number of Steiner points, and call algorithm *Ref* full edge sequence conversion step itself, and denote the new path as $\Pi_{Ref\text{-}1}(v_s, v_e)$, i.e., the sequence of consecutive purple dashed line segments between ϕ_1 and ϕ_5. We substitute $\Pi_{Roug}(v_s, v_e)$, i.e., the sequence of consecutive orange line segments between ϕ_1 and ϕ_5, as $\Pi_{Ref\text{-}1}(v_s, v_e)$, if $|\Pi_{Ref\text{-}1}(v_s, v_e)| < |\Pi_{Roug}(v_s, v_e)|$.

– **Single endpoint** (see Figure 8 (b)): If both v_n and v_p are on the edge, it means only v_c is on the vertex (e.g., $v_c = \phi_2$, $v_n = \phi_3$ and $v_p = \phi_1$). We add new Steiner points at the midpoints between v_c and the nearest Steiner points of v_c on the edges that adjacent to v_c. There are three possible ways to go v_p to v_n, which are (1) passes the original path $\Pi_{Roug}(v_p, v_n)$, i.e., the sequence of consecutive orange line segments between ϕ_1 and ϕ_3, (2) & (3) passes the set of newly added Steiner points on the left (resp. right) side of the path (v_p, v_c, v_n), which is $\Pi_l(v_p, v_n)$ (resp. $\Pi_r(v_p, v_n)$), i.e., the sequence of consecutive purple (resp. pink) dashed line segments between ϕ_1 and ϕ_3. We compare the weighted distance among them, and substitute $\Pi_{Roug}(v_p, v_n)$ as the path with the shortest weighted distance. We run this step for maximum ζ times (i.e., keep adding new Steiner points at the midpoints between v_c and the nearest Steiner points of v_c on the edges adjacent to v_c), if $\Pi_{Roug}(v_p, v_n)$ is still the longest path, where ζ is a constant and normally is set as 10.

Then, we move forward in $\Pi_{Roug}(s, t)$ by setting v_c to be v_n, and updating v_n and v_p accordingly. After we process all the vertices in $\Pi_{Roug}(s, t)$, we return $\Pi_{Ref\text{-}1}(s, t)$ and its corresponding edge sequence S based on T.

3.3.2 ***Snell's law path refinement***. We introduce some notations first. Given an edge sequence S, a source point s, and a point c_1 on $e_1 \in S$, we can obtain a *3D surface Snell's ray* $\Pi_c = (s, c_1, c_2, \ldots, c_g, R_g^c)$ starting from s, hitting c and following Snell's law on other edges in S, where $1 \le g \le l$, each c_i for $i \in \{1, \ldots, g\}$ is an intersection point in Π_c, and R_g^c is the last out-ray at $e_g \in S$. Figure 5 (a) shows an example of $\Pi_m = (s, m_1, R_1^m)$ (i.e., the blue line) that does not pass the whole $S = (e_1, e_2)$, and $\Pi_{m^1} = (s, m_1^1, m_2^1, R_2^{m^1})$ (i.e., the purple line) that passes the whole S. So, if we could find the point ρ_1 on e_1 such that the 3D surface Snell's ray $\Pi_\rho = (s, \rho_1, \ldots, \rho_l, R_l^\rho)$ which hits t, then Π_ρ is the result of $\Pi_{Ref\text{-}2}(s, t)$. Furthermore, given two sequence of points X and X', we define $X \oplus X'$ to be a new sequence of points X by appending X' to the end of X. In Figure 4, we have $\Pi^*(s, t) = \Pi^*(s, r_i) \oplus \Pi^*(r_i, t)$. We then introduce three sub-steps in this step.

Binary search initial path finding: We use binary search to find the initial path on S that follows Snell's law (see Figure 5 (a)). For $i = 1$, we let $[a_i, b_i]$ be a candidate interval on $e_i \in S$, let m_i be the midpoint of $[a_i, b_i]$, and initial set $a_i = p_i$ and $b_i = q_i$, where p_i and q_i are the left and right endpoint of e_i, respectively. Then, we can find the 3D surface Snell's ray $\Pi_m = (s, m_1, \ldots, m_g, R_g^m)$ with $g \le l$ (e.g., we have the blue line $\Pi_m = (s, m_1, R_1^m)$ when $i = 1$). Depending on whether Π_m pass the whole edge sequence S or not, there are two cases:

- Π_m does not pass the whole S (e.g., the blue line $\Pi_m = (s, m_1, R_1^m)$ when $i = 1$ does not pass the whole S): If e_{g+1} is on the left (resp. right) side of R_g^m, then we need to search in $[a_i, m_i]$ (resp. $[m_i, b_i]$), so we set $b_i = m_i$ (resp. $a_i = m_i$) for $i \in \{1, \ldots, g\}$. (E.g., e_2 is on the left side of the blue line R_1^m, we set $b_1 = m_1$, and we have $[a_1, b_1] = [p_1, m_1]$.) We iterate this step until Π_m passes the whole S based on T for the first time.
- Π_m passes the whole S (e.g., the purple line $\Pi_{m^1} = (s, m_1^1, m_2^1, R_2^{m^1})$ when $i = 1$ passes the whole S): If t is on R_g^m, then we could just return Π_m as the result. If t is on the left (resp. right) side of R_g^m, then we need to search in $[a_i, m_i]$ (resp. $[m_i, b_i]$), so we set $b_i = m_i$ (resp. $a_i = m_i$) for $i \in \{1, \ldots, l\}$. (E.g., t is on the right side of the purple line $R_2^{m^1}$, we set $a_1 = m_1^1$ and $a_2 = m_2^1$, and we have $[a_1, b_1] = [m_1^1, m_1]$ and $[a_2, b_2] = [m_2^1, q_2]$.)

Effective weight pruning: We use effective weight to prune out unnecessary checking (see Figure 5 (a)-(c)). For $i = 1$, suppose that we have found a 3D surface Snell's ray $\Pi_m = (s, m_1, \ldots, m_l, R_l^m)$ (e.g., the purple line $\Pi_{m^1} = (s, m_1^1, m_2^1, R_2^{m^1})$) that passes the whole S based on T for the first time.

- Firstly (see Figure 5 (a)), given two edges that are adjacent to t in the last face f_l in $F(S)$, we calculate the intersection point between R_l^m and these two edges (either one of these two edges), and denote it as u (e.g., the purple line $R_2^{m^1}$ intersect with the left edge $\overline{p_1 t}$ of f_2 that adjacent to t).
- Secondly (see Figure 5 (b)), we project u onto f_0 (i.e., the first face in $F(S)$) into 2D, and denote the projection point as u_p. Now, the whole $F(S)$ could be divided into two parts using e_1, (1) f_0, and (2) all the faces in $F(S)$ except f_0. For the latter one, we regard them as one *effective face* and denote it as f_{ef}, where the weight of f_{ef} is called *effective weight* and we denote it as w_{ef} (e.g., $f_{ef} = \triangle u_p p_1 q_1$ is an effective face for f_1 and f_2).
- Thirdly (see Figure 5 (b)), by using $\overline{sm_1^1}$ (e.g., the purple line), $\overline{m_1^1 u_p}$ (e.g., the orange dashed line), and the weight for f_0 (i.e., w_0), we could use Snell's law to calculate w_{ef}, i.e., the effective weight for f_{ef}.
- Fourthly (see Figure 5 (b)), we project t onto f_0 into 2D, and denote the projection point as t_p. We apply Snell's law again to find the effective intersection point m_{ef} on e_1 using w_0, w_{ef}, s and t_p in a quartic equation (note that only f_0 and f_{ef} are involved, so the equation will have the unknown at the power of four). Specifically, we set m_{ef} to be unknown and use Snell's law in vector form [8], then build a quartic equation using w_0, w_{ef}, $\overline{sm_{ef}}$ (e.g., the dark blue line $\overline{sm_{ef}}$) and $\overline{m_{ef} t_p}$ (e.g., the green dashed line $\overline{m_{ef} t_p}$). Then, we use root formula [7] to solve m_{ef}.
- Fifthly (see Figure 5 (c)), we compute the 3D surface Snell's ray Π_m from s with the initial ray through m_{ef} (e.g., the dark blue line $\Pi_{m^2} = (s, m_{ef}, m_2^2, R_2^{m^2})$), and update $[a_i, b_i]$ for $i \in \{1, \ldots, l\}$ depending on whether Π_m passes the whole S based on T and whether t (or e_{g+1}) is on the left or right side of R_l^m (or R_g^m) for $i \in \{1, \ldots, l\}$ (or for $i \in \{1, \ldots, g\}$) (e.g., the dark blue line Π_{m^2} passes the whole S and t is on the left side of the dark blue line $R_2^{m^1}$, we set $b_1 = m_{ef}$ and $b_2 = m_2^2$, and we have $[a_1, b_1] = [m_1^1, m_{ef}]$ and $[a_2, b_2] = [m_2^1, m_2^2]$).

Binary search refined path finding: We use binary search iteratively to find the refined path on S (see Figure 5 (d)). We perform binary search iteratively, until $|a_ib_i| < \delta$ (e.g., $|a_1b_1| = |m_1^1 m_{ef}| < \delta$ when $i = 1$), where $\delta = \frac{h\epsilon w}{6lW}$, h is the minimum height of any face in F, W and w are the maximum and minimum weights of face in F, and l is the number of edges in S, respectively. We calculate the midpoint of $[a_i, b_i]$ as ρ_i (e.g., ρ_1 is the midpoint of $[a_1, b_1] = [m_1^1, m_{ef}]$ when $i = 1$), and store ρ_i in $\Pi_{Ref\text{-}2}(s,t)$ using $\Pi_{Ref\text{-}2}(s,t) \oplus (\rho_i)$ where $\Pi_{Ref\text{-}2}(s,t)$ is initialized to be (s) (e.g., we have the pink dashed line $\Pi_{Ref\text{-}2}(s,t) = (s, \rho_1)$ when $i = 1$). Then, we move forward (i.e., $i = i + 1$) and let ρ_i be the starting point of new Π_m that passing S based on T by starting at m_{i+1} on e_{i+1}, and to m_g on e_g (e.g., we have the green line $\Pi_{m^3} = (\rho_1, m_2^3, R_2^{m^3})$ and the yellow line $\Pi_{m^4} = (\rho_1, m_2^4, R_2^{m^4})$ when $i = 2$). We iterate this step until we process all the edges in S (e.g., until we process all the edges in $S = (e_1, e_2)$, we get result path $\Pi_{Ref\text{-}2}(s,t) = (s, \rho_1, \rho_2, t)$).

3.3.3 ***Path checking.*** If $|\Pi_{Ref\text{-}2}(s,t)| \leq \frac{(1+\epsilon)}{(1+\eta\epsilon)}|\Pi_{Roug}(s,t)|$, we guarantee that $|\Pi_{Ref\text{-}2}(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$ and we can return $\Pi_{Ref\text{-}2}(s,t)$ as output. Otherwise, we need the next step.

3.3.4 ***Error guaranteed path refinement.*** We introduce some notations first. In Dijkstra algorithm, given a source point s, a set of points $G_A.V$ where A could be *Roug* or *Ref*, for each $u \in G_A.V$, we define $dist_A(u)$ to be the current shortest distance from s to point u, define $prev_A(u)$ to be the previous point on the shortest path from s to point u. After algorithm *Roug*, *state information* stores $dist_{Roug}(u)$ and $prev_{Roug}(u)$ for each $u \in G_{Roug}.V$. In Figure 7, for $G_{Roug}.V$ (i.e., three blue points and three black points), we have $dist_{Roug}(c) = 5 + 2 = 7$ and $prev_{Roug}(c) = a$. Recall that $G_{Roug}.V \subseteq G_{Ref}.V$. We define Q to be a priority queue that stores points $u \in G_{Ref}.V$ that are being preprocessed, as $\{u, dist_{Ref}(u)\}$. In Figure 7, Q stores $\{\{a, 5\}, \{b, 4.7\}\}$. We then perform Dijkstra algorithm on G_{Ref} as follows (see Figure 7).

- **Distance and previous point information initialization using state information**: For each $u \in G_{Ref}.V$, if $u \in G_{Ref}.V \setminus G_{Roug}.V$, we initialize $dist_{Ref}(u)$ to be ∞ and $prev_{Ref}(u)$ to be *NULL*, otherwise, if $u \in G_{Roug}.V$, we initialize $dist_{Ref}(u)$ to be $dist_{Roug}(u)$ and $prev_{Ref}(u)$ to be $prev_{Roug}(u)$ (e.g., for green point d, we initialize $dist_{Ref}(d) = \infty$ and $prev_{Ref}(d) = NULL$, for blue point c, we initialize $dist_{Ref}(c) = dist_{Roug}(c) = 5 + 2 = 7$ and $prev_{Ref}(c) = prev_{Roug}(c) = a$).
- **Priority queue looping**: While Q is not empty:
  - **Dequeue** Q: Dequeue Q with smallest distance value, let v be the dequeued point and $dist_{Ref}(v)$ be the dequeued shortest distance value (e.g., suppose Q stores $\{\{a, 5\}, \{b, 4.7\}\}$, we dequeue b with shortest distance value 4.7).
    - **Pruning using state information, next point updating and Q updating**: For each adjacent vertex v' of v (such that $\overline{vv'} \in G_{Ref}.E$), if $dist_{Ref}(v') > dist_{Ref}(v) + wleng(v, v')$ (where $wleng(v, v') = w_{vv'} \cdot d(v, v')$ is the weighted length for $\overline{vv'}$), then we set $dist_{Ref}(v') = dist_{Ref}(v) + wleng(v, v')$, set $prev_{Ref}(v') = v$ and inqueue $\{v', dist_{Ref}(v')\}$ into Q (i.e., one of adjacent vertex of b is c, since using the state information, we already have $dist_{Ref}(c) = 7$, so $7 < dist_{Ref}(b) + wleng(b, c) = 4.7 + 3 = 7.7$, there is no need to inqueue $\{c, dist_{Ref}(c) = 7.7\}$ into Q, which saves the inqueue and dequeue time).

In Figure 7, if there is no state information, we just know $dist_{Ref}(c) = \infty$, and we need to inqueue $\{c, dist_{Ref}(c) = 7.7\}$ into Q, which increase the running time. Finally, we calculate the refined path $\Pi_{Ref\text{-}3}(s,t)$, and return it as output.

3.4 Theoretical Analysis

Let $\Pi(s,t)$ be the final calculated weighted shortest path of our algorithm. The running time, memory usage, and error of algorithm *Roug-Ref* are in Theorem 3.1.

Theorem 3.1. *The total running time for algorithm Roug-Ref is $O(n \log n + l)$, the total memory usage is $O(n + l)$. It guarantees that $|\Pi(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$.*

Proof Sketch. The running time contains $O(n \log n)$ for algorithm *Roug* and $O(n \log n + l)$ for algorithm *Ref*. The memory usage contains $O(n)$ for algorithm *Roug* and $O(n + l)$ for algorithm *Ref*. The error bound is due to the path checking step and the error guaranteed path refinement step using Dijkstra algorithm in algorithm *Ref*. For the sake of space, the detailed proof could be found in the appendix. □

4 RELATED WORK & BASELINE

4.1 Related Work

As mentioned in Section 1.2, there are three categories of algorithms for solving the weighted region problem approximately: (1) *Continuous wavefront* approach, (2) *heuristics* approach, and (3) *Steiner point* approach,

(1) For the *Continuous wavefront* approach, it aims to calculate the weighted shortest path which follows Snell's law on a continuous surface by exploiting Snell's law using continuous Dijkstra [35] algorithm. The algorithm will return a path with $(1+\epsilon)$-approximation weighted shortest distance in $O(n^8 \log(\frac{nNW}{w\epsilon}))$ time, and its running time is very large.

(2) The *heuristics* approach uses simulated annealing [28] and genetic algorithm [39] to calculate the weighted shortest path, but it cannot guarantee on the quality of the result path returned (in terms of the length of the path) with a given time limit and a given maximum memory.

(3) For the *Steiner point* approach, it uses Dijkstra algorithm on the weighted graph constructed using Steiner points on edges in E and V to calculate the weighted shortest path. Even though the work [25, 30], i.e., algorithm *FixSP*, is regarded as the best-known algorithm for solving the weighted region problem and it runs in $O(n^3 \log n)$ time, it is still very slow. Our experiment shows that for a terrain with 50k faces with $\epsilon = 0.1$, our algorithm's total query time is 73s ($\approx$ 1.2 min), and total memory usage is 43MB, but the best-known algorithm's [25, 30] total query time is 119,000s ($\approx$ 1.5 days), and total memory usage is 2.9GB.

4.2 Baseline

Algorithm *FixSP* [25, 30] in the *Steiner point* approach is regarded as the best-known algorithm for solving the weighted region problem, we set it as a baseline. Note that the algorithm in [25] is for

calculating the unweighted shortest path, we adapt it by assigning each face a weight for solving the weighted region problem. The algorithm that uses Dijkstra algorithm on the weighted graph constructed by the original Steiner point and V, i.e., the rough path calculation step of our algorithm *Roug* (by changing the input error from $\eta\epsilon$ to ϵ), called algorithm *Logarithmic Steiner Point placement scheme* (*LogSP*), is also included as a baseline. Furthermore, in our algorithm *Roug-Ref*, (1) we can choose not to use the state information for pruning out in Dijkstra algorithm in the error guaranteed path refinement step of algorithm *Ref*, (2) we can use *FixSP* to substitute *LogSP* as Steiner points placement scheme in algorithm *Roug* and the error guaranteed path refinement in *Ref*, (3) we can remove the full edge sequence conversion step in algorithm *Ref* (such that the input edge sequence of the Snell's law path refinement step in algorithm *Ref* may be a non-full edge sequence, then we need to try different combinations of edges when using Snell's law), and (4) we can remove the effective weight pruning out sub-step in the Snell's law path refinement step of algorithm *Ref* (after the removing of the effective weight pruning out sub-step, the Snell's law path refinement step is a naive method in [35]), for **ablation study**. We use (1) *Roug-Ref*(A, •, •, •), (2) *Roug-Ref*(•, B, •, •), (3) *Roug-Ref*(•, •, C, •), and (4) *Roug-Ref*(•, •, •, D), to denote these variations, and we also add (5) *Roug-Ref*(A, B, C, D), (6) *Roug-Ref*(A, B, C, •), (7) *Roug-Ref*(A, B, •, D), (8) *Roug-Ref*(A, •, C, D), (9) *Roug-Ref*(•, B, C, D), (10) *Roug-Ref*(A, B, •, •), (11) *Roug-Ref*(A, •, C, •), (12) *Roug-Ref*(A, •, •, D), (13) *Roug-Ref*(•, B, C, •), (14) *Roug-Ref*(•, B, •, D), (15) and *Roug-Ref*(•, •, C, D), to denote apply these variations at the same time in our framework, where $\{A = NoPrunDijk, B = FixSP, C = NoEdgSeqConv, D = NoEffWeig\}$, and • means keeping the same component as in algorithm *Roug-Ref*.

In general, we have 17 baseline algorithms, i.e., *FixSP*, *LogSP*, and 15 variations of *Roug-Ref*. We do not use the *Continuous wavefront* approach as the baseline, due to its large running time. In addition, there is no implementation of the *Continuous wavefront* approach so far [29]. We do not use the *heuristics* approach as the baseline, because it cannot guarantee on the quality of the result path returned (in terms of the length of the path) with a given time limit and a given maximum memory.

Comparisons: We compare 8 algorithms in Table 1 (for the sake of space, we just include the first 5 out of 15 variations of *Roug-Ref* in the table, the comparisons of all algorithms could be found in the appendix). *Roug-Ref* is the best algorithm since it has the smallest running time and memory usage.

5 EMPIRICAL STUDIES

5.1 Experimental Setup

We conducted our experiments on a Linux machine with 2.67 GHz CPU and 48GB memory. All algorithms were implemented in C++. For the following experiment setup, we mainly follow the experiment setup in the work [25, 26, 34, 37, 38].

Datasets: Following some existing studies on terrain data [19, 34, 36], we conducted our experiment based on 17 real terrain datasets, namely (1) BearHead (*BH*, with 280k faces) [4, 37, 38], (2) EaglePeak (*EP*, with 300k faces) [4, 37, 38], (3) SeaBed (*SB*, with 2k faces) [13], (4) CyberPunk (*CP*, with 2k faces) [3], (5) PathAdvisor (*PA*, with 1k faces) [42], (6) a small-version of *BH* (*BH-small*, with 3k faces), (7) a small-version of *EP* (*EP-small*, with 3k faces), (8) - (12) multi-resolution of *EP* (with 1M, 2M, 3M, 4M, 5M faces), and (13) - (17) multi-resolution of *EP-small* (with 10k, 20k, 30k, 40k, 50k faces). For *SB* [13] and *CP* [3] dataset, we first obtained the height / satellite map from [3, 13], then used Blender [1] to generate the 3D terrain model. For *BH-small* and *EP-small* datasets, we generate them using *BH* and *EP* following the procedure in [34, 37, 38] (which creates a terrain with different resolutions and the procedure could be found in the appendix), since algorithm *FixSP* [25, 30] is not feasible on any of the full datasets due to its expensive running time. Following the work [22], we set the weight of a triangle in terrain datasets to be the slope of that face. In order to test the scalability of our algorithm, we also generate multi-resolution of *EP* and *EP-small* datasets.

Algorithms: Our algorithm *Roug-Ref*, and the baseline algorithms, i.e., *FixSP* [25, 30], *LogSP*, and 15 variations of *Roug-Ref* are studied. Since *FixSP* is not feasible on large datasets due to its expensive running time, we (1) compared all algorithms on small datasets, and (2) compared algorithms not involving *FixSP* (i.e., both *LogSP* and variations of *Roug-Ref* not involving *FixSP*) on full datasets.

Query Generation: We randomly chose two vertices in V, one as a source and the other as a destination. For each measurement, 100 queries were answered, and the average, minimum and maximum result was returned.

Factors & Measurements: We studied three factors in the experiments, namely (1) ϵ, (2) removing value (i.e., in the η calculation step of algorithm *Roug*, the number of removed Steiner points whenever we encounter a new Steiner point), and (3) dataset size (i.e., the number of faces in a terrain surface). In addition, we used four measurements to evaluate the algorithm performance, namely (1) *preprocessing time* (i.e., the time for constructing the weighted graph using Steiner points), (2a) *query time for the first algorithm* (i.e., the time in *Roug*), (2b) *query time for the second algorithm* (i.e., the time in *Ref*), (2c) *total query time* (i.e., the time in *Roug-Ref*), (3a) *memory usage for the first algorithm* (i.e., the space consumption in *Roug*), (3b) *memory usage for the second algorithm* (i.e., the space consumption in *Ref*), (3c) *total memory usage* (i.e., the space consumption in *Roug-Ref*), and (4) *distance error* (i.e., the error of the distance returned by the algorithm compared with the exact weighted shortest path).

When varying ϵ and dataset size, we already selected the optimal removing value (= 2), our experimental result shows that there is no need to use the error guaranteed path refinement step in algorithm *Ref*, so all the measurements for 15 variations of *Roug-Ref* with or without *NoPrunDijk* are same. But, when varying removing value, it may happen that the error guaranteed path refinement step in algorithm *Ref* is used, so the 15 variations of *Roug-Ref* with or without *NoPrunDijk* will have different effects on the measurements. Thus, for clarity, (1) when varying removing value, we compared all 15 variations of *Roug-Ref* (both with and without *NoPrunDijk*), (2) when varying ϵ and dataset size, we compared variations of *Roug-Ref* using the original error guaranteed path refinement step in algorithm *Ref*, i.e., not involving *NoPrunDijk*.

Since no algorithm could solve the weighted region problem exactly so far, we use *Roug-Ref*(*NoPrunDijk*, *FixSP*, *NoEdgSeqConv*, *NoEffWeig*) and set $\epsilon = 0.05$ and the removing value to be 2 to simulate the exact weighted shortest path on small-version of datasets

Algorithm	Time		Size		Error
FixSP [25, 30]	$O(n^3 \log n)$	Gigantic	$O(n^3)$	Large	$(1+\epsilon)$
LogSP	$O(n \log \frac{LN}{\epsilon} \log(n \log \frac{LN}{\epsilon}))$	Large	$O(n \log \frac{LN}{\epsilon})$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, *FixSP*, *NoEdgSeqConv*, *NoEff Weig*)	$O(n^3 \log n + nl^2 \log(\frac{nNW}{w\epsilon}))$	Large	$O(n^3 + nl)$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, •, •, •)	$O(n \log \frac{LN}{\epsilon} \log(n \log \frac{LN}{\epsilon}) + l)$	Large	$O(n \log \frac{LN}{\epsilon} + l)$	Medium	$(1+\epsilon)$
Roug-Ref(•, *FixSP*, •, •)	$O(n^2 \log n + l)$	Large	$O(n^2 + l)$	Medium	$(1+\epsilon)$
Roug-Ref(•, •, *NoEdgSeqConv*, •)	$O(n \log n + nl)$	Medium	$O(nl)$	Small	$(1+\epsilon)$
Roug-Ref(•, •, •, *NoEff Weig*)	$O(n \log n + l^2 \log(\frac{nNW}{w\epsilon}))$	Medium	$O(n+l)$	Small	$(1+\epsilon)$
***Roug-Ref* (ours)**	$O(n \log n + l)$	**Small**	$O(n+l)$	**Small**	$(1+\epsilon)$

Table 1: Comparison of 8 algorithms (comparison of all algorithms could be found in the appendix)

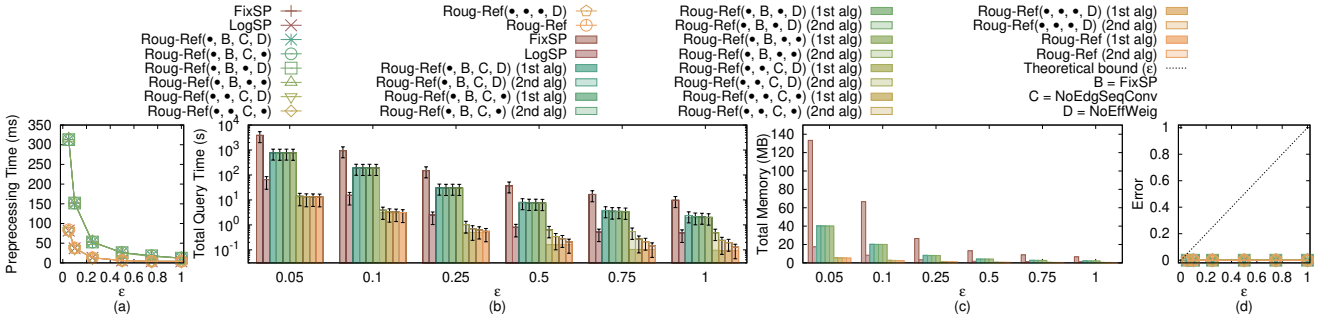

Figure 11: Effect of ϵ on *BH-small* dataset

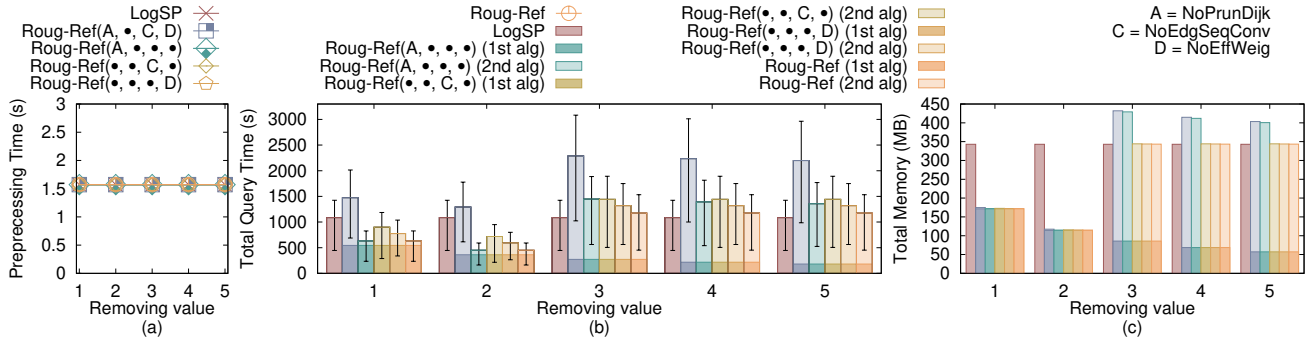

Figure 12: Effect of removing value on *EP* dataset

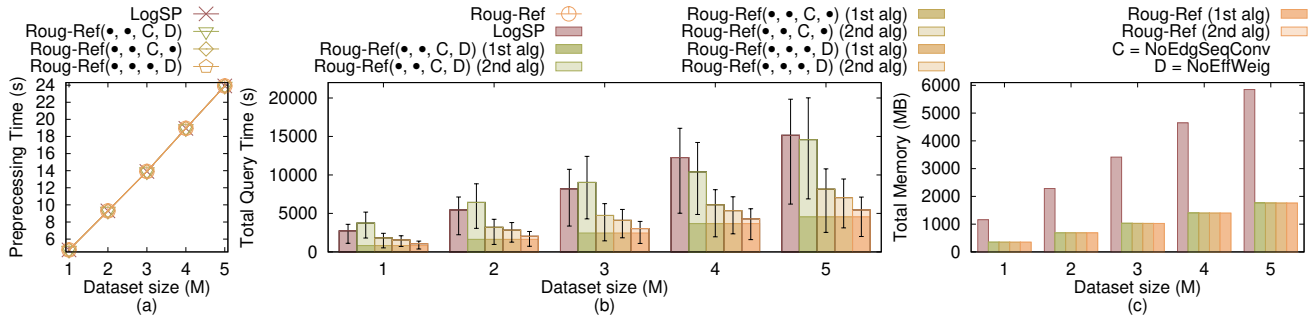

Figure 13: Effect of dataset size on multi-resolution of *EP* datasets

for measuring distance error. Since *FixSP* is not feasible on full datasets, the distance error is omitted on full datasets.

5.2 Experimental Results

Figure 11, Figure 12, Figure 13 show the V2V path query result on the *BH-small* dataset, *EP* dataset, and multi-resolution of *EP* datasets when varying ϵ, removing value, and dataset size, respectively. For

the (1) total query time, (2) query time for algorithm *Roug* and (3) query time for algorithm *Ref*, the vertical bar means the minimum and maximum (1) total query time, (2) query time for algorithm *Roug* and (3) query time for algorithm *Ref*. The results on other combinations of dataset and the variation of ϵ, removing value, dataset size, the separation of two steps in query time and memory usage calculated using all algorithms, and A2A path query could be found in the appendix.

Effect of ϵ for V2V path query. We tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} by setting removing value to be 2, and the results on *BH-small* dataset calculated using default algorithms, i.e., *FixSP*, *LogSP*, and variations of *Roug-Ref* without *NoPrunDijk*, are in Figure 11. *Roug-Ref* performs better than all the remaining algorithms in terms of query time and memory usage, and it is clearer to observe the superior performance of *Roug-Ref* when $\epsilon = 1$. The error of all algorithms is close to 0%.

Effect of removing value for V2V path query. We tested 5 values of removing value from {1, 2, 3, 4, 5} on small-version datasets by setting ϵ to be 0.1, on full datasets by setting ϵ to be 0.25, and the results on *EP* dataset calculated using *LogSP*, and some important variations of *Roug-Ref* without *FixSP* (i.e., with only one component change), are in Figure 12 (the result calculated using all the variations of *Roug-Ref* could be found in the appendix). By setting the removing value to be 2, the total running time and total memory usage of the algorithms using *Roug-Ref* framework are the smallest. This is because when the removing value is larger, *Roug* could run faster, but it has a higher chance that *Ref* needs to perform the error guaranteed path refinement step. Thus, we select the optimal removing value, i.e., 2.

Effect of dataset size (scalability test) for V2V path query. We tested 5 values of dataset size from {11M, 2M, 3M, 4M, 5M} on multi-resolution of *EP* datasets by setting ϵ to be 0.25 and removing value to be 2, and the results calculated using default algorithms, i.e., *FixSP*, *LogSP*, and variations of *Roug-Ref* without *NoPrunDijk* and *FixSP*, are in Figure 13. *Roug-Ref* could still beat other algorithms in terms of query time and memory usage. We also tested 5 values of dataset size from {10k, 20k, 30k, 40k, 50k} on multi-resolution of *EP-small* datasets by setting ϵ to be 0.1 and removing value to be 2. The figure could be found in the appendix.

Arbitrary point-to-arbitrary point (A2A) path query. We tested the A2A path query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting removing value to be 2 on *EP-small* and *EP* datasets. The result could be found in the appendix. *Roug-Ref* still performs better than all the remaining algorithms in terms of query time and memory usage.

5.3 Case Study

5.3.1 User study. We conducted a user study on a campus map weighted shortest path finding tool, which allows users to find the shortest path between any two rooms in a university campus, namely *Path Advisor* [42]. The floor of the building is represented in a terrain surface. It is expected that the path should not be too close to the obstacle (e.g., the distance between the path to the obstacle should be at least 0.2 meter). Based on this, when a face on the floor is closer to the boundary of aisle in a building (resp. the aisle center), the face is assigned with a larger (resp. smaller) weight. We adopted our 18 algorithms to their tool by using *PA* dataset [42]. We chose two places in Path Advisor as source and destination, respectively, and repeated it for 100 times to calculate the path (with $\epsilon = 0.5$). Figure 2 shows an example of different paths in Path Advisor. The blue and purple dashed paths are the weighted shortest path (with distance 105.8m) and the unweighted shortest path (with distance 98.4m), respectively. We presented Figure 2 and the path distance result to 30 users (i.e., university students), and 96.7% of users think the blue path is the most realistic one since it is not close to the obstacle and it does not have sudden direction changes. In addition, the average query time for the state-of-the-art algorithm *FixSP* and our algorithm *Roug-Ref* are 16.62s and 0.1s, respectively. The result on other measurements could be found in the appendix.

5.3.2 Motivation study. We also conducted a motivation study on the placement of undersea optical fiber cable on the seabed as mentioned in Section 1.1 by using *SB* dataset [13]. For a face with a deeper sea level, the hydraulic pressure is higher, the cable's lifespan is reduced, and it is more expensive to repair and maintain the cable, so the face will have a larger weight. The average life expectancy of the cable is 25 years [33], and if the cable is in deep waters (e.g., 8.5km or greater), the cable needs to be repaired frequently (e.g., its life expectancy is reduced to 20 years) [33]. We randomly selected two points as source and destination, respectively, and repeated it 100 times to calculate the path (with $\epsilon = 0.5$). Figure 3 shows an example of different paths on seabed. The blue and purple dashed paths are the weighted shortest path (with a distance 457.9km) and the unweighted shortest path (with a distance 438.3km), respectively. According to [21], the cost of undersea optical fiber cable is USD \$200M/km. Consider constructing a cable that will be used for 100 years, the total estimated cost for the green and red paths are USD \$366B ($= \frac{100\text{years}}{25\text{years}} \times 457.9\text{km} \times \$200\text{M/km}$) and \$438B ($= \frac{100\text{years}}{20\text{years}} \times 438.3\text{km} \times \$200\text{M/km}$), respectively. The average query time for the state-of-the-art algorithm *FixSP* and our algorithm *Roug-Ref* are 22.50s and 0.2s, respectively. The result on other measurements could be found in the appendix.

5.4 Experimental Results Summary

Our algorithm *Roug-Ref* consistently outperforms the state-of-the-art algorithm, i.e., *FixSP*, in terms of all measurements. Specifically, our algorithm runs up to 1630 times faster than the state-of-the-art algorithm. When the dataset size is 50k with $\epsilon = 0.1$, our algorithm's total query time is 73s ($\approx$ 1.2 min), and total memory usage is 43MB, but the state-of-the-art algorithm's total query time is 119,000s ($\approx$ 1.5 days), and total memory usage is 2.9GB. The case study also shows that *Roug-Ref* is the best algorithm since it is the fast algorithm that supports real-time responses.

6 CONCLUSION

In our paper, we propose a two-step approximation algorithm for calculating the weighted shortest path in the 3D weighted region problem using algorithm *Roug-Ref*. Our algorithm could bound the error ratio, and the experimental results show that it runs up to 1630 times faster than the state-of-the-art algorithm. The future work could be that proposing a new pruning step to reduce the algorithm running time further.

REFERENCES

[1] 2022. *Blender.* https://www.blender.org
[2] 2022. *Cyberpunk 2077.* https://www.cyberpunk.net
[3] 2022. *Cyberpunk 2077 Map.* https://www.videogamecartography.com/2019/08/cyberpunk-2077-map.html
[4] 2022. *Data Geocomm.* http://data.geocomm.com/
[5] 2022. *Google Earth.* https://earth.google.com/web
[6] 2022. *Metaverse.* https://about.facebook.com/meta
[7] 2022. *Quartic function.* https://en.wikipedia.org/wiki/Quartic_function
[8] 2022. *Snell's law in vector form.* https://physics.stackexchange.com/questions/435512/snells-law-in-vector-form
[9] 2023. *Fermat's principle.* https://en.wikipedia.org/wiki/Fermat%27s_principle
[10] 2023. *Newton's method.* https://en.wikipedia.org/wiki/Newton%27s_method
[11] 2023. *Snell's law.* https://en.wikipedia.org/wiki/Snell%27s_law
[12] Lyudmil Aleksandrov, Mark Lanthier, Anil Maheshwari, and Jörg-R Sack. 1998. An ε—approximation algorithm for weighted shortest paths on polyhedral surfaces. In *Scandinavian Workshop on Algorithm Theory*. Springer, 11–22.
[13] Christopher Amante and Barry W Eakins. 2009. ETOPO1 arc-minute global relief model: procedures, data sources and analysis. (2009).
[14] Harri Antikainen. 2013. Comparison of Different Strategies for Determining Raster-Based Least-Cost Paths with a Minimum Amount of Distortion. *Transactions in GIS* 17, 1 (2013), 96–108.
[15] Christian Bueger, Tobias Liebetrau, and Jonas Franken. 2022. Security threats to undersea communications cables and infrastructure–consequences for the EU. *Report for SEDE Committee of the European Parliament, PE702* 557 (2022).
[16] Jindong Chen and Yijie Han. 1990. Shortest Paths on a Polyhedron. In *SOCG*. New York, NY, USA, 360–369.
[17] Jean-Lou De Carufel, Carsten Grimm, Anil Maheshwari, Megan Owen, and Michiel Smid. 2014. A note on the unsolvability of the weighted region shortest path problem. *Computational Geometry* 47, 7 (2014), 724–727.
[18] Ke Deng, Heng Tao Shen, Kai Xu, and Xuemin Lin. 2006. Surface k-NN query processing. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 78–78.
[19] Ke Deng and Xiaofang Zhou. 2004. Expansion-based algorithms for finding single pair shortest path on surface. In *International Workshop on Web and Wireless Geographical Information Systems*. Springer, 151–166.
[20] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.
[21] RL Gallawa. 1981. Estimated cost of a submarine fiber cable system. *Fiber & Integrated Optics* 3, 4 (1981), 299–322.
[22] Amin Gheibi, Anil Maheshwari, Jörg-Rüdiger Sack, and Christian Scheffer. 2018. Path refinement in weighted regions. *Algorithmica* 80, 12 (2018), 3766–3802.
[23] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. 2010. A survey of motion planning algorithms from the perspective of autonomous UAV guidance. *Journal of Intelligent and Robotic Systems* 57, 1 (2010), 65–100.
[24] Norman Jaklin, Mark Tibboel, and Roland Geraerts. 2014. Computing high-quality paths in weighted regions. In *Proceedings of the Seventh International Conference on Motion in Games*. 77–86.
[25] Manohar Kaul, Raymond Chi-Wing Wong, and Christian S Jensen. 2015. New lower and upper bounds for shortest distance queries on terrains. *Proceedings of the VLDB Endowment* 9, 3 (2015), 168–179.
[26] Manohar Kaul, Raymond Chi-Wing Wong, Bin Yang, and Christian S Jensen. 2013. Finding shortest paths on terrains by killing two birds with one stone. *Proceedings of the VLDB Endowment* 7, 1 (2013), 73–84.
[27] Jonathan Kim. 2022. *Submarine cables: the invisible fiber link enabling the Internet.* https://dgtlinfra.com/submarine-cables-fiber-link-internet/
[28] Mark R Kindl and Neil C Rowe. 2012. Evaluating simulated annealing for the weighted-region path-planning problem. In *2012 26th International Conference on Advanced Information Networking and Applications Workshops*. IEEE, 926–931.
[29] Mark Lanthier. 2000. *Shortest path problems on polyhedral surfaces.* Ph.D. Dissertation. Carleton University.
[30] Mark Lanthier, Anil Maheshwari, and J-R Sack. 2001. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica* 30, 4 (2001), 527–562.
[31] Lik-Hang Lee, Tristan Braud, Pengyuan Zhou, Lin Wang, Dianlei Xu, Zijun Lin, Abhishek Kumar, Carlos Bermejo, and Pan Hui. 2021. All one needs to know about metaverse: A complete survey on technological singularity, virtual ecosystem, and research agenda. *arXiv preprint arXiv:2110.05352* (2021).
[32] Lik-Hang Lee, Zijun Lin, Rui Hu, Zhengya Gong, Abhishek Kumar, Tangyao Li, Sijia Li, and Pan Hui. 2021. When creators meet the metaverse: A survey on computational arts. *arXiv preprint arXiv:2111.13486* (2021).
[33] Jean-François Libert and Gary Waterworth. 2016. 13 - Cable technology. In *Undersea Fiber Communication Systems (Second Edition)*. Academic Press, 465–508.
[34] Lian Liu and Raymond Chi-Wing Wong. 2011. Finding shortest path on land surface. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 433–444.
[35] Joseph SB Mitchell and Christos H Papadimitriou. 1991. The weighted region problem: finding shortest paths through a weighted planar subdivision. *Journal of the ACM (JACM)* 38, 1 (1991), 18–73.
[36] Cyrus Shahabi, Lu-An Tang, and Songhua Xing. 2008. Indexing land surface for efficient knn query. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1020–1031.
[37] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, and David M. Mount. 2017. Distance oracle on terrain surface. In *SIGMOD/PODS'17*. New York, NY, USA, 1211–1226.
[38] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, David M Mount, and Hanan Samet. 2022. Proximity queries on terrain surface. *ACM Transactions on Database Systems (TODS)* (2022).
[39] Elias K Xidias. 2019. On designing near-optimum paths on weighted regions for an intelligent vehicle. *International Journal of Intelligent Transportation Systems Research* 17, 2 (2019), 89–101.
[40] Songhua Xing, Cyrus Shahabi, and Bei Pan. 2009. Continuous monitoring of nearest neighbors on land surface. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1114–1125.
[41] Da Yan, Zhou Zhao, and Wilfred Ng. 2012. Monochromatic and bichromatic reverse nearest neighbor queries on land surfaces. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 942–951.
[42] Yinzhao Yan and Raymond Chi-Wing Wong. 2021. Path Advisor: a multi-functional campus map tool for shortest path. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2683–2686.
[43] Xiaoming Zheng, Sven Koenig, David Kempe, and Sonal Jain. 2010. Multirobot forest coverage for weighted and unweighted terrain. *IEEE Transactions on Robotics* 26, 6 (2010), 1018–1031.

A SUMMARY OF FREQUENT USED NOTATIONS

Table 2 shows a summary of frequent used notations.

B COMPARISON OF ALL ALGORITHMS

Table 3 shows a comparison of all algorithms.

C EXAMPLE ON THE GOOD PERFORMANCE OF THE EFFECTIVE WEIGHT PRUNING SUB-STEP IN THE SNELL'S LAW PATH REFMENT STEP OF ALGORITHM *REF*

We use a 1D example to illustrate why the effective weight pruning sub-step in the Snell's law path refinement step of algorithm *Ref* could prune out unnecessary checking. Let 0 and 100 to be the position of the two endpoints of e_1, and we have $[a_1 b_1] = [0, 100]$. Assume that the position of the optimal point r_1 is 87.32. Then, without the effective weight pruning sub-step in the Snell's law path refinement step of algorithm *Ref*, the searching interval will be $[50, 100]$, $[75, 100]$, $[75, 87.5]$, $[81.25, 87.5]$, $[84.375, 87.5]$, $[85.9375, 87.5]$, $[86.71875, 87.5]$, $[87.109375, 87.5] \cdots$. But, with the effective weight pruning sub-step in the Snell's law path refinement step of algorithm *Ref*, assume that we still need to use several iterations of the original binary search to let Π_m pass the whole S based on T, and we need to check $[50, 100]$, $[75, 100]$, $[75, 87.5]$. After checking the interval $[75, 87.5]$, we get a Π_m that passes the whole S based on T. Assume we calculate m_{ef} as 87 using effective weight pruning step. As a result, in the next checking, we could directly limit the searching interval to be $[87, 87.5]$, which could prune out some unnecessary interval checking including $[81.25, 87.5]$, $[84.375, 87.5]$, $[85.9375, 87.5]$, $[86.71875, 87.5]$, and thus, the effective weight pruning sub-step in the Snell's law path refinement step of algorithm *Ref* could save the running time and memory usage.

Notation	Meaning
T	The terrain surface
$V/E/F$	The set of vertices / edges / faces of T
n	The number of vertices of T
N	The smallest integer value which is larger than or equal to the coordinate value of any vertex
W/w	The maximum / minimum weights of T
L	The length of the longest edge in T
h	The minimum height of any face in F
ϵ	The error parameter
η	The constant reversely calculated using Steiner points for controlling the error
k	The removing value
$\Pi^*(s,t)$	The optimal weighted shortest path
r_i	The intersection point of $\Pi^*(s,t)$ and an edge in E
$\Pi(s,t)$	The final calculated weighted shortest path
$\lvert\Pi(s,t)\rvert$	The weighted distance of $\Pi(s,t)$
$\overline{pq}$	A line between two points p and q on a face
$\Pi_{Roug}(s,t)$	The rough path calculated using algorithm *Roug*
$\Pi_{Ref\text{-}1}(s,t)$	The modified rough path calculated using the full edge sequence conversion step of algorithm *Ref*
$\Pi_{Ref\text{-}2}(s,t)$	The refined path calculated using the Snell's law path refinement step of algorithm *Ref*
$\Pi_{Ref\text{-}3}(s,t)$	The refined path calculated using the error guaranteed path refinement step of algorithm *Ref*
ρ_i	The intersection point of $\Pi_{Ref\text{-}2}(s,t)$ and an edge in E
ξ	The iteration counts of single endpoint cases in the full edge sequence conversion step of algorithm *Ref*
S	The edge sequence that $\Pi_{Ref\text{-}1}(s,t)$ connects from s to t in order based on T
l	The number of edges in S
Π_c	A 3D surface Snell's ray
SP_{Roug} / SP_{Ref}	The set of Steiner points on each edge of E used in algorithm *Roug* / *Ref*
G_{Roug} / G_{Ref}	The connected weighted graph used in algorithm *Roug* / *Ref*
$G_{Roug}.V$ / $G_{Ref}.V$	The set of vertices in the connected weighted graph used in algorithm *Roug* / *Ref*
$G_{Roug}.E$ / $G_{Ref}.E$	The set of edges in the connected weighted graph used in algorithm *Roug* / *Ref*

Table 2: Summary of frequent used notations

D A2A PATH QUERY

Apart from the V2V path query that we discussed in the main body of this paper, we also present a method to answer the *arbitrary point-to-arbitrary point (A2A) path query* in the weighted region problem based on our method. We give a definition first. Given a vertex v, a face f with three vertices v_1, v_2, v_3 and weight w, we define v *belongs to* f if the area of $f = \triangle v_1v_2v_3$ is equal to the sum of the area of $\triangle vv_1v_2$, $\triangle vv_1v_3$ and $\triangle vv_1v_2v_3$. With the definition, we are ready to introduce the adapted method to answer the A2A path query in the weighted region. This adapted method is similar to the one presented in Section 3, the only difference is that if s or t is not in V, we could simply make them as vertices by adding new triangles between them and the three vertices of the face f that s or t *belongs* in T, and the newly added triangle face's weight is the same as the weight of f. We need to remove f from F, and add the newly added triangle faces into F. We also add s or t in V, and add the edges of the newly added triangle faces into E. In the worst case, both s and t are not in V, we need to create six new faces, and add two new vertices, so the total number of vertices is still n. Thus, the running time, memory usage and error bound of the adapted method for answering the A2A path query in the weighted region problem is still the same as the method in the main body of the paper, that is, the same as the values in the Theorem 3.1.

E EMPIRICAL STUDIES

E.1 Experimental Results on the V2V Path Query

(1) Figure 11 and Figure 14, (2) Figure 15 and Figure 16 show the result on the *BH-small* dataset when varying ϵ and removing value, respectively. (3) Figure 17 and Figure 18, (4) Figure 19 and Figure 20 show the result on the *BH* dataset when varying ϵ and removing value, respectively. (5) Figure 21 and Figure 22, (6) Figure 23 and Figure 24 show the result on the *EP-small* dataset when varying ϵ and removing value, respectively. (7) Figure 25 and Figure 26, (8) Figure 27 and Figure 28 show the result on the *EP-small* dataset when varying ϵ and removing value, respectively. (9) Figure 29 and Figure 30 show the result on multi-resolution of *EP-small* datasets when varying dataset size. (10) Figure 13 and Figure 31 show the result on multi-resolution of *EP* datasets when varying dataset size.

Effect of ϵ. In Figure 11, Figure 17, Figure 21 and Figure 25, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *BH-small*, *BH*, *EP-small* and *EP* datasets by setting removing value to be 2. Figure 14, Figure 18, Figure 22 and Figure 26 are the separated query time and memory usage in two steps for these results. *Roug-Ref* performs better than all the remaining algorithms in terms of query time and memory usage. The error of all algorithms is close to 0%.

Effect of removing value. In Figure 15, Figure 19, Figure 23 and Figure 27, we tested 5 values of removing value from {1, 2, 3, 4, 5} on *BH-small*, *BH*, *EP-small* and *EP* datasets by setting ϵ to be 0.1 on small-version datasets (i.e., *BH-small* and *EP-small* dataset), and 0.25 on full datasets (i.e., *BH* and *EP* dataset). Figure 16, Figure 20, Figure 24 and Figure 28 are the separated query time and memory usage in two steps for these results. By setting the removing value to be 2, the total running time and total memory usage of using *Roug-Ref* framework are the smallest. This is because when the removing value is larger, *Roug* could run faster, but it has a higher chance that *Ref* needs to perform the error guaranteed path refinement step. Thus, we select the optimal removing value, i.e., 2.

Effect of dataset size (scalability test). In Figure 29 and Figure 13, we tested 5 values of dataset size from {10k, 20k, 30k, 40k, 50k} on multi-resolution of *EP-small* datasets by setting ϵ to be 0.1, and {1M, 2M, 3M, 4M, 5M} on multi-resolution of *EP* datasets

Algorithm	Time		Size		Error
FixSP [25, 30]	$O(n^3 \log n)$	Gigantic	$O(n^3)$	Large	$(1+\epsilon)$
LogSP	$O(n \log \frac{LN}{\epsilon} \log(n \log \frac{LN}{\epsilon}))$	Large	$O(n \log \frac{LN}{\epsilon})$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, *FixSP*, *NoEdgSeqConv*, *NoEffWeig*)	$O(n^3 \log n + nl^2 \log(\frac{nNW}{w\epsilon}))$	Large	$O(n^3 + nl)$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, *FixSP*, *NoEdgSeqConv*, •)	$O(n^3 \log n + nl)$	Large	$O(n^3 + nl)$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, *FixSP*, •, *NoEffWeig*)	$O(n^3 \log n + l^2 \log(\frac{nNW}{w\epsilon}))$	Large	$O(n^3 + l)$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, *FixSP*, •, •)	$O(n^3 \log n + l)$	Large	$O(n^3 + l)$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, •, *NoEdgSeqConv*, *NoEffWeig*)	$O(n \log \frac{LN}{\epsilon} \log(n \log \frac{LN}{\epsilon}) + nl^2 \log(\frac{nNW}{w\epsilon}))$	Large	$O(n \log \frac{LN}{\epsilon} + nl)$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, •, *NoEdgSeqConv*, •)	$O(n \log \frac{LN}{\epsilon} \log(n \log \frac{LN}{\epsilon}) + nl)$	Large	$O(n \log \frac{LN}{\epsilon} + nl)$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, •, •, *NoEffWeig*)	$O(n \log \frac{LN}{\epsilon} \log(n \log \frac{LN}{\epsilon}) + l^2 \log(\frac{nNW}{w\epsilon}))$	Large	$O(n \log \frac{LN}{\epsilon} + l)$	Medium	$(1+\epsilon)$
Roug-Ref(*NoPrunDijk*, •, •, •)	$O(n \log \frac{LN}{\epsilon} \log(n \log \frac{LN}{\epsilon}) + l)$	Large	$O(n \log \frac{LN}{\epsilon} + l)$	Medium	$(1+\epsilon)$
Roug-Ref(•, *FixSP*, *NoEdgSeqConv*, *NoEffWeig*)	$O(n^2 \log n + nl^2 \log(\frac{nNW}{w\epsilon}))$	Large	$O(n^2 + nl)$	Medium	$(1+\epsilon)$
Roug-Ref(•, *FixSP*, *NoEdgSeqConv*, •)	$O(n^2 \log n + nl)$	Large	$O(n^2 + nl)$	Medium	$(1+\epsilon)$
Roug-Ref(•, *FixSP*, •, *NoEffWeig*)	$O(n^2 \log n + l^2 \log(\frac{nNW}{w\epsilon}))$	Large	$O(n^2 + l)$	Medium	$(1+\epsilon)$
Roug-Ref(•, *FixSP*, •, •)	$O(n^2 \log n + l)$	Large	$O(n^2 + l)$	Medium	$(1+\epsilon)$
Roug-Ref(•, •, *NoEdgSeqConv*, *NoEffWeig*)	$O(n \log n + nl^2 \log(\frac{nNW}{w\epsilon}))$	Medium	$O(nl)$	Small	$(1+\epsilon)$
Roug-Ref(•, •, *NoEdgSeqConv*, •)	$O(n \log n + nl)$	Medium	$O(nl)$	Small	$(1+\epsilon)$
Roug-Ref(•, •, •, *NoEffWeig*)	$O(n \log n + l^2 \log(\frac{nNW}{w\epsilon}))$	Medium	$O(n+l)$	Small	$(1+\epsilon)$
***Roug-Ref* (ours)**	$\boldsymbol{O(n \log n + l)}$	**Small**	$\boldsymbol{O(n+l)}$	**Small**	$\boldsymbol{(1+\epsilon)}$

Table 3: Comparison of all algorithms

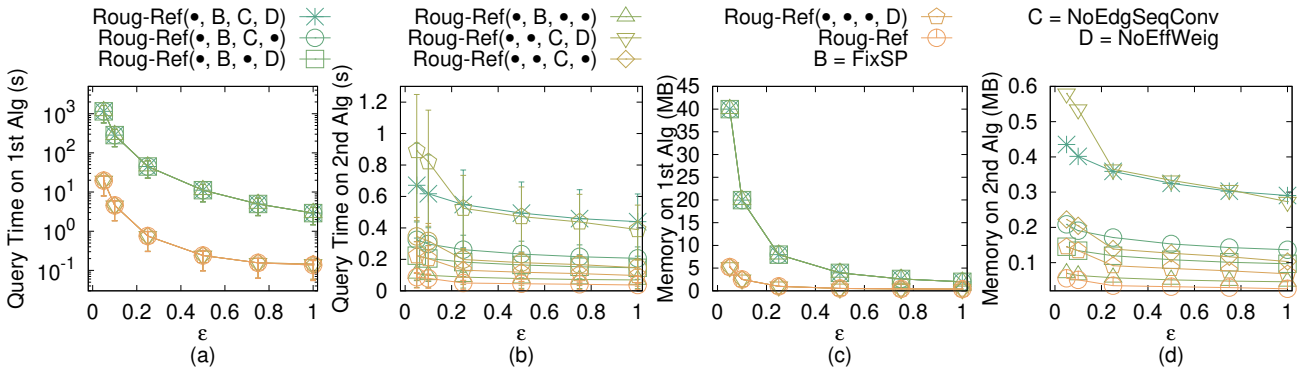

Figure 14: Effect of ϵ on *BH-small* dataset with separated query time and memory usage in two steps (V2V path query)

by setting ϵ to be 0.25 for scalability test. Figure 30 and Figure 31 are the separated query time and memory usage in two steps for these results. On multi-resolution of *EP-small* datasets, *Roug-Ref* could still beat other algorithms in terms of query time and memory usage. When the dataset size is 50k with $\epsilon = 0.1$, the state-of-the-art algorithm's (i.e., *FixSP*) total query time is 119,000s (≈ 1.5 days), and total memory usage is 2.9GB, while our algorithm's (i.e., *Roug-Ref*) total query time is 73s (≈ 1.2 min), and total memory usage is 43MB.

E.2 Experimental Results on the A2A Path Query

In Figure 32 and Figure 34, we tested the A2A path query by varying ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting removing value to be 2on *EP-small* and *EP* datasets. Figure 33 and Figure 35 are the separated query time and memory usage in two steps for these results. Similar to the V2V path query, for the A2A path query, *Roug-Ref* still performs better than all the remaining algorithms in terms of query time and memory usage. This is because A2A path query is very similar to V2V path query.

E.3 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 [34, 37, 38]. Let $T_t = (V_t, E_t, F_t)$ be our target terrain 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 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.

E.4 Case Study

E.4.1 User Study (Path Advisor). Figure 36 and Figure 37 show the result for Path Advisor user study when varying ϵ. Our user

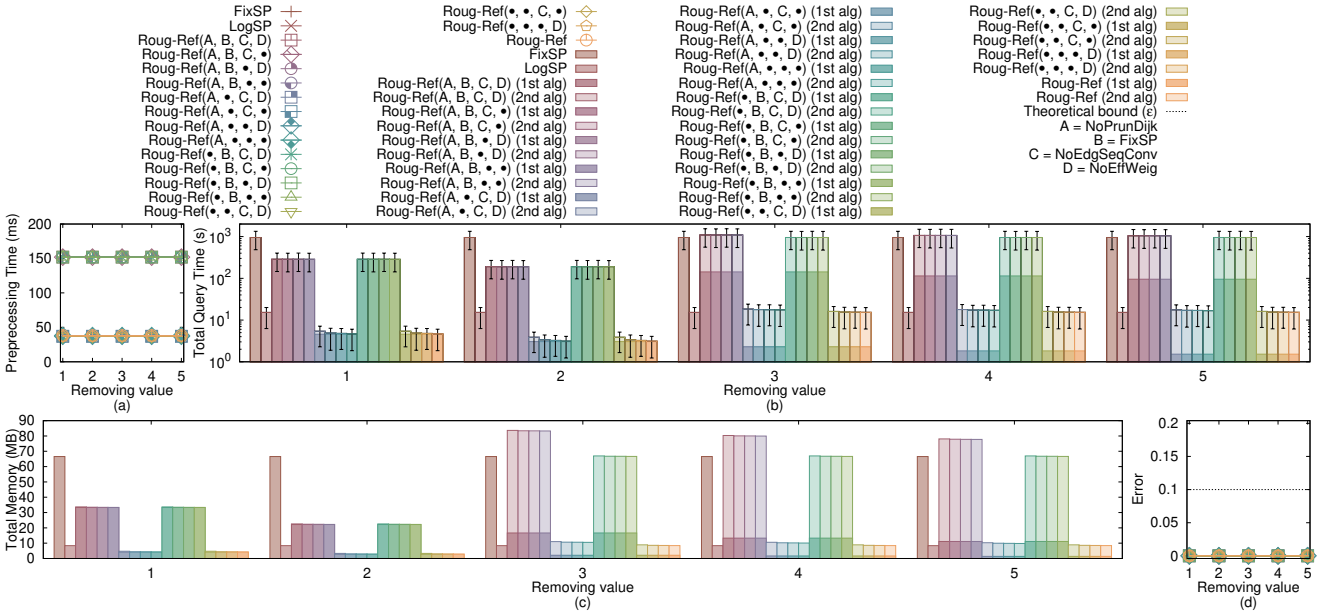

Figure 15: Effect of removing value on *BH-small* dataset (V2V path query)

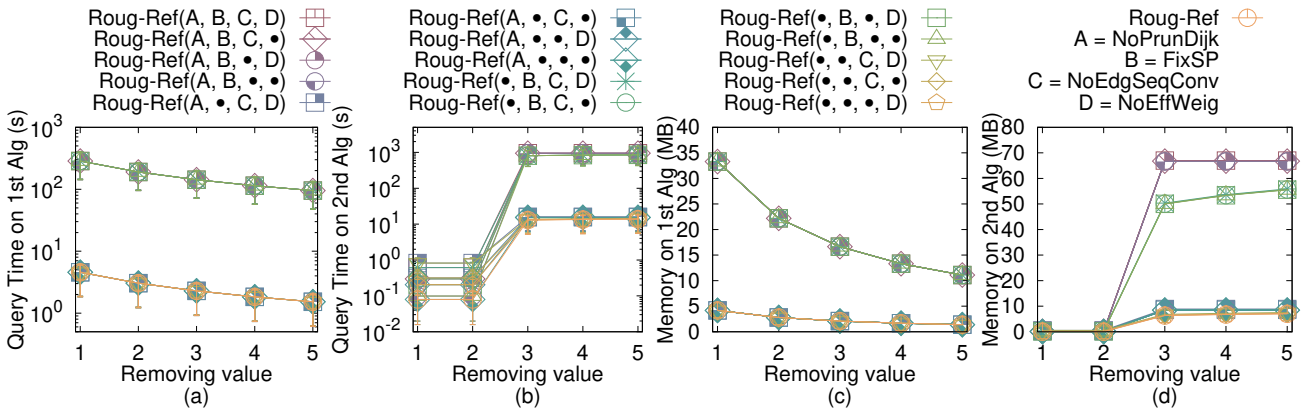

Figure 16: Effect of removing value on *BH-small* dataset with separated query time and memory usage in two steps (V2V path query)

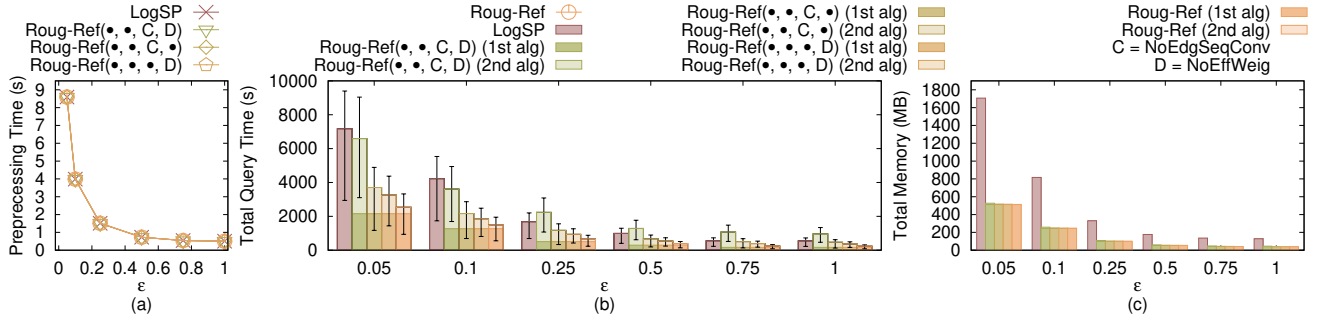

Figure 17: Effect of ϵ on *BH* dataset (V2V path query)

study in Section 5.3.1 has already shown that most users think the blue path (i.e., the weighted shortest path) is the most realistic one.

Our user study in Section 5.3.1 has also already shown that when $\epsilon = 0.5$, the average query time for the state-of-the-art algorithm

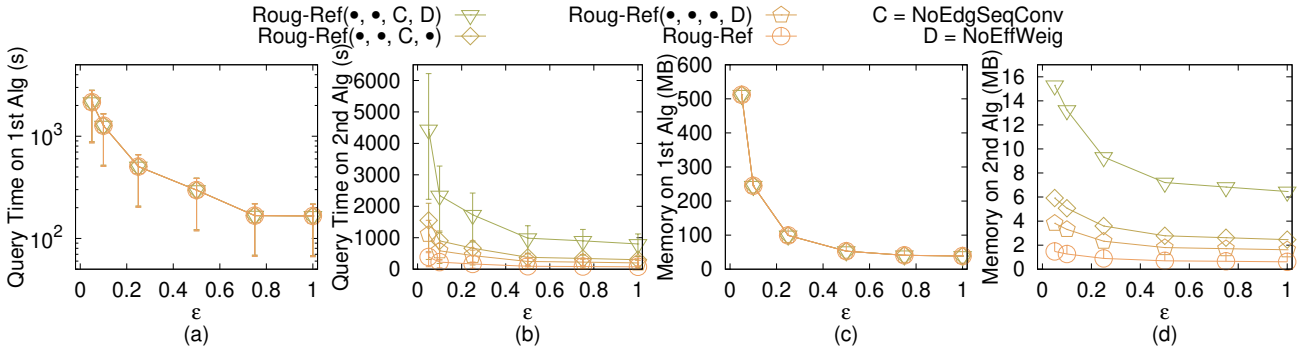

Figure 18: Effect of ϵ on *BH* dataset with separated query time and memory usage in two steps (V2V path query)

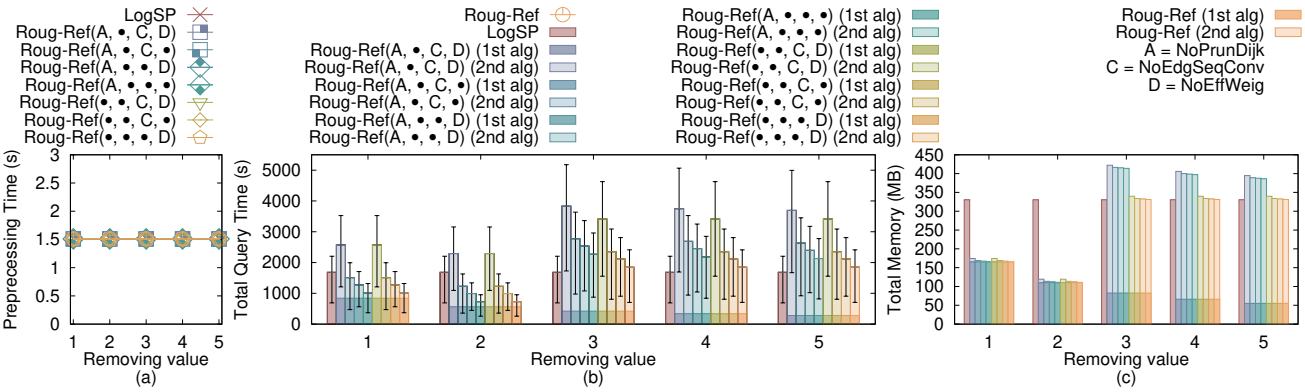

Figure 19: Effect of removing value on *BH* dataset (V2V path query)

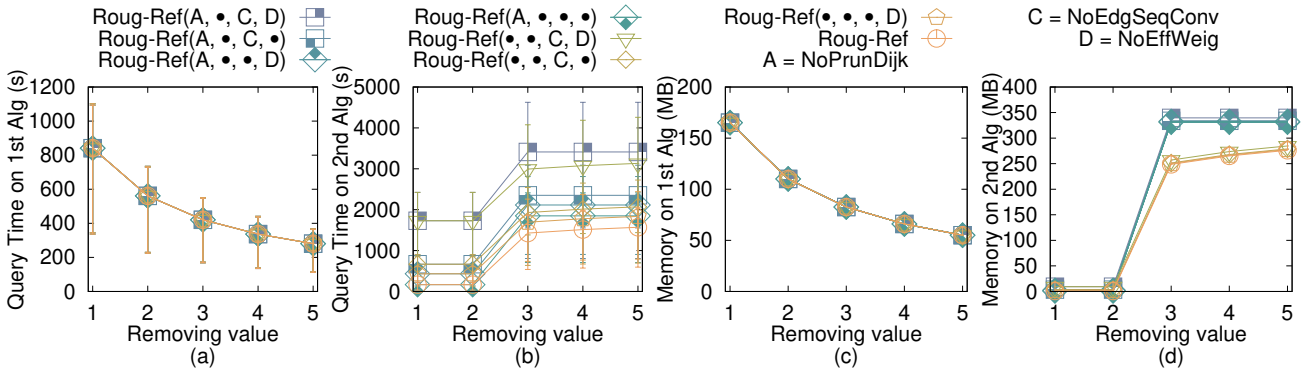

Figure 20: Effect of removing value on *BH* dataset with separated query time and memory usage in two steps (V2V path query)

FixSP and our algorithm *Roug-Ref* are 16.62s and 0.1s, respectively. In addition, in a map application, the query time is the most crucial factor since users would like to get the result in a shorter time. Thus, *Roug-Ref* is the most suitable algorithm for Path Advisor.

E.4.2 User Study (Cyberpunk 2077). We conducted another user study on Cyberpunk 2077 [2], a popular 3D computer game. The dataset is *CP* dataset [3] used in our experiment. We set the weight of a triangle in terrain to be the slope of that face. We randomly selected two points as source and destination, respectively, and repeated it 100 times to calculate the path. Figure 38 and Figure 39 show the result for Cyberpunk 2077 user study when varying ϵ. When $\epsilon = 0.5$, the average query time for the state-of-the-art algorithm *FixSP* and our algorithm *Roug-Ref* are 21.61s and 0.2s, respectively. It is important to get real-time responses in computer games. Thus, *Roug-Ref* is the most suitable algorithm for Cyberpunk 2077.

E.4.3 Motivation Study. Figure 40 and Figure 41 show the result for seabed motivation study when varying ϵ. Our motivation study in Section 5.3.2 has already shown that the blue path (i.e., the weighted shortest path) is the most realistic one since it could avoid the

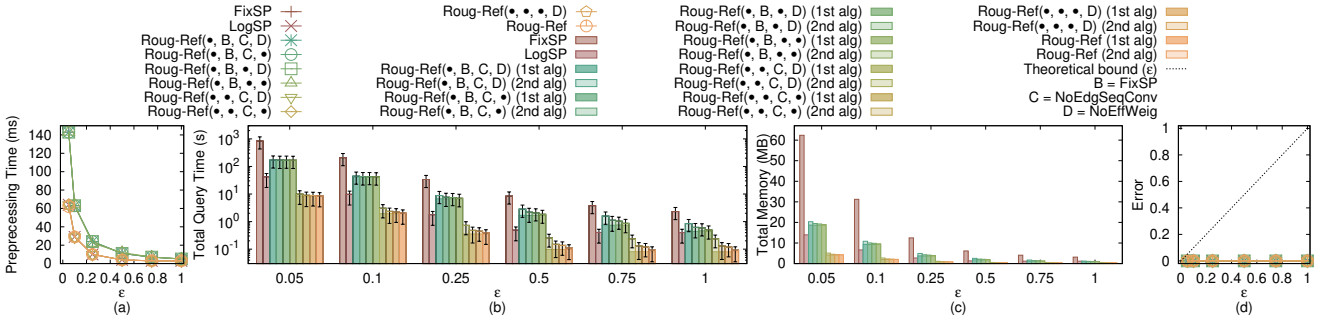

Figure 21: Effect of ϵ on *EP-small* dataset (V2V path query)

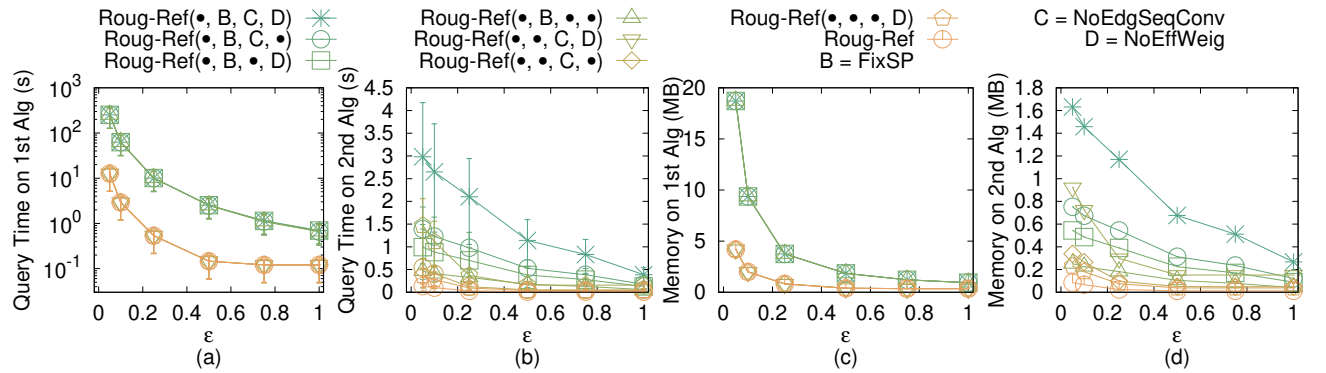

Figure 22: Effect of ϵ on *EP-small* dataset with separated query time and memory usage in two steps (V2V path query)

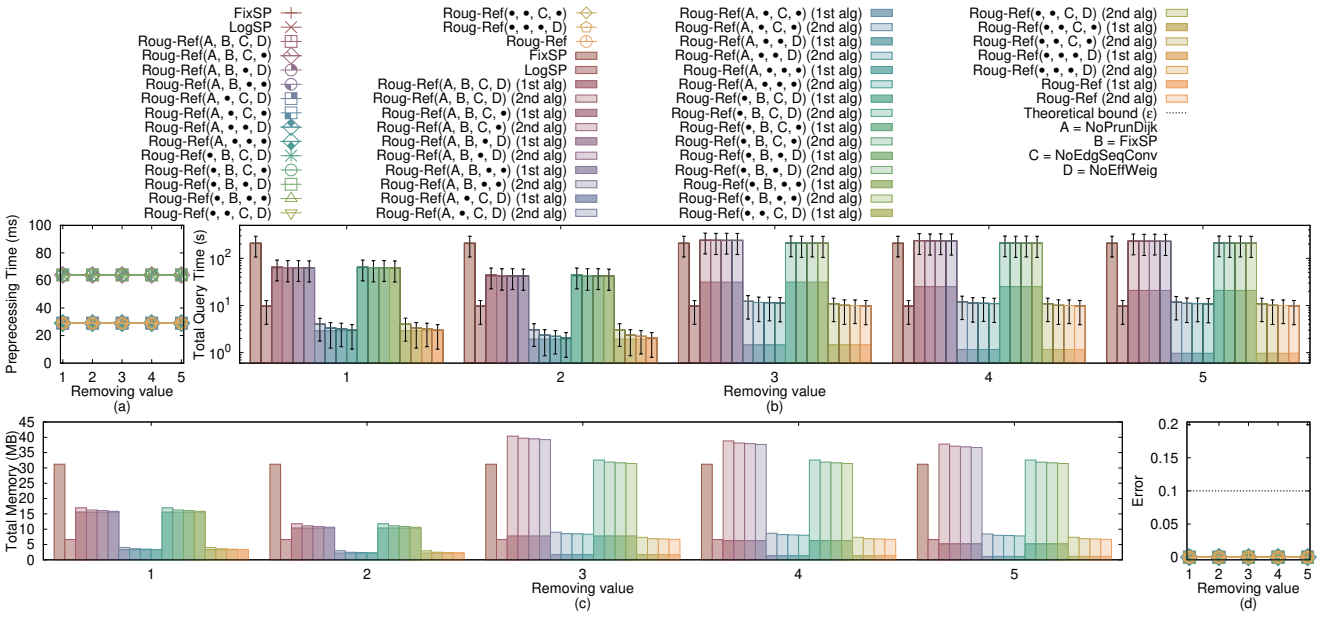

Figure 23: Effect of removing value on *EP-small* dataset (V2V path query)

regions with higher hydraulic pressure, and thus, could reduce the construction cost of undersea optical fiber cable. *Roug-Ref* still has the smallest query time and memeory usage.

F PROOFS

LEMMA F.1. *There are at most* $k_{SP} \leq 2(1 + \log_\lambda \frac{L}{r})$ *Steiner points on each edge in* E *when placing Steiner point based on* ϵ *in algorithm Roug.*

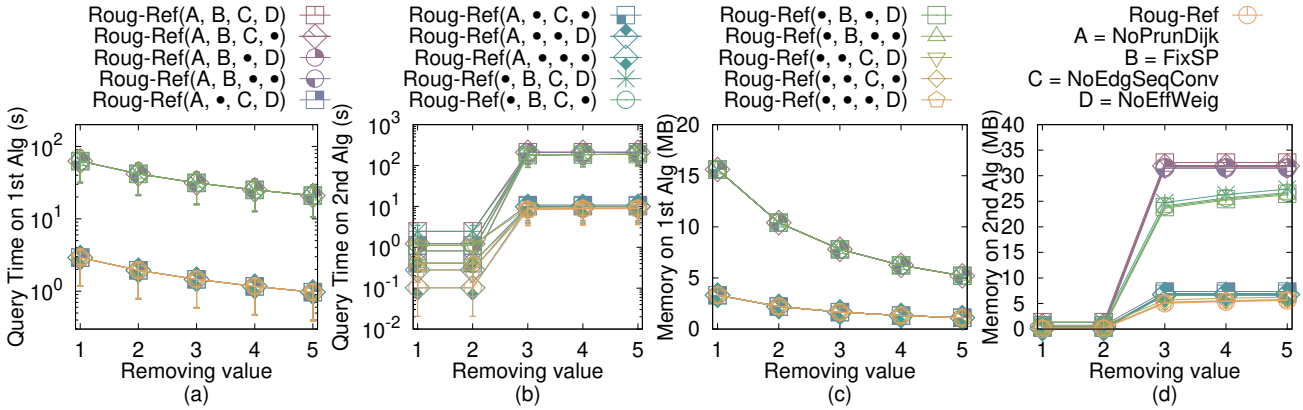

Figure 24: Effect of removing value on *EP-small* dataset with separated query time and memory usage in two steps (V2V path query)

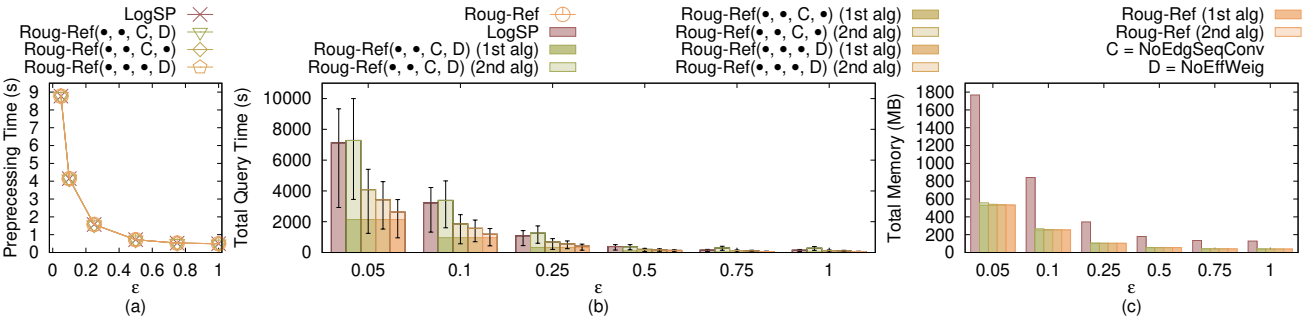

Figure 25: Effect of ϵ on *EP* dataset (V2V path query)

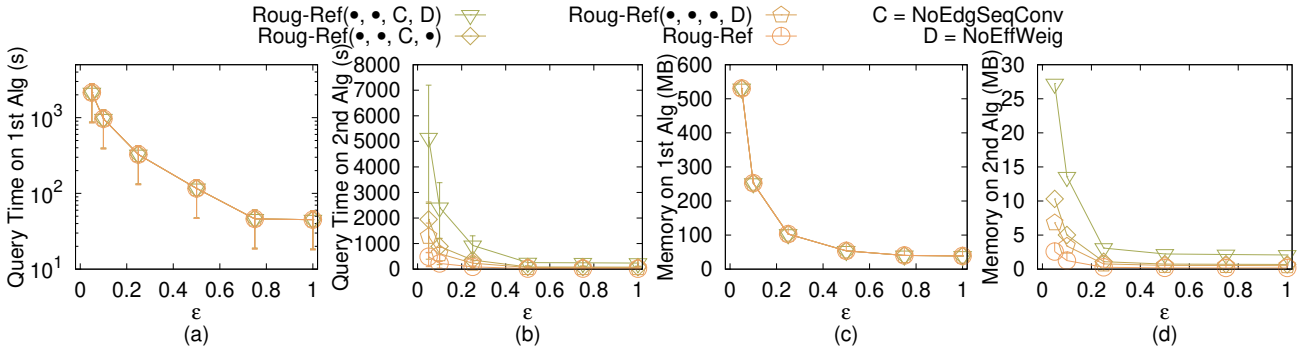

Figure 26: Effect of ϵ on *EP* dataset with separated query time and memory usage in two steps (V2V path query)

Proof. We prove it for the extreme case, i.e., k_{SP} is maximized. This case happens when the edge has maximum length L and it joins two vertices has minimum radius r. Since each edge contains two endpoints, we have two sets of Steiner points from both endpoints, and we have the factor 2. When placing Steiner point based on ϵ in algorithm *Roug*, each set of Steiner points contains at most $(1+\log_\lambda \frac{L}{r})$ Steiner points, where the 1 comes from the first Steiner point that is the nearest one from the endpoint. Therefore, we have $k_{SP} \leq 2(1+\log_\lambda \frac{L}{r})$. □

Lemma F.2. *When placing Steiner point based on ϵ in algorithm Roug, $\epsilon' = \frac{1+\epsilon+\frac{W}{w}-\sqrt{(1+\epsilon+\frac{W}{w})^2-4\epsilon}}{4}$ with $0 < \epsilon' < \frac{1}{2}$ and $\epsilon > 0$ after we express ϵ' in terms of ϵ.*

Proof. The mathematical derivation is like we regard ϵ' as an unknown and solve a quadratic equation. The derivation is as follows.

$$(2 + \frac{2W}{(1-2\epsilon')\cdot w})\epsilon' = \epsilon$$

$$2 + \frac{2W}{(1-2\epsilon')\cdot w} = \frac{\epsilon}{\epsilon'}$$

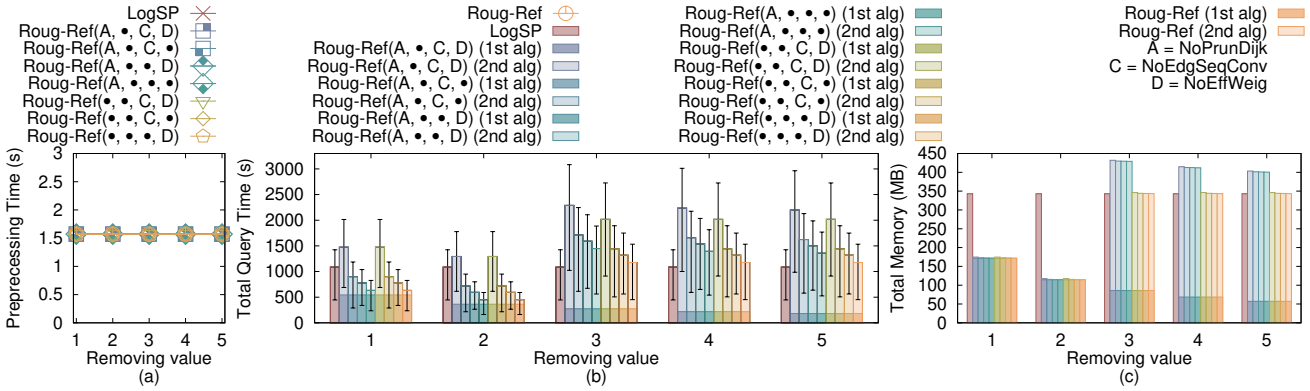

Figure 27: Effect of removing value on *EP* dataset (V2V path query)

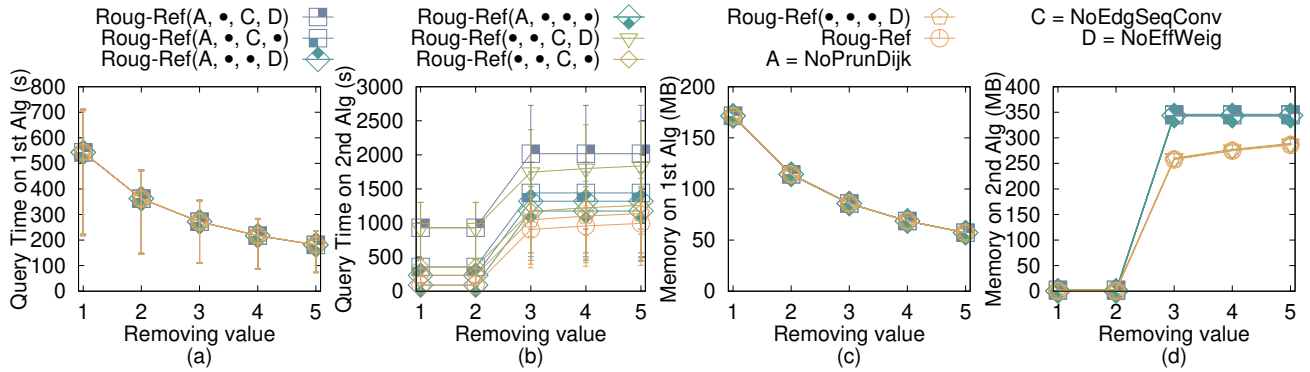

Figure 28: Effect of removing value on *EP* dataset with separated query time and memory usage in two steps (V2V path query)

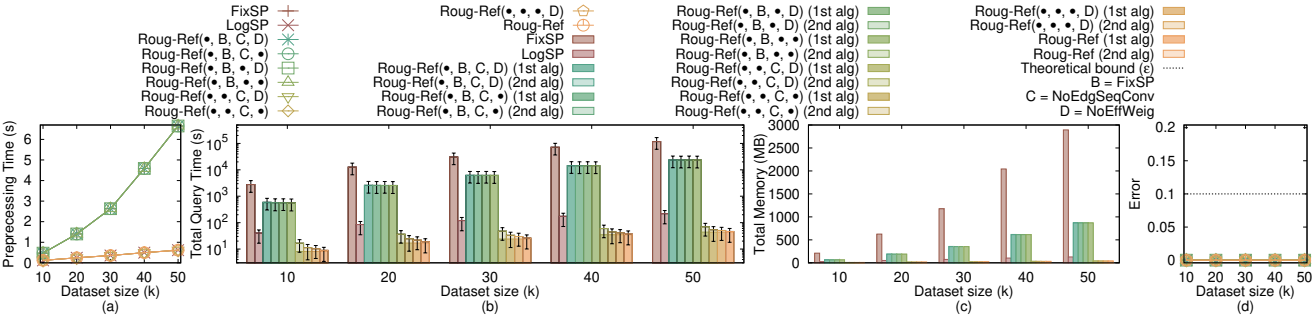

Figure 29: Effect of dataset size on multi-resolution of *EP-small* datasets (V2V path query)

$$\frac{2W}{(1-2\epsilon')\cdot w} = \frac{\epsilon - 2\epsilon'}{\epsilon'}$$

$$2\frac{W}{w}\epsilon' = \epsilon - (2+2\epsilon)\epsilon' + 4\epsilon'^2$$

$$4\epsilon'^2 - (2+2\epsilon+2\frac{W}{w})\epsilon' + \epsilon = 0$$

$$\epsilon' = \frac{2+2\epsilon+2\frac{W}{w} \pm \sqrt{4(1+\epsilon+\frac{W}{w})^2 - 16\epsilon}}{8}$$

$$\epsilon' = \frac{1+\epsilon+\frac{W}{w} \pm \sqrt{(1+\epsilon+\frac{W}{w})^2 - 4\epsilon}}{4}$$

Finally, we take $\epsilon' = \frac{1+\epsilon+\frac{W}{w} - \sqrt{(1+\epsilon+\frac{W}{w})^2-4\epsilon}}{4}$ since $0 < \epsilon' < \frac{1}{2}$ (we could plot the figure for this expression, and will found that the upper limit is always $\frac{1}{2}$ if we use −). □

Lemma F.3. *Let h be the minimum height of any face in F whose vertices have non-negative integer coordinates no greater than N. Then, $h \geq \frac{1}{N\sqrt{3}}$.*

Proof. Let a and b be two non-zero vectors with non-negative integer coordinates no greater than N, and a and b are not co-linear. Since we know $\frac{|a\times b|}{2}$ is the face area of a and b, we have

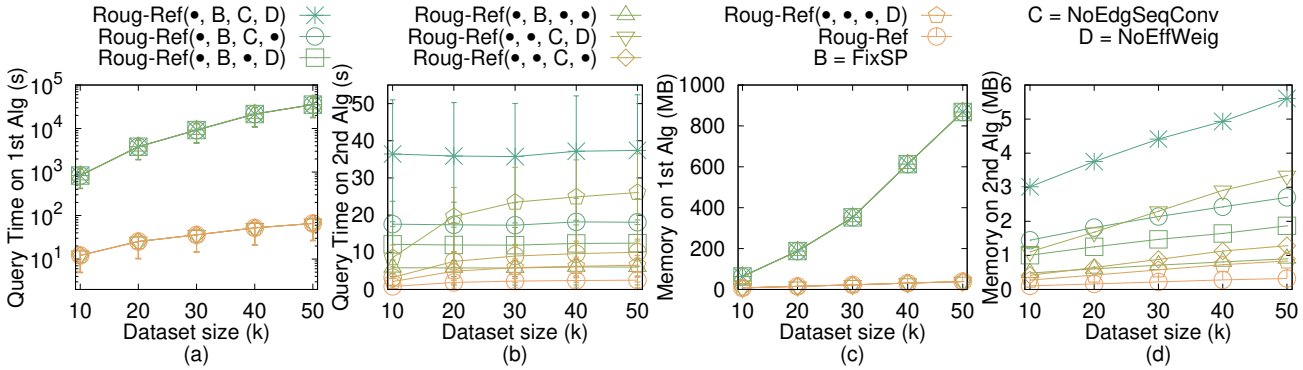

Figure 30: Effect of dataset size on multi-resolution of *EP-small* datasets with separated query time and memory usage in two steps (V2V path query)

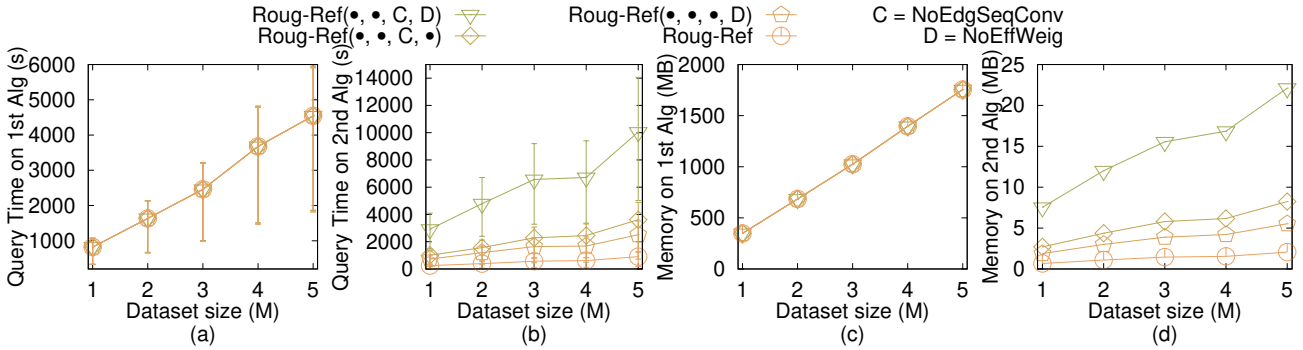

Figure 31: Effect of dataset size on multi-resolution of *EP* datasets with separated query time and memory usage in two steps (V2V path query)

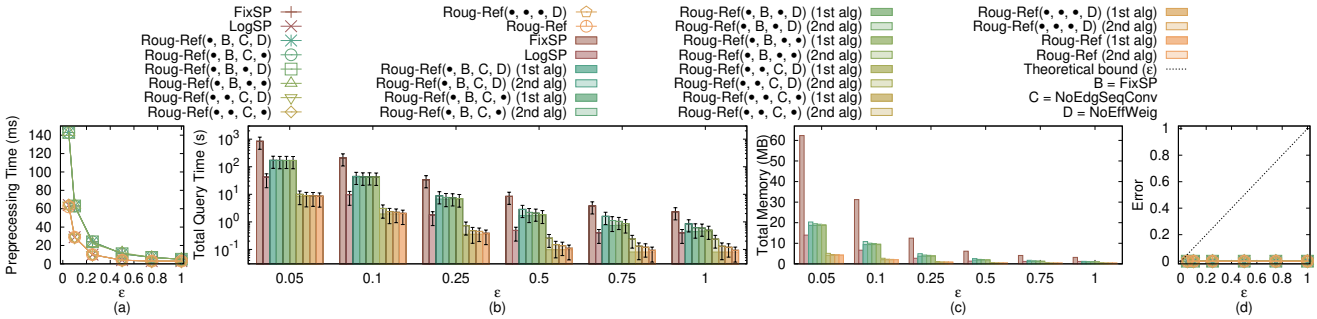

Figure 32: A2A path query on *EP-small* dataset

$h = \min_{a,b} \frac{|a \times b|}{|b|} = \min_{a,b} \frac{\sqrt{\omega}}{\sqrt{x_a^2+y_a^2+z_a^2}} \geq \frac{1}{N\sqrt{3}} \min_{a,b} \sqrt{\omega} \geq \frac{1}{N\sqrt{3}}$, where $\omega = (y_a z_b - z_a y_b)^2 + (z_a x_b - x_a z_b)^2 + (x_a y_b - y_a x_b)^2$. □

Theorem F.4. *The running time for algorithm Roug is $O(n \log n)$ and the memory usage is $O(n)$.*

Proof of Theorem F.4. Originally, if we do not remove Steiner points in algorithm *Roug*, i.e., we place Steiner point based on ϵ, then following Lemma F.1, the number of Steiner points k_{SP} on each edge is $O(\log_\lambda \frac{L}{r})$, where $\lambda = (1+\epsilon' \cdot \sin\theta)$ and $r = \epsilon' h$. Following Lemma F.2 and Lemma F.3, $r = O(\frac{\epsilon}{N})$, and thus $k_{SP} = O(\log \frac{LN}{\epsilon})$. But, since we have removed Steiner points for η calculation and rough path calculation, the remaining Steiner points on each edge is $O(1)$. So $|G_{Roug}.V| = n$. Since we know for a graph with n vertices, the running time and memory usage of Dijkstra algorithm on this graph are $O(n \log n)$ and n, so the running time of algorithm *Roug* is $O(n \log n)$ and the memory usage is $O(n)$. □

Theorem F.5. *The running time for the full edge sequence conversion step in algorithm Ref is $O(n \log n)$ and the memory usage is $O(n)$.*

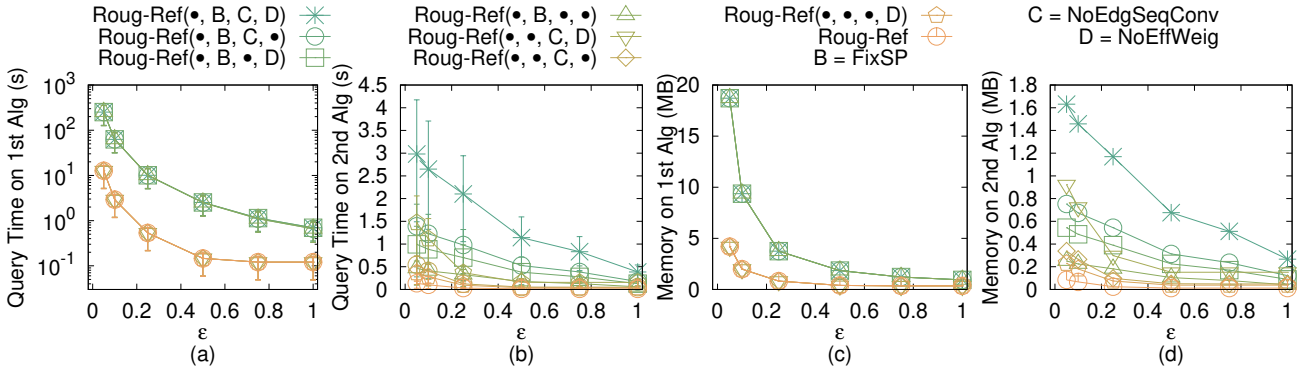

Figure 33: A2A path query on*EP-small* dataset with separated query time and memory usage in two steps

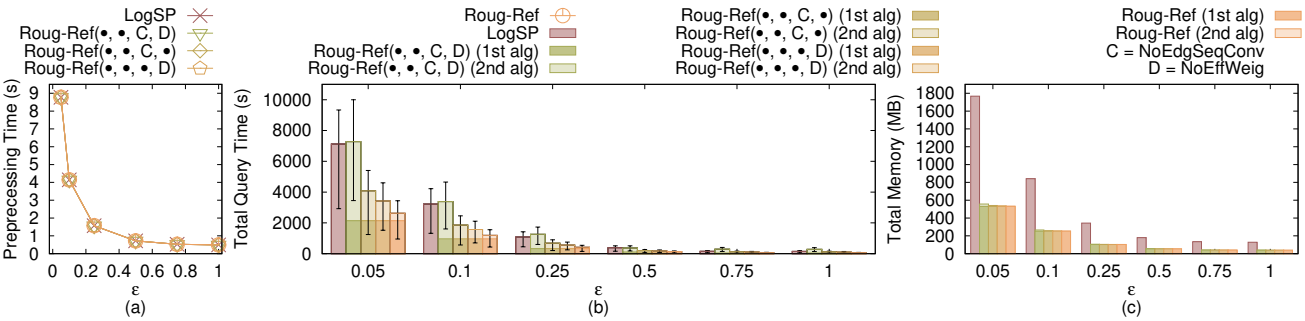

Figure 34: A2A path query on *EP* dataset

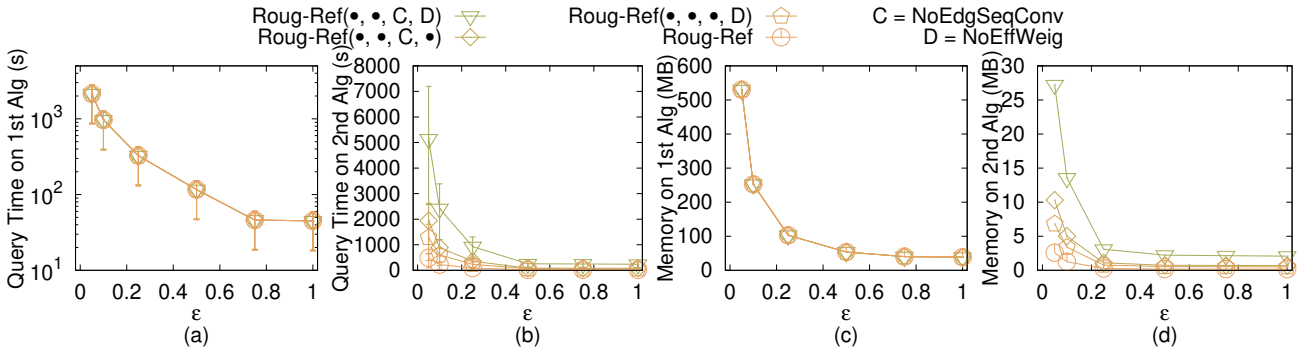

Figure 35: A2A path query on *EP* dataset with separated query time and memory usage in two steps

Proof of Theorem F.5. Firstly, we prove the running time. Given $\Pi_{Roug}(s,t)$, there are three cases on how to apply the full edge sequence conversion step in algorithm *Ref* on $\Pi_{Roug}(s,t)$, i.e., (1) some segments of $\Pi_{Roug}(s,t)$ passes on the edges (i.e., no need to use algorithm *Ref* full edge sequence conversion step), (2) some segments of $\Pi_{Roug}(s,t)$ belongs to single endpoint case, and (3) some segments of $\Pi_{Roug}(s,t)$ belongs to successive endpoint case. For the first case, there is no need to case about it. For the second case, we just need to add more Steiner points on the edges adjacent to the vertices passed by $\Pi_{Roug}(s,t)$, and using Dijkstra algorithm to refine it, and the running time is the same as the one in algorithm *Roug*, which is $O(n \log n)$. For the third case, we just need to add more Steiner points on the edge adjacent to the vertex passed by $\Pi_{Roug}(s,t)$, and find a shorter path by running for ζ times, and there are at most $O(n)$ such vertices, so the running time is $O(\zeta n) = O(n)$. Therefore, the running time for the full edge sequence conversion step in algorithm *Ref* is $O(n \log n)$.

Secondly, we prove the memory usage. Algorithm *Roug* needs $O(n)$ memory since it is a common Dijkstra algorithm, whose memory usage is $O(|G_{Roug}.V|)$, where $|G_{Roug}.V|$ is size of vertices in the Dijkstra algorithm. Handling one single endpoint case needs $O(1)$ memory. Since there could be at most n single endpoint cases, the memory usage is $O(n)$. Handling successive endpoint cases needs $O(n)$ memory since algorithm *Roug* needs $O(n)$ memory. Therefore, the memory usage for the full edge sequence conversion step in algorithm *Ref* is $O(n)$. □

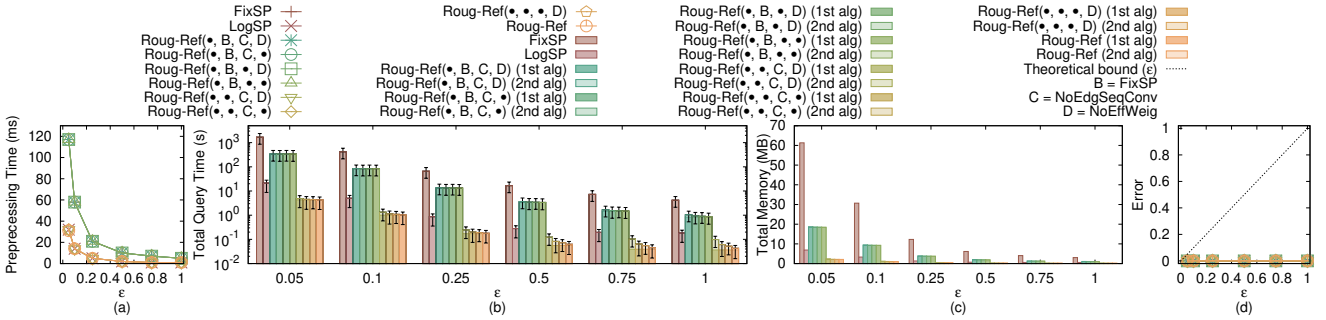

Figure 36: Effect of ϵ for Path Advisor user study

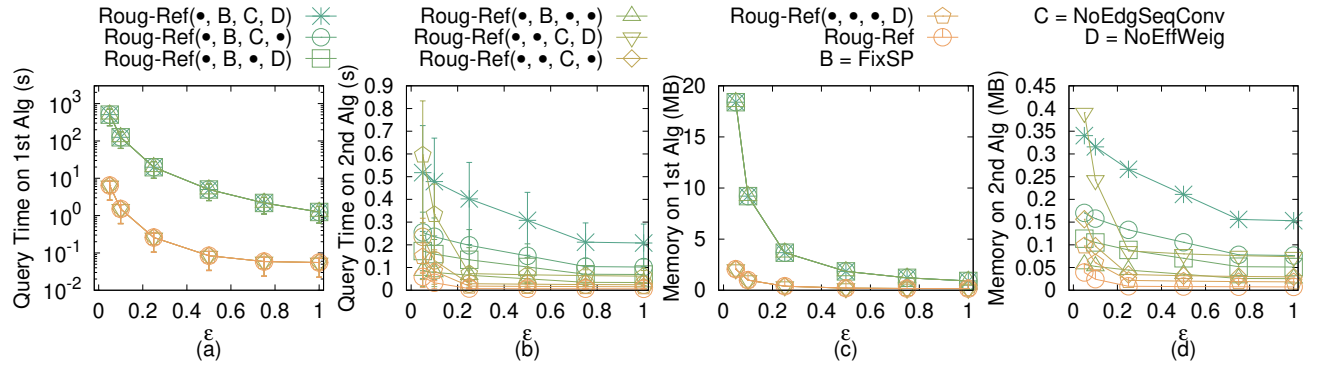

Figure 37: Effect of ϵ for Path Advisor user study with separated query time and memory usage in two steps

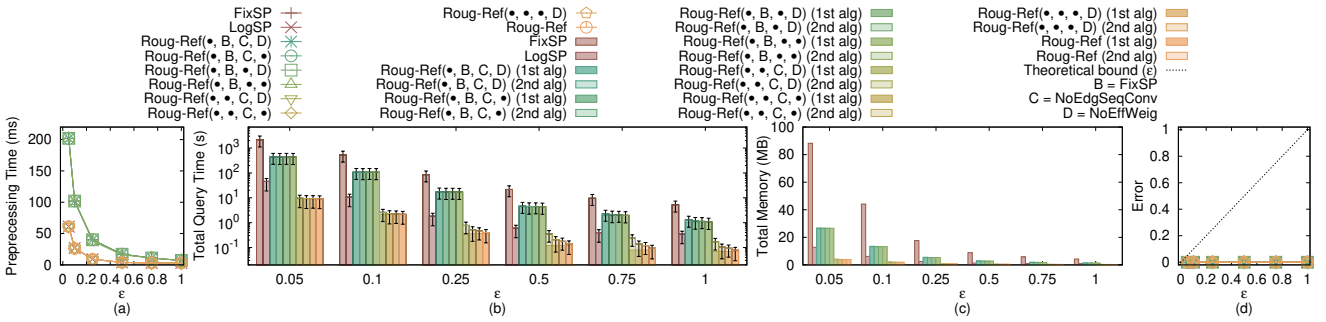

Figure 38: Effect of ϵ for Cyberpunk 2077 user study

Theorem F.6. *The running time for the Snell's law path refinement step in algorithm Ref is $O(l)$, and the memory usage is $O(l)$.*

Proof of Theorem F.5. Firstly, we prove the running time. Let l be the number of edges in S. Since the effective weight pruning sub-step could directly find the optimal position of the intersection point on the first edge in S in $O(1)$ time, so the running time of the Snell's law path refinement step in algorithm *Ref* is $O(l)$.

Secondly, we prove the memory usage, since the refined path will pass l edges, so the memory usage of the Snell's law path refinement step in algorithm *Ref* is $O(l)$. □

Proof of Theorem 3.1. Firstly, we prove the total running time. (1) In most of the cases, there is no need to use the error guaranteed path refinement step in algorithm *Ref*. In this case, the total running time is the sum of the running time using algorithm *Roug* and the first three steps in algorithm *Ref*. From Theorem F.4, Theorem F.5 and Theorem F.6, we obtain the total running time $O(n \log n + l)$. (2) In some special cases, we need to use the error guaranteed path refinement step in algorithm *Ref* for error guarantee. The sum of the running time of algorithm *Roug* and the error guaranteed path refinement step in algorithm *Ref* is exactly the same as the running time that we perform Dijkstra algorithm on the weighted graph G_{Ref} constructed by the original Steiner points (i.e., $k_{SP} = O(\log \frac{LN}{\epsilon})$ Steiner points per edge) and V, which is $O(n \log \frac{LN}{\epsilon} \log(n \log \frac{LN}{\epsilon}) + l)$. But the constant term $O(\log \frac{LN}{\epsilon})$ is not important and could be omitted, so we obtain the total running time $O(n \log n + l)$. (3) In general, the total running time is $O(n \log n + l)$.

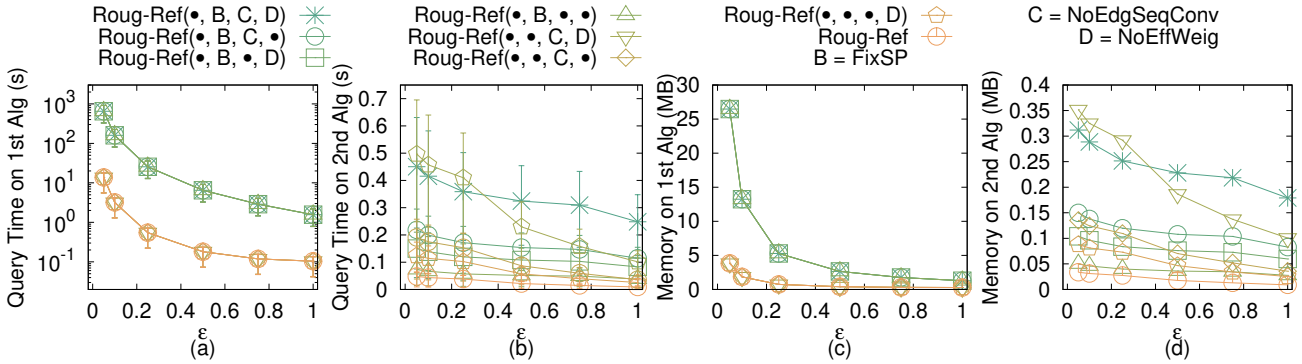

Figure 39: Effect of ϵ for Cyberpunk 2077 user study with separated query time and memory usage in two steps

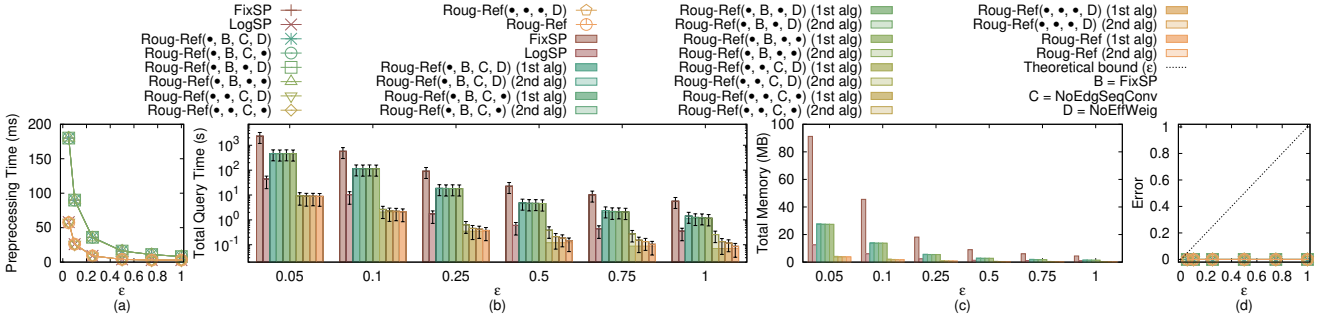

Figure 40: Effect of ϵ for seabed motivation study

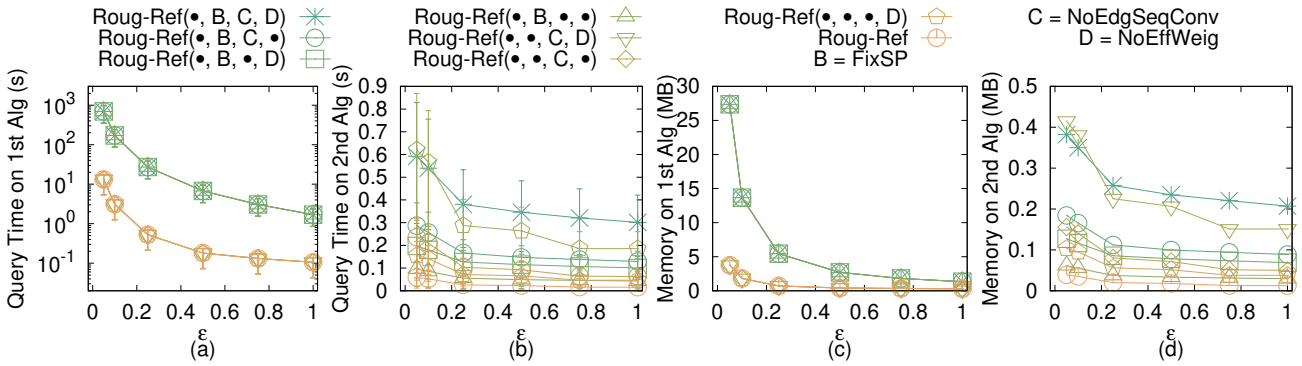

Figure 41: Effect of ϵ for seabed motivation study with separated query time and memory usage in two steps

Secondly, we prove the total memory usage. (1) In most of the cases, there is no need to use the error guaranteed path refinement step in algorithm *Ref*. In this case, the total memory usage is the sum of the memory usage using algorithm *Roug* and the first three steps in algorithm *Ref*. From Theorem F.4, Theorem F.5 and Theorem F.6, we obtain the average case total memory usage $O(n+l)$. (2) In some cases, we need to use the error guaranteed path refinement step in algorithm *Ref* for error guarantee. The sum of the memory usage of algorithm *Roug* and the error guaranteed path refinement step in algorithm *Ref* is exactly the same as the memory usage that we perform Dijkstra algorithm on the weighted graph G_{Ref} constructed by the original Steiner points (i.e., $k_{SP} = O(\log \frac{LN}{\epsilon})$ Steiner points per edge) and V, which is $O(n \log \frac{LN}{\epsilon} + l)$. But the constant term $O(\log \frac{LN}{\epsilon})$ is not important and could be omitted, so we obtain the total memory usage $O(n+l)$. (3) In general, the total memory usage is $O(n+l)$.

Finally, we prove the error bound. Recall one baseline algorithm *LogSP*, the algorithm that uses Dijkstra algorithm on the weighted graph constructed by the original Steiner point (following the Steiner point placement scheme in our algorithm with ϵ as input error) and V, i.e., the rough path calculation step of our algorithm *Roug* (by changing the input error from $\eta\epsilon$ to ϵ). We define the path calculated by algorithm *LogSP* between s and t to be $\Pi_{LogSP}(s,t)$. [12, 29] show that $|\Pi_{LogSP}(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$.

A proof sketch could be found in Theorem 1 of [12] and a detailed proof could be found in Theorem 3.1 of [29]. But, in [12, 29], they have $|\Pi_{LogSP}(s,t)| \leq (1+(2+\frac{2W}{(1-2\epsilon')\cdot w})\epsilon')|\Pi^*(s,t)|$ where $0<\epsilon'<\frac{1}{2}$. After substituting $(2+\frac{2W}{(1-2\epsilon')\cdot w})\epsilon'=\epsilon$ with $0<\epsilon'<\frac{1}{2}$ and $\epsilon>0$, we have $|\Pi_{LogSP}(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$ where $\epsilon>0$. This error bound is always true no matter whether the edge sequence passed by $\Pi_{LogSP}(s,t)$ is the same as the edge sequence passed by $\Pi^*(s,t)$ or not. Then, in algorithm *Roug*, we first remove some Steiner points in the rough path calculation step, and then calculate $\eta\epsilon$ based on the remaining Steiner points, and then use $\eta\epsilon$ as the input error to calculate $\Pi_{Roug}(s,t)$ in the rough path calculation step, so by adapt $\eta\epsilon$ as the input error in algorithm *LogSP*, we have $|\Pi_{Roug}(s,t)| \leq (1+\eta\epsilon)|\Pi^*(s,t)|$. Next, in the path checking step of algorithm *Ref*, if $|\Pi_{Ref\text{-}2}(s,t)| \leq \frac{(1+\epsilon)}{(1+\eta\epsilon)}|\Pi_{Roug}(s,t)|$, we return $\Pi_{Ref\text{-}2}(s,t)$ as output $\Pi(s,t)$, which implies that $|\Pi_{Ref\text{-}2}(s,t)| = |\Pi(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$. Otherwise, we use the error guaranteed path refinement step in algorithm *Ref*, and we return $\Pi_{Ref\text{-}3}(s,t)$ as output $\Pi(s,t)$, where the error bound is the same as in algorithm *LogSP*, i.e., $|\Pi_{Ref\text{-}3}(s,t)| = |\Pi(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$. Therefore, algorithm *Roug-Ref* guarantees that $|\Pi(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$. □

Theorem F.7. *The total running time for algorithm Roug-Ref(NoPrunDijk, •, •, •) is* $O(n\log\frac{LN}{\epsilon}\log(n\log\frac{LN}{\epsilon})+l)$*, the total memory usage is* $O(n\log\frac{LN}{\epsilon}+l)$*. It guarantees that* $|\Pi(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$.

Proof of Theorem F.7. The difference between *Roug-Ref*(*NoPrunDijk*, •, •, •) and *Roug-Ref* is that in the former one, we do not use the state information for pruning out in Dijkstra algorithm in the error guaranteed path refinement step of algorithm *Ref*. This will only affect the total running time and the total memory usage. Thus, we only prove the total running time and the total memory usage.

Firstly, we prove the total running time. (1) In most of the cases, there is no need to use the error guaranteed path refinement step in algorithm *Ref*. In this case, the total running time is the sum of the running time using algorithm *Roug* and the first three steps in algorithm *Ref*. From Theorem F.4, Theorem F.5 and Theorem F.6, we obtain the total running time $O(n\log n+l)$. (2) In some special cases, we need to use the error guaranteed path refinement step in algorithm *Ref* for error guarantee. Since we do not use the state information for pruning out in Dijkstra algorithm in the error guaranteed path refinement step of algorithm *Ref*, we need to perform Dijkstra algorithm on the weighted graph G_{Ref} constructed by the original Steiner points (i.e., $k_{SP}=O(\log\frac{LN}{\epsilon})$ Steiner points per edge) and V in the error guaranteed path refinement step, and the running time is $O(n\log\frac{LN}{\epsilon}\log(n\log\frac{LN}{\epsilon}))$, so we obtain the total running time $O(n\log\frac{LN}{\epsilon}\log(n\log\frac{LN}{\epsilon})+l)$. (3) In general, the total running time is $O(n\log\frac{LN}{\epsilon}\log(n\log\frac{LN}{\epsilon})+l)$.

Secondly, we prove the total memory usage. (1) In most of the cases, there is no need to use the error guaranteed path refinement step in algorithm *Ref*. In this case, the total memory usage is the sum of the memory usage using algorithm *Roug* and the first three steps in algorithm *Ref*. From Theorem F.4, Theorem F.5 and Theorem F.6, we obtain the average case total memory usage $O(n+l)$. (2) In some cases, we need to use the error guaranteed path refinement step in algorithm *Ref* for error guarantee. Since we do not use the state information for pruning out in Dijkstra algorithm in the error guaranteed path refinement step of algorithm *Ref*, we need to perform Dijkstra algorithm on the weighted graph G_{Ref} constructed by the original Steiner points (i.e., $k_{SP}=O(\log\frac{LN}{\epsilon})$ Steiner points per edge) and V in the error guaranteed path refinement step, and the memory usage is $O(n\log\frac{LN}{\epsilon})$, so we obtain the total memory usage $O(n\log\frac{LN}{\epsilon}+l)$. (3) In general, the total memory usage is $O(n\log\frac{LN}{\epsilon}+l)$. □

Theorem F.8. *The total running time for algorithm Roug-Ref(•, FixSP, •, •) is* $O(n^2\log n+l)$*, the total memory usage is* $O(n^2+l)$*. It guarantees that* $|\Pi(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$.

Proof of Theorem F.8. The difference between *Roug-Ref*(•, *FixSP*, •, •) and *Roug-Ref* is that in the former one, we use *FixSP* to substitute *LogSP* as Steiner points placement scheme in algorithm *Roug* and the error guaranteed path refinement in *Ref*. This will only affect the total running time and the total memory usage. Thus, we only prove the total running time and the total memory usage.

For both the total running time and the total memory usage, by using *FixSP*, we need to place $O(n^2)$ Steiner points per edge on E [30]. Since there are total n edges, there are total $O(n^3)$ Steiner points in the weighted graph, so the running time and memory usage of using Dijkstra algorithm on this graph are $O(n^3\log n)$ and $O(n^3)$. By using the framework of *Roug-Ref*, we obtain the total running time and total memory usage, i.e., $O(n^2\log n+l)$ and $O(n^2+l)$. □

Theorem F.9. *The total running time for algorithm Roug-Ref(•, •, NoEdgSeqConv, •) is* $O(n\log n+nl)$*, the total memory usage is* $O(nl)$*. It guarantees that* $|\Pi(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$.

Proof of Theorem F.9. The difference between *Roug-Ref*(•, •, *NoEdgSeqConv*, •) and *Roug-Ref* is that in the former one, we remove the full edge sequence conversion step in algorithm *Ref* (such that the input edge sequence of the Snell's law path refinement step in algorithm *Ref* may be a non-full edge sequence, then we need to try different combinations of edges when using Snell's law). This will only affect the total running time and the total memory usage. Thus, we only prove the total running time and the total memory usage.

For both the total running time and the total memory usage, there are total n different cases of the edge sequence that we need to perform Snell's law on, that is, we need to use the Snell's law path refinement step in algorithm *Ref* for n times, and select the path with the shortest distance. Therefore, the total running time and the total memory usage need to include the Snell's law path refinement step in algorithm *Ref* for n times. By using the framework of *Roug-Ref*, we obtain the total running time and total memory usage, i.e., $O(n\log n+nl)$ and $O(nl)$. □

Theorem F.10. *The total running time for algorithm Roug-Ref(•, •, •, NoEff Weig) is* $O(n\log n+l^2\log(\frac{nNW}{w\epsilon}))$*, the total memory usage is* $O(n+l)$*. It guarantees that* $|\Pi(s,t)| \leq (1+\epsilon)|\Pi^*(s,t)|$.

Proof of Theorem F.10. The difference between *Roug-Ref*(•, •, •, *NoEff Weig*) and *Roug-Ref* is that in the former one, we remove the effective weight pruning out sub-step in the Snell's law path

refinement step of algorithm *Ref*. This will only affect the total running time and the total memory usage. Thus, we only prove the total running time and the total memory usage.

Firstly, we prove the total running time. Let l be the number of edges in S. In the binary search initial path and binary search refined path finding sub-step, they first take $O(l)$ time for computing the 3D surface Snell's ray Π_m since there are l edges in S and we need to use Snell's law l times to calculate the intersection point on each edge. Then, they take $O(\log \frac{L_i}{\delta})$ time for deciding the position of m_i because we stop the iteration when $|a_i b_i| < \delta$, and they are binary search approach, where L_i is the length of e_i. Since $\delta = \frac{h\epsilon w}{6lW}$ and $L_i \leq L$ for $\forall i \in \{1, \ldots, l\}$, the running time is $O(\log \frac{lWL}{h\epsilon w})$. Since we run the above two nested loops l times, the total running time is $O(l^2 \log \frac{lWL}{h\epsilon w})$. So the running time of the Snell's law path refinement step without the effective weight pruning sub-step in algorithm *Ref* is $O(l^2 \log \frac{nWL}{h\epsilon w})$. By using the framework of *Roug-Ref*, we obtain the total running time is $O(n \log n + l^2 \log(\frac{nNW}{w\epsilon}))$.

Secondly, we prove the total memory usage, since the refined path will pass l edges, so the memory usage of the Snell's law path refinement step without the effective weight pruning sub-step in algorithm *Ref* is $O(l)$. By using the framework of *Roug-Ref*, we obtain the total memory usage is $O(n + l)$. □