SHORTEST PATH QUERIES ON TRIANGULAR IRREGULAR NETWORKS AND POINT CLOUDS

by

YINZHAO YAN

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

August 2025, Hong Kong

SHORTEST PATH QUERIES ON TRIANGULAR IRREGULAR NETWORKS AND POINT CLOUDS

by

YINZHAO YAN

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

Performing shortest path queries on a 3D surface is a topic of widespread interest in both industry and academia. Among different representations of a 3D surface, the most popular representations are *Triangular Irregular Network*, i.e., *TIN*, and *point cloud*. However, all existing shortest path query algorithms on a *TIN* are inefficient, and there is no existing shortest path query algorithm on a point cloud. In this thesis, we study how to effectively calculate the shortest path passing on a *TIN* and a point cloud in three aspects. (1) We propose an efficient on-the-fly shortest path algorithm answering the shortest path passing different regions on a weighted *TIN*, where different regions are assigned different weights. (2) We propose an efficient updatable shortest path oracle answering the shortest path query for a set of *Points-Of-Interests* (*POIs*) on an updated *TIN*. (3) We propose an efficient shortest path oracle answering the shortest path query for a set of *POIs* on a point cloud, and an efficient proximity query algorithm using our oracle. Our experimental results show that they outperform the best-known algorithms or oracles concerning time and memory.

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

_____

YINZHAO YAN

1 August 2025

SHORTEST PATH QUERIES ON TRIANGULAR IRREGULAR NETWORKS AND POINT CLOUDS

by

YINZHAO YAN

This is to certify that I have examined the above Ph.D. thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Prof. Raymond Chi-Wing WONG, Thesis Supervisor

Prof. Xiaofang ZHOU, Head of Department

Department of Computer Science and Engineering

1 August 2025

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to my esteemed supervisor, Prof. Raymond Chi-Wing Wong, for his consistent support of my research, career, and personal growth over the years. He has been a friend to discuss daily life, a wise scholar to guide me through challenges, and a mentor providing insightful advice. Under his patient mentorship, I have successfully explored the fascinating world of academic research. Throughout my PhD journey, he always provided me with valuable support and professional advice. I particularly appreciate the academic freedom that he granted me in choosing research topics. His role as both supervisor and life mentor has been truly exceptional. Apart from conducting research, his excellent interpersonal style also influenced me throughout my future career.

I would also like to express my gratitude to Professor Christian S. Jensen for his exceptional academic guidance. His insightful comments on my paper and detailed manuscript revisions significantly enhanced the paper's quality.

I am truly grateful to my thesis defense committee members, Prof. Andrew Wing On Poon (chairperson), Prof. Cyrus Shahabi, Prof. Xueqing Zhang, Prof. Xiaofang Zhou and Dr. Binhang Yuan. I am equally indebted to my thesis proposal defense committee members and my PhD qualifying examination committee members, Prof. Dimitris Papadias, Prof. Pedro Sander and Prof. Xiaofang Zhou. Their valuable insights, constructive feedback, and expertise greatly elevated the quality of my work.

Finally, I would like to express my heartfelt thanks to my family. My parents have always been a source of unwavering support and love during my PhD journey. Your backing has truly empowered me.

TABLE OF CONTENTS

Title Page i

Abstract ii

Authorization Page iii

Signature Page iv

Acknowledgments v

Table of Contents vi

List of Figures ix

List of Tables xi

Chapter 1 Introduction 1

1.1 Motivations 2

1.1.1 On-the-fly Algorithm for Shortest Path Queries on Weighted *TIN*s 2

1.1.2 Oracle for Shortest Path Queries on Updated *TIN*s 3

1.1.3 Oracle for Proximity Queries on Point Clouds 3

1.1.4 Example 4

1.2 Three Studies 5

1.2.1 On-the-fly Algorithm for Shortest Path Queries on Weighted *TIN*s 6

1.2.2 Oracle for Shortest Path Queries on Updated *TIN*s 8

1.2.3 Oracle for Proximity Queries on Point Clouds 10

1.3 Organizations 12

Chapter 2 Related Work 13

2.1 On-the-fly Algorithm for Shortest Path Queries on Weighted *TIN*s 13

2.1.1 On-the-fly Algorithms on Weighted *TIN*s 13

2.1.2 On-the-fly Algorithms on Unweighted *TIN*s 15

2.2 Oracle for Shortest Path Queries on Updated *TIN*s 16
2.2.1 On-the-fly Algorithms on *TIN*s 16
2.2.2 Oracles on *TIN*s 17
2.2.3 Oracles on Road Networks 18
2.2.4 Sub-graph Generation Algorithms 18
2.3 Oracle for Proximity Queries on Point Clouds 19
2.3.1 On-the-fly Algorithms on Point Clouds 19
2.3.2 Oracles for the Shortest Path Query on Point Clouds 20
2.3.3 Oracles for Other Proximity Queries on Point Clouds 22

Chapter 3 On-the-fly Algorithm for Shortest Path Queries on Weighted TINs 23
3.1 Preliminary 23
3.1.1 Notation and Definitions 23
3.1.2 Problem 24
3.2 Methodology 25
3.2.1 Overview of Algorithm *Roug-Ref* 25
3.2.2 Key Ideas of Algorithm *Rough-Refine* 29
3.2.3 Key Ideas of Additional Techniques 29
3.2.4 Implementation Details of Algorithm *Roug* 32
3.2.5 Implementation Details of Algorithm *Ref* 32
3.2.6 Theoretical Analysis 36
3.3 Empirical Studies 41
3.3.1 Experimental Setup 41
3.3.2 Experimental Results 43

Chapter 4 Oracle for Shortest Path Queries on Updated TINs 48
4.1 Preliminary 48
4.1.1 Notation and Definitions 48
4.1.2 Problem 50
4.1.3 Property 51
4.2 Methodology 52
4.2.1 Overview of *UP-Oracle* 52
4.2.2 Key Ideas of *UP-Oracle*'s Update Phase 54

4.2.3 Implementation Details of *UP-Oracle*'s Update Phase 58
4.2.4 Handling Subsequent Changes 67
4.2.5 Adaptation to Multi-layer Structure (*UP-Oracle-MuLa*) 67
4.2.6 Adaptation to the AR2AR Query (*UP-Oracle-AR2AR*) 69
4.2.7 Theoretical Analysis 69
4.3 Empirical Studies 72
4.3.1 Experimental Setup 72
4.3.2 Experimental Results 74

Chapter 5 Oracle for Proximity Queries on Point Clouds 81
5.1 Preliminary 81
5.1.1 Notation and Definitions 81
5.1.2 Problem 84
5.2 Methodology 84
5.2.1 Overview of *RC-Oracle* 84
5.2.2 Key Ideas of *RC-Oracle* 86
5.2.3 Implementation Details of *RC-Oracle* 88
5.2.4 Adaptation to *RC-Oracle-A2A* 91
5.2.5 Proximity Query Algorithms 92
5.2.6 Theoretical Analysis 94
5.3 Empirical Studies 98
5.3.1 Experimental Setup 98
5.3.2 Experimental Results for *TIN*s 101
5.3.3 Experimental Results for Point Clouds 104

Chapter 6 Conclusion 110

Publication 112

References 113

LIST OF FIGURES

1.1 A 3D surface, a *TIN* and a point cloud 2

1.2 An illustration of Snell's law 7

3.1 Algorithm *Roug-Ref* framework overview description 25

3.2 Algorithm *Roug-Ref* framework overview 26

3.3 Full edge sequence conversion 27

3.4 Placement of Steiner points 30

3.5 Snell's law path refinement step in algorithm *Ref* 31

3.6 Ablation study (effect of k on *BH-small* dataset) regarding algorithm *Roug-Ref* 44

3.7 Ablation study (effect of k on *BH* dataset) regarding algorithm *Roug-Ref* 44

3.8 Baseline comparisons (effect of ϵ on *EP-small* dataset) regarding algorithm *Roug-Ref* 45

3.9 Baseline comparisons (effect of dataset size on multi-resolution of *EP-small* datasets) regarding algorithm *Roug-Ref* 46

3.10 Scalability test regarding algorithm *Roug-Ref* 46

4.1 *UP-Oracle* framework overview 49

4.2 An unaffected path 50

4.3 *UP-Oracle* framework overview description 52

4.4 Exact shortest path update step in *UP-Oracle* 54

4.5 Sub-graph generation step in *UP-Oracle* 57

4.6 Ablation study on *TJ* dataset with fewer POIs for the P2P query regarding *UP-Oracle* 75

4.7 Ablation study on *SC* dataset with more POIs for the P2P query regarding *UP-Oracle* 75

4.8 Baseline comparisons (effect of ϵ on *GI* dataset with fewer POIs for the P2P query) regarding *UP-Oracle* 76

4.9 Baseline comparisons (effect of n on *AU* dataset with fewer POIs for the P2P query) regarding *UP-Oracle* 77

4.10 Scalability test (effect of *DS* on *LH* dataset with more POIs for the P2P query) regarding *UP-Oracle* 77

4.11 Baseline comparisons on *SC* dataset for multi-layer structure regarding *UP-Oracle* 78

4.12 Baseline comparisons on *SC* dataset for the AR2AR query regarding *UP-Oracle* 79

5.1 A small point cloud and a *TIN* 82

5.2 *RC-Oracle* framework overview description 84

5.3 *RC-Oracle* framework overview 84

5.4 An illustration of *SE-Oracle-Adapt* 88

5.5 Baseline comparisons (effect of ϵ on BH_t-*small TIN* dataset for the P2P query) regarding *RC-Oracle-Adapt* 103

5.6 Baseline comparisons (effect of π on EP_t-*small TIN* dataset for the P2P query) regarding *RC-Oracle-Adapt* 103

5.7 Baseline comparisons (effect of ϵ on EP_p-*small* point cloud dataset for the P2P query) regarding *RC-Oracle* 105

5.8 Baseline comparisons (effect of π on GF_p point cloud dataset for the P2P query) regarding *RC-Oracle* 106

5.9 Baseline comparisons (effect of N on LM_p point cloud dataset for the P2P query) regarding *RC-Oracle* 106

5.10 Baseline comparisons on EP_p point cloud dataset for the A2A query regarding *RC-Oracle-A2A* 107

5.11 Ablation study on RM_p point cloud dataset for the P2P query regarding *RC-Oracle* 107

5.12 The shortest path passing on a point cloud, the shortest surface and network path passing on a *TIN* 109

LIST OF TABLES

1.1 P2P, AR2AR and A2A queries 8

3.1 Frequent used notations in the study of shortest path queries on weighted *TIN*s 24

3.2 *TIN* datasets in the study of shortest path queries on weighted *TIN*s 42

3.3 Comparison of algorithms on a weighted *TIN* regarding algorithm *Roug-Ref* 42

4.1 Frequent used notations in the study of shortest path queries on updated *TIN*s 50

4.2 *TIN* datasets in the study of shortest path queries on updated *TIN*s 72

4.3 Comparison of algorithms on an updated *TIN* regarding *UP-Oracle* 73

5.1 Frequent used notations in the study of proximity queries on point clouds 83

5.2 Point cloud datasets in the study of proximity queries on point clouds 99

5.3 Comparison of algorithms (support the shortest path query) on a point cloud regarding *RC-Oracle* 100

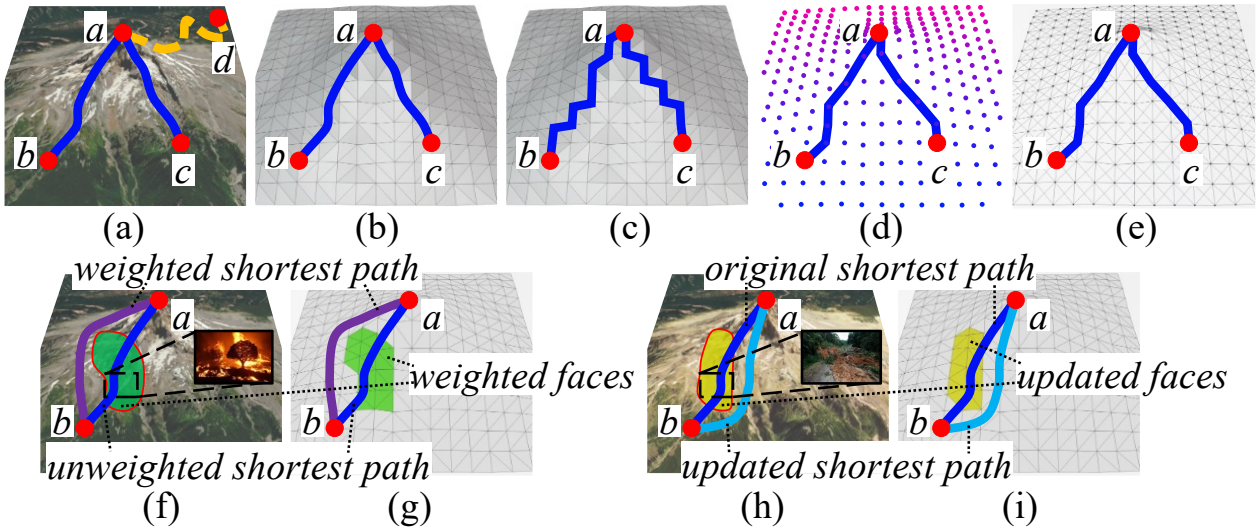
CHAPTER 1

INTRODUCTION

Performing the *shortest path query* on a 3D surface is a topic of widespread interest in both industry and academia [31, 33, 55, 64, 72, 76, 77, 82]. In industry, Google Earth [3] and Metaverse [9] utilize shortest paths passing on 3D surfaces (such as Earth and virtual reality) for route planning. In academia, the shortest path query on a 3D surface is a prevalent research topic in the field of databases [45, 46, 51, 61, 62, 71, 72, 73, 75, 78, 79, 82, 83].

***TIN* and point cloud**: There are different representations of a 3D surface, including *Triangular Irregular Network*, i.e., *TIN* [72, 76, 77], and *point cloud* [82]. (1) A *TIN* contains a set of contiguous *faces* each of which is denoted by a triangle. Each face consists of three *edges* connecting at three *vertices*. Figure 1.1 (a) shows a 3D surface in a 20km $\times$ 20km region in Santa Monica Mountains National Recreation Area [63], California, USA. Figures 1.1 (b) and (c) show a *TIN* of this surface, and the shortest *surface* path [46] passing on (the faces of) and the *network* path [46] passing on (the edges of) the *TIN*, respectively. (2) A point cloud contains a set of *points* in 3D space (correspond to the vertices of the *TIN*). Figure 1.1 (d) shows a point cloud of this surface, and the shortest path passing on a point cloud (graph). For the *point cloud graph*, its *vertices* consist of the points in the point cloud. Its *edges* consist of a set of edges between each vertex and its 8 neighbor vertices in the 2D plane. Each edge's weight is set to the Euclidean distance between its two vertices. Figure 1.1 (e) shows a point cloud graph.

Existing studies: There are some existing on-the-fly algorithms [25, 45, 46, 49, 54, 64, 71, 72, 75, 76] for answering shortest path queries on *TIN*s. But, there is no study answering shortest path queries on point clouds directly. Existing algorithms [59, 65, 86] convert a point cloud to a *TIN* via triangulation [37] (where all vertices of faces are points in the point cloud), and then calculate the shortest path passing on this *TIN*.



Remark: (a) Shortest paths passing on a 3D surface, shortest (b) surface and (c) network paths passing on a *TIN*, shortest paths passing on (d) a point cloud and (e) a point cloud graph, weight and unweighted shortest path passing on (f) a 3D surface and (g) a *TIN*, original and updated shortest path passing on (h) an updated 3D surface and (i) an updated *TIN*.

Figure 1.1. A 3D surface, a *TIN* and a point cloud

1.1 Motivations

1.1.1 On-the-fly Algorithm for Shortest Path Queries on Weighted *TIN*s

Given a *TIN*, existing *TIN* shortest path query algorithms [25, 45, 46, 49, 54, 64, 71, 72, 75, 76] only answer the *unweighted* shortest path passing on an *unweighted TIN*, where all faces of the *TIN* have the same weights (i.e., their face weights are set to a fixed value, e.g., 1). They cannot calculate *weighted* shortest path passing on a *weighted TIN*, where each face of the *TIN* has a *weight* (e.g., different regions assigned different weights depending on the application nature). The unweighted case is a particular instance of the weighted case where all face weights are set to a fixed value. We aim to efficiently find the weighted shortest path passing on a weighted *TIN*. Figures 1.1 (f) and (g) show weighted and unweighted shortest paths in the purple and blue lines from source a to destination b. Computing the weighted shortest path passing on a *TIN* is involved in numerous applications with different interpretations of the faces' weights on the *TIN* (e.g., water and forest), including autonomous vehicles' obstacle avoidance path planning [38, 74] and human's route-recommendation systems [69, 78].

1.1.2 Oracle for Shortest Path Queries on Updated *TIN*s

1) Usage of oracles Given a *TIN*, existing *TIN* shortest path query algorithms [25, 45, 46, 49, 54, 64, 71, 72, 75, 76] are slow, especially when we need to calculate more than one shortest path with different sources and destinations. But, if we pre-compute the shortest paths among these possible query points, known as *Points-Of-Interests* (*POIs*), by means of indexing (called an *oracle*) on a *TIN*, then we can use the oracle to answer the shortest path query more efficiently. POIs can be viewing platforms, shelters and hotels in a national park.

2) Updated *TIN* Existing algorithms are also slow on an updated *TIN*. Thus, we also aim to construct an oracle on an updated *TIN*. For example, after landslides or earthquakes, we aim to find the shortest evacuation paths efficiently for life-saving. After marsquake (observed by NASA's InSight lander on May 4, 2022 [47]), mars rovers aim to find the shortest escape paths quickly to avoid damage. If we pre-compute shortest paths (among viewing platforms, shelters, hotels or Mars rover working stations) using an oracle on *TIN*s prone to these disasters, and efficiently update the oracle after the disaster, then we can use it to efficiently return shortest paths. Figures 1.1 (a) and (b) (resp. Figures 1.1 (h) and (i)) show the original and updated shortest path in blue and light blue lines from source a to destination b on a 3D surface (resp. a *TIN*) before and after *TIN* updates.

1.1.3 Oracle for Proximity Queries on Point Clouds

1) Advantages of point cloud Given a point cloud, although we can convert the point cloud to a *TIN* and calculate the shortest path passing on this converted *TIN* using existing *TIN* shortest path query algorithms [25, 45, 46, 49, 54, 64, 71, 72, 75, 76], there are four advantages point clouds compared with *TIN*s, such that we aim at directly calculate the shortest path passing on a point cloud. (i) *More direct access to point cloud data.* We can obtain a point cloud using an iPhone LiDAR scanner in 10s [66] or using a satellite in 3s for a region of $1km^2$ [60]. But, a *TIN* is usually converted from a point cloud via *triangulation* [37]. (ii) *Lower hard disk usage of a point cloud.* We only store point information of a point cloud, but store vertex, edge and face information of a *TIN*. (iii) *Faster shortest path query time on a point cloud.* Calculating the shortest path passing on a point cloud

is faster than calculating the shortest surface path passing on a *TIN*. Since a *TIN* is more complicated than a point cloud. (iv) *Small distance error of the shortest path passing on a point cloud (compared with the shortest network path passing on a TIN).* In Figures 1.1 (b) and (c), the shortest surface and network paths passing on a *TIN* are very different (since each vertex only connects with 6 neighbor vertices). But, in Figures 1.1 (b) and (d), the shortest surface path passing on a *TIN* is similar to the shortest path passing on a point cloud (since each point connects with 8 neighbor points).

2) Usage of oracles in proximity queries Although calculating the shortest path passing on a point cloud is faster compared with on a *TIN*, it is still costly in some proximity queries [83], including *k-Nearest Neighbor* (*kNN*) *queries* [31, 33, 64, 72, 76, 83] and *range queries* [55, 68, 83]. We can pre-compute the shortest paths among POIs using an oracle on a point cloud, and then we can use the oracle to answer the proximity query more efficiently.

1.1.4 Example

In January 2025, the wildfire in Southern California, USA [6] affected the Los Angeles metropolitan area and surrounding regions. Over 30 people have died, and more than 17,000 buildings have been destroyed or damaged. Figure 1.1 (a) shows a 3D surface of Santa Monica Mountains National Recreation Area [63] affected by this wildfire. In this case, staffs will evacuate tourists in the mountain to the closest shelters or hotels immediately for tourists' safety.

1) On-the-fly algorithm for shortest path queries on weighted *TIN*s In Figures 1.1 (f) and (g), the 3D surface and the *TIN* consist of destroyed and non-destroyed regions (faces with green and gray color). During the wildfire, especially when the fire is very strong, it is difficult and dangerous to pass through the destroyed region, so we set each *TIN* face's weight to be the *damage level* [67] (a static value) of each region, i.e., *TIN* faces of the destroyed regions will have a larger weight, while the *TIN* faces of the non-destroyed regions will have a smaller weight. The purple and blue paths between point a (a viewing platform) and point b (a shelter/hotel) have *weighted distances* of 15.2km and 209km, respectively, so we choose the blue path that does not pass the destroyed region as the evacuation path. We cannot pre-compute the paths passing on the weighted *TIN* before

the wildfire since we do not know the damage level of the destroyed regions beforehand. But, after the wildfire, we efficiently calculate the weighted shortest path for evacuation.

2) Oracle for shortest path queries on updated *TIN*s Since the wildfire also causes landslides [7], resulting the changes to the *TIN*. If the fire is not strong but causes a landslide, there is no need to consider the destroyed and non-destroyed regions on the weighted *TIN*, but we need to calculate paths passing on the updated *TIN*. In Figures 1.1 (h) and (i), the 3D surface and the *TIN* consist of updated faces (faces with yellow color). When we need to calculate more than one shortest path with different sources and destinations among POIs a to d (viewing platform or shelter/hotel), we can build an oracle before the wildfire. After the landslide, we efficiently update the oracle and use it to efficiently answer the shortest path query for evacuation.

3) Oracle for proximity queries on point clouds If we can start the evacuation earlier when the fire just started, there is no need to consider the destroyed and non-destroyed regions on the weighted *TIN*, and no need to consider the updated *TIN*. We can use the point cloud for faster calculation. In Figure 1.1 (a), when the number of tourists is large and the capacity of each shelter/hotel is limited, we need to find the shortest paths (in blue and yellow lines) from POI a (one of the viewing platforms) to its k-nearest POIs b to d (k-nearest shelters/hotels). In Figures 1.1 (d), POIs b and c are the k-nearest POIs to POI a where $k = 2$. We build an oracle on the point cloud and use it to efficiently answer proximity queries for evacuation.

1.2 Three Studies

There are three studies in this thesis. Firstly, we investigate shortest path queries on a weighted *TIN* using an on-the-fly algorithm, published in MDM 2024 [79]. Secondly, we explore shortest path queries on an updated *TIN* using an oracle, published in TKDE 2024 [84]. Thirdly, we examine proximity queries on a point cloud using an oracle, published in SIGMOD 2024 [82]. We list these three studies as follows.

- **Yinzhao Yan** and Raymond Chi-Wing Wong,
  "*Efficient Shortest Path Queries on 3D Weighted Terrain Surfaces for Moving Objects*",

the 25th IEEE International Conference on Mobile Data Management (MDM 2024), Brussels, Belgium on 24-27 June, 2024 (Acceptance 21/78 = 26.92%), Selected as the best paper (out of 21 accepted papers)

- **Yinzhao Yan**, Raymond Chi-Wing Wong and Christian S. Jensen, "*An Efficiently Updatable Path Oracle for Terrain Surfaces*", IEEE Transactions on Knowledge and Data Engineering (TKDE), 2024

- **Yinzhao Yan** and Raymond Chi-Wing Wong, "*Proximity Queries on Point Clouds using Rapid Construction Path Oracle*", the 2024 ACM Conference on Management of Data (SIGMOD), Santiago, Chile on 9-15 June, 2024

The high-level problems of these three studies are the same, i.e., we study the shortest path queries on a 3D surface. We then present our contribution of these three studies as follows.

1.2.1 On-the-fly Algorithm for Shortest Path Queries on Weighted *TIN*s

1) Snell's Law Before we present our study, we give the concept of Snell's law, which is an important geometric information of the *TIN* and can be applied to calculating the weighted shortest path passing on the weighted *TIN*. In Physics, when a light ray passes the boundary between two different media (e.g., air and glass), it bends at the boundary since light seeks the path with the minimum time. The incidence angle and refraction angle for the light satisfy *Snell's law* [53] (see Figures 1.2 (a) and (b)). In Figure 1.2 (a), if all faces have the same weights (=1), the blue line is the (unweighted) shortest path. When faces f_1 and f_2 have weights 3 and 2, the purple line with a weighted distance of 4 that satisfies Snell's law is the weighted shortest path, but the weighted distance of the blue line is $5.2 > 4$. Figure 1.2 (b) shows a similar example.

2) Challenges There is no algorithm capable of calculating the exact shortest path passing on the weighted *TIN* when the number of faces in the weighted *TIN* exceeds two [30]. Existing algorithms [20, 43, 45, 49, 69] can only solve it on-the-fly *approximately*. Even the best-known $(1+\epsilon)$-approximate algorithm [43, 49] for calculating the weighted shortest

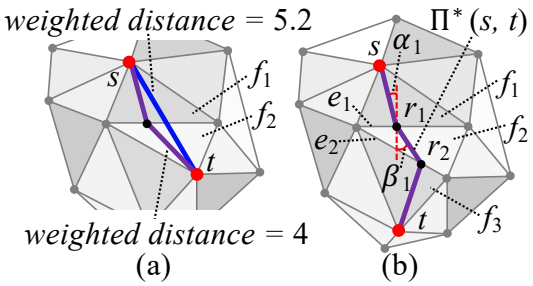


Figure 1.2. An illustration of Snell's law

path is slow, since it does not utilize any geometric information on the weighted *TIN*. Our experimental results show that it needs more than one day for querying.

3) Our algorithm We propose an efficient two-step $(1+\epsilon)$-approximate on-the-fly shortest path algorithm that answers the shortest path query on a weighted *TIN* called *Rough-Refine*, i.e., *Roug-Ref*, where $\epsilon > 0$ is called the *error parameter*. Algorithm *Roug-Ref* is a step-by-step algorithm involving algorithm *Roug* and algorithm *Ref*. Algorithm *Roug-Ref* achieves outstanding performance concerning the query time and memory usage due to the rough-refine concept, i.e., (i) a *novel* pruning step in algorithm *Roug* during the rough path calculation with error guarantee (achieved by transferring the pruned-out information from algorithm *Roug* to algorithm *Ref*, and after conducting necessary checks in algorithm *Ref*, there is no need to perform calculations on the pruned-out information anymore), (ii) an *efficient* reduction of the search area in algorithm *Roug* before Snell's law refinement (achieved by the calculation of the rough path), and (iii) the usage of *Snell's law* in algorithm *Ref* for efficient refinement.

We have four additional novel techniques for further speedup, including (i) an *efficient* Steiner point placement scheme in algorithm *Roug* for reducing the number of Steiner points during the rough path calculation (achieved by considering geometric information of each face on the weighted *TIN*, such as face weight, internal angle and edge length), (ii) a *progressive* approach in algorithm *Ref* for minimizing the search area before Snell's law refinement (achieved by progressively exploring the local search area instead of directly using the global search area), (iii) an *effective weight* pruning technique in algorithm *Ref* for faster processing during Snell's law refinement (achieved by considering additional information on the weighted *TIN*), and (iv) a *novel* error guaranteed pruning technique in algorithm *Ref* for handling rare cases when we are unable to use Snell's law to refine a

rough path to a $(1+\epsilon)$-approximate weighted shortest path, and then an additional step is required for error guarantee, but the algorithm total query time will not increase a lot (achieved by re-using the calculated information in algorithm *Roug*).

4) Contributions We summarize our major contributions.

(i) We propose algorithm *Roug-Ref*, which is the first algorithm that efficiently answers shortest path queries on a weighted *TIN*, since the rough-refine concept and four additional techniques are absent in algorithms [20, 43, 45, 49, 69].

(ii) We provide theoretical analysis on the query time, memory usage and error bound of our algorithm.

(iii) Our experimental results show that our algorithm is up to 1,630 times faster than the best-known algorithm [43, 49] on benchmark real datasets with the same error ratio. For a *TIN* with 50k faces with $\epsilon = 0.1$, our algorithm runs in 73s $\approx$ 1.2 min and uses 43MB of memory, but the best-known $(1+\epsilon)$-approximate algorithm [43, 49] runs in 119,000s $\approx$ 1.5 days and uses 2.9GB of memory.

1.2.2 Oracle for Shortest Path Queries on Updated *TIN*s

1) P2P and AR2AR query Consider a set of POIs on a *TIN*. We study the shortest path query between a pair of objects X and Y. (i) In the *POI-to-POI* (*P2P*) *query*, both X and Y are POIs. (ii) In the *ARbitrary point-to-ARbitrary point* (*AR2AR*) *query*, i.e., POIs are not given as input, both X and Y are arbitrary points on the *TIN*. The first two rows in Table 1.1 illustrate this.

Table 1.1. P2P, AR2AR and A2A queries

Query	Source	Destination	3D surfaces
P2P	POI	POI	*TIN*/point cloud
AR2AR	arbitrary point[a]	arbitrary point[a]	*TIN*
A2A	any point[b]	any point[b]	point cloud

Remark: [a]on (the faces of) a *TIN* in continuous space, and [b]on (a set of points of) a point cloud in discrete space.

2) Challenges (i) All existing *TIN* on-the-fly shortest path query algorithms [25, 45, 46, 49, 54, 64, 71, 72, 75, 76, 79] are slow when many shortest path queries are involved. Our experimental results show that they run for more than one day. (ii) Existing studies [43, 71,

72, 82] can construct oracles on *static TIN*s, but no study can accommodate updated *TIN*s. When a *TIN* is updated, straightforward adaptations of the best-known oracles [71, 72, 43] for the P2P and AR2AR queries on a *TIN* must re-construct the oracles. Our experimental results show that their oracle construction time is 7 to 10 hours.

3) Our oracle We propose an efficiently updatable $(1+\epsilon)$-approximate shortest path oracle that answers the P2P shortest path query on an updated *TIN* called *Updatable Path Oracle*, i.e., *UP-Oracle*. *UP-Oracle* can be easily adapted to answering AR2AR queries on T_{aft} (we denote it as *UP-Oracle-AR2AR*).

(i) **Achieving a short oracle update time.** The ideas for achieving a short oracle update time of *UP-Oracle* follow from a novel property and the useful information on T_{bef}. (a) The property is the *non-updated TIN shortest path intact* property. It implies that given a path between two POIs u and v passing on T_{bef} (with distance d), if the distances from both u and v to the updated faces are large enough (i.e., both larger than $\frac{d}{2}$), then the path between u and v passing on T_{aft} remains the same and does not need to be updated. (b) The useful information is the stored pairwise P2P exact shortest paths passing on T_{bef} when *UP-Oracle* is constructed. The *exact* shortest distances are no larger than the *approximate* shortest distances. So given an exact (resp. approximate) shortest path with two POIs u and v on T_{bef}, based on the property, it is likely (resp. unlikely) that the distances from both u and v to the updated faces are both larger than half of the exact (resp. approximate) length of this path, and it reduces (resp. increases) the likelihood of updating this path passing on T_{aft}.

(ii) **Efficiently achieving a small output size.** Although we have the pairwise P2P exact shortest paths passing on T_{aft}, we aim to return fewer paths to reduce the output size (i.e., the space complexity of the output oracle). In the landslide and earthquake example, after disasters, since it is time-consuming to build evacuation paths in the damaged region, fewer paths imply that the total time to build evacuation paths is smaller, enabling the rescue teams to focus on saving lives. In the marsquake example, we cannot store the pairwise paths in rovers due to their limited memory size. That is, given a complete graph (with the POIs as vertices and with the exact shortest paths between POIs on T_{aft} as edges), we hope that *UP-Oracle* can efficiently generate a sub-graph of it with a small output size. We propose an algorithm called *Hierarchy Greedy Spanner*, i.e., *HGSpan*, to solve it.

4) Contributions We summarize our major contributions.

(i) We propose *UP-Oracle*, which is the first oracle that efficiently answers shortest path queries on an updated *TIN*. We also develop an efficient algorithm *HGSpan* for reducing the output size of *UP-Oracle*, and we adapt *UP-Oracle* for handling subsequent changes, adapt *UP-Oracle* to *UP-Oracle-AR2AR* for AR2AR queries, and adapt *UP-Oracle* to a multi-layer structure called *UP-Oracle Multi Layer*, i.e., *UP-Oracle-MuLa*.

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

(iii) *UP-Oracle* and *UP-Oracle-AR2AR* outperform the best-known oracle [72, 71] for the P2P query and the best-known oracle [43] for the AR2AR query on *TIN*s concerning the oracle update time, output size and shortest path query time. Our experimental results show that for the P2P query on a *TIN* with 0.5M faces and 250 POIs, when computing 100 shortest paths with different sources and destinations, these values for *UP-Oracle* are 400s $\approx$ 6.7 min, 22MB and 0.1s, but for the best-known oracle [72, 71] are 35,100s $\approx$ 9.8 hours, 250MB and 0.3s, respectively. (iii) For the AR2AR query on a *TIN* with 20k faces with the same query, these values for *UP-Oracle-AR2AR* are 480s $\approx$ 7 min, 3MB and 0.05s, but for the best-known oracle [43] are 7,100s $\approx$ 2 hours, 150MB and 5s, respectively.

1.2.3 Oracle for Proximity Queries on Point Clouds

1) P2P and A2A query Similar to *TIN*, we have similar queries on a point cloud. But, since a point cloud does not have a face, we change "arbitrary point on (the faces of) the *TIN*" to "any point on (a set of points of) the point cloud", to obtain *Any point-to-Any point* (*A2A*) *query* on the point cloud. "Arbitrary point on the *TIN*" comes from a continuous space (i.e., the faces of the *TIN*), and "any point on the point cloud" comes from a discrete space (i.e., a set of a certain number of points of the point cloud). The AR2AR query on the *TIN* is more general than the A2A query on the point cloud since an object may lie on the face of the *TIN*. The P2P query is the same on both *TIN* and point cloud. The first and third rows in Table 1.1 illustrate this. For P2P and A2A queries, we also have *kNN* and range queries among a query object X and target objects Y on a point cloud, where X and Y have the same definition mentioned above.

2) Challenges (i) There is no study answering shortest path query on a point cloud *on-the-fly* directly. Existing algorithms [59, 65, 86] convert a point cloud to a *TIN*, and then calculate the shortest path passing on this *TIN* using algorithms [25, 45, 46, 49, 54, 64, 71, 72, 75, 76]. Our experimental results show that they need several days for *kNN* or range queries. (ii) There is no study answering shortest path query on a point cloud using *oracles* directly. The only closely related works are oracles [71, 72, 43] on a *TIN*. We can adapt them to a point cloud by converting the point cloud to a *TIN*, and then constructing these oracles on this *TIN*. But, the best-known adapted *TIN* oracles [71, 72, 43] for the P2P and A2A queries on a point cloud have a large oracle construction time. This is because their *loose criterion for algorithm earlier termination* drawback. That is, although they provide a criterion to terminate their algorithms earlier, the criterion is not tight. Our experimental results show that their oracle construction time is half to one day.

3) Our oracle and proximity query algorithms We propose an efficient $(1+\epsilon)$-approximate shortest path oracle that answers the P2P shortest path query on a point cloud called *Rapid Construction path Oracle*, i.e., *RC-Oracle*. *RC-Oracle* can be easily adapted to answer the A2A shortest path query on the point cloud if POIs are not given as input (we denote it as *RC-Oracle-A2A*). Based on them, we develop efficient $(1+\epsilon)$-approximate proximity query algorithms. We introduce the key idea of the small oracle construction time of *RC-Oracle*.

(i) **Rapid point cloud on-the-fly shortest path query algorithm.** When constructing *RC-Oracle*, we propose algorithm *Fast on-the-Fly shortest path query*, i.e., *FastFly*, which is a Dijkstra-based algorithm [34] returning its calculated shortest path passing on a point cloud. It can significantly reduce the algorithm's shortest path query time, since computing the shortest path passing on a *TIN* is expensive.

(ii) **Rapid oracle construction.** When constructing *RC-Oracle*, we use algorithm *FastFly*, i.e., a *Single-Source All-Destination* (*SSAD*) algorithm [25, 73, 45, 79, 46], to calculate the shortest path passing on the point cloud from for each POI to other POIs *simultaneously*, and set *tight* earlier termination criterion for different POIs.

4) Contributions We summarize our major contributions.

(i) We propose *RC-Oracle*, which is the first oracle that efficiently answers shortest path queries on a point cloud. We also propose algorithm *FastFly* used for constructing

RC-Oracle, and develop efficient proximity query algorithms using *RC-Oracle*.

(ii) We provide theoretical analysis on the oracle construction time, oracle size (i.e., the space complexity of the oracle), shortest path query time and error bound of *RC-Oracle*. We also provide theoretical analysis on the query time and error bound of algorithm *Fast-Fly* and proximity query algorithms.

(iii) *RC-Oracle*, *RC-Oracle-A2A* and the proximity query algorithm perform much better than the best-known adapted *TIN* oracle [71, 72] for the P2P query and the best-known adapted *TIN* oracle [43] for the A2A query on a point cloud concerning the oracle construction time, oracle size and proximity (e.g., *kNN*) query time. Our experimental results show that for the P2P query on a point cloud with 2.5M points and 500 POIs, these values for *RC-Oracle* are 200s $\approx$ 3.2 min, 50MB and 12.5s, but for the best-known adapted *TIN* oracle [71, 72] are 78,000s $\approx$ 21.7 hours, 1.5GB and 150s, respectively. For the A2A query on a point cloud with 100k points and 5000 objects, these values for *RC-Oracle-A2A* are 100s $\approx$ 1.6 min, 150M and 0.25s, but for the best-known adapted *TIN* oracle [43] are 50,000s $\approx$ 13.9 hours, 21GB and 12.5s.

1.3 Organizations

The remainder of this thesis is organized as follows. Chapter 2 covers the related work, Chapter 3 studies shortest path queries on a weighted *TIN* using an on-the-fly algorithm, Chapter 4 studies shortest path queries on a *TIN* using an oracle, Chapter 5 studies proximity queries on a point cloud using an oracle, and Chapter 6 concludes this thesis with future directions.

CHAPTER 2

RELATED WORK

This chapter studies the related work to our three studies. Chapter 2.1 covers the related work of the on-the-fly algorithm for shortest path queries on weighted *TIN*s. Chapter 2.2 covers the related work of the oracle for shortest path queries on updated *TIN*s. Chapter 2.3 covers the related work of the oracle for proximity queries on point clouds.

2.1 On-the-fly Algorithm for Shortest Path Queries on Weighted *TIN*s

2.1.1 On-the-fly Algorithms on Weighted *TIN*s

Consider a weighted *TIN* T with N vertices. T contains a set of vertices V, edges E and faces F. Two types of algorithms can compute the shortest path passing on a weighted *TIN on-the-fly*.

1) Steiner point approach Algorithms [20, 43, 45, 49] use Dijkstra's algorithm on a weighted graph constructed by Steiner points and V to calculate an approximate weighted shortest path. Algorithm *Fixed Steiner Point* (*FixSP*) [43, 49] and algorithm *Fixed Steiner Point No Weight Adaption* (*FixSP-NoWei-Adp*) [45] calculates the result path with an error ratio $(1+\epsilon)$, and algorithm *Logarithmic Steiner Point* (*LogSP*) [20] calculates the result path with an error ratio much larger than $(1+\epsilon)$. (i) Algorithm *FixSP* places a uniform number of Steiner points (i.e., $O(N^2)$) on each edge. It runs in $O(N^3 \log N)$ time and is regarded as the best-known $(1+\epsilon)$-approximate algorithm for calculating the shortest path passing on the weighted *TIN*. (ii) Algorithm [45] also places a uniform number of Steiner points on each edge, but it was originally designed in the unweighted case. We adapt it to be algorithm *FixSP-NoWei-Adp* [45] in the weighted case by assigning each face a weight. Then, it runs in $O(N^3 \log N)$ time and it is equivalent to *FixSP*. (iii) In algorithm *LogSP*, given an error parameter $\epsilon' > 0$ that determines how far the Steiner points are placed

(different from our ϵ that controls the distance error bound), it places $O(\log \frac{c}{\epsilon'})$ Steiner points on each edge non-uniformly, i.e., places more Steiner points near a vertex, and has a distance error ratio $(1 + (2 + \frac{2W}{(1-2\epsilon')\cdot w})\epsilon')$, where $c \in [0.2, 1]$ is a constant that depends on T, W is the maximum weight of the face in F and w is the minimum weight of the face in F. We adapt it to be algorithm *LogSP-Adp* that runs in $O(N \log \frac{c}{\epsilon} \log(N \log \frac{c}{\epsilon}))$ time, such that given ϵ, we can place Steiner points using ϵ and have a distance error ratio $(1 + \epsilon)$, by finding the relationship between ϵ and ϵ', so these two algorithms can place the same number of Steiner points on each edge, and their distance error ratios are the same, i.e., $1 + (2 + \frac{2W}{(1-2\epsilon')\cdot w})\epsilon' = 1 + \epsilon$, and get $\epsilon' = \frac{1+\epsilon+\frac{W}{w}-\sqrt{(1+\epsilon+\frac{W}{w})^2-4\epsilon}}{4}$. Although we can change Dijkstra's algorithm in these algorithms to A* algorithm [41], it is not our focus since the latter is a heuristic algorithm without any error guarantee. This also applies to the rest of Dijkstra's algorithm in this thesis.

Drawbacks of algorithm *FixSP* and *FixSP-NoWei-Adp*: They are very slow due to two reasons. (i) *Absence of Snell's law*: They do not utilize any geometric information *between two adjacent faces in* F *that contain the same edge* on T, i.e., Snell's law. So, they place many Steiner points on edges in E. But, we utilize Snell's law to avoid this. (ii) *Uniform number of Steiner points on each edge*: They do not utilize any geometric information of *each face in* F on T, such as face weight, internal angle and edge length, and always place a uniform number of Steiner points (i.e., $O(N^2)$) on each edge (the distance between a pair of adjacent Steiner points on the same edge is the same) to bound the error. But, we utilize this information and place only $O(\log c')$ Steiner points on each edge (the distance between a pair of adjacent Steiner points on the same edge is different), where $c' \in [2, 5]$ is a constant that depends on T and ϵ. Our experimental results show that for a *TIN* with 50k faces and $\epsilon = 0.1$, *Roug-Ref* just places 10 Steiner points on each edge to find a rough path in 71s $\approx$ 1.2 min, and finds a refined path in 2s, but both *FixSP* and *FixSP-NoWei-Adp* place more than 600 Steiner points on each edge to find the result path in 119,000s $\approx$ 1.5 days.

Drawback of algorithm *LogSP* and *LogSP-Adp*: (i) *LogSP* has the *larger distance error ratio* drawback, since its distance error ratio is always larger than that of other algorithms, making it difficult to compare with other algorithms. (ii) Both of them also have the *absence of Snell's law* drawback. In the same experimental setting of the previous paragraph, *Roug-*

Ref runs in 73s $\approx$ 1.2 min, but *LogSP-Adp* runs in 220s $\approx$ 3.7 min.

2) Edge sequence approach Algorithm *Edge Sequence* (*EdgSeq*) [69] uses *FixSP* with only slight variations (i.e., places a uniform and constant number of Steiner points, e.g., 3 and not $O(N^2)$, on each edge) to calculate a path without error guarantee, and then uses Snell's law on the edge sequence passed by this path based on T to calculate a shorter path. We adapt it to be algorithm *EdgSeq-Adp*, that uses *FixSP* to calculate a $(1+\epsilon)$-approximate shortest path, and then uses Snell's law on the edge sequence of this path to compute a shorter path that runs in $O(N^3 \log N + N^4 \log(\frac{N^2 IWL}{w\epsilon}))$ time, where L is the longest length of edges in T, and I is the minimum integer which is no less than the coordinate value of any vertex in V.

Drawbacks of algorithm *EdgSeq* and *EdgSeq-Adp*: (i) *EdgSeq* does not have an error guarantee of the returned path's distance with a given time limit. (ii) *EdgSeq-Adp* is still very slow due to two reasons. (a) *Absence of pruning technique for the edge sequence finding*: It uses Snell's law after *FixSP*, so its query time is an additive result of the query time of *FixSP* and usage of Snell's law. But, we use a pruning technique to further prune out more than half of the Steiner points. (b) *Absence of pruning technique when using Snell's law*: It solely uses binary search in Snell's law on the edge sequence. But, we use *effective weight* information of T for pruning. In the same experimental setting of the previous three paragraphs, *EdgSeq-Adp* runs in 131,000s $\approx$ 1.7 days, but *Roug-Ref* runs in 73s $\approx$ 1.2 min. If we substitute *FixSP* to *LogSP-Adp* in *EdgSeq-Adp*, the new algorithm has the same drawback as of *EdgSeq-Adp*, and its query time is always larger than that of *LogSP-Adp*, so there is no need to consider this adaptation.

2.1.2 On-the-fly Algorithms on Unweighted *TIN*s

Algorithm [73] is the best-known exact on-the-fly algorithm for the unweighted shortest path calculation, but there is no known algorithm that can adapt it to the weighted case, and it is not our focus.

2.2 Oracle for Shortest Path Queries on Updated *TIN*s

2.2.1 On-the-fly Algorithms on *TIN*s

Consider a *TIN* T with N vertices. Two types of algorithms can compute the shortest path passing on a *TIN on-the-fly*.

1) Exact algorithms The running times of the four exact algorithms [25, 44, 73, 75] are $O(\mathsf{N}\log^2\mathsf{N})$, $O(\mathsf{N}^2)$, $O(\mathsf{N}^2)$ and $O(\mathsf{N}^2\log\mathsf{N})$, respectively. They are *SSAD* algorithms [25, 43, 44, 71, 73, 75], i.e., given a source, they can calculate the shortest path from it to all other vertices *simultaneously*. According to existing studies [43, 71, 64, 76], algorithm [44] that runs in $O(\mathsf{N}\log^2\mathsf{N})$ time is hard to implement (no implementation exists). So, the implementable algorithm *WAVefront on-the-Fly Algorithm* (*WAV-Fly-Algo*) [25, 73] that runs in $O(\mathsf{N}^2)$ time is recognized as the practical algorithm of choice. It uses a continuous version of Dijkstra's algorithm and needs to consider continuous points on the edges of the *TIN* by unfolding the 3D *TIN* into a 2D plane (which is not needed in the plain Dijkstra's algorithm) during wavefront propagation, so its running time cannot be reduced to $O(\mathsf{N}\log\mathsf{N})$ in the plain Dijkstra's algorithm. *WAV-Fly-Algo* comes in two variants with the same time complexity: an initial version [25] and an extended version [73] with better empirical running time.

2) Approximate algorithms Approximate algorithms [45, 46, 51, 79] aim to reduce the running time. The best-known approximate algorithm *Efficient Steiner Point on-the-Fly Algorithm* (*ESP-Fly-Algo*) [45, 79] on *TIN*s places Steiner points on edges in E, and then constructs a graph using these points and V to calculate a $(1+\epsilon)$-approximate shortest path passing on a *TIN* using Dijkstra's algorithm. It runs in $O(\frac{\mathsf{l}_{max}\mathsf{N}}{\epsilon\mathsf{l}_{min}\sqrt{1-\cos\theta}}\log(\frac{\mathsf{l}_{max}\mathsf{N}}{\epsilon\mathsf{l}_{min}\sqrt{1-\cos\theta}}))$ time, where l_{max} (resp. l_{min}) is the longest (resp. shortest) length of edges in T, and θ is the minimum angle of the face in F. Algorithm [45] runs on an unweighted *TIN* and algorithm [79] runs on a weighted *TIN* where each *TIN* face has a weight. They are the same if we set each face's weight in algorithm [79] to be 1, so we regard them as one algorithm.

Drawback: All these algorithms are inefficient for computing multiple shortest path queries. Our experimental results show that *WAV-Fly-Algo* and *ESP-Fly-Algo* need 11,600s $\approx$ 3.2 hours and 8,600s $\approx$ 2.4 hours to compute 100 paths with different sources and desti-

nations on a *TIN* with 0.5M faces, respectively.

2.2.2 Oracles on *TIN*s

The *Space Efficient Oracle* (*SE-Oracle*) [71, 72] (resp. the *Efficiently ARbitrary pints-to-arbitrary points Oracle* (*EAR-Oracle*) [43]) is regarded as the best-known oracle for answering *approximate* P2P (resp. AR2AR) queries on *TIN*s. Further, an existing oracle [82], originally designed for answering *approximate* P2P queries on point clouds, can be adapted to the *Rapid-Constuction TIN Oracle* (*RC-TIN-Oracle*) for answering *approximate* P2P queries on *TIN*s [82]. Yet, no existing oracle can accommodate updated *TIN*s, where the oracle needs to be updated efficiently. A straightforward adaptation is to re-construct them from scratch when the *TIN* is updated. A smart adaptation is to leverage Property 1 in *UP-Oracle*, such that we only re-calculate the paths passing on T_{aft} that require updating to reduce the oracle update time. We denote the adapted oracles as *SE-UP-Oracle*, *EAR-UP-Oracle* and *RC-TIN-UP-Oracle*.

1) *SE-Oracle* and *SE-UP-Oracle* These use a *compressed partition tree*, algorithm *SSAD* and *well-separated node pair sets* to index the $(1+\epsilon)$-approximate pairwise P2P shortest paths. (i) *SE-Oracle*'s oracle construction time, output size and shortest path query time is $O(\frac{nN^2}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ and $O(h^2)$, respectively, where n is the number of POIs on the *TIN*, h is the compressed partition tree's height and $\beta \in [1.5, 2]$ is the largest capacity dimension [71, 72]. (ii) *SE-UP-Oracle*'s oracle update time is $O(\mu_1 N^2 + n\log^2 n)$, where μ_1 is a data-dependent variable, and $\mu_1 \in [5, 20]$ in our experiments.

2) *EAR-Oracle* and *EAR-UP-Oracle* These use the same idea as *SE-Oracle* and *SE-UP-Oracle*, i.e., well-separated node pair sets. Their differences are that they adapt *SE-Oracle* and *SE-UP-Oracle* from the P2P query to the AR2AR query by using Steiner points on the *TIN* faces and using *highway nodes* (i.e., not POIs in *SE-Oracle* and *SE-UP-Oracle*) for well-separated node pair sets construction. (i) *EAR-Oracle*'s oracle construction time, output size and shortest path query time is $O(\lambda\xi(mN)^2 + \frac{N^3}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ and $O(\lambda\xi\log(\lambda\xi))$, respectively, where λ is the number of highway nodes in one square, ξ is the number of boxes under square root, and m is the number of Steiner points on each face. (ii) *EAR-UP-Oracle*'s oracle update time is $O(\mu_2 N^2 + n\log^2 n)$, where μ_2 is a

data-dependent variable, and $\mu_2 \in [12, 45]$ in our experiments.

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

Drawbacks: (1) *SE-Oracle*, *EAR-Oracle* and *RC-TIN-Oracle* only support the *static TIN* and do not address how to update the oracle on an *updated TIN*, since they do not utilize Property 1. (2) Although *SE-UP-Oracle*, *EAR-UP-Oracle* and *RC-TIN-UP-Oracle* utilize Property 1, they do not *fully utilize it*. Since they only store the pairwise P2P *approximate* shortest paths passing on T_{bef}, the oracle update time remains large. (3) In the P2P query, the oracle update time for *SE-Oracle*, *SE-UP-Oracle*, *RC-TIN-Oracle*, *RC-TIN-UP-Oracle* and *UP-Oracle* are 35,100s $\approx$ 9.8 hours, 8,400s $\approx$ 2.4 hours, 28,100s $\approx$ 7.5 hours, 10,100s $\approx$ 2.9 hours and 400s $\approx$ 6.7 min on a *TIN* dataset with 0.5M faces and 250 POIs, respectively. In the AR2AR query, the oracle update time for *EAR-Oracle*, *EAR-UP-Oracle* and *UP-Oracle-AR2AR* are 7,100s $\approx$ 2 hours, 4,300s $\approx$ 1.2 hours and 480s $\approx$ 7 min on a *TIN* with 20k faces, respectively.

2.2.3 Oracles on Road Networks

There are some existing studies [19, 39] focusing on using oracles on road networks with landmark-based ideas. However, they are heuristic approaches without an error guarantee, so they are not our main focus. There are also some existing studies [42, 70] focusing on using oracles on updated road networks. However, they do not have the theoretical worst case oracle update time. Instead, they only provide a theoretical oracle update time with high probability, so they are also not our main focus.

2.2.4 Sub-graph Generation Algorithms

Given a complete graph and ϵ, algorithm *Greedy Spanner* (*GSpan*) [21] that runs in $O(n^3 \log n)$ time is the best-known $(1+\epsilon)$-sub-graph generation algorithm. A $(1+\epsilon)$-

sub-graph has the property that the distance between any pair of its vertices is at most $(1+\epsilon)$ times the exact distance. Given a $(1+\epsilon')$-sub-graph and ϵ, where $\epsilon \geqslant \epsilon' > 0$, algorithm *Sparse Greedy Spanner* (*SGSpan*) [29] uses a simpler structure to approximate the internal graph for faster processing and generates a $(1+\epsilon)$-sub-graph.

Drawbacks: (1) Algorithm *GSpan* is very slow since it does not use any simpler structure to approximate the internal graph when performing Dijkstra's algorithm [34] on the sub-graph. (2) Although algorithm *SGSpan* uses a simpler structure for approximation, it cannot take a complete graph, i.e., $\epsilon' = 0$, as input. If we force $\epsilon' = 0$, algorithm *SGSpan* degenerates to algorithm *GSpan*. (3) On a *TIN* with 0.5M faces and 500 POIs, algorithm *HGSpan* uses 24s to generate a $(1+\epsilon)$-sub-graph of size 44MB, but algorithm *GSpan* uses 101s to generate a $(1+\epsilon)$-sub-graph of size 41MB.

2.3 Oracle for Proximity Queries on Point Clouds

2.3.1 On-the-fly Algorithms on Point Clouds

Consider a point cloud C with N points. There is no study answering shortest path query on a point cloud *on-the-fly* directly. Existing algorithms [59, 65, 86] convert a point cloud to a *TIN* T in $\mathsf{O}(\mathsf{N})$ time, and then calculate the shortest path passing on this *TIN* [25, 45, 46, 49, 54, 64, 71, 72, 75, 76], which are very slow. There are two types of *TIN* shortest path query algorithms, i.e., (1) the *shortest surface path* [25, 45, 49, 54, 75] and (2) the *shortest network path* [46] query algorithms.

1) *TIN* shortest surface path query algorithms There are two more sub-types. (i) *Exact algorithms*: Algorithm [54] (resp. algorithm [75]) uses continuous Dijkstra (resp. checking window) algorithm to calculate the result in $\mathsf{O}(\mathsf{N}^2 \log \mathsf{N})$ (resp. $\mathsf{O}(\mathsf{N}^2 \log \mathsf{N})$) time, and the best-known *TIN* exact shortest surface path query algorithm *Chen and Han*, i.e., algorithm *CH* [25] (as recognized by [45, 46, 64, 76]) unfolds the 3D *TIN* into a 2D *TIN*, and then connects the source and destination using a line segment on this 2D *TIN* to calculate the result in $\mathsf{O}(\mathsf{N}^2)$ time. (ii) *Approximate algorithms*: All algorithms [45, 49] place discrete points (i.e., Steiner points) on edges of a *TIN*, and then construct a graph using these Steiner points together with the original vertices to calculate the result. The best-known

TIN $(1+\epsilon)$-approximate shortest surface path query algorithm, i.e., algorithm *Kaul* [45] (as recognized by [71, 72]) runs in $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ time.

2) *TIN* shortest network path query algorithm Since the shortest network path passing on *TIN* is the shortest surface path restricted on passing along edges of the *TIN* without traversing the faces of the *TIN*, it is an approximate path. The best-known *TIN* approximate shortest network path query algorithm *Dijkstra*, i.e., algorithm *Dijk* [46] runs in $O(N \log N)$ time.

Adaptations: Given a point cloud, we adapt algorithms *CH* [25], *Kaul* [45] and *Dijk* [46] to be algorithms *CH-Adapt*, *Kaul-Adapt* and *Dijk-Adapt*, by first converting the point cloud to a *TIN*, and then compute the corresponding shortest path passing on the *TIN*. Since we regard the shortest path passing on a point cloud as the exact shortest path, they return the approximate shortest path passing on a point cloud.

Drawbacks of the on-the-fly algorithms: Although we can pre-process the point cloud and store the generated *TIN* as a data structure in the memory, all these algorithms are still time-consuming. Since the time for calculating the shortest path passing on a *TIN* is 10^2 to 10^5 times larger than the time for converting a point cloud to a *TIN*. Thus, the latter time can be neglected. Our experimental results show algorithm *CH-Adapt*, *Kaul-Adapt* and *Dijk-Adapt* first needs to convert a point cloud with 2.5M points to a *TIN* in 21s, and then perform the *kNN* query for all 500 objects on this *TIN* in 290,000s $\approx$ 3.4 days, 161,000s $\approx$ 1.9 days and 3,990s $\approx$ 1.1 hours, respectively. Performing the same query for algorithm *FastFly* on the point cloud needs 4,000s $\approx$ 1.1 hours.

2.3.2 Oracles for the Shortest Path Query on Point Clouds

There is no study answering shortest path query on a point cloud using *oracles* directly. The only closely related works are oracles on a *TIN*. Specifically, *Space Efficient Oracle* (*SE-Oracle*) [71, 72] answers P2P shortest path queries on a *TIN* using an oracle. *Efficiently ARbitrary points-to-arbitrary points Oracle* (*EAR-Oracle*) [43] answers AR2AR shortest path queries on a *TIN* using an oracle. They store *TIN* shortest surface paths. By performing a linear scan using the shortest path query results, they can answer other proximity queries.

Adaptations: Given a point cloud, we adapt *SE-Oracle* [71, 72] and *EAR-Oracle* [43] to

be *SE-Oracle-Adapt* and *EAR-Oracle-Adapt* by first converting the point cloud to a *TIN*, and then construct these oracles on the *TIN*.

1) SE-Oracle-Adapt It uses a *compressed partition tree* [71, 72] and *well-separated node pair sets* [24] to index the $(1+\epsilon)$-approximate pairwise P2P shortest surface paths passing on the converted *TIN*. Its oracle construction time, oracle size and shortest path query time are $O(\frac{nN^2}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ and $O(h^2)$, respectively, where n is the number of POIs on the point cloud. It is regarded as the best-known adapted *TIN* oracle for the P2P query on a point cloud.

Drawbacks of *SE-Oracle-Adapt*: Its oracle construction time is large due to the *loose criterion for algorithm earlier termination*. For POIs in the same level of the compressed partition tree, they have the *same* earlier termination criteria, which are not tight for some of such POIs. Even after the *SSAD* algorithm has visited most POIs, its earlier termination criterion is still not reached. But, in *RC-Oracle*, we have *tight* earlier termination criteria for each different POI, to minimize the running time of the *SSAD* algorithm. In the P2P query on a point cloud, for a point cloud with 2.5M points and 500 POIs, the oracle construction time of *SE-Oracle-Adapt* is 78,000s $\approx$ 21.7 hours, while *RC-Oracle* just needs 200s $\approx$ 3.2 min.

2) EAR-Oracle-Adapt It also uses well-separated node pair sets, which is similar to *SE-Oracle-Adapt*. But, *EAR-Oracle-Adapt* adapts *SE-Oracle-Adapt* from the P2P query on a point cloud to the A2A query on a point cloud by using Steiner points on the faces of the converted *TIN* and *highway nodes* as POIs in well-separated node pair sets construction. Its oracle construction time, oracle size and shortest path query time are $O(\lambda\xi(mN)^2 + \frac{N^3}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$, $O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ and $O(\lambda\xi\log(\lambda\xi))$, respectively. It is regarded as the best-known adapted *TIN* oracle for the A2A query on a point cloud.

Drawbacks of *EAR-Oracle-Adapt*: It also has the *loose criterion for algorithm earlier termination* drawback. In the A2A query on a point cloud, for a point cloud with 100k points, the oracle construction time of *EAR-Oracle-Adapt* is 50,0000s $\approx$ 13.9 hours, while *RC-Oracle-A2A* just needs 100s $\approx$ 1.6 min.

2.3.3 Oracles for Other Proximity Queries on Point Clouds

There is no existing study focusing on using oracle to answer proximity queries on a point cloud. But, studies [32, 33, 64] build an oracle to answer proximity queries on a *TIN*. Specifically, studies [32, 33] use a multi-resolution *TIN* model (resp. *SUrface Oracle* (*SU-Oracle*) [64] uses a surface index) to answer the *kNN* query on a *TIN* in $O(N^2)$ (resp. $O(N \log^2 N)$) time. We adapt *SU-Oracle* to be *SU-Oracle-Adapt* in a similar way of *SE-Oracle-Adapt*. Although *SU-Oracle-Adapt* is regarded as the best-known adapted *TIN* oracle to directly answer the *kNN* query, studies [71, 72] show the *kNN* query time of *SU-Oracle-Adapt* is up to 5 times larger than that of using *SE-Oracle-Adapt* with a linear scan of the shortest path query result.

CHAPTER 3

ON-THE-FLY ALGORITHM FOR SHORTEST PATH QUERIES ON WEIGHTED TINS

This chapter studies shortest path queries on weighted *TIN*s using an on-the-fly algorithm. Chapter 3.1 provides the preliminary. Chapter 3.2 presents the methodology. Chapter 3.3 presents the experimental results.

3.1 Preliminary

3.1.1 Notation and Definitions

1) *TIN*s and paths Consider a weighted *TIN* T with N vertices. T contains a set of vertices V, edges E and faces F. Let x_v, y_v and z_v be the three coordinate values of each vertex $v \in V$. Let L be the longest length of edges in T, and I be the minimum integer which is no less than the coordinate value of any vertex in V. If two faces contain the same edge, they are said to be *adjacent*. Each face $f_i \in F$ has a weight $w_i > 0$, and an edge's weight can be the smaller or larger weight of the two faces containing the edge, depending on the problem setting. For example, if we want (resp. do not want) the path passing on the edge, we set the edge's weight to be the smaller (resp. larger) weight of the two faces containing the edge. The maximum weight of the face in F is denoted by W, and the minimum weight of the face in F is denoted by w. The minimum height (of the 2D triangle with a base) of a face in F is denoted by h. Given a face f_i, and two points p and q on f_i, let $\overline{pq}$ be a line segment between them, $d(p, q)$ be the Euclidean distance between them, and $D(p, q) = w_i \cdot d(p, q)$ be the *weighted (surface) distance* between them. Given s and t in V, let $\Pi^*(s, t) = \langle s, r_1, \ldots, r_l, t\rangle$ be the *optimal weighted shortest path* passing on T (with $l \geqslant 0$). Here, for each $i \in \{1, \ldots, l\}$, r_i is a point on an edge in E, is named as an *optimal intersection point* in $\Pi^*(s, t)$. The purple path in Figure 1.2 (b) shows $\Pi^*(s, t) = \langle s, r_1, r_2, t\rangle$ passing on T. We define $|\cdot|$ to a path's weighted distance, e.g., $|\Pi^*(s, t)|$ is $\Pi^*(s, t)$'s weighted distance. Let $\Pi(s, t)$ be the path result of algorithm *Roug-Ref*. If point s (or t) is not in V, we

can regard it as a new vertex and then add three new triangles each involving this point and three vertices of the face containing this point.

2) Snell's law Let $S = \langle (v_1, v_1'), \ldots, (v_l, v_l') \rangle = \langle e_1, \ldots, e_l \rangle$ be a sequence of l edges that $\Pi^*(s,t)$ connects from s to t in order based on T, where S is said to be *passed by* $\Pi^*(s,t)$. Let $F(S) = \langle f_1, f_2, \ldots, f_l, f_{l+1} \rangle$ be an adjacent face sequence corresponds to S, such that for every f_i with $i \in \{2, \ldots l\}$, f_i is the face that contains e_i and e_{i+1} in S, while f_1 is the face that adjacent to f_2 at e_1 and f_{l+1} is the face that adjacent to f_l at e_l. We define α_i and β_i to be the incidence and refraction angles of $\Pi^*(s,t)$ on e_i with $i \in \{1, \ldots l\}$, respectively. In Figure 1.2 (b), $\Pi^*(s,t)$ satisfies Snell's law, $w_i \cdot \sin \alpha_i = w_{i+1} \cdot \sin \beta_i$, with $i \in \{1, \ldots, l\}$. Table 3.1 shows a notation table.

Table 3.1. Frequent used notations in the study of shortest path queries on weighted *TIN*s

Notation	**Meaning**
T	The weighted *TIN*
V, E, F	The set of vertices, edges and faces of T
N	The size of V
L	The longest length of edges in T
W	The maximum weight of the face in F
w	The minimum weight of the face in F
ϵ	The error parameter
$\Pi^*(s,t)$	The optimal weighted shortest path
$\Pi(s,t)$	The final calculated weighted shortest path
$\lvert\Pi(s,t)\rvert$	$\Pi(s,t)$'s weighted distance
S	The edge sequence that $\Pi^*(s,t)$ connects from s to t in order based on T
l	The size of S

3.1.2 Problem

Given T, s and t, the problem is to efficiently calculate a weighted shortest path passing on T with the best performance concerning the query time and memory usage such that $|\Pi(s,t)| \leqslant (1+\epsilon)|\Pi^*(s,t)|$.

3.2 Methodology

3.2.1 Overview of Algorithm *Roug-Ref*

We first illustrate algorithm *Roug-Ref* with an example. In Figure 3.1 and Figures 3.2 (a) to (d), for algorithm *Roug*, given an weighted *TIN* T, a source s, a destination t, an error parameter ϵ and a user parameter k (a small positive constant), we efficiently find a $(1+\eta\epsilon)$-approximate *rough path* between s and t using Steiner points, where $\eta > 1$ is a constant and is calculated based on T, ϵ and k (note that $\eta \in (1, 2]$ on average in our experiments). In Figure 3.1 and Figures 3.2 (e) to (i), for algorithm *Ref*, given the rough path, we efficiently *refine* it to be a $(1+\epsilon)$-approximate weighted shortest path using Snell's law. Next, we introduce four concepts and overviews of algorithms *Roug* and *Ref*.

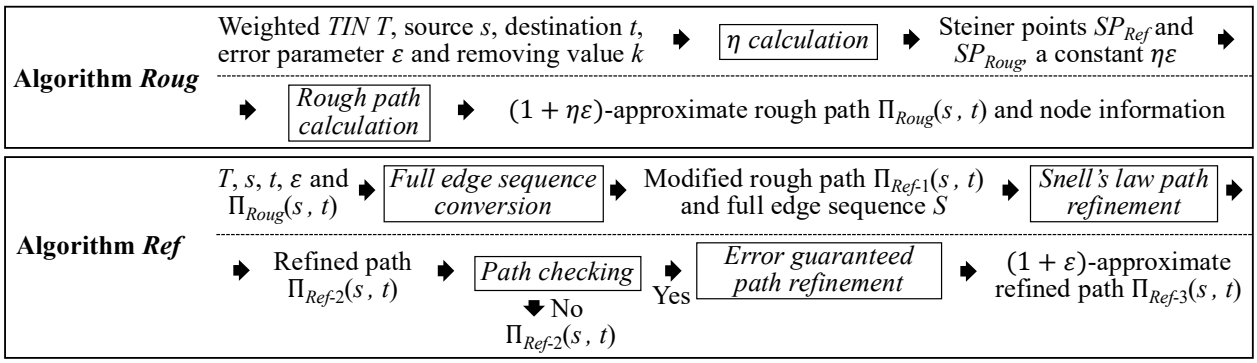
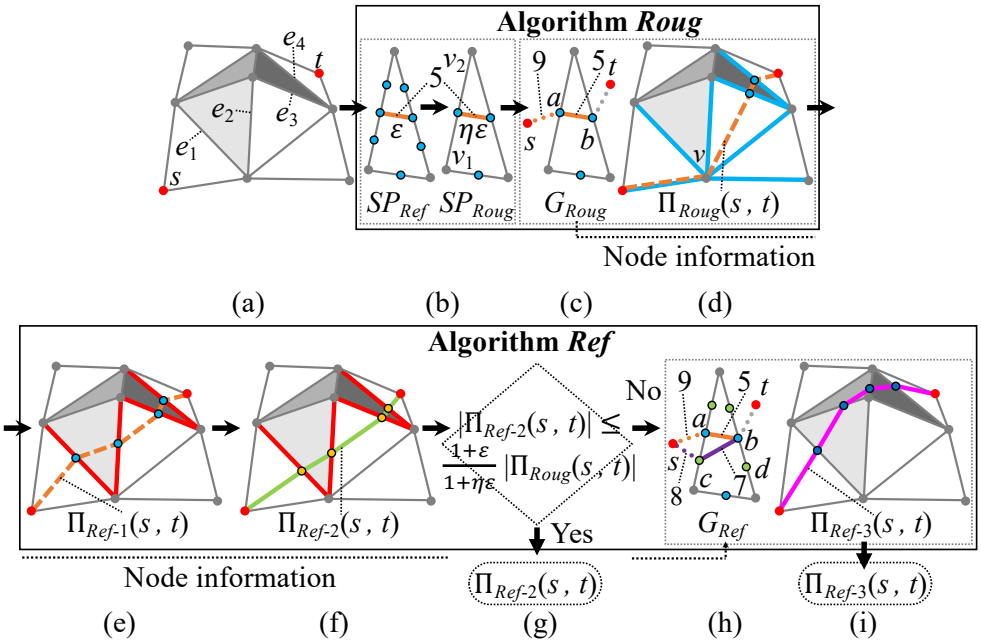


Figure 3.1. Algorithm *Roug-Ref* framework overview description

1) Concepts We give four concepts:

(i) **The weighted graph**: It is used in Dijkstra's algorithm. Let G_A be a weighted graph used in algorithm *Roug* or *Ref*, $\mathsf{G}_\mathsf{A}.\mathsf{V}$ and $\mathsf{G}_\mathsf{A}.\mathsf{E}$ be the sets of nodes and weighted edges of G_A, where A is a placeholder that can be *Roug* or *Ref*. To build G_A, we define a set of Steiner points on each edge of E as SP_A. Let $\mathsf{G}_\mathsf{A}.\mathsf{V} = SP_\mathsf{A} \bigcup \mathsf{V}$. For each node p and q in $\mathsf{G}_\mathsf{A}.\mathsf{V}$ that lie on the same face of T, if they are adjacent nodes on the same edge or lie on different edges, we connect them with a weighted edge $\overline{\mathsf{pq}} \in \mathsf{G}_\mathsf{A}.\mathsf{E}$, with the weighted (surface) distance $w_{\mathsf{pq}} \cdot \mathsf{d}(\mathsf{p}, \mathsf{q})$, where w_{pq} means the weight associated with the face or the edge that p and q lie on. In Figure 3.2 (b), the blue nodes are SP_{Ref} and SP_{Roug}, the blue and gray nodes are $\mathsf{G}_{Ref}.\mathsf{V}$ and $\mathsf{G}_{Roug}.\mathsf{V}$, the orange lines are the weighted edge with



Remark: SP_{Roug} and SP_{Ref} are the set of Steiner points on edges in E used in algorithm *Roug* and *Ref*, $\Pi_{Roug}(s,t)$ is the rough path calculated using algorithm *Roug*, $\Pi_{Ref\text{-}1}(s,t)$ is the modified rough path calculated using the full edge sequence conversion step of algorithm *Ref*, $\Pi_{Ref\text{-}2}(s,t)$ is the refined path calculated using the Snell's law path refinement step of algorithm *Ref*, and $\Pi_{Ref\text{-}3}(s,t)$ is the refined path calculated using the error guaranteed path refinement step of algorithm *Ref*.

Figure 3.2. Algorithm *Roug-Ref* framework overview

weighted distance 5 in $G_{Roug}.E$ and $G_{Ref}.E$.

(ii) **The removing value**: It is a small positive constant (denoted by k) used for calculating SP_{Roug}. In Figure 3.2 (b), we have a set of Steiner points SP_{Ref}. When moving from v_1 to v_2 (including v_1 and v_2), if we encounter a Steiner point, we iteratively keep one and remove the next $k = 1$ point(s). That is, we keep v_1, remove the next $k = 1$ point(s), keep one point, remove the next $k = 1$ point(s), and keep v_2. We repeat it for all edges in E to obtain a set of remaining Steiner points SP_{Roug}.

(iii) **The node information**: It is calculated in algorithm *Roug*, and is can reduce the query time in algorithm *Ref*. In Dijkstra's algorithm, given a source node s, a set of nodes $G_A.V$ where A can be *Roug* or *Ref*, for each $u \in G_A.V$, we let $dist_A(u)$ be the weighted shortest distance from s to u, let $prev_A(u)$ be the previous node of u along the weighted shortest path from s to u. Give s, after running algorithm *Roug*, *node information* stores $dist_{Roug}(u)$ and $prev_{Roug}(u)$ (based on s) for each $u \in G_{Roug}.V$. In Figure 3.2 (c), suppose

that this face's weight is 1. After running algorithm *Roug*, the weighted shortest distance from s to a is 9, the Euclidean distance of the orange edge is 5, the weighted shortest path from s to b is $s \to a \to b$, so we have $dist_{Roug}(b) = 9 + 1 \times 5 = 14$ and $prev_{Roug}(b) = a$ as node information of b.

(iv) **The full or non-full edge sequence**: Given an edge sequence S, if each edge length in S is larger than 0 (resp. there exists at least one edge in S whose length is 0), then S is a *full* (resp. *non-full*) *edge sequence* or S is *full* (resp. *non-full*). In Figure 3.3 (a), given the path in the purple dashed line and orange line between s and t, the edge sequence $S_1 = \langle (a,b), (b, c = \phi_2), (c,d), (c, e = \phi_3), (e,f), (e,g), (g, h = \phi_4), (h,i), (i,j) \rangle$ and $S_2 = \langle (a,b), (c = \phi_2, c), (e = \phi_3, e), (h = \phi_4, h), (i,j) \rangle$ passed by the two paths are full and non-full, since each edge length in S_1 is larger than 0, while the edge length at $c = \phi_2$, $e = \phi_3$ and $h = \phi_4$ in S_2 are 0. Figure 3.3 (b) has a similar case. As we will discuss later, a non-full edge sequence increases algorithm *Ref*'s query time.

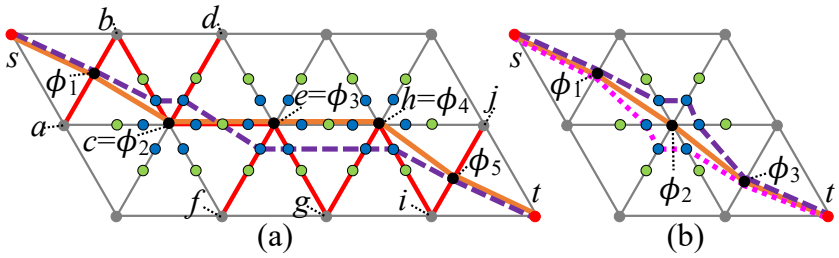


Figure 3.3. Full edge sequence conversion

2) Overview of algorithm *Roug* In Figure 3.1, given T, s, t, ϵ and k, we find a $(1+\eta\epsilon)$-approximate rough path between s and t, and calculate the node information for s in two steps:

(i) η **calculation**: In Figure 3.2 (a), given T, ϵ and k, we first use ϵ and an efficient Steiner point placement scheme to get SP_{Ref}, use k to remove some Steiner points and get SP_{Roug} in Figure 3.2 (b), and then use SP_{Roug} to calculate $\eta\epsilon$ (using constrained programming and the reverse technique of calculating SP_{Ref} with ϵ). We need this step to prune some Steiner points, which can reduce the query time of the algorithm.

(ii) **Rough path calculation**: In Figure 3.2 (c), given T, s, t and SP_{Roug}, we use Dijkstra's algorithm on G_{Roug} constructed by SP_{Roug} and V (we must include V in SP_{Roug} for error guarantee) to calculate a $(1+\eta\epsilon)$-approximate rough path between s and t in Figure 3.2 (d)

(denoted by $\Pi_{Roug}(s,t)$, i.e., the orange dashed line), and store $dist_{Roug}(u)$ and $prev_{Roug}(u)$ (based on s) for each $u \in G_{Roug}.V$ as node information. We need this step to reduce the search area when applying Snell's law (since directly using Snell's law on T is slow), which can reduce the query time of the algorithm.

3) Overview of algorithm *Ref* In Figure 3.1, given T, s, t, ϵ, $\Pi_{Roug}(s,t)$ and the node information, we refine $\Pi_{Roug}(s,t)$ and calculate a $(1+\epsilon)$-approximate weighted shortest path in four steps:

(i) **Full edge sequence conversion**: In Figure 3.2 (d), given $\Pi_{Roug}(s,t)$ whose corresponding edge sequence S' is non-full (the edge length at v is 0), we progressively convert it to a modified rough path in Figure 3.2 (e) (denoted by $\Pi_{Ref\text{-}1}(s,t)$, i.e., the orange dashed line) whose corresponding edge sequence $S = \langle e_1, e_2, e_3, e_4 \rangle$ (edges in red) is full. We need this step to further reduce the search area when applying Snell's law, which can reduce the query time of the algorithm.

(ii) **Snell's law path refinement**: In Figure 3.2 (f), given T, s, t, ϵ and S, we use Snell's law and S to efficiently get a refined path (denoted by $\Pi_{Ref\text{-}2}(s,t)$, i.e., the green line) on S. We need this step to avoid placing many Steiner points, which can reduce the query time of the algorithm.

(iii) **Path checking**: In Figure 3.2 (g), given $\Pi_{Roug}(s,t)$, $\Pi_{Ref\text{-}2}(s,t)$, ϵ and η, if $|\Pi_{Ref\text{-}2}(s,t)| \leqslant \frac{(1+\epsilon)}{(1+\eta\epsilon)}|\Pi_{Roug}(s,t)|$, since we guarantee $|\Pi_{Roug}(s,t)| \leqslant (1+\eta\epsilon)|\Pi^*(s,t)|$ due to the error bound of Dijkstra's algorithm, we have $|\Pi_{Ref\text{-}2}(s,t)| \leqslant (1+\epsilon)|\Pi^*(s,t)|$, and we return $\Pi_{Ref\text{-}2}(s,t)$. Otherwise, we need the next step for error guarantee. We need this step to terminate the algorithm earlier, which can reduce the query time of the algorithm.

(iv) **Error guaranteed path refinement**: In Figures 3.2 (h), given T, s, t, SP_{Ref} and the node information (based on s), we use Dijkstra's algorithm on G_{Ref} constructed by SP_{Ref} and V to efficiently calculate a $(1+\epsilon)$-approximate weighted shortest path between s and t in Figure 3.2 (i) (denoted by $\Pi_{Ref\text{-}3}(s,t)$, i.e., the purple line). We can guarantee $|\Pi_{Ref\text{-}3}(s,t)| \leqslant (1+\epsilon)|\Pi^*(s,t)|$ due to the error bound of Dijkstra's algorithm. We need this step to ensure error guarantee and hope that the query time of the algorithm will not increase a lot (by using the node information).

3.2.2 Key Ideas of Algorithm *Rough-Refine*

Algorithm *Roug-Ref* is efficient due to the rough-refine concept, which involves the following three techniques.

1) Novel Steiner point pruning step (in the η calculation step) In Figure 3.2 (b), we prune out some Steiner points in SP_{Ref} using k, and use the remaining Steiner points SP_{Roug} for the rough path calculation, i.e., $SP_{Roug} \subseteq SP_{Ref}$ and $G_{Roug}.V \subseteq G_{Ref}.V$. The pruned Steiner points are transferred to the error guaranteed path refinement step, and our experimental results show that it is very likely after the Snell's law path refinement, the path checking finds that there is no need to use Dijkstra's algorithm on these pruned Steiner points.

2) Efficient reduction of the search area (in the rough path calculation step) In Figure 3.2 (d), the edge sequence S' passed by $\Pi_{Roug}(s,t)$ is used in the Snell's law path refinement step. If we do not know S', we need to try Snell's law on different combinations of edges in E (so the search area is large) and select the shortest result path, which is time-consuming.

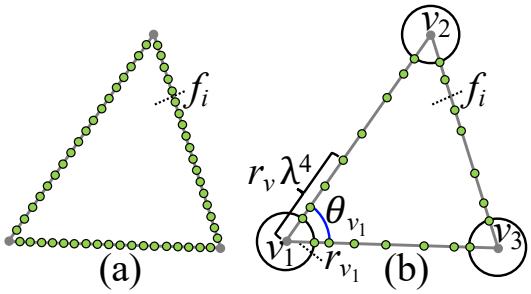
3) Usage of Snell's law (in the Snell's law path refinement step) In Figure 3.2 (e), we utilize Snell's law on S' to efficiently calculate $\Pi_{Ref\text{-}2}(s,t)$, such that the distance between the intersection point of $\Pi_{Ref\text{-}2}(s,t)$ on each edge of S' and that of $\Pi^*(s,t)$ is smaller than an error value $\delta = \frac{h\epsilon w}{6lW}$, and there is no need to place many Steiner points on edges in E.

3.2.3 Key Ideas of Additional Techniques

We also have the following four additional novel techniques to make algorithm *Roug-Ref* more efficient.

1) Efficient Steiner point placement scheme with a $(1+\epsilon)$ error bound (in the η calculation step) In Figure 3.2 (b), we have an efficient Steiner point placement scheme (i) by considering the additional geometric information of T, such as face weight, internal angle, and edge length, etc., so that we can place *different* numbers of Steiner points on different edges in E (for minimizing the total number of Steiner points), and (ii) by using mathematical transformations to express the error ratio concerning $(1+\epsilon)$ (so that with the η

calculation step, the rough path has an error ratio $(1+\eta\epsilon)$). Specifically, Figures 3.4 (a) and (b) show the placement of Steiner points under the same error ratio for two baseline algorithms *FixSP* and *FixSP-NoWei-Adp*, and for our algorithm *Roug-Ref*.



Remark: Placement of Steiner points in (a) *FixSP* and *FixSP-NoWei-Adp*, and (b) *Roug-Ref*.

Figure 3.4. Placement of Steiner points

2) Novel full edge sequence conversion technique using progressive approach (in the full edge sequence conversion step) In Figure 3.2 (d), S' passed by $\Pi_{Roug}(s,t)$ is non-full at vertex v. If we use Snell's law on S' to calculate a refined path, after the path exits v, we do not know where it goes next, so we need to add *all* the edges connected to v n S' (edges in blue) for error guarantees, and try Snell's law on different combinations of edge sequences to select the shortest result path. Indeed, only a subset of these edges is required. To solve it, we need the full edge sequence conversion step.

Illustration: In Figure 3.3 (a), given a rough path $\langle s, \phi_1, \phi_2, \phi_3, \phi_4, \phi_5, t\rangle$ (i.e., the orange line) whose corresponding edge sequence is non-full, we divide it into a smaller segment $\langle \phi_1, \phi_2, \phi_3, \phi_4, \phi_5\rangle$ that all the edges passed by this segment have length equal to 0, i.e., at ϕ_2, ϕ_3 and ϕ_4. We add more Steiner points to *progressively* find a new path segment, i.e., the purple dashed line between ϕ_1 and ϕ_5, until the edge sequence (in red) passes by this path segment is full. If the distance of the new path segment is smaller than that of the original one, we replace the new one with the original one, and obtain a modified rough path (i.e., the purple dashed line between s and t). Figure 3.3 (b) shows a similar example.

3) Novel effective weight pruning technique (in the Snell's law path refinement step) In Figure 3.2 (f), we use Snell's law to find optimal intersection points on each edge of S, and then connect them to form $\Pi_{Ref\text{-}2}(s,t)$. The basic idea is to use binary search, but we can efficiently prune out some checking by utilizing *effective weight* information on T.

Illustration: In Figure 3.5 (a), 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. We use binary search to adjust the position of m_1 to the left or right by checking whether t is on the left or right of this ray, and repeat until the ray passes the entire S, i.e., the purple ray from s to m_1^1. In Figure 3.5 (b), we regard all the faces in $F(S)$ except f_1 as one *effective face* $\triangle u_p p_1 q_1$, and use the ray in the purple line for calculating its effective weight. Then, we can calculate the position of m_{ef} on e_1 in one quartic equation using f_1's weight and $\triangle u_p p_1 q_1$'s effective weight. In Figure 3.5 (c), we trace the ray starting from s and passing m_{ef} (i.e., the dark blue line). We iterate the midpoint selection step until (i) the ray hits t or (ii) the distance between the new m_1 and the previous m_1 is smaller than δ. In Figure 3.5 (d), we continue on other edges in S.

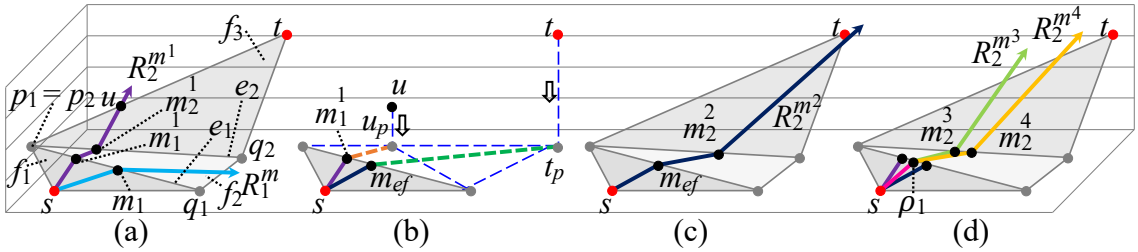


Figure 3.5. Snell's law path refinement step in algorithm *Ref*

4) Novel error guaranteed path refinement using pruning technique (in the error guaranteed path refinement step) In Figure 3.2 (g), for the path checking step, when $k \leqslant 2$, only in a rare case (i.e., the edge sequence passed by the rough path differs from that of the optimal weighted shortest path) which never occurs in our experiments, we need the error guaranteed path refinement step in Figures 3.2 (h) and (i). When k is larger, more Steiner points are removed, and η is larger, so the chance that $|\Pi_{Ref\text{-}2}(s,t)| > \frac{(1+\epsilon)}{(1+\eta\epsilon)}|\Pi_{Roug}(s,t)|$ becomes larger, we need this step to ensure error guarantee and hope that the query time of algorithm *Roug-Ref* will not increase a lot, by using the node information.

Illustration: (i) In the rough path calculation step, see Figure 3.2 (c), we have $dist_{Roug}(b) = 9 + 1 \times 5 = 14$ and $prev_{Roug}(b) = a$ as node information of b. (ii) In the error guaranteed path refinement step, see Figure 3.2 (h), since $G_{Roug}.V \subseteq G_{Ref}.V$, we know $dist_{Ref}(b) = 14$ and $prev_{Ref}(b) = a$. We maintain a *priority queue* [26] $Q = \{\{u_1, dist_{Ref}(u_1)\}, \dots\}$ in Dijkstra's algorithm on G_{Ref} that stores a set of nodes $u_i \in G_{Ref}.V$ waiting for processing. Suppose we need to process a and c, so Q stores $\{\{a, 9\}, \{c, 8\}\}$. We

dequeue c with a shorter distance value ($8 < 9$), and one adjacent node of c is b. Since $dist_{Ref}(b) = 14 < dist_{Ref}(c) + w_{bc} \cdot d(b, c) = 8 + 1 \times 7 = 15$, we do not need to insert b and $dist_{Ref}(b) = 15$ into the queue for time-saving. Since $G_{Ref}.V$ contains $G_{Roug}.V$ and the removed Steiner points, the query time by performing Dijkstra's algorithm on G_{Ref} without the node information is the same as the time of the rough path calculation step plus the time of error guaranteed path refinement step.

3.2.4 Implementation Details of Algorithm *Roug*

We give implementation details of algorithm *Roug* (see Figures 3.2 (a) to (d)). We mainly discuss our efficient Steiner points placement scheme with a $(1+\epsilon)$ error bound, (i.e., the technique in algorithm *LogSP-Adp*).

Detail: Given a vertex v in V, we let h_v be the minimum height starting from v on the faces containing v, let C_v be a *sphere* centered at v with radius $r_v = \frac{1+\epsilon+\frac{w}{w}-\sqrt{(1+\epsilon+\frac{w}{w})^2-4\epsilon}}{4} \cdot h_v$, and let θ_v be the angle of two edges of T connecting with v. In Figure 3.4 (b), there is a sphere C_{v_1} centered at v_1, with the radius r_{v_1} and the angle θ_{v_1} of v_1. For each vertex v of face f_i, we place Steiner points $p_1, p_2, \ldots, p_{\tau_p - 1}$ on two edges of f_i connecting with v, such that $|\overline{vp_j}| = r_v \lambda^{j-1}$, where $\tau_p = \log_\lambda \frac{|e_p|}{2 \cdot r_v}$ for every integer $2 \leqslant j \leqslant \tau_p - 1$, and $\lambda = (1 + \frac{1+\epsilon+\frac{w}{w}-\sqrt{(1+\epsilon+\frac{w}{w})^2-4\epsilon}}{4} \cdot \sin\theta_v)$. Algorithm *LogSP* has $r_v = \epsilon' \cdot h_v$ and $\lambda = (1 + \epsilon' \cdot \sin\theta_v)$. Since we adapt *LogSP* to be *LogSP-Adp* by setting $\epsilon' = \frac{1+\epsilon+\frac{w}{w}-\sqrt{(1+\epsilon+\frac{w}{w})^2-4\epsilon}}{4}$, we obtain r_v and λ w.r.t. ϵ.

3.2.5 Implementation Details of Algorithm *Ref*

We give implementation details of algorithm *Ref*.

1) Full edge sequence conversion See Figures 3.2 (e) and 3.3.

Detail and example: A point is said to be *on the edge* (resp. *vertex*) if it lies in the internal of an edge in E (resp. it lies on the vertex in V). In both Figures 3.3 (a) and (b), ϕ_1 is on the edge, and ϕ_2 is on the vertex. Algorithm 1 shows this step, and the following is an example.

Algorithm 1 *EdgSeqConv* ($\Pi_{Roug}(s,t)$, ζ)

Input: the rough path $\Pi_{Roug}(s,t)$ and ζ (a constant and normally set as 10)
Output: the edge sequence S of the modified rough path $\Pi_{Ref\text{-}1}(s,t)$

1: $v_s \leftarrow NULL, v_e \leftarrow NULL, E' \leftarrow \emptyset, \Pi_{Ref\text{-}1}(s,t) \leftarrow \Pi_{Roug}(s,t)$
2: **for** each point v that $\Pi_{Roug}(s,t)$ intersects with an edge in E (except s and t), such that v is on the vertex **do**
3: $v_c \leftarrow v, v_n \leftarrow v_c.next, v_p \leftarrow v_c.prev$
4: **if** v_n is on the vertex and v_p is on the edge **then**
5: $v_s \leftarrow v_c$, $E' \leftarrow E' \cup$ edges with v_c as one endpoint
6: **else if** both v_n and v_p are on the edge **then**
7: $E' \leftarrow E' \cup$ edges with v_c as one endpoint
8: **else if** v_n is on the edge and v_n is on the vertex **then**
9: $v_e \leftarrow v_n$, $E' \leftarrow E' \cup$ edges with v_c as one endpoint
10: add new Steiner points at the midpoints between the vertices and original Steiner points on E'
11: $\Pi'_{Roug}(v_s, v_e) \leftarrow$ path calculated using Dijkstra's algorithm on the weighted graph constructed by these new Steiner points and V
12: $\Pi'_{Ref\text{-}1}(v_s, v_e) \leftarrow$ *EdgSeqConv* ($\Pi'_{Roug}(v_s, v_e)$, ζ)
13: **if** $|\Pi'_{Ref\text{-}1}(v_s, v_e)| < |\Pi_{Roug}(v_s, v_e)|$ **then**
14: $\Pi_{Ref\text{-}1}(s,t) \leftarrow \Pi_{Ref\text{-}1}(s, v_s) \cup \Pi'_{Ref\text{-}1}(v_s, v_e) \cup \Pi_{Ref\text{-}1}(v_e, t)$
15: **else if** both v_p and v_n are on the edge **then**
16: **for** $i \leftarrow 1$ to ζ **do**
17: 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
18: $\Pi_B(v_p, v_n) \leftarrow$ path passes newly added Steiner points on the B side of the path (v_p, v_c, v_n), where $B = \{left, right\}$
19: **if** $|\Pi_B(v_p, v_n)| < |\Pi_{Roug}(v_p, v_n)|$ **then**
20: $\Pi_{Ref\text{-}1}(s,t) \leftarrow \Pi_{Ref\text{-}1}(s, v_p) \cup \Pi_B(v_p, v_n) \cup \Pi_{Ref\text{-}1}(v_n, t)$
21: **break**
22: **return** the edge sequence S of $\Pi_{Ref\text{-}1}(s,t)$ based on T

(i) *Successive point*: Lines 4-14, see Figure 3.3 (a) with the orange path as input. $v_s = \phi_1$ and $v_n = \phi_5$ are *start* and *end vertices*, respectively. The blue points are the new Steiner points. The orange line and purple dashed line between ϕ_1 and ϕ_5 are $\Pi_{Roug}(v_s, v_e)$ and $\Pi'_{Ref\text{-}1}(v_s, v_e)$.

(ii) *Single point*: Lines 15-21, see Figure 3.3 (b) with the orange path as input. The blue points are new Steiner points. The orange, purple line-dashed and pink dot-dashed lines between ϕ_1 and ϕ_3 are $\Pi_{Roug}(v_p, v_n)$, $\Pi_{left}(v_p, v_n)$ and $\Pi_{right}(v_p, v_n)$.

2) Snell's law path refinement See Figures 3.2 (f) and 3.5.

Detail and example: Let $\Pi_{\mathit{Ref}\text{-}2}(s,t) = \langle s, \rho_1, \ldots, \rho_l, t\rangle$, where ρ_i is a point on an edge in E with $i \in \{1,\ldots,l\}$. Given S, s and a point c_1 on $e_1 \in S$, we can obtain a *surface ray* $\Pi_c = \langle s, c_1, \ldots, c_g, R^c_g\rangle$ starting from s, hitting c_1 and following Snell's law on S, where $1 \leqslant g \leqslant l$, each c_i is an intersection point in Π_c with $i \in \{1,\ldots,g\}$, and R^c_g is the final out-ray at $e_g \in S$. In Figure 3.5 (a), $\Pi_m = \langle s, m_1, R^m_1\rangle$ in blue line does not pass the whole $S = \langle e_1, e_2\rangle$, but $\Pi_{m^1} = \langle s, m^1_1, m^1_2, R^{m^1}_2\rangle$ in purple line passes the whole S. Algorithm 2 shows this step, and the following is an example.

(i) *Binary search initial path finding*: Lines 6-15, see Figure 3.5 (a). In lines 6-9, we first have the blue ray $\Pi_m = \langle s, m_1, R^m_1\rangle$ that does not pass the whole S, we set $[a_1, b_1] = [p_1, m_1]$. In lines 10-15, we then have the purple ray $\Pi_{m^1} = \langle s, m^1_1, m^1_2, R^{m^1}_2\rangle$ passes the whole S, we set $[a_1, b_1] = [m^1_1, m_1]$ and $[a_2, b_2] = [m^1_2, q_2]$.

(ii) *Effective weight pruning*: Lines 16-24. The purple ray passes the whole S for the first time, we can use effective weight pruning. In line 17 and Figure 3.5 (a), we get u. In lines 18-22 and Figure 3.5 (b), we (1) get u_p and $f_{ef} = \triangle u_p p_1 q_1$, (2) calculate w_{ef} using purple line $\overline{sm^1_1}$, the orange dashed line $\overline{m^1_1 u_p}$, f_1, f_{ef}, w_1 and Snell's law, (3) get t_p, (4) set m_{ef} to be unknown and use Snell's law in vector form [11], build a quartic equation with unknown at the power of four using w_1, w_{ef}, the dark blue line $\overline{sm_{ef}}$ and the green dashed line $\overline{m_{ef} t_p}$, and use root formula [50] to solve m_{ef}. In lines 23-24 and Figure 3.5 (c), we compute the dark blue ray $\Pi_{m^2} = \langle s, m_{ef}, m^2_2, R^{m^2}_2\rangle$, and set $[a_1, b_1] = [m^1_1, m_{ef}]$ and $[a_2, b_2] = [m^1_2, m^2_2]$.

(iii) *Binary search refined path finding*: Lines 2, 6-15, 25-26, see Figure 3.5 (d). In lines 6-15, we iterate until $|a_1 b_1| = |m^1_1 m_{ef}| < \delta$. In line 25, we have ρ_1, the pink dashed link $\Pi_{\mathit{Ref}\text{-}2}(s,t) = \langle s, \rho_1\rangle$, and *root* $= \rho_1$. In line 2, we iterate to obtain the green ray $\Pi_{m^3} = \langle \rho_1, m^3_2, R^{m^3}_2\rangle$ and the yellow ray $\Pi_{m^4} = \langle \rho_1, m^4_2, R^{m^4}_2\rangle$. Until we process all the edges in $S = \langle e_1, e_2\rangle$, we get result path $\Pi_{\mathit{Ref}\text{-}2}(s,t) = \langle s, \rho_1, \rho_2, t\rangle$.

3) Error guaranteed path refinement See Figure 3.2 (h).

Detail and example: Algorithm 3 shows this step, and the following is an example.

(i) *Distance and previous node initialization*: Lines 2-5. For d, $\mathit{dist}_{\mathit{Ref}}(d) = \infty$ and $\mathit{prev}_{\mathit{Ref}}(d) = \mathit{NULL}$; for b, $\mathit{dist}_{\mathit{Ref}}(b) = \mathit{dist}_{\mathit{Roug}}(b) = 9 + 1 \times 5 = 14$ and $\mathit{prev}_{\mathit{Ref}}(b) =$

Algorithm 2 *SneLawRef* (s, t, δ, S)

Input: source s, destination t, user parameter δ and edge sequence S
Output: the refined path $\Pi_{Ref\text{-}2}(s, t)$

1: $\Pi_{Ref\text{-}2}(s, t) \leftarrow \{s\}$, $root \leftarrow s$
2: **for** each $e_i \in S$ with $i \leftarrow 1$ to $|S|$ **do**
3: $\quad a_i \leftarrow e_i$ left endpoint, $b_i \leftarrow e_i$ right endpoint, $[a_i, b_i] \leftarrow$ an interval
4: **for** each $e_i \in S$ with $i \leftarrow 1$ to $|S|$ **do**
5: $\quad$ **while** $|a_i b_i| \geqslant \delta$ **do**
6: $\quad\quad m_i \leftarrow$ midpoint of $[a_i, b_i]$ and calculate a surface ray with $\Pi_m = \langle root, m_i, \ldots, m_g, R_g^m \rangle$ with $g \leqslant l$
7: $\quad\quad$ **if** Π_m does not pass the whole S, i.e., $g < l$ **then**
8: $\quad\quad\quad$ **if** e_{g+1} is on R_g^m's left or right side **then**
9: $\quad\quad\quad\quad [a_j, b_j] \leftarrow [a_j, m_j]$ or $[m_j, b_j]$ for each $j \leftarrow i$ to g
10: $\quad\quad$ **else if** Π_m passes the whole S, i.e., $g = l$ **then**
11: $\quad\quad\quad$ **if** t is on R_g^m **then**
12: $\quad\quad\quad\quad \Pi_{Ref\text{-}2}(s, t) \leftarrow \Pi_{Ref\text{-}2}(s, t) \cup \{m_i, \ldots, m_g, t\}$
13: $\quad\quad\quad\quad$ **return** $\Pi_{Ref\text{-}2}(s, t)$
14: $\quad\quad\quad$ **else if** t is on R_g^m's left or right side **then**
15: $\quad\quad\quad\quad [a_j, b_j] \leftarrow [a_j, m_j]$ or $[m_j, b_j]$ for each $j \leftarrow i$ to g
16: $\quad\quad\quad$ **if** have not used effective weight pruning on e_i **then**
17: $\quad\quad\quad\quad u \leftarrow$ the intersection point between R_l^m and one of the two edges that are adjacent to t in the last face f_{l+1} in $F(S)$
18: $\quad\quad\quad\quad u_p \leftarrow$ projected point of u on the first face f_1 in $F(S)$
19: $\quad\quad\quad\quad f_{ef} \leftarrow$ *effective face* contains all faces in $F(S) \setminus \{f_1\}$
20: $\quad\quad\quad\quad w_{ef} \leftarrow$ *effective weight* for f_{ef}, calculated using $\overline{sm_i}$, $\overline{m_i u_p}$, f_1, f_{ef}, w_1 (the weight for f_1) and Snell's law
21: $\quad\quad\quad\quad t_p \leftarrow$ the projected point of t on f_1
22: $\quad\quad\quad\quad m_{ef} \leftarrow$ effective intersection point on e_1, calculated using w_1, w_{ef}, s, t_p and Snell's law in a quartic equation
23: $\quad\quad\quad\quad m_i \leftarrow m_{ef}$, compute $\Pi_m = \langle root, m_i, \ldots, m_g, R_g^m \rangle$
24: $\quad\quad\quad\quad$ update $[a_j, b_j]$ for each $j \leftarrow i$ to g same as in lines 11-15
25: $\quad \rho_i \leftarrow [a_i, b_i]$ midpoint, $\Pi_{Ref\text{-}2}(s, t) \leftarrow \Pi_{Ref\text{-}2}(s, t) \cup \{\rho_i\}$, $root \leftarrow \rho_i$
26: $\Pi_{Ref\text{-}2}(s, t) \leftarrow \Pi_{Ref\text{-}2}(s, t) \cup \{t\}$
27: **return** $\Pi_{Ref\text{-}2}(s, t)$

$prev_{Roug}(b) = a$.

(ii) *Priority queue looping*: Lines 6-13. Suppose Q stores $\{\{a, 9\}, \{c, 8\}\}$, we dequeue c. One adjacent node of c is b, since $dist_{Ref}(b) = 14 < dist_{Ref}(c) + w_{bc} \cdot d(b, c) = 8 + 1 \times 7 = 15$, there is no need to update $dist_{Ref}(b)$ and $prev_{Ref}(b)$, and no need to enqueue $\{b, dist_{Ref}(b) = 15\}$ into Q for time-saving. If we change the weighted shortest distance from s to a from 9 to 6, we have the following. Suppose Q stores $\{\{a, 6\}, \{c, 8\}\}$, we dequeue a. One adjacent

Algorithm 3 *ErrGuarRef* $(s, t, SP_{Ref}, NodeInfo)$

Input: source s, destination t, Steiner points SP_{Ref}, node information $dist_{Roug}(u)$ and $prev_{Roug}(u)$ (based on s) for each $u \in G_{Roug}.V$

Output: the refined path $\Pi_{Ref\text{-}3}(s, t)$

1: build a weighted graph G_{Ref} using SP_{Ref}, enqueue $\{s, 0\}$ into Q
2: **for** each $u \in G_{Ref}.V$ **do**
3: **if** $u \in G_{Ref}.V \setminus G_{Roug}.V$ (resp. $u \in G_{Roug}.V$) **then**
4: $dist_{Ref}(u) \leftarrow \infty$ (resp. $dist_{Roug}(u)$)
5: $prev_{Ref}(u) \leftarrow NULL$ (resp. $prev_{Roug}(u)$)
6: **while** Q is not empty and the to-be-dequeued node is not t **do**
7: dequeue node v from Q with smallest distance value $dist_{Ref}(v)$
8: **for** each adjacent vertex v' of v, such that $\overline{vv'} \in G_{Ref}.E$ **do**
9: **if** $dist_{Ref}(v') > dist_{Ref}(v) + w_{vv'} \cdot d(v, v')$ **then**
10: $dist_{Ref}(v') \leftarrow dist_{Ref}(v) + w_{vv'} \cdot d(v, v')$, $prev_{Ref}(v') \leftarrow v$
11: enqueue $\{v', dist_{Ref}(v')\}$ into Q
12: **if** $dist_{Ref}(v') = dist_{Ref}(v) + w_{vv'} \cdot d(v, v')$ **then**
13: enqueue $\{v', dist_{Ref}(v')\}$ into Q
14: $u \leftarrow prev_{Ref}(t)$, $\Pi_{Ref\text{-}3}(s, t) \leftarrow \{t\}$
15: **while** $u \neq s$ **do**
16: $\Pi_{Ref\text{-}3}(s, t) \leftarrow \Pi_{Ref\text{-}3}(s, t) \cup \{u\}$, $u \leftarrow prev_{Ref}(u)$
17: $\Pi_{Ref\text{-}3}(s, t) \leftarrow \Pi_{Ref\text{-}3}(s, t) \cup \{s\}$, reverse $\Pi_{Ref\text{-}3}(s, t)$
18: **return** $\Pi_{Ref\text{-}3}(s, t)$

node of a is b, since $dist_{Ref}(b) = 11 = dist_{Ref}(a) + w_{ab} \cdot d(a, b) = 6 + 1 \times 5 = 11$, we enqueue $\{b, dist_{Ref}(b) = 11\}$ into Q, and there is no need to update $dist_{Ref}(a)$ and $prev_{Ref}(a)$ for time-saving.

(iii) *Path retrieving*: Lines 14-17. We obtain $\Pi_{Ref\text{-}3}(s, t)$.

3.2.6 Theoretical Analysis

Theorem 3.2.1 shows the analysis of algorithm *Roug-Ref*.

Theorem 3.2.1 *The query time and memory usage for algorithm Roug-Ref is* $O(N \log N + l)$ *and* $O(N + l)$. *It guarantees that* $|\Pi(s, t)| \leqslant (1 + \epsilon)|\Pi^*(s, t)|$.

Proof. To finish the proof, we need Lemmas 3.2.1, 3.2.2 and 3.2.3, and Theorems 3.2.2, 3.2.3 and 3.2.4.

Lemma 3.2.1 *There are at most* $k_{SP} \leqslant 2(1+\log_{\lambda}\frac{L}{2\cdot r})$ *Steiner points on each edge in* E *when placing Steiner point based on* ϵ *in algorithm Roug, where* r *is the minimum* r_v *for all* $v \in V$.

Proof. We prove it in 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 $(1+\log_{\lambda}\frac{L}{2\cdot r})$ Steiner points, where the 1 comes from the first Steiner point that is the nearest one from the endpoint. Therefore, we have $k_{SP} \leqslant 2(1+\log_{\lambda}\frac{L}{2\cdot r})$. □

Lemma 3.2.2 *When placing Steiner point based on* ϵ *in algorithm Roug, based on* $1+(2+\frac{2W}{(1-2\epsilon')\cdot w})\epsilon' = 1+\epsilon$, *we obtain* $\epsilon' = \frac{1+\epsilon+\frac{W}{w}-\sqrt{(1+\epsilon+\frac{W}{w})^2-4\epsilon}}{4}$ *with* $0 < \epsilon' < \frac{1}{2}$ *and* $\epsilon > 0$.

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

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

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

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

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

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

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

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

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

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 can plot the figure for this expression, and will found that the upper limit is always $\frac{1}{2}$ if we use $-$). □

Lemma 3.2.3 *Let* h *be the minimum height of any face in* F *whose vertices have non-negative integer coordinates no greater than* I*. Then,* $h \geqslant \frac{1}{I\sqrt{3}}$.

Proof. Let a and b be two non-zero vectors with non-negative integer coordinates no greater than I, 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 $h = \min_{a,b} \frac{|a\times b|}{|b|} = \min_{a,b} \frac{\sqrt{\omega}}{\sqrt{x_a^2+y_a^2+z_a^2}} \geqslant \frac{1}{I\sqrt{3}} \min_{a,b} \sqrt{\omega} \geqslant \frac{1}{I\sqrt{3}}$, where $\omega = (x_a y_b - y_a x_b)^2 + (y_a z_b - z_a y_b)^2 + (z_a x_b - x_a z_b)^2$. □

Theorem 3.2.2 *The query time for algorithm Roug is* $O(N \log N)$ *and the memory usage is* $O(N)$.

Proof. If we do not remove Steiner points in algorithm *Roug*, i.e., we place Steiner point based on ϵ, and then following Lemma 3.2.1, the number of Steiner points k_{SP} on each edge is $O(\log_\lambda \frac{L}{r})$, where $\lambda = (1 + \frac{1+\epsilon+\frac{W}{w}-\sqrt{(1+\epsilon+\frac{W}{w})^2-4\epsilon}}{4} \cdot \sin\theta)$, $r = \frac{1+\epsilon+\frac{W}{w}-\sqrt{(1+\epsilon+\frac{W}{w})^2-4\epsilon}}{4} \cdot h$ and θ is the minimum θ_v for all $v \in V$. Following Lemma 3.2.2 and Lemma 3.2.3, $r = O(\frac{\epsilon}{I})$, and thus $k_{SP} = O(\log \frac{LI}{\epsilon})$. But, since we remove Steiner points for η calculation and rough path calculation, the remaining Steiner points on each edge are $O(1)$. So $|G_{Roug}.V| = N$. Due to the usage of Dijkstra's algorithm, the query time of algorithm *Roug* is $O(N \log N)$ and the memory usage is $O(N)$. □

Theorem 3.2.3 *The query time for the full edge sequence conversion step in algorithm Ref is* $O(N \log N)$ *and the memory usage is* $O(N)$.

Proof. Firstly, we prove the *query 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 care 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's algorithm to refine it, and the query 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 query time is $O(\zeta n) = O(N)$. Therefore, the query 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's algorithm, whose memory usage is $O(|G_{Roug}.V|)$, where $|G_{Roug}.V|$ is size of vertices in the Dijkstra's algorithm. Handling one single endpoint case needs $O(1)$ memory. Since there can 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 $O(N)$. □

Theorem 3.2.4 *The query time for the Snell's law path refinement step in algorithm Ref is* $O(l)$*, and the memory usage is* $O(l)$*.*

Proof. Firstly, we prove the *query time*. Since the effective weight pruning sub-step can directly find the optimal position of the intersection point on each edge in S in $O(1)$ time, the query 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 $O(l)$. □

With these lemmas and theorems, we can prove Theorem 3.2.1.

Firstly, we prove the *query time*. (1) In most cases, there is no need to use the error guaranteed path refinement step in algorithm *Ref*. In this case, the query time is the sum of the query time using algorithm *Roug* and the first three steps in algorithm *Ref*. From Theorems 3.2.2, 3.2.3 and 3.2.4, we obtain the query 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 query time of algorithm *Roug* and the error guaranteed path refinement step in algorithm *Ref* is exactly the same as the query time that we use Dijkstra's algorithm on the weighted graph G_{Ref} constructed by the original Steiner points (i.e., $k_{SP} = O(\log \frac{LI}{\epsilon})$ Steiner points on each edge) and V, which is $O(N \log \frac{LI}{\epsilon} \log(N \log \frac{LI}{\epsilon}) + l)$. But the constant term $O(\log \frac{LI}{\epsilon})$ is not important and can be omitted, so we obtain the query time $O(N \log N + l)$. (3) In general, the query time is $O(N \log N + l)$.

Secondly, we prove the *memory usage*. (1) In most cases, there is no need to use the error guaranteed path refinement step in algorithm *Ref*. In this case, the memory usage is the sum of the memory usage using algorithm *Roug* and the first three steps in algorithm *Ref*. From Theorems 3.2.2, 3.2.3 and 3.2.4, we obtain the average case 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 use Dijkstra's algorithm on the weighted graph G_{Ref} constructed by the original Steiner points (i.e., $k_{SP} = O(\log \frac{LI}{\epsilon})$ Steiner points on each edge) and V, which is $O(N \log \frac{LI}{\epsilon} + l)$. But the constant term $O(\log \frac{LI}{\epsilon})$ is not important and can be omitted, so we obtain the memory usage $O(N + l)$. (3) In general, the memory usage is $O(N + l)$.

Finally, we prove the *error bound*. Recall one baseline algorithm *LogSP*. We define the path calculated by algorithm *LogSP* between s and t to be $\Pi_{LogSP}(s, t)$. The study [20, 48] show that $|\Pi_{LogSP}(s, t)| \leqslant (1 + (2 + \frac{2W}{(1-2\epsilon')\cdot w})\epsilon')|\Pi^*(s, t)|$. A proof sketch appears in Theorem 1 of study [20] and a detailed proof appears in Theorem 3.1 of study [48]. We adapt algorithm *LogSP* to be algorithm *LogSP-Adp* by substituting $(2 + \frac{2W}{(1-2\epsilon')\cdot w})\epsilon' = \epsilon$ with $0 < \epsilon' < \frac{1}{2}$ and $\epsilon > 0$, we have $|\Pi_{LogSP\text{-}Adp}(s, t)| \leqslant (1 + \epsilon)|\Pi^*(s, t)|$ where $\epsilon > 0$ and $\Pi_{LogSP\text{-}Adp}(s, t)$ is the path result returned by algorithm *LogSP-Adp*. Recall that algorithm *LogSP-Adp* corresponds to the efficient Steiner point placement scheme in algorithm

Roug. This error bound is always true no matter whether the edge sequence passed by $\Pi_{\textit{LogSP-Adp}}(s,t)$ 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, calculate $\eta\epsilon$ based on the remaining Steiner points, and then use $\eta\epsilon$ as the input error ratio to calculate $\Pi_{\textit{Roug}}(s,t)$ in the rough path calculation step, so by adapt $\eta\epsilon$ as the input error ratio in algorithm *LogSP-Adp*, we have $|\Pi_{\textit{Roug}}(s,t)| \leqslant (1+\eta\epsilon)|\Pi^*(s,t)|$. Next, in the path checking step of algorithm *Ref*, if $|\Pi_{\textit{Ref-2}}(s,t)| \leqslant \frac{(1+\epsilon)}{(1+\eta\epsilon)}|\Pi_{\textit{Roug}}(s,t)|$, we return $\Pi_{\textit{Ref-2}}(s,t)$ as output $\Pi(s,t)$, which implies that $|\Pi_{\textit{Ref-2}}(s,t)| = |\Pi(s,t)| \leqslant (1+\epsilon)|\Pi^*(s,t)|$. Otherwise, we use the error guaranteed path refinement step in algorithm *Ref*, and we return $\Pi_{\textit{Ref-3}}(s,t)$ as output $\Pi(s,t)$, where the error bound is the same as in algorithm *LogSP-Adp*, i.e., $|\Pi_{\textit{Ref-3}}(s,t)| = |\Pi(s,t)| \leqslant (1+\epsilon)|\Pi^*(s,t)|$. Therefore, algorithm *Roug-Ref* guarantees that $|\Pi(s,t)| \leqslant (1+\epsilon)|\Pi^*(s,t)|$. □

3.3 Empirical Studies

3.3.1 Experimental Setup

We performed experiments on a Linux machine with 2.67 GHz CPU and 48GB memory. We implemented algorithms in C++. Our experimental setup generally follows the setups in the literature [45, 46, 51, 71, 72].

1) Datasets We conducted our experiment on 27 ($= 5+2+20$) real *TIN* datasets in Table 3.2. For 5 original datasets, *EP* has more mountains compared with the other 4 original datasets. For 2 small-version and 20 multi-resolution datasets, we generated them using *BH* and *EP* following the procedure in [51, 71, 72] (which creates a *TIN* with different resolutions). We use the slope of a face in *TIN* as that face's weight [43].

2) Algorithms We compared *Roug-Ref* with $(1+\epsilon)$-approximate algorithms, i.e., (i) *EdgSeq-Adp* [69], (ii) the best-known algorithm *FixSP* [43, 49], (iii) *FixSP-NoWei-Adp* [45], (iv) *LogSP-Adp* [20], and (v) variations of *Roug-Ref*. We also compared with (i) *EdgSeq* [69] without error bound, (ii & iii) *LogSP* [20] and *Roug* with distance error ratios larger than $(1+\epsilon)$ in [80]. We compare them, together with *EdgSeq* (without error guarantee) and

Table 3.2. *TIN* datasets in the study of shortest path queries on weighted *TIN*s

Name	$\|F\|$
Original dataset	
BearHead (*BH*) [71, 72]	280k
EaglePeak (*EP*) [71, 72]	300k
SeaBed (*SB*) [22]	2k
ValaisSwitzerland (*VS*) [18]	2k
PathAdvisor (*PA*) [78]	1k
Small-version dataset	
BH small-version (*BH-small*)	3k
EP small-version (*EP-small*)	3k
Multi-resolution dataset	
Multi-resolution of *BH*	1M, 2M, 3M, 4M, 5M
Multi-resolution of *BH-small*	10k, 20k, 30k, 40k, 50k
Multi-resolution of *EP*	1M, 2M, 3M, 4M, 5M
Multi-resolution of *EP-small*	10k, 20k, 30k, 40k, 50k

LogSP (which have an error ratio much larger than $(1+\epsilon)$) in Table 3.3. *Roug-Ref* has the smallest query time and memory usage. For these algorithms, we cannot run them in parallel (i.e., using a distributed implementation), since we only execute them once for one source.

Table 3.3. Comparison of algorithms on a weighted *TIN* regarding algorithm *Roug-Ref*

Algorithm	Query time		Size		Error
EdgSeq [69]	$O(N^4 \log(\frac{N^2 I W L}{w\epsilon}))$	Large	$O(N^2)$	Large	Large
EdgSeq-Adp [69]	$O(N^3 \log N + N^4 \log(\frac{N^2 N W L}{w\epsilon}))$	Large	$O(N^3)$	Large	Small
FixSP [43, 49]	$O(N^3 \log N)$	Large	$O(N^3)$	Large	Small
FixSP-NoWei-Adp [45]	$O(N^3 \log N)$	Large	$O(N^3)$	Large	Small
LogSP [20]	$O(N \log \frac{LI}{\epsilon'} \log(N \log \frac{LI}{\epsilon'}))$	Medium	$O(N \log \frac{LI}{\epsilon'})$	Medium	Large
LogSP-Adp [20]	$O(N \log \frac{LI}{\epsilon} \log(N \log \frac{LI}{\epsilon}))$	Medium	$O(N \log \frac{LI}{\epsilon})$	Medium	Small
***Roug-Ref* (ours)**	$\boldsymbol{O(N \log N + l)}$	**Small**	$\boldsymbol{O(N + l)}$	**Small**	**Small**

Remark: I is the minimum integer which is no less than the coordinate value of any vertex in V, W is the maximum weight of the face in F, w is the minimum weight of the face in F, L is the longest length of edges in T, and l is size of S.

3) Query generation We randomly chose pairs of vertices in V as source and destination, and we report the average, minimum, and maximum results of 100 queries.

4) Factors and measurements We studied three factors, namely (i) k (i.e., the removing value), (ii) ϵ (i.e., the error parameter), and (iii) dataset size (i.e., the number of faces in a

TIN). We used five measurements to evaluate the algorithm performance, namely (i) *query time*, (ii) *memory usage*, (iii) *chances of using error guaranteed path refinement step*, (iv) *average number of Steiner points on each edge (used in path calculation)*, and (v) *distance error ratio* (we use *EdgSeq-Adp* with $\epsilon = 0.05$ to simulate the exact result since no algorithm can calculate the exact shortest path passing on the weighted *TIN*).

3.3.2 Experimental Results

We compared (1) all algorithms on datasets with less than 250k faces, and (2) algorithms not involving *FixSP* components on datasets with more than 250k faces due to the expensive query time. The vertical bar means the minimum and maximum results.

1) Ablation study of *Roug-Ref* In our algorithm *Roug-Ref*, we have 4 variations: (i) we do not use our efficient Steiner points placement scheme, i.e., we use algorithm *FixSP* for Steiner point placement, (ii) we remove the full edge sequence conversion step, (iii) we remove the effective weight pruning out sub-step in the Snell's law path refinement step, and (iv) we do not use the node information for pruning in the error guaranteed path refinement step, for ablation study (they correspond to four techniques in Chapter 3.2.3). We use (i) *Roug-Ref-NoEffSP*, (ii) *Roug-Ref-NoEdgSeqConv*, (iii) *Roug-Ref-NoEffWeig*, and (iv) *Roug-Ref-NoPrunDijk*, to denote these variations. If we do not use the rough-refine concept, *Roug-Ref* becomes *LogSP-Adp*. We adapt *FixSP*, *FixSP-NoWei-Adp* and *EdgSeq-Adp* by using our rough-refine concept, and denote it as *Roug-Ref-Navie1*. We adapt *LogSP-Adp* similarly to be *Roug-Ref-Navie2*. Since k will only affect our algorithm *Roug-Ref* and its variations, we study the effect of k here.

Effect of *k*: In Figure 3.6 and Figure 3.7, we tested 5 values of k in $\{1, 2, 3, 4, 5\}$ on *BH-small* and *BH* dataset by setting ϵ to be 0.1 and 0.25 for ablation study, respectively. (i) When $k \leqslant 2$ and k increases, more Steiner points are removed and *Roug* runs faster, so the query time and memory usage of these algorithms decreases. But when $k > 2$, it has a higher chance (more than 99%) that *Ref* needs to perform the error guaranteed path refinement step, so the query time and memory usage have a sudden increase. Thus, the optimal k is 2 (also verified on other datasets shown in [80]). (ii) When $k = 2$, the query time of *Roug-Ref-Naive1* and *Roug-Ref-NoEffSP* are 230s and 210s on *BH-small* dataset, but their difference is small due to the log-scale in Figure 3.6 (a). Due to the same reason, the

difference in query time of the other four algorithms on *BH-small* dataset is also small, so we compare them with linear-scale in Figure 3.7 (which shows the usefulness of each component). We provided some selected metrics performance figures, the full sets of metrics performance figures in [80].

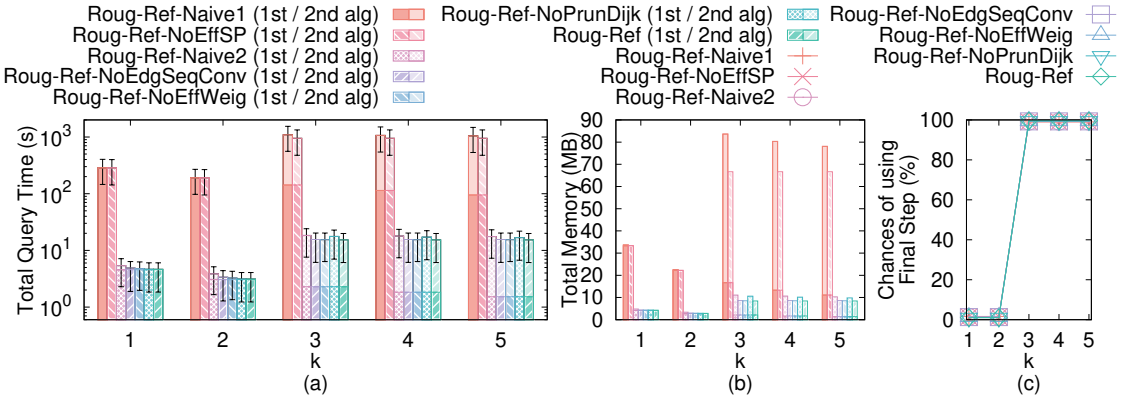


Figure 3.6. Ablation study (effect of k on *BH-small* dataset) regarding algorithm *Roug-Ref*

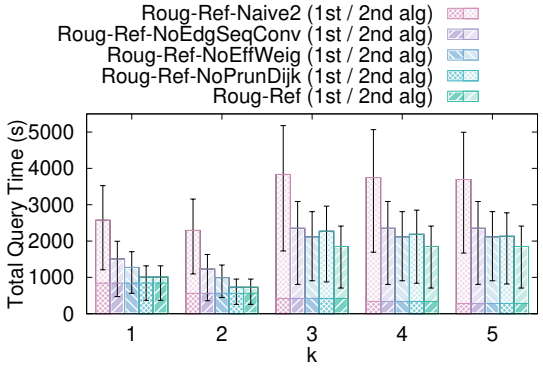


Figure 3.7. Ablation study (effect of k on *BH* dataset) regarding algorithm *Roug-Ref*

2) Baseline comparisons We then study the effect of ϵ and dataset size on other baselines when $k = 2$.

Effect of ϵ: In Figure 3.8, we tested 6 values of ϵ in $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on *EP-small* dataset. (i) When ϵ increases, fewer Steiner points are required so the query time and memory usage decrease. (ii) The query time and memory usage of *Ref* is much smaller than that of *Roug*. (iii) *Roug-Ref* performs better than all other algorithms concerning the query time and memory usage, and it is clear to observe the superior performance of *Roug-Ref*, due to the rough-refine concept and four novel techniques. The distance error ratio of all algorithms is very small, the value is 0.0004 when $\epsilon = 1$ for *Roug-Ref*. (iv)

When $\epsilon = 0.05$, *Roug-Ref* has 160 fewer Steiner points on each edge than *FixSP* and *FixSP-NoWei-Adp*, the query time and calculated path's distance of *Roug-Ref* is 14.6s and 105.3m, but are both 23,800s $\approx$ 7.2 hours and 105.2m for *FixSP* and *FixSP-NoWei-Adp*, respectively, since *Roug-Ref* uses the rough-refine concept. *EdgSeq-Adp* performs worse than *FixSP* since *EdgSeq-Adp* first uses *FixSP* and then uses Snell's law for the weighted shortest path calculation. *LogSP-Adp* does not perform well since it does not utilize Snell's law.

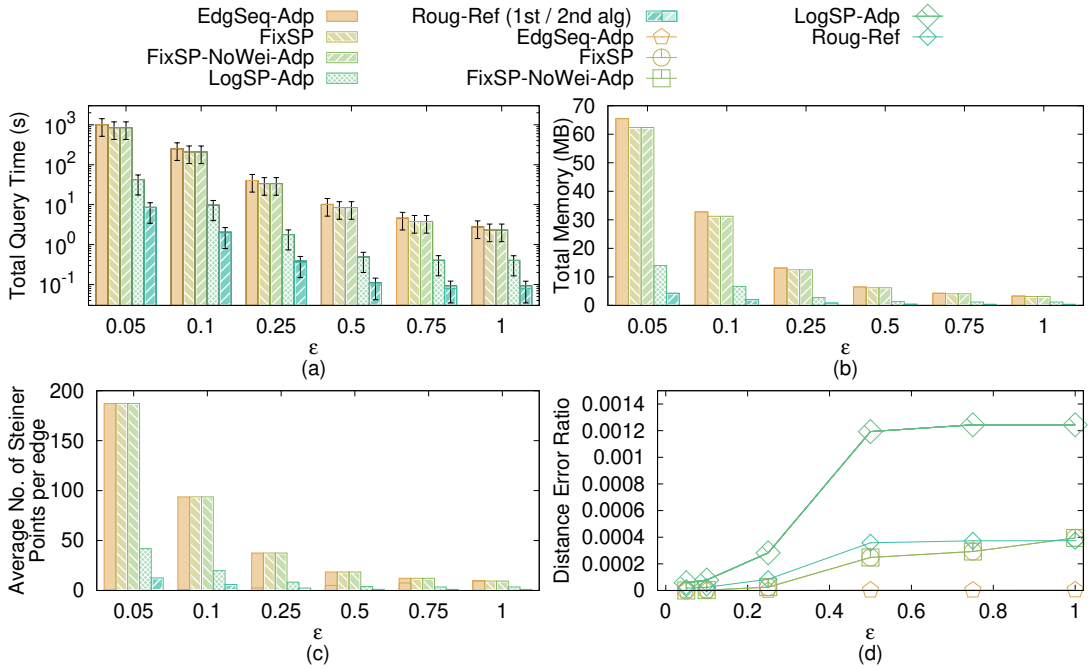


Figure 3.8. Baseline comparisons (effect of ϵ on *EP-small* dataset) regarding algorithm *Roug-Ref*

Effect of dataset size: In Figure 3.9, we tested 5 values of dataset size in {10k, 20k, 30k, 40k, 50k} on multi-resolution of *EP-small* datasets by setting ϵ to be 0.1. When the dataset size is 50k, *Roug-Ref*'s query time is 10^3 times, 10^3 times, 10^3 times and 6 times smaller than that of *EdgSeq-Adp*, *FixSP*, *FixSP-NoWei-Adp* and *LogSP-Adp*, respectively.

3) Scalability test In Figure 3.10, we tested 5 values of dataset size in {1M, 2M, 3M, 4M, 5M} on multi-resolution of *EP* datasets by setting ϵ to be 0.25. When the dataset size is 5M, *Roug-Ref*'s query time is still reasonable. However, the query times for *EdgSeq-Adp*, *FixSP* and *FixSP-NoWei-Adp* are larger than 7 days, so they are excluded from the figure.

4) Other algorithms comparisons Given ϵ, *Roug-Ref* can calculate a path with distance d. But, *EdgSeq* does not have error bound, *LogSP* and *Roug* have distance error ratios larger

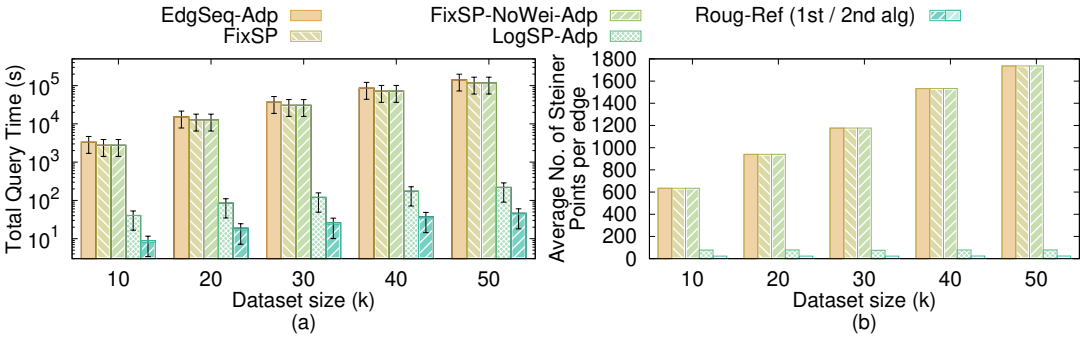


Figure 3.9. Baseline comparisons (effect of dataset size on multi-resolution of *EP-small* datasets) regarding algorithm *Roug-Ref*

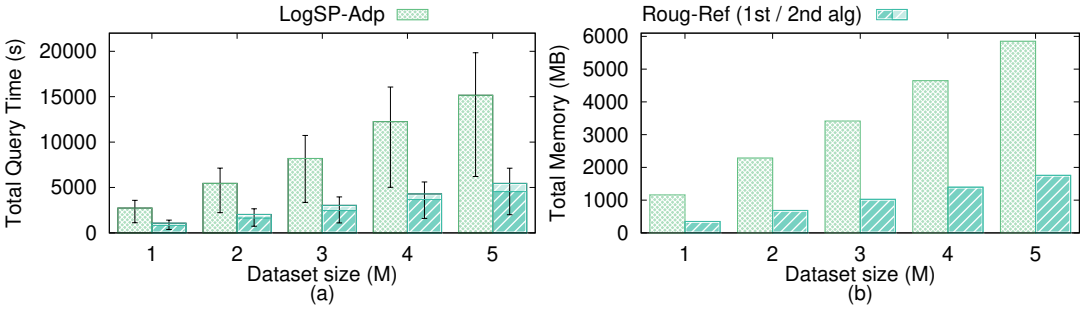


Figure 3.10. Scalability test regarding algorithm *Roug-Ref*

than $(1+\epsilon)$. We finetune their input to make their calculated path also with distance d. When $\epsilon = 0.1$, *Roug-Ref*, *EdgSeq*, *LogSP* and *Roug* run in 3s, 140s, 113s and 110s on *BH-small* dataset, respectively. In addition, we also compare *Roug-Ref* and algorithm [46] which calculates the shortest network path passing on the edges of the weighted *TIN*. When $\epsilon = 0.1$, they run in 3s and 2.1s on *BH-small* dataset, with distance error 0.0004 and 0.09, respectively. Thus, the shortest network path is not our main focus due to its large error.

5) Case study We performed a case study on the wildfire evacuation in Santa Monica Mountains National Recreation Area in Chapter 1. During the wildfire, we set each *TIN* face's weight to be the *damage level* [67] (a static value) of each region. In Figures 1.1 (f) and (g), the purple and blue paths between point a (a viewing platform) and point b (a shelter/hotel) have *weighted distances* of 15.2km and 209km, respectively, so we choose the blue path that does not pass the destroyed region as the evacuation path. The *distance* of the blue path is 11.2km [4], and the average car driving speed is 90km/h, so the evacuation

can be finished in 7.4($= \frac{11.2\text{km}\times 60\text{mins/h}}{90\text{km/h}}$) mins. The query time for the best-known algorithm *FixSP* and *Roug-Ref* are 11,900s $\approx$ 3.3 hours and 7.3s, respectively. So, the weighted shortest path calculated by *Roug-Ref* is the best evacuation path.

6) Summary *Roug-Ref* is up to 1,630 times and 40 times better than the best-known algorithm *FixSP* concerning the query time and memory usage, respectively. When the dataset size is 50k with $\epsilon = 0.1$, *Roug-Ref*'s query time is 73s $\approx$ 1.2 min, and memory usage is 43MB, but *FixSP*'s query time is 119,000s $\approx$ 1.5 days, and memory usage is 2.9GB.

CHAPTER 4

ORACLE FOR SHORTEST PATH QUERIES ON UPDATED TINS

This chapter studies shortest path queries on updated *TIN*s using an oracle. Chapter 4.1 provides the preliminary. Chapter 4.2 presents the methodology. Chapter 4.3 presents the experimental results.

4.1 Preliminary

4.1.1 Notation and Definitions

1) *TIN*s and POIs Consider a *TIN* T_{bef} with N vertices. T_{bef} contains a set of vertices V, edges E and faces F. Let l_{max} be the longest length of edges in E. Let x_v, y_v and z_v be the three coordinate values of each vertex $v \in \mathsf{V}$. If the vertices in V have position update, we obtain a new *TIN*, T_{aft}. Figures 1.1 (b) and (i) show T_{bef} and T_{aft}, respectively. Similarly, Figures 4.1 (a) to (c) show T_{bef}, and Figures 4.1 (d) to (g) show T_{aft}. There is no need to consider the case when N changes because T_{bef} and T_{aft} have the same 2D grid with $\overline{x} \times \overline{y} = \mathsf{N}$ vertices [51, 72, 71]. Specifically, for T_{bef} in Figure 4.1 (a), given a *fixed* region, existing methods [76, 43, 71, 64] sample a *fixed* set of points on T_{bef} on the 2D grid in the x-y plane and use the elevations of these points as the z-coordinates, yielding the final set of vertices on T_{bef}. For T_{aft} in Figure 4.1 (d), since we focus on the *same* region (although the *shape* of the region on T_{aft} may change), the set of points is *fixed*, and N is also *fixed*. In the P2P query, let P be a set of n POIs on T_{bef}. There is no need to consider when n changes, or when $n > \mathsf{N}$. The set of red points in Figure 4.1 (a) is P. When a POI is added, we create an oracle that answers the AR2AR query, which implies we consider all possible POIs to be added. When a POI is removed, we continue to use the original oracle. When $n > \mathsf{N}$, we still create an oracle that answers the AR2AR query.

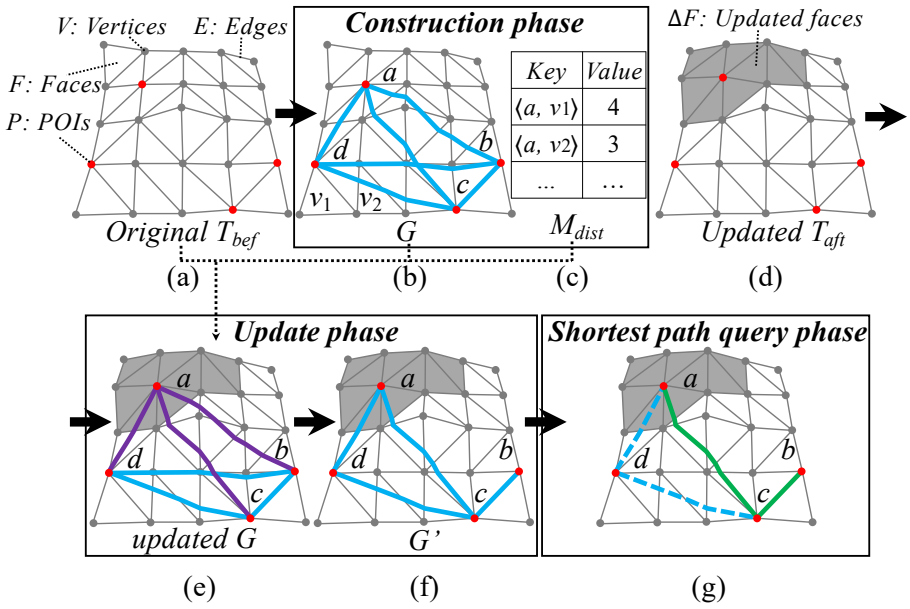


Figure 4.1. *UP-Oracle* framework overview

2) Path Given s and t in P, and a *TIN* T (which can be T_{bef} or T_{aft}), let $\Pi(s, t|T)$ be the exact shortest path between s and t passing on T, and $|\cdot|$ be a path's distance (e.g., $|\Pi(s, t|T)|$ is $\Pi(s, t|T)$'s).

3) Updated and non-updated faces Given T_{bef}, T_{aft} and P, let $\Delta F = \{f_1, f_2, \ldots, f_{|\Delta F|}\}$ be a set of *updated faces*, such that f_i is a face in F with at least one of its three vertices' coordinates differing between T_{bef} and T_{aft} and $|\Delta F|$ is size of ΔF. It is easy to obtain ΔF by comparing T_{bef} and T_{aft}. In Figure 1.1 (i) (resp. Figure 4.1 (d)), the yellow (resp. gray) region is ΔF based on T_{bef} and T_{aft}. There is no need to consider the case with two or more *disjoint* non-empty sets of updated faces. If this happens, we can create a larger set of faces that contains these disjoint sets. Thus, in Figures 1.1 (i) and Figure 4.1 (d), the set of updated faces that we consider is connected [56]. We say that a point (either a vertex or a POI) is in ΔF if it is on a face in ΔF, and we say that a path passes ΔF if any segment of this path is on a face ΔF. In Figure 4.1 (e), a is in ΔF, and the purple path between a and b passes ΔF.

4) Disk Given a point p on T_{bef} and a constant $r > 0$, let $D(p, r)$ be a disk with p as center and r as radius, which consists of points on T_{bef} that have exact shortest distance to p is no more than r. Given a face f_i, if a point q exists on f_i such that the shortest distance between p and q is no more than r, then disk $D(p, r)$ *intersects with* face f_i. Figure 4.2 shows two disks with u and v as centers, and both $\frac{|\Pi(u,v|T_{bef})|}{2}$ as radii, that do not intersect with any

updated faces. Table 4.1 shows a notation table.

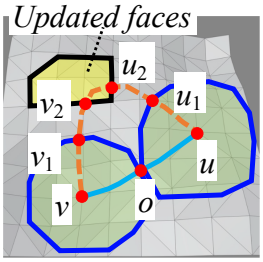


Figure 4.2. An unaffected path

Table 4.1. Frequent used notations in the study of shortest path queries on updated *TIN*s

Notation	**Meaning**
T_{bef}	The *TIN* before updates
T_{aft}	The *TIN* after updates
V, E, F	The set of vertices, edges and faces of T_{bef} and T_{aft}
l_{max}	The longest length of edges in T_{bef}
N	The size of V
P	The set of POI
n	The size of P
$\Pi(s, t\|T)$	The exact shortest path between s and t passing on T (which can be T_{bef} or T_{aft})
$\|\Pi(s, t\|T)\|$	$\Pi(s, t\|T)$'s distance
ΔF	The updated faces of T_{bef} and T_{aft}
$D(p, r)$	A disk with p as center and r as radius
ϵ	The error parameter

4.1.2 Problem

Given T_{bef}, T_{aft} and P, the problem is design an efficient $(1+\epsilon)$-approximate shortest path oracle to calculate shortest paths passing on T_{aft} (using shortest paths passing on T_{bef}) with the best performance concerning the oracle update time, output size and shortest path query time such that $|\Pi'(s, t|T_{aft})| \leqslant (1+\epsilon)|\Pi(s, t|T_{aft})|$ for any pairs of s and t in P, where $\Pi'(s, t|T_{aft})$ is the shortest path between s and t passing on T_{aft}.

4.1.3 Property

Property 1 describes an important property, called non-updated *TIN* shortest path intact property, for solving our problem.

Property 1 (Non-updated *TIN* Shortest Path Intact Property) *In Figure 4.2, given* T_{bef}, T_{aft} *and* $\Pi(u, v|T_{bef})$, *if two disks* $D(u, \frac{|\Pi(u,v|T_{bef})|}{2})$ *and* $D(v, \frac{|\Pi(u,v|T_{bef})|}{2})$ *do not intersect with* ΔF, *then* $\Pi(u, v|T_{aft})$ *is the same as* $\Pi(u, v|T_{bef})$.

Proof. We prove by contradiction. Suppose that two disks $D(u, \frac{|\Pi(u,v|T_{bef})|}{2})$ and $D(v, \frac{|\Pi(u,v|T_{bef})|}{2})$ do not intersect with ΔF, but $\Pi(u, v|T_{aft})$ is different from $\Pi(u, v|T_{bef})$, and we need to update $\Pi(u, v|T_{bef})$ to $\Pi(u, v|T_{aft})$ due to the smaller distance of $\Pi(u, v|T_{aft})$, i.e., $|\Pi(u, v|T_{aft})| < |\Pi(u, v|T_{bef})|$. This case will only happen when $\Pi(u, v|T_{aft})$ passes ΔF. In Figure 4.2, we let u_1 (resp. v_1) be the point on $\Pi(u, v|T_{aft})$ that the exact shortest distance $|\Pi(u, u_1|T_{aft})|$ (resp. $|\Pi(v, v_1|T_{aft})|$) on T_{aft} is the same as $|\frac{\Pi(u,v|T_{bef})}{2}|$. We let u_2 (resp. v_2) be the point on $\Pi(u, v|T_{aft})$ that u_2 (resp. v_2) is a point in ΔF and the exact shortest distance $|\Pi(u, u_2|T_{aft})|$ (resp. $|\Pi(v, v_2|T_{aft})|$) on T_{aft} is the minimum one. Clearly, u_2 (resp. v_2) is the intersection point between $\Pi(u, v|T_{aft})$ and ΔF, such that the exact shortest distance $|\Pi(u, u_2|T_{aft})|$ (resp. $|\Pi(v, v_2|T_{aft})|$) on T_{aft} is the minimum one. Note that a point is said to be in ΔF if this point is on a face in ΔF. We let o be the mid-point on $\Pi(u, v|T_{bef})$, clearly we have $|\Pi(u, o|T_{bef})| = |\Pi(o, v|T_{bef})| = |\frac{\Pi(u,v|T_{bef})}{2}|$. We also know that $|\Pi(u, u_1|T_{aft})| = |\Pi(u, o|T_{bef})| = |\Pi(v, v_1|T_{aft})| = |\Pi(v, o|T_{bef})| = |\frac{\Pi(u,v|T_{bef})}{2}|$. The light blue path is $\Pi(u, v|T_{bef})$ and the yellow path is $\Pi(u, v|T_{aft})$. Since the minimum distance from both u and v to the updated faces ΔF is no smaller than $|\frac{\Pi(u,v|T_{bef})}{2}|$, we know $|\Pi(u, o|T_{bef})| = |\Pi(u, u_1|T_{aft})| \leqslant |\Pi(u, u_2|T_{aft})|$ and $|\Pi(v, o|T_{bef})| = |\Pi(v, v_1|T_{aft})| \leqslant |\Pi(v, v_2|T_{aft})|$. Since $\Pi(u, v|T_{aft})$ passes ΔF, $|\Pi(u_2, v_2|T_{aft})| \geqslant 0$. Thus, we have $|\Pi(u, u_2|T_{aft})| + |\Pi(v, v_2|T_{aft})| + |\Pi(u_2, v_2|T_{aft})| = |\Pi(u, v|T_{aft})| \geqslant |\Pi(u, v|T_{bef})| = |\Pi(u, o|T_{bef})| + |\Pi(v, o|T_{bef})|$, which is a contradiction of our assumption $|\Pi(u, v|T_{aft})| < |\Pi(u, v|T_{bef})|$. Thus, we finish the proof. □

4.2 Methodology

4.2.1 Overview of *UP-Oracle*

We first illustrate *UP-Oracle* with an example. In Figure 4.1 (a) and Figure 4.3, we have an original *TIN* and a set of POIs. In Figures 4.1 (b), (c) and Figure 4.3, we construct *UP-Oracle* by calculating the exact shortest paths among these POIs. In Figures 4.1 (d) - (f) and Figure 4.3, we have an updated *TIN* and an error parameter ϵ, we compare the original and updated *TIN* to detect updated faces, update *UP-Oracle* by calculating the updated exact shortest paths, and generate a sub-graph using the exact shortest paths. In Figure 4.1 (g) and Figure 4.3, we answer the shortest path query between a pair of POIs using *UP-Oracle*. Next, we introduce the three components and three phases of *UP-Oracle*.

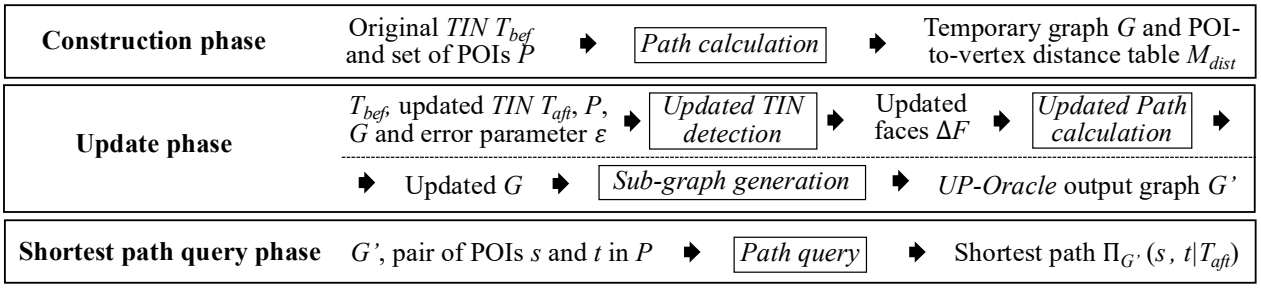


Figure 4.3. *UP-Oracle* framework overview description

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

(i) **The temporary graph *G*** is a complete graph that stores the pairwise P2P exact shortest paths. G contains a set of vertices G.V and edges G.E (where each POI in P is a vertex in G.V). Given a pair of POIs u and v, the exact shortest path $\Pi(u, v|T)$ passing on T is an edge $e(u, v|T)$ in G.E with a weight $|e(u, v|T)| = |\Pi(u, v|T)|$, where T can be T_{bef} or T_{aft}. Figure 4.1 (b) shows a G with 4 vertices and 6 edges. The light blue edge $e(a, c|T_{bef})$ in G denotes a path $\Pi(a, c|T_{bef})$.

(ii) **POI-to-vertex distance table M_{dist}** is a *hash table* [27] that stores the exact shortest distance between each POI in P and each vertex in V on T_{bef}, used for reducing the oracle update time of *UP-Oracle*. A POI u and a vertex v is stored as a key $\langle u, v\rangle$, and the distance between them $|\Pi(u, v|T_{bef})|$ is stored as a value. In Figures 4.1 (c), the exact shortest distance between POI a and vertex v_1 is 4.

(iii) **The *UP-Oracle* output graph G'** is a sub-graph of G used for answering pairwise P2P $(1+\epsilon)$-approximate shortest paths. G' contains a set of vertices $G'.V$ and edges $G'.E$. Given a pair of vertices u and v in $G'.V$, let $e'(u, v|T_{aft})$ be an edge with a weight $|e'(u, v|T_{aft})|$, and let $\Pi_{G'}(u, v|T_{aft})$ be the shortest path passing on G'. Figure 4.1 (f) shows a G' with 4 edges. The light blue edge $e'(a, c|T_{aft})$ in G' denotes a path $\Pi(a, c|T_{aft})$. The shortest path $\Pi_{G'}(a, b|T_{aft})$ consists of edges $e'(a, c|T_{aft})$ and $e'(c, b|T_{aft})$.

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

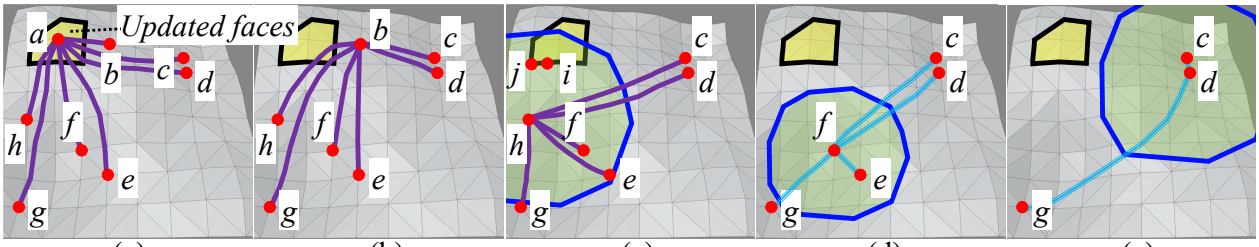
(i) **Construction phase**: Given T_{bef} and P, considering each POI in P as the source, we use algorithm *SSAD* [43, 71, 44, 73, 25, 75] to simultaneously: (1) calculate the exact shortest paths between this POI and other POIs passing on T_{bef}, and store these in G, and (2) calculate the exact shortest distance between this POI and all vertices on T_{bef}, and store these in M_{dist}. In Figures 4.1 (b) and (c), we first use algorithm *SSAD* with a as the source to calculate paths between a and $\{b, c, d\}$ (the light blue paths), and distances between a and all vertices. Next, we use b, c as sources and repeat this.

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

- *Updated TIN detection*: Given T_{bef} and T_{aft}, we compare the coordinates of their vertices to detect ΔF.
- *Exact shortest path update*: Given T_{aft}, P, G, M_{dist} and ΔF, we select some POIs in P as sources in algorithm *SSAD* to update the exact shortest paths passing on T_{aft} if Property 1 is not satisfied for paths connecting to these POIs, and we update G. In Figure 4.1 (e), we use a as the source to update paths between a and $\{b, c, d\}$ (the purple paths). Figure 4.4 shows more details. In Figures 4.4 (a) to (c), since a is on a face in ΔF, the path with b as the source passes on ΔF, and the blue disk $D(h, \frac{|\Pi(c,h|T_{bef})|}{2})$ with h as center intersects with ΔF, Property 1 is not satisfied for all possible paths connecting a, b, h. So, we use them as sources in algorithm *SSAD* for path updating on T_{aft}, and we update G. In Figures 4.4 (d) and (e), Property 1 is satisfied, and path update is not needed. We give more details in Chapters 4.2.2 and 4.2.3.

- *Sub-graph generation*: Given G and ϵ, we use algorithm *HGSpan* to efficiently generate G' for output size reduction, such that $|\Pi_{G'}(s,t|T_{aft})| \leqslant (1+\epsilon)|\Pi(s,t|T_{aft})|$ for any pair of POIs s and t in P. In Figure 4.1 (f), we obtain G' from G. We give more details in Chapters 4.2.2 and 4.2.3.

(iii) **Shortest path query phase**: Given G', and a pair of POIs s and t in P, we use Dijkstra's algorithm for finding the shortest path between s and t on G', i.e., $\Pi_{G'}(s,t|T_{aft})$. In Figure 4.1 (g), given a source a and a destination b, we calculate $\Pi_{G'}(a,b|T_{aft})$, see the green path.



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

Figure 4.4. Exact shortest path update step in *UP-Oracle*

4.2.2 Key Ideas of *UP-Oracle*'s Update Phase

1) Exact shortest path update step *UP-Oracle* has a short oracle update time due to our design in the *exact shortest path update* step of the update phase. Recall from Chapter 1.2.2 that the short oracle update time is mainly enabled by the *non-updated TIN shortest path intact* property in Property 1, and the stored pairwise P2P exact shortest paths passing on T_{bef}. We consider three additional issues and propose four techniques (one for each of the first two issues, and two for the third issue) to further reduce the oracle update time.

(i) **Which POI to select first for path updating before Property 1 is utilized - Optimal POI selection sequence:** In Figures 4.4 (a) to (c), (i) a is in ΔF, (ii) one of b's exact shortest paths, $\Pi(b,h|T_{bef})$, passes ΔF, and (iii) h is near ΔF. As Property 1 is not satisfied for the paths connecting a, b, h, we use a as the source in algorithm *SSAD* to update the paths passing on T_{aft} to other POIs simultaneously, and repeat this for b and h. In Figures 4.4 (d) and (e), Property 1 is satisfied, so we do not need to use algorithm *SSAD* with f and d

as sources. Here, we only need to use algorithm *SSAD* 3 times, and the optimal sequence is selecting the POIs: (i) on a face in ΔF, (ii) connecting to the path passing ΔF, and (iii) near ΔF (correspond to the sequence a, b, h). But, if we do not use this optimal sequence, e.g., we first update the paths with c, d, e, f, g as sources, then we still need to update the paths with a, b, h as sources. Here, we need to use algorithm *SSAD* 8 times. Note that the sequence only aims to identify the POI to select first as the source in algorithm *SSAD*, and we still update paths in parallel (as in the construction phase).

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

(iii) **How to efficiently determine whether Property 1 is satisfied - Efficient distance approximation:** In Figure 4.4 (c), given h, let i be the point belonging to ΔF which is closest to h, and let j be the vertex in ΔV that is closest to h. When determining whether Property 1 is satisfied, for the path between h and c, we determine whether the blue disk $D(h, r)$ intersects with ΔF (where $r = \frac{|\Pi(h,c|T_{bef})|}{2}$), by efficiently determining whether $r < |\Pi(h, j|T_{bef})| - l_{max}$ in $O(1)$ time (where $|\Pi(h, j|T_{bef})|$ is stored in M_{dist}). If so, $r < |\Pi(h, i|T_{aft})|$, i.e., Property 1 is satisfied, and we do not need to update the path, since we have the distance approximation $|\Pi(h, j|T_{bef})| - l_{max} \leqslant |\Pi(h, i|T_{aft})|$ from the triangle inequality. Otherwise, we update the path. But, if we do not use this efficient approximation, we need to calculate $|\Pi(h, i|T_{aft})|$ using algorithm *SSAD* in $O(N^2)$ time.

(iv) **How to efficiently determine whether Property 1 is satisfied - Efficient disk and updated face intersection check:** In Figure 4.4 (c), we sort the third type of POI in the optimal POI selection sequence from near to far based on their minimum distance to any vertex in ΔV on T_{bef} using M_{dist}. Thus, we get the ordering h, f, e, d, c, g. When determining whether Property 1 is satisfied, we just need to create one blue disk $D(h, r)$ (where $r = \frac{|\Pi(h,c|T_{bef})|}{2}$ is half of the longest distance of the paths between h and each POI in $\{c, d, e, f, g\}$), and determine whether it intersects with ΔF. If the disk with the largest radius and with the center closest to ΔF intersects with ΔF, Property 1 is not satisfied and there is no need to check other disks. Otherwise, Property 1 is always satisfied. In total,

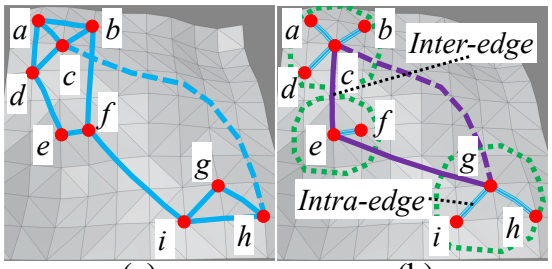
there are $O(n)$ POIs, we need to create $O(n)$ disks for efficient checking. But, if we do not use this efficient checking, we need to create ten disks, i.e., five disks $D(h, \frac{|\Pi(X,h|T_{bef})|}{2})$ and five disks $D(X, \frac{|\Pi(X,h|T_{bef})|}{2})$ for checking, where $X \in \{c, d, e, f, g\}$. In total, there are $O(n^2)$ paths, it needs to create $O(n^2)$ disks.

2) Sub-graph generation step *UP-Oracle* can efficiently reduce the output size due to our design in the *sub-graph generation* step (using algorithm *HGSpan*) of the update phase. Due to this, the output size of *UP-Oracle* is 44MB on a *TIN* with 0.5M faces and 500 POIs, but the value is 520MB for *SE-Oracle* and 416MB for *RC-TIN-Oracle*. Intuitively, given a complete graph and ϵ, the best-known $(1+\epsilon)$-sub-graph generation algorithm *GSpan* [21] is inefficient. But, we use a simpler structure to approximate the internal graph for faster processing and generate a $(1+\epsilon)$-sub-graph.

Concept: **The hierarchy graph *H*** is a graph that has a simpler structure than G', and it is used for efficiently generating G' using G. Let $Q_{G'}$ be a *group* of vertices in $G'.V$ with *group center* $v \in Q_{G'}$ and *radius* r. For every vertex $u \in Q_{G'}$, we have $|\Pi_{G'}(u, v|T_{aft})| \leqslant r$. We create a set of groups $Q^1_{G'}, Q^2_{G'}, \ldots, Q^k_{G'}$ such that every vertex in $G'.V$ belongs to at least one group, where k is the number of groups. H can form a set of *groups* by regarding several vertices in G' that are close to each other as one vertex. As a result, H is an approximation of G'. Similar to G', H contains a set of edges $H.E$. Given a group $Q^i_{G'}$, let an *intra-edge* be an edge between the group center of $Q^i_{G'}$ and a vertex in $Q^i_{G'}$, and let an *inter-edge* be an edge between two group centers. Given a pair of vertices u and v in $H.E$, let $e_H(u, v|T_{aft})$ be an (intra- or inter-) edge with a weight $|e_H(u, v|T_{aft})|$. Given a pair of group centers s and t, let $\Pi_H(s, t|T_{aft})$ be the shortest path between them in H that only consists of inter-edges. Figures 4.5 (a) and (b) show a G' and its corresponding H. There are three groups with centers c, e, g in H. The light blue paths are intra-edges and the purple paths are inter-edges. The shortest path $\Pi_H(c, g|T_{aft})$ consists of edges $e_H(c, e|T_{aft})$ and $e_H(e, g|T_{aft})$.

Method: We introduce algorithm *HGSpan* as follow.

(i) **Why algorithm *HGSpan* is efficient:** Given a complete graph G and ϵ, algorithm *HGSpan* sorts edges in G based on their length in ascending order, and we initialize sub-graph G' to be empty. For each sorted edge in $G.E$, e.g., $|e(c, h|T_{aft})|$ in Figure 4.5 (a), if



Remark: (a) *UP-Oracle* output graph G' and (b) hierarchy graph H of G'

Figure 4.5. Sub-graph generation step in *UP-Oracle*

$|\Pi_{G'}(c, h|T_{aft})| > (1+\epsilon)|e(c, h|T_{aft})|$, it is inserted into G', where $|\Pi_{G'}(c, h|T_{aft})|$ is approximated by the distance between c and h (which are group centers) on H, i.e., $|\Pi_H(c, g|T_{aft})|$ in Figure 4.5 (b). It is calculated using Dijkstra's algorithm in $O(1)$ time. But, if we do not use H for approximation, we need to use Dijkstra's algorithm on G' to calculate $|\Pi_{G'}(c, h|T_{aft})|$ in $O(n \log n)$ time, as in algorithm *GSpan* [21]. We repeat this for all edges in G, and return G' as the output.

(ii) **How to construct *H*:** To construct H, we sort the edges in G according to their length in ascending order. We then divide them into $\log n$ intervals, where each interval contains edges with lengths in $(\frac{2^{i-1}D}{n}, \frac{2^iD}{n}]$ for $i \in [1, \log n]$ and D is the longest length of edges in G. In Figure 4.5 (b), for the i-th iteration, we select c, e, g as group centers and create groups with $\delta\frac{2^iD}{n}$ as radius, where $\delta = \frac{\sqrt{\epsilon^2+36\epsilon+36}-(\epsilon+6)}{24} \in (0, \frac{1}{2})$ since $\epsilon \in (0, \infty)$. We insert intra-edges (light blue paths) into H between each group center and vertices in this group, and we insert inter-edges (solid purple paths) into H between every two group centers such that the distances between them on G' are at most $\frac{2^iD}{n} + 2\delta\frac{2^iD}{n}$. Then, we use H to approximate G'. For each sorted edge in G with length in the current length interval, e.g., $e(c, h|T_{aft})$ in Figure 4.5 (a), if $|\Pi_{G'}(c, h|T_{aft})| > (1+\epsilon)|e(c, h|T_{aft})|$, where $|\Pi_{G'}(c, h|T_{aft})|$ is approximated by $|\Pi_H(c, g|T_{aft})|$, we insert this edge (dashed light blue path) into G' and also insert an inter-edge between the group centers of c and h, i.e., $e_H(c, g|T_{aft})$ (dashed purple path), into H. Having processed all edges in the current length interval, for the $(i+1)$-st iteration, we repeat the above process to re-construct H, so that H is a valid approximation of G'. But, in algorithm *SGSpan* [29], if we force it to use a complete graph as input, the radius of each group in H becomes 0, and it degenerates to algorithm *GSpan*.

In algorithm *HGSpan*, we use a different process to construct H (i.e., we insert inter-edges into H at two different stages: one before processing edges in G, and another one during this process), and we set the radius of each group in H to exceed 0 (i.e., $\delta\frac{2^i D}{n} > 0$ since $\delta > 0$), to avoid the degeneration by sacrificing the output size.

4.2.3 Implementation Details of *UP-Oracle*'s Update Phase

1) Exact shortest path update step We give the implementation details of the *exact shortest path update* step of the update phase in *UP-Oracle*.

Notation: Before we provide the algorithm, we introduce some notation. Let $P_{rem} = \{p'_1, p'_2, \ldots, p'_{|P_{rem}|}\}$ be a set of remaining POIs in P on T_{aft} that we have not processed, where $|P_{rem}|$ is the size of P_{rem}. P_{rem} is initialized to be P. In each update iteration, when we have processed a POI, we remove it from P_{rem}. In Figures 4.4 (a) and (b), $P_{rem} = \{b, c, d, e, f, g, h\}$ and $P_{rem} = \{c, d, e, f, g, h\}$. Given a POI $u \in P_{rem}$, let $\Pi(u) = \{\Pi(u, v_1|T_{bef}), \Pi(u, v_2|T_{bef}), \ldots, \Pi(u, v_{|\Pi(u)|}|T_{bef})\}$ be a set of the exact shortest paths stored in G passing on T_{bef} with u as an endpoint and each $v_i \in P_{rem} \setminus \{u\}$ as the other endpoint, such that these paths have not been updated. $\Pi(u)$ is initialized to be all the exact shortest paths stored in G with u as an endpoint, where $|\Pi(u)|$ is the size of $\Pi(u)$. In Figures 4.4 (a) to (c), the purple paths denote $\Pi(a)$, $\Pi(b)$ and $\Pi(h)$.

Detail and example: We summarize the methods in Algorithms 4 and 5. Algorithm 4 is used in three places in Algorithm 5. The following is an example of them.

Algorithm 4 *OnePoiUpdate* (T_{aft}, G, u, P_{rem})

Input: an updated *TIN* T_{aft}, a temporary graph G, a POI u and a set of remaining POIs in P on T_{aft} that we have not processed P_{rem}
Output: an updated G and an updated P_{rem}

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

Algorithm 5 *Update* (T_{aft}, P, G, M_{dist}, ΔF)

Input: an updated *TIN* T_{aft}, a set of POIs P, a temporary graph G, a POI-to-vertex distance table M_{dist} and a set of updated faces ΔF
Output: updated G

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

(i) P_{rem} *and* $\Pi(u)$ *initialization*: Lines 1-3. In Figure 4.4 (a), we initialize $P_{rem} = \{a, b, c, d, e, f, g, h\}$, $\Pi(a) = \{\Pi(a, b|T_{bef}), \Pi(a, c|T_{bef}), \ldots, \Pi(a, h|T_{bef})\}$, $\Pi(b) = \{\Pi(b, a|T_{bef}), \Pi(b, c|T_{bef}), \ldots, \Pi(b, h|T_{bef})\}$, $\ldots$, and $\Pi(h) = \{\Pi(h, a|T_{bef}), \Pi(h, b|T_{bef}), \ldots, \Pi(h, g|T_{bef})\}$. Next, we use the *optimal POI selection sequence* for path updating.

(ii) *Path update for POI in* ΔF: Lines 4-6. In Figure 4.4 (a), a is on a face in ΔF, so Property 1 is not satisfied. We first use *OnePoiUpdate* (T_{aft}, G, a, P_{rem}) to update the purple paths passing on T_{aft}, and update G. Then, we remove $\Pi(a, X|T_{bef})$ in $\Pi(X)$ for each $X \in P_{rem}$, so $\Pi(a)$ becomes empty, $\Pi(b) = \{\Pi(b, c|T_{bef}), \Pi(b, d|T_{bef}), \ldots, \Pi(b, h|T_{bef})\}$, $\ldots$, and $\Pi(h) = \{\Pi(h, b|T_{bef}), \Pi(h, c|T_{bef}), \ldots, \Pi(h, g|T_{bef})\}$. Finally, we remove a from P_{rem} to get $P_{rem} = \{b, c, d, e, f, g, h\}$.

(iii) *Path update for POI connecting to the path passing* ΔF: Lines 7-9. In Figure 4.4 (b), b

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

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

- In Figure 4.4 (c), the sorted POIs are h, f, e, d, c, g. We process h, and the path with the longest distance is $\Pi(c, h|T_{bef})$. Since Property 1 with the *minimum disk radius* is not satisfied, i.e., only one blue disk intersects with ΔF (determined by checking $|\frac{\Pi(h,c|T_{bef})}{2}| > |\Pi(h, j|T_{bef})| - l_{max}$ according to *efficient distance approximation* and *efficient disk and updated face intersection check*), we first use *OnePoiUpdate* (T_{aft}, G, h, P_{rem}) to update the purple paths passing on T_{aft}, and update G. Then, we remove $\Pi(h, X|T_{bef})$ in $\Pi(X)$ for each $X \in P_{rem}$, so $\Pi(a)$, $\Pi(b)$ and $\Pi(h)$ become empty, $\Pi(c) = \{\Pi(c, d|T_{bef}), \Pi(c, e|T_{bef}), \ldots, \Pi(c, g|T_{bef})\}$, $\ldots$, and $\Pi(g) = \{\Pi(g, c|T_{bef}), \Pi(g, d|T_{bef}), \ldots, \Pi(g, f|T_{bef})\}$. Finally, we remove h from P_{rem} to get $P_{rem} = \{c, d, e, f, g\}$.

- In Figure 4.4 (d), the sorted POIs are f, e, d, c, g. We process f, and the path with the longest distance is $\Pi(c, f|T_{bef})$. Since Property 1 is satisfied, i.e., the blue disk does not intersect with ΔF (determined by checking $|\frac{\Pi(f,c|T_{bef})}{2}| < \min_{\forall v \in \Delta V} |\Pi(f, v|T_{bef})| - l_{max}$), we do not need to update the paths connect to f. We remove f from P_{rem} to get $P_{rem} = \{c, d, e, g\}$. Then, the sorted POIs are e, d, c, g, and we process e similar to above.

- In Figure 4.4 (e), the case is also similar.

Lemma: We give three important lemmas as follows.

(i) **Necessity of storing the pairwise P2P exact shortest paths (i.e., *G*) on T_{bef}:** Let $U(A) = \frac{n'}{n}$ be the *Update ratio* of an oracle A, where n' is the number of POIs in P that need to be used as a source in algorithm *SSAD* (for path updating on T_{aft}). In Figures 4.4

(a) to (c), we use algorithm *SSAD* with a, b, h as sources to update shortest paths passing on T_{aft} for *UP-Oracle* and *SE-UP-Oracle*. In Figure 4.4 (d), *UP-Oracle* (resp. *SE-UP-Oracle*) calculates an *exact* (resp. *approximate*) path between c and f passing on T_{bef}. The radius of the disk with f as center is smaller (resp. larger), so the disk does not (resp. may) intersect with ΔF, and *UP-Oracle* does not need to (resp. *SE-UP-Oracle* may need to) use algorithm *SSAD* with f as source to update shortest paths passing on T_{aft}. The case also happens for the paths between c and e. In Figure 4.4 (e), the case also happens for the path between g and each POI in $\{c, d\}$. Thus, for *UP-Oracle* (resp. *SE-UP-Oracle*), we perform algorithm *SSAD* with three POIs a, b, h (resp. seven POIs a, b, c, d, e, f, g) as sources for path updating on T_{aft}. As there is a total of eight POIs, $U(\textit{UP-Oracle}) = \frac{3}{8}$ (resp. $U(\textit{SE-UP-Oracle}) = \frac{7}{8}$). The oracle update time of *SE-UP-Oracle* is 2.4 times larger than that of *UP-Oracle*. *RC-TIN-UP-Oracle* is similar to *SE-UP-Oracle*. Given an oracle A, a higher $U(A)$ means that the oracle update time of A is larger. Lemma 4.2.1 shows the necessity of storing G.

Lemma 4.2.1 *Given T_{bef}, T_{aft}, P and an oracle A that does not store the pairwise P2P exact shortest paths passing on T_{bef}, $U(\textit{UP-Oracle}) \leqslant U(A)$.*

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

(ii) **Correctness of the efficient distance approximation:** In the distance approximation, we have "$|\Pi(h, j|T_{bef})| - l_{max} \leqslant |\Pi(h, i|T_{aft})|$ due to the triangle inequality" in Figure 4.4 (c). Lemma 4.2.2 shows the correctness of this inequality, implying that the correctness of the efficient distance approximation. In Lemma 4.2.2, u can be h, any point on a face in ΔF can be i, and v can be j in Figure 4.4 (c).

Lemma 4.2.2 *The minimum distance from a POI u to any point on a face in ΔF on T_{aft} is at least $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{bef})| - l_{max}$.*

Proof. According to [72, 71], the exact shortest distance on a *TIN* follows triangle inequality. Given an edge e which belongs to a face in ΔF with two endpoints u_1 and u_2,

suppose that the exact shortest path from u to ΔF intersects with any point on e for the first time. There are two cases:

- If the intersection point is one endpoint of e (e.g., u_1), since u_1 is a vertex of a face in ΔF, so the minimum distance from u to ΔF in non-updated faces of T_{aft} is the same as $|\Pi(u, u_1|T_{bef})|$. Since $|\Pi(u, u_1|T_{bef})|$ is at least $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{bef})|$, we obtain that the minimum distance from u to ΔF in non-updated faces of T_{aft} is at least $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{bef})|$.
- If the intersection point is on e, we denote this intersection point as u_3. Suppose that $|\Pi(u, u_1|T_{bef})| - |u_1u_3| < |\Pi(u, u_2|T_{bef})| - |u_2u_3|$, where $|u_1u_3|$ (resp. $|u_2u_3|$) is the segment's length between u_1 and u_3 (resp. between u_2 and u_3) on edge e. According to triangle inequality, the minimum distance from u to ΔF in non-updated faces of T_{aft} is at least $|\Pi(u, u_1|T_{aft})| - |u_1u_3|$. Since we only care about the minimum distance, $|\Pi(u, u_1|T_{aft})|$ is the same as $|\Pi(u, u_1|T_{bef})|$. Since the $|\Pi(u, u_1|T_{bef})|$ is at least $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{bef})|$ and $|u_1u_3|$ is at most l_{max}, we obtain that the minimum distance from u to ΔF in non-updated faces of T_{aft} is at least $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{bef})| - l_{max}$.

□

(iii) **Correctness of the efficient disk and updated face intersection check:** In the intersection check, we just need to create one blue disk $D(h, r)$ (where $r = \frac{|\Pi(h,c|T_{bef})|}{2}$ is half of the longest distance of the paths between h and each POI in $\{c, d, e, f, g\}$), and determine whether it intersects with ΔF in Figure 4.4 (c), instead of creating ten disks. Lemma 4.2.3 shows the correctness of this check. The disk in Lemma 4.2.3 can be $D(h, \frac{|\Pi(h,c|T_{bef})|}{2})$ in Figure 4.4 (c).

Lemma 4.2.3 *If the disk, with u as center, and half of the longest distance among all non-updated paths adjacent to u as radius, intersects with ΔF, Property 1 is not satisfied, and we need to use algorithm SSAD to update all non-updated paths adjacent to u. Otherwise, Property 1 is satisfied, and there is no need to update shortest paths adjacent to u.*

Proof. If the disk with the largest radius intersects with ΔF, we just need to update the paths and there is no need to check other disks. In Figure 4.4 (c), the sorted POIs are h, f, e, d, c, g. We create one disk $D(h, \frac{|\Pi(c,h|T_{bef})|}{2})$, since it *intersects* with ΔF, we use algorithm *SSAD* to update all shortest paths adjacent to h that have not been updated. We do not need to create ten disks, i.e., five disks $D(h, \frac{|\Pi(X,h|T_{bef})|}{2})$ and five disks $D(X, \frac{|\Pi(X,h|T_{bef})|}{2})$, where $X = \{c, d, e, f, g, h\}$. Since the disk $D(h, \frac{|\Pi(c,h|T_{bef})|}{2})$ with the largest radius already intersects with ΔF, so there is no need to check other disks.

If the disk with the largest radius and with the center closest to ΔF does not intersect with ΔF, then other disks cannot intersect with ΔF, so there is no need to update the paths. In Figure 4.4 (d), the sorted POIs are f, e, d, c, g. We create one disk $D(f, \frac{|\Pi(c,f|T_{bef})|}{2})$, since it *does not intersect* with ΔF, there is no need to update shortest paths adjacent to f. We do not need to create eight disks, i.e., four disks $D(f, \frac{|\Pi(X,f|T_{bef})|}{2})$ and four disks $D(X, \frac{|\Pi(X,f|T_{bef})|}{2})$, where $X = \{c, d, e, f, g\}$. Since the disk $D(f, \frac{|\Pi(c,f|T_{bef})|}{2})$ with the largest radius does not intersect with ΔF, so the disks $D(f, \frac{|\Pi(X,f|T_{bef})|}{2})$ with smaller radius and the disks $D(X, \frac{|\Pi(X,f|T_{bef})|}{2})$ with centers further away from ΔF compared with f cannot intersect with ΔF. Recall that given a POI u, we use $\min_{\forall v \in \Delta V} |\Pi(u, v|T_{bef})| - l_{max}$ as the lower bound of the minimum distance from u to any point in ΔF on T_{aft}. If $D(f, \frac{|\Pi(c,f|T_{bef})|}{2})$ does not intersect with ΔF, then $\min_{\forall v \in \Delta V} |\Pi(c, v|T_{bef})| - l_{max} > \frac{|\Pi(c,f|T_{bef})|}{2}$, and $\min_{\forall v \in \Delta V} |\Pi(X, v|T_{bef})| - l_{max} > \frac{|\Pi(c,f|T_{bef})|}{2}$ (since we sort X from near to far based on their minimum distance to any vertex in ΔV on T_{bef}), and then $\min_{\forall v \in \Delta V} |\Pi(X, v|T_{bef})| - l_{max} > \frac{|\Pi(X,f|T_{bef})|}{2}$ (since $|\Pi(c, f|T_{bef})| \geqslant |\Pi(X, f|T_{bef})|$), i.e., the disks $D(X, \frac{|\Pi(X,f|T_{bef})|}{2})$ cannot intersect with ΔF, where $X = \{c, d, e, f, g\}$. □

2) Sub-graph generation step We give the implementation details of the *sub-graph generation* step of the update phase in *UP-Oracle*.

Detail and example: Algorithm 6 details *HGSpan*. The following is an example of it.

(i) *Edge sorting, interval splitting and* G' *initialization*: Lines 2-6. We insert edges of G with lengths in $I_0 = (0, \frac{D}{N}]$ into G'.

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

Algorithm 6 *HGSpan* (G, ϵ)

Input: a temporary graph G and an error parameter ϵ
Output: a *UP-Oracle* output graph G' (a sub-graph of G)

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

- Lines 9-15 (*group construction and* H*'s intra-edge insertion*): When $i = 1$, there is a limited number of edges in G', i.e., most pairs of vertices in $G'.V$ do not have an edge connecting them. It is likely that every vertex in $G'.V$ forms a group itself in H, and there are no intra-edges in H since each group only contains one vertex.

- Lines 16-19 (H*'s first type inter-edge insertion*): Similarly, it is likely that there are no

inter-edges in H.

- Lines 20-25 (G'*'s edge insertion and* H*'s second type inter-edge insertion*): For each edge $e(u, v|T_{aft})$ in G with a weight in $I_1 = (\frac{D}{N}, \frac{2D}{N}]$, it is likely that the group center of u (resp. v) in H is u (resp. v) itself, i.e., $w = u$ and $x = v$. Since there is a limited number of edges in G', line 23 is likely true, and we insert $e(u, v|T_{aft})$ into G' and $e_H(w, x|T_{aft})$ into H.

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

- Lines 9-15: Suppose that we have G' as shown in Figure 4.5 (a). Based on $G.V$, we create three groups with centers c, e, g in H, see Figure 4.5 (b). We insert intra-edges $e_H(a, c|T_{aft}), \ldots, e_H(g, i|T_{aft})$ (light blue paths) into H in Figure 4.5 (b).

- Lines 16-19: We insert inter-edges $e_H(c, e|T_{aft})$ and $e_H(e, g|T_{aft})$ (solid purple paths) into H, see Figure 4.5 (b).

- Lines 20-25: Suppose that we need to examine edge $e(c, h|T_{aft})$ in G in Figure 4.5 (a) with a weight in $I_i = (\frac{2^{i-1}D}{n}, \frac{2^iD}{n}]$, and the group center of c (resp. h) in H is c (resp. g). Then, we check whether $|\Pi_H(c, g|T_{aft})| > (1+\epsilon)|e(c, h|T_{aft})|$. If so, we insert edge $e(c, h|T_{aft})$ (dashed light blue path) into G', and insert inter-edge $e_H(c, g|T_{aft})$ (dashed purple path) with a weight $|e(c, g|T_{aft})| + |e_H(g, h|T_{aft})|$ into H. Next, we repeat this by starting the $(i+1)$-st iteration and clear H. This way, we construct G'.

Lemma: Lemma 4.2.4 analyzes algorithm *HGSpan*.

Lemma 4.2.4 *The running time of HGSpan is* $O(n \log^2 n)$. *The output of HGSpan, i.e.,* G', *satisfies* $|\Pi_{G'}(u, v|T_{aft})| \leqslant (1+\epsilon)|\Pi(u, v|T_{aft})|$ *for any pairs of vertices* u *and* v *in* $G'.V$.

Proof. Firstly, we prove the *running time* of algorithm *HGSpan*.

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

- In the G''s construction step, for each edge interval, it needs $O(n \log n + n) = O(n \log n)$ time (shown as follows). Since there are total $\log n$ intervals, it needs $O(n \log^2 n)$ time.

  - In the group construction and H's intra-edge insertion step, it needs $O(n \log n)$ time. Since according to Lemma 6 in [29], we know that a vertex in H belongs to at most $O(1)$ groups (i.e., there are at most $O(1)$ group centers in H), so we just need to run Dijkstra's algorithm (in $O(n \log n)$ time) on G' for $O(1)$ times to calculate intra-edges for H.
  - In the H's first type inter-edge insertion step, it needs $O(n \log n)$ time. Since there are at most $O(1)$ group centers in H, so we just need to run Dijkstra's algorithm (in $O(n \log n)$ time) on G' for $O(1)$ times to calculate inter-edges for H.
  - In the G''s edge insertion and H's second type inter-edge insertion step, it needs $O(n)$ time. According to [29], there are $O(n)$ edges in each interval. Since there are at most $O(1)$ group centers in H, so answering the shortest path query using Dijkstra's algorithm on H needs $O(1)$ time. So, in order to examine $O(n)$ edges, this step needs Dijkstra's algorithm (in $O(1)$ time) on H for $O(n)$ times, and the total running time is $O(n)$.

In general, the running time for algorithm *HGSpan* is $O(n) + O(n \log^2 n) = O(n \log^2 n)$.

Secondly, we prove the *error bound* of algorithm *HGSpan*. According to algorithm *FAST-GREEDY* [29], by setting $t = t' = \epsilon + 1$, we can adapt it so that it takes a $\frac{t}{\sqrt{tt'}}$-sub-graph as input and generates a t-sub-graph for our algorithm *HGSpan*. According to Lemma 8 in [29], by setting $\delta = \frac{\sqrt{tt'+34\sqrt{tt'}+1}-(\sqrt{tt'}+5)}{24}$, it can guarantee that H is a valid approximation of G'. By setting $t = t' = \epsilon + 1$ in algorithm *HGSpan*, we have $\delta = \frac{\sqrt{\epsilon^2+36\epsilon+36}-(\epsilon+6)}{24}$. Since we also set δ to be $\frac{\sqrt{\epsilon^2+36\epsilon+36}-(\epsilon+6)}{24}$, we know that in algorithm *HGSpan*, when processing any group of edges $G.E^i$, H is always a valid approximation of G'. Note that there is no need to consider the right formula in $\delta = \min\{\frac{1}{2}(\frac{\sqrt{t}-\sqrt{t'}}{\sqrt{t}+3\sqrt{t'}}), \frac{\sqrt{tt'+34\sqrt{tt'}+1}-(\sqrt{tt'}+5)}{24}\}$ of algorithm *FAST-GREEDY* [29]. If we force the input to be a complete graph, we set $t = t'$, and δ is 0, i.e., it degenerates to

algorithm *GSpan*. This is because the right formula $\delta = \frac{1}{2}(\frac{\sqrt{t}-\sqrt{t'}}{\sqrt{t}+3\sqrt{t'}})$ controls the path with the second longest length in G' in Lemma 9 of [29], and it has no relationship with algorithm *GSpan*. Thus, in the G''s edge insertion and H's second type inter-edge insertion step of algorithm *HGSpan*, for each edge $e(u, v|T_{aft}) \in G.E^i$ between two vertices u and v, when we need to check whether $|\Pi_H(w, x|T_{aft})| > (1+\epsilon)|e(u, v|T_{aft})|$, where $\Pi_H(w, x|T_{aft})$ is the shortest path of group centers calculated using Dijkstra's algorithm on H, w and x are two group centers, such that, u is in w's group, and v is in x's group, $\Pi_H(w, x|T_{aft})$ is a valid approximation of $\Pi_{G'}(u, v|T_{aft})$. In other words, we are actually checking whether $|\Pi_{G'}(u, v|T_{aft})| > (1+\epsilon)|e(u, v|T_{aft})|$ or not. Consider any edge $e(u, v|T_{aft}) \in G.E$ between two vertices u and v which is not inserted into G' by algorithm *HGSpan*. Since $e(u, v|T_{aft})$ is discarded, it implies that $|\Pi_{G'}(u, v|T_{aft})| \leqslant (1+\epsilon)|e(u, v|T_{aft})|$. Since $|e(u, v|T_{aft})| = |\Pi(u, v|T_{aft})|$, so on the output graph of algorithm *HGSpan*, i.e., G', we always have $|\Pi_{G'}(u, v|T_{aft})| \leqslant (1+\epsilon)|\Pi(u, v|T_{aft})|$ for any pairs of vertices u and v in $G'.V$. We finish the proof. □

4.2.4 Handling Subsequent Changes

So far, we have handled a single change. After one change, we have the updated G and the sub-graph G'. There is no old G, since we update G partially by using the new paths passing on T_{bef} to replace the original paths passing on T_{aft}. We keep G in the hard disk and use G' for shortest paths queries. To adapt *UP-Oracle* to handle subsequent changes, we also update M_{dist} simultaneously when using algorithm *SSAD* for path updating in the *exact shortest path update* step of the update phase. Then, if subsequent changes occur, we update G and M_{dist}, and generate G' to support querying.

4.2.5 Adaptation to Multi-layer Structure (*UP-Oracle-MuLa*)

We can maintain a multi-layer structure. Long-range queries can utilize the approximate results calculated using G' (a sub-graph), as G (a complete graph) is excessively large. Short-range queries can utilize exact results with higher accuracy. In the landslide and earthquake (resp. marsquake) example, popular viewing platforms (resp. Mars rover

frequent work regions) can be regarded as short-range query regions, and we use the exact results for faster evacuation and escape, respectively. Depending on different areas of these regions, we can select different *Level-Of-Details* (*LODs*) of short-range query regions for customized querying.

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

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

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

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

Note that when $LOD = i > 1$, H'_{i-1} and H'_i have the same group centers, but the group radius of H'_i is larger than that of H'_{i-1}, so if a pair of POIs belong to the same group in H'_{i-1} (such that their corresponding exact path is stored in one of $M_1, M_2, \ldots, M_{i-1}$), they

also belong to the same group in H'_i, and we do not need to store this exact path again in M_i. Thus, the sum of exact paths stored in M_{LOD} does not exceed the total number of exact paths in G.

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

4.2.6 Adaptation to the AR2AR Query (*UP-Oracle-AR2AR*)

We can adapt *UP-Oracle* to be *UP-Oracle-AR2AR* for the AR2AR query. We first place Steiner points on T_{bef} using the method in study [35], and then use them as input (not the POIs) to construct *UP-Oracle-AR2AR* (as of *UP-Oracle*). When T_{bef} changes to T_{aft}, the positions of Steiner points (based on T_{aft}) also change, we update *UP-Oracle-AR2AR* using these Steiner points accordingly. For the shortest path query phase, given arbitrary point s (resp. t) on face f_s (resp. f_t), we let $\mathcal{S}(s)$ (resp. $\mathcal{S}(t)$) be a set of Steiner points on f_s (resp. f_t) and the adjacent faces [35]. Then, we return $\Pi_{G'}(s, t|T_{aft})$ in *UP-Oracle-AR2AR* (which has the same definition in *UP-Oracle*) by concatenating $\Pi(s, p|T_{aft})$, $\Pi_{G'}(p, q|T_{aft})$, and $\Pi(q, t|T_{aft})$ such that $|\Pi_{G'}(s, t|T_{aft})| = \min_{p\in\mathcal{S}(s), q\in\mathcal{S}(t)}[|\Pi(s, p|T_{aft})| + |\Pi_{G'}(p, q|T_{aft})| + |\Pi(q, t|T_{aft})|]$, where $|\Pi(s, p|T_{aft})|$ and $|\Pi(q, t|T_{aft})|$ can be calculated in $O(1)$ time using algorithm *SSAD* and $|\Pi_{G'}(p, q|T_{aft})|$ is distance of the path between p and q returned by *UP-Oracle-AR2AR*.

4.2.7 Theoretical Analysis

Theorem 4.2.1 analyzes *UP-Oracle* and its two adaptations.

Theorem 4.2.1 *The oracle construction time, oracle update time, output size and shortest path query time of (1) UP-Oracle and (2) UP-Oracle-MuLa are both* $O(nN^2)$, $O(N^2 + n\log^2 n)$, $O(n)$ *and* $O(\log n)$, *and (3) UP-Oracle-AR2AR are* $O(\frac{N^3}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon})$, $O(N^2 + \frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log^2(\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon})$, $O(\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon})$ *and* $O(\log(\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$, *respectively. (1)*

UP-Oracle, (2) UP-Oracle-MuLa, and (3) UP-Oracle-AR2AR satisfy $|\Pi_{G'}(s,t|T_{aft})| \leqslant (1+\epsilon)|\Pi(s,t|T_{aft})|$ *for any pairs of (1 & 2) POIs* s *and* t *in* P*, and (3) points* s *and* t *on* T_{aft}*, respectively.*

Proof. We give the proof for *UP-Oracle* as follows.

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

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

- In the updated *TIN* detection step, it needs $O(N)$ time. Since we just need to iterate each face in T_{aft} and T_{bef}. Since the number of faces in T_{aft} and T_{bef} is $O(N)$, it needs $O(N)$ time.
- In the Exact shortest path update step, it needs $O(N^2)$ time. Since we just need to update a constant number of POIs (which is shown by our experiments) using algorithm *SSAD* for calculating the exact shortest path from this POI to other POI passing on T_{aft}, and each algorithm *SSAD* needs $O(N^2)$ time, so it needs $O(N^2)$ time in total.
- In the sub-graph generation step, it needs $O(n\log^2 n)$ time. Since this step is using algorithm *HGSpan*, and algorithm *HGSpan* runs in $O(n\log^2 n)$ time as stated in Lemma 4.2.4.

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

Thirdly, we prove the *output size* of *UP-Oracle*. According to [29], we know that the output graph of algorithm *HGSpan*, i.e., G', has $O(n)$ edges. So, the output size of *UP-Oracle* is $O(n)$.

Fourthly, we prove the *shortest path query time* of *UP-Oracle*. Since we need to perform Dijkstra's algorithm on G', and in our experiments, G' has $O(1)$ edges and n vertices, so using a Fibonacci heap in Dijkstra's algorithm, the shortest path query time of *UP-Oracle* is $O(\log n)$.

Fifthly, we prove the *error bound* of *UP-Oracle*. The error bound of *UP-Oracle* is due to the error bound of algorithm *HGSpan*. As stated in Lemma 4.2.4, on the output graph of algorithm *HGSpan*, i.e., G', we always have $|\Pi_{G'}(u, v|T_{aft})| \leqslant (1+\epsilon)|\Pi(u, v|T_{aft})|$ for any pairs of vertices u and v in $G'.V$. Thus, we have the error bound of *UP-Oracle*, i.e., *UP-Oracle* satisfies $|\Pi_{G'}(u, v|T_{aft})| \leqslant (1+\epsilon)|\Pi(u, v|T_{aft})|$ for any pairs of POIs u and v in P.

We give the proof for *UP-Oracle-MuLa* as follows.

Firstly, the oracle construction time of *UP-Oracle-MuLa* is the same as *UP-Oracle*.

Secondly, we prove the *oracle update time* of *UP-Oracle-MuLa*. Apart from the oracle update time $O(N^2 + n\log^2 n)$ of *UP-Oracle*, we need $O(n\log n)$ time in the multi-layer structure generation step. Since to construct a temporary hierarchy graph, there are at most $O(1)$ group centers in this temporary hierarchy graph according to Lemma 6 in [29], we just need to run Dijkstra's algorithm (in $O(n\log n)$ time) on G for $O(1)$ times in order to calculate intra-edges and inter-edges for this temporary hierarchy graph. Since there are total $O(1)$ temporary hierarchy graphs, this step needs $O(n\log n)$ time. In general, the oracle update time of *UP-Oracle-MuLa* is $O(N^2 + n\log^2 n + nlogn) = O(N^2 + n\log^2 n)$.

Thirdly, we prove the *oracle size* of *UP-Oracle-MuLa*. The *output size* includes the output graph size $O(n)$ of algorithm *HGSpan*, and the size of M_{LOD} $O(n)$ (since not all the exact paths are stored in M_{LOD}). In general, the oracle size of *UP-Oracle-MuLa* is $O(n + n) = O(n)$.

Fourthly, we prove the *shortest path query time* and *error bound*. The experimental *shortest path query time* and *error bound* are better than *UP-Oracle* since *UP-Oracle-MuLa* stores some exact paths in M_{LOD}, but the theoretical time and error are the same as *UP-Oracle*.

We give the proof for *UP-Oracle-A2A* as follows. For the *oracle construction time*, *oracle update time*, *output size* and *shortest path query time*, since we place total $\frac{N}{\sin\theta\sqrt{\epsilon}}\log\frac{1}{\epsilon}$ Steiner points [35] on the faces of the *TIN*, so we use this value to substitute n in the oracle construction time, oracle update time, output size and shortest path query time of *UP-Oracle* to obtain the results for *UP-Oracle-A2A*. For the *error bound*, we first know that $|\Pi_{G'}(p, q|T_{aft})| \leqslant (1+\epsilon)|\Pi(p, q|T_{aft})|$ for any two Steiner points p and q due to the error bound of *UP-Oracle*. According to study [35], if $|\Pi_{G'}(p, q|T_{aft})| \leqslant (1+\epsilon)|\Pi(p, q|T_{aft})|$, then $|\Pi_{G'}(s, t|T_{aft})| \leqslant (1+\epsilon)|\Pi(s, t|T_{aft})|$ for two arbitrary points s and t.

In general, we finish the proof. □

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

4.3 Empirical Studies

4.3.1 Experimental Setup

We performed experiments on a Linux machine with 2.20 GHz CPU and 512GB memory. We implemented algorithms in C++. Our experimental setup generally follows the setups in the literature [45, 46, 51, 71, 72].

1) Datasets We conducted our experiments on 30 ($= 5 \times 6$) real before and after earthquake *TIN* datasets listed in Table 4.2 with 0.5M faces. We obtained the earthquake 3D satellite models with a 5km $\times$ 5km region from Google Earth with a resolution of 10m [64, 71, 72, 79, 82], and then we use Blender [1] to generate the *TIN*. To study the scalability, we follow an existing multi-resolution *TIN* dataset generation procedure [72, 79, 82] to obtain different resolutions of these datasets with 1M, 1.5M, 2M, 2.5M faces. This procedure appears in [85]. We extracted 500 POIs using OpenStreetMap [72, 79, 82].

Table 4.2. *TIN* datasets in the study of shortest path queries on updated *TIN*s

Name	**Magnitude**	**Date**
Tohoky, Japan (*TJ*) [15]	9.0	Mar 11, 2011
Sichuan, China (*SC*) [58]	8.0	May 12, 2008
Gujarat, India (*GI*) [14]	7.6	Jan 26, 2001
Alaska, USA (*AU*) [13]	7.1	Nov 30, 2018
Leogane, Haiti (*LH*) [57]	7.0	Jan 12, 2010
Valais, Switzerland (*VS*) [18]	4.1	Oct 24, 2016

Remark: Real earthquake *TIN* datasets.

2) Algorithms We included (i) the best-known exact on-the-fly algorithm *WAV-Fly-Algo* [25, 73], (ii) the best-known approximate on-the-fly algorithm *ESP-Fly-Algo* [45, 79], (iii) the best-known oracle *SE-Oracle* [71, 72] for the P2P query, (iv) its adaptation *SE-UP-Oracle*, (v) the best-known oracle *EAR-Oracle* [43] for the AR2AR query, (vi) its adaptation

EAR-UP-Oracle, (vii) the adapted oracle from point clouds to *TIN*s *RC-TIN-UP-Oracle* [82] and (viii) its adaptation *EAR-UP-Oracle* as baselines. In Table 4.3, we compared these algorithms with *UP-Oracle*. The comparisons of all algorithms can be found in [85]. Since the adapted *UP-Oracle* for subsequent changes has the same update phase as *UP-Oracle*, they have the same complexity and we omit the former oracle. For on-the-fly algorithms, we cannot run them in parallel (i.e., using a distributed implementation), since we only execute them once for one source. For oracles, we can run them in parallel for faster processing, since we execute them multiple times for different sources. But, for the comparison fairness for on-the-fly algorithms, we do not run oracles in parallel.

Table 4.3. Comparison of algorithms on an updated *TIN* regarding *UP-Oracle*

Algorithm	Oracle construction time		Oracle update time		Output size		Shortest path query time	
Oracle-based algorithm								
SE-Oracle [71, 72]	$O(\frac{nN^2}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nN^2}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Large	$O(h^2)$	Small
SE-UP-Oracle [71, 72]	$O(\frac{nN^2}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\mu_1 N^2 + n\log^2 n)$	Large	$O(n)$	Small	$O(\log n)$	Small
EAR-Oracle [43]	$O(\lambda\xi(mN)^2 + \frac{N^3}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\lambda\xi(mN)^2 + \frac{N^3}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi\log(\lambda\xi))$	Medium
EAR-UP-Oracle [43]	$O(\lambda\xi(mN)^2 + \frac{N^3}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\mu_2 N^2 + n\log^2 n)$	Large	$O(n)$	Small	$O(\log n)$	Small
RC-TIN-Oracle [82]	$O(\frac{nN^2}{\epsilon} + n\log n)$	Small	$O(\frac{nN^2}{\epsilon} + n\log n)$	Large	$O(\frac{nN}{\epsilon})$	Medium	$O(1)$	Small
RC-TIN-UP-Oracle [82]	$O(\frac{nN^2}{\epsilon} + n\log n)$	Small	$O(\mu_3 N^2 + n\log^2 n)$	Large	$O(n)$	Small	$O(\log n)$	Small
***UP-Oracle* (ours)**	$O(nN^2)$	**Small**	$O(N^2 + n\log^2 n)$	**Small**	$O(n)$	**Small**	$O(\log n)$	**Small**
On-the-fly algorithm								
WAV-Fly-Algo [25, 73]	-	N/A	-	N/A	-	N/A	$O(N^2)$	Large
ESP-Fly-Algo [45, 79]	-	N/A	-	N/A	-	N/A	$O(\gamma\log\gamma)$	Large

Remark: $n << N$, h is the compressed partition tree's height, β is the largest capacity dimension [71, 72], λ is the number of highway nodes in one square, ξ is the number of boxes under square root, m is the number of Steiner points on each face, μ_1, μ_2, μ_3, μ_4, μ_5 and μ_6 are data-dependent variables, $\mu_1 \in [5, 20]$, $\mu_2 \in [12, 45]$, $\mu_3 \in [30, 65]$, $\mu_4 \in [50, 200]$, $\mu_5 \in [300, 650]$ and $\mu_6 \in [5, 10]$ in our experiments, θ is the minimum angle of the face in the *TIN*, l_{max} (resp. l_{min}) is longest (resp. shortest) length of edges in *TIN*, and $\gamma = \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}$.

3) Query generation We randomly chose pairs of POIs in P for the P2P query, or arbitrary points on T_{aft} for the AR2AR query, and we reported the average, minimum and maximum results of 100 queries.

4) Factors and measurements We studied three factors, namely (i) ϵ (i.e., the error parameter), (ii) n (i.e., the number of POIs) and (iii) dataset size *DS* (i.e., the number of faces in a *TIN*). We used five measurements to evaluate the algorithm performance, namely (i)

oracle construction time, (ii) *oracle update time*, (iii) *oracle size* (i.e., the space usage of G, M_{dist} and H), (iv) *output size* (i.e., the space usage of G'), (v) *shortest path query time* and (vi) *distance error ratio*.

4.3.2 Experimental Results

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

1) Ablation study for the P2P query We consider 6 variations of *UP-Oracle*, i.e., (i) we use a random POI selection sequence, instead of using our optimal POI selection sequence, (ii) we use the full shortest distance of a shortest path as the disk radius, instead of using our minimum disk radius, (iii) we do not store the POI-to-vertex distance information and re-calculate the shortest path passing on T_{aft} for determining whether the disk intersects with ΔF, instead of using our efficient distance approximation, (iv) we create two disks for each path when checking whether we need to re-calculate the shortest path between a pair of POIs, instead of using our efficient disk and updated face intersection check, (v) we remove the sub-graph generation step, i.e., algorithm *HGSpan* in the update phase and use a hash table to store the pairwise P2P exact shortest paths passing on T_{aft} in G and (vi) we use algorithm *GSpan* [21] or algorithm *SGSpan* [29] (degenerates to algorithm *GSpan* when the input is a complete graph), instead of using algorithm *HGSpan* in the sub-graph generation step of the update phase. We use *UP-Oracle-**X*** where $\boldsymbol{X} \in$ {*RanSelSeq*, *FullRad*, *NoDistAppr*, *NoEffIntChe*, *NoEdgPru*, *NoEffEdgPru*} to denote these variations. The first four oracles correspond to the four techniques in Chapter 4.2.2. The last two oracles correspond to the idea covered in Chapter 4.2.2.

In Figure 4.6 (resp. Figure 4.7), we test the 5 values of n in {50, 100, 150, 200, 250} on *TJ* (resp. {500, 1000, 1500, 2000, 2500} on *SC*) dataset while fixing ϵ at 0.1 and *DS* at 0.5M (resp. ϵ to 0.25 and *DS* to 0.5M) for the ablation study involving 6 variations (resp. the last 3 variations, since the first 3 variations have excessive oracle update times with

500 POIs) and *UP-Oracle*. The oracle update time for *UP-Oracle-**X***, where ***X*** $\in$ {*RanSelSeq*, *FullRad*, *NoDistAppr*, *NoEffIntChe*, *NoEffEdgPru*} exceeds that of *UP-Oracle* due to the four techniques from Chapter 4.2.2 and the use of algorithm *HGSpan* from Chapter 4.2.2. Although the oracle update time and the shortest path query time of *UP-Oracle-NoEdgPru* are slightly smaller than those of *UP-Oracle*, the output size for *UP-Oracle-NoEdgPru* is 10^4 times due to the use of algorithm *HGSpan*. Thus, *UP-Oracle* is the best oracle among the variations.

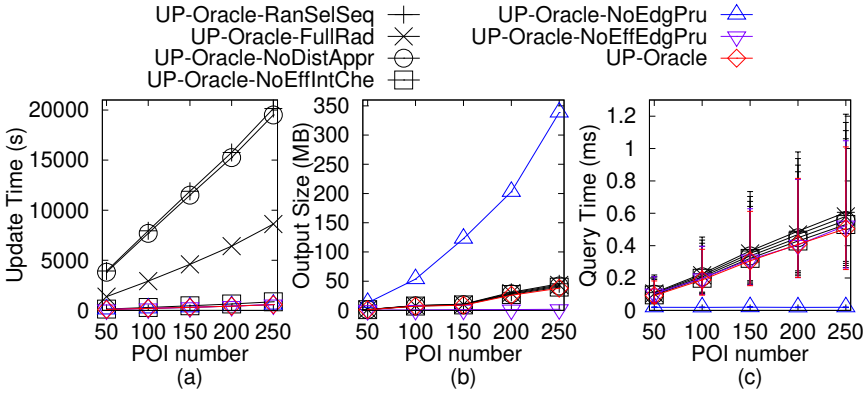


Figure 4.6. Ablation study on *TJ* dataset with fewer POIs for the P2P query regarding *UP-Oracle*

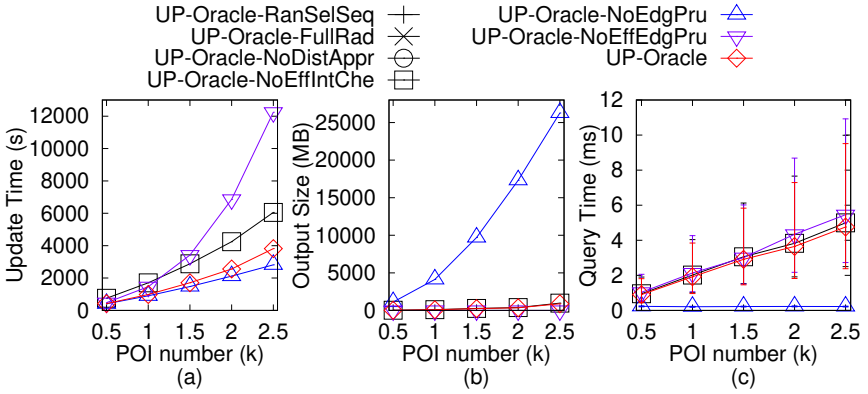


Figure 4.7. Ablation study on *SC* dataset with more POIs for the P2P query regarding *UP-Oracle*

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

Effect of ϵ: In Figure 4.8, we test the 6 values of ϵ in {0.05, 0.1, 0.25, 0.5, 0.75, 1} on *GI* dataset with fewer POIs while fixing n at 50 and *DS* at 0.5M. Although all algorithms

have errors close to 0%, *UP-Oracle* offers superior performance over other baselines concerning the oracle construction time, oracle update time, output size and shortest path query time due to the *non-updated TIN shortest path intact* property, the stored pairwise P2P exact shortest paths passing on T_{bef}, and the use of algorithm *HGSpan* in *UP-Oracle*. Although the oracle size of *UP-Oracle* is slightly larger than those of *SE-Oracle*, *SE-UP-Oracle*, *RC-TIN-Oracle* and *RC-TIN-UP-Oracle*, the oracle update time of *UP-Oracle* is 88 times, 21 times, 70 times and 17 times smaller, respectively. Varying ϵ has (i) no impact on the oracle construction time of *UP-Oracle* since it is independent of ϵ, (ii) a small impact on the oracle update time of *UP-Oracle*, since when n is small, the exact shortest path update step dominates the sub-graph generation step, and the former step is independent of ϵ and (iii) a small impact on the oracle construction time and oracle update time of other oracles since their early termination criteria of using algorithm *SSAD* are loose (i.e., they need to use algorithm *SSAD* to cover most of the POIs or highway nodes as destinations even when ϵ is large).

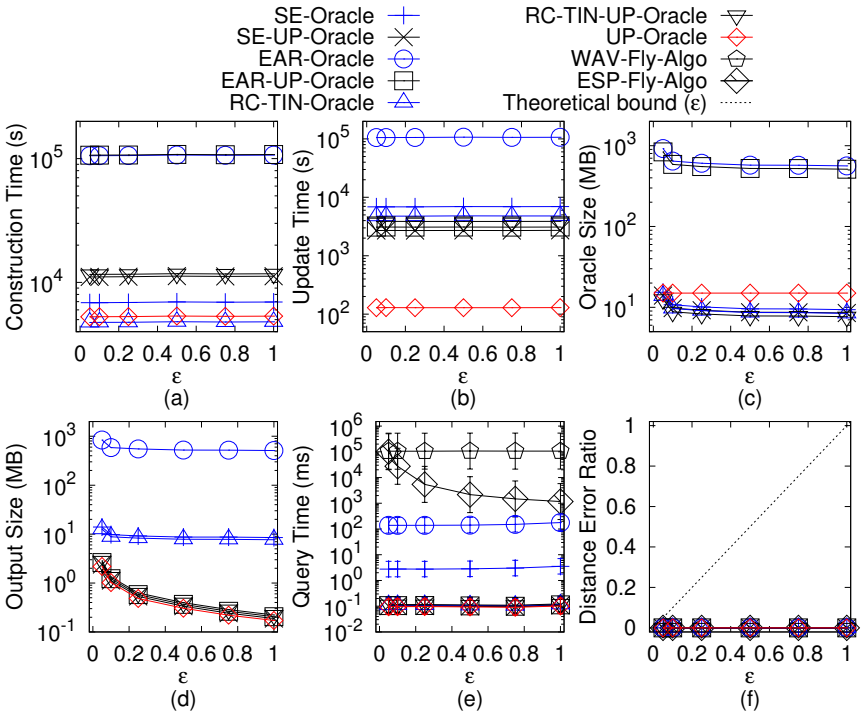


Figure 4.8. Baseline comparisons (effect of ϵ on *GI* dataset with fewer POIs for the P2P query) regarding *UP-Oracle*

Effect of *n*: In Figure 4.9, we test the 5 values of n in {50, 100, 150, 200, 250} on *AU* dataset while fixing ϵ at 0.1 (we also have the results with 5 values of n in {500, 1000, 1500,

2000, 2500} while fixing ϵ at 0.25 in [85]) and *DS* at 0.5M. The oracle update time, output size and shortest path query time of *UP-Oracle* remain better than those of the baselines. Specifically, the oracle update time of *UP-Oracle* is 21 times, 23 times and 17 times smaller than those of *SE-UP-Oracle*, *EAR-UP-Oracle* and *RC-TIN-UP-Oracle*, respectively. Since *SE-UP-Oracle*, *EAR-UP-Oracle* and *RC-TIN-UP-Oracle* have output graph G′ (which is similar to *UP-Oracle*), their output size and shortest path query time are similar to those of *UP-Oracle*.

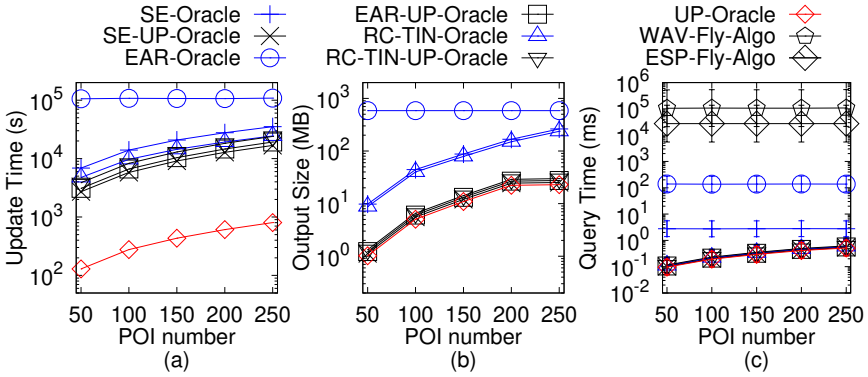


Figure 4.9. Baseline comparisons (effect of n on *AU* dataset with fewer POIs for the P2P query) regarding *UP-Oracle*

3) Scalability test for the P2P query (effect of *DS*) In Figure 4.10, we test 5 values of *DS* in {0.5M, 1M, 1.5M, 2M, 2.5M} on *LH* dataset with more POIs while fixing ϵ at 0.25 and n at 500. *UP-Oracle* can scale up to a large dataset with 2.5M points. Since *UP-Oracle* is an oracle, its shortest path query time is 10^5 times smaller than that of *ESP-Fly-Algo*.

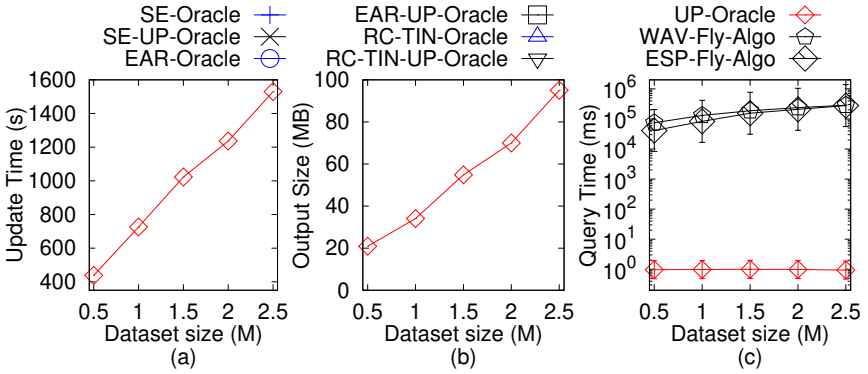


Figure 4.10. Scalability test (effect of *DS* on *LH* dataset with more POIs for the P2P query) regarding *UP-Oracle*

4) Baseline comparisons for multi-layer structure In Figure 4.11, we compare *UP-Oracle* and *UP-Oracle-MuLa* by varying ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ and fixing $\mathfrak{n}$ at 500 and *DS* at 0.5M on *SC* dataset. The oracle update time, output size and shortest path query time of *UP-Oracle-MuLa* are 1.1 times larger, 10 times larger and 2 times smaller than those of *UP-Oracle*, since *UP-Oracle-MuLa* uses M_{LOD} to store some exact paths to accelerate shortest-range queries.

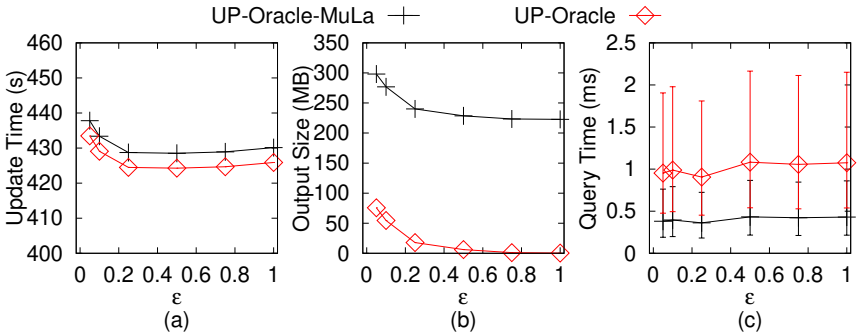


Figure 4.11. Baseline comparisons on *SC* dataset for multi-layer structure regarding *UP--Oracle*

5) Baseline comparisons for the AR2AR query In Figure 4.12, we test the AR2AR query by varying ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ while fixing *DS* at 2k on a multi-resolution of *SC* dataset. We adapt *SE-Oracle*, *SE-UP-Oracle*, *RC-TIN-Oracle* and *RC-TIN-UP-Oracle* that answer the P2P query in a similar way to *UP-Oracle-AR2AR*, and denote them as *SE-Oracle-AR2AR*, *SE-UP-Oracle-AR2AR*, *RC-TIN-Oracle-AR2AR* and *RC-TIN-UP-Oracle-AR2AR* (such that they can answer the AR2AR query). The oracle update time of *UP-Oracle-AR2AR* is 15 times better than the best-known oracle *EAR-Oracle* on *TIN*s for the AR2AR query.

6) Case study (landslide) We performed a case study on the wildfire evacuation in Santa Monica Mountains National Recreation Area in Chapter 1. Recall that the wildfire causes a landslide. We construct the oracle before the wildfire and landslide. When the wildfire and landslide happen, we capture the newest *TIN*, and then we update the oracle. In this case study, on a *TIN* with 0.5M faces and 250 POIs, *UP-Oracle* just needs 400s $\approx$ 6.7 min to update the oracle, but the best-known oracle *SE-Oracle* for the P2P query needs 35,100s $\approx$ 9.8 hours. Answering 100 paths takes 0.1s for *UP-Oracle*, 8,600s $\approx$ 2.4 hours for the best-known on-the-fly algorithm *ESP-Fly-Algo*, and 0.3s for *SE-Oracle*. Thus, only *UP-*

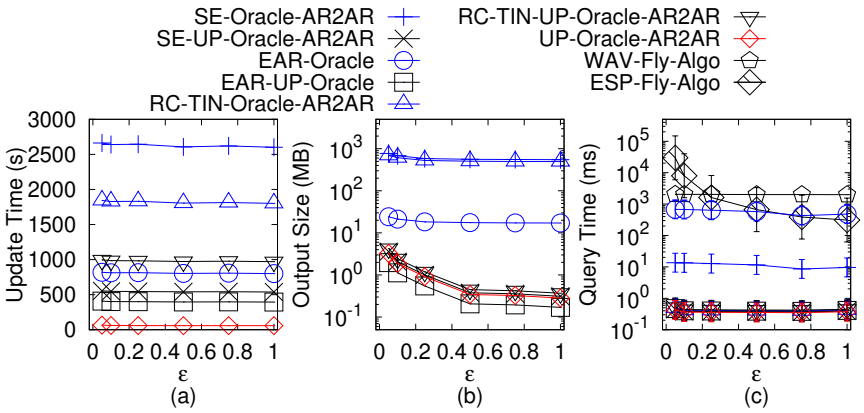


Figure 4.12. Baseline comparisons on *SC* dataset for the AR2AR query regarding *UP-Oracle*

Oracle is suitable for evacuation to save lives. In addition, since time-consuming to build evacuation paths in the damaged region, fewer paths imply that the total time to build evacuation paths is smaller, enabling the rescue teams to focus on saving lives. For other nearby regions, rescue teams need to keep roads clear to avoid road blockages caused by panic. Fewer paths imply that the number of rescue teams needed for road maintenance is smaller, enabling an increased focus on saving lives. Thus, we also aim at reducing the number of edges in the *UP-Oracle* output graph.

7) Case study (marsquake) We performed another case study on the marsquake in Chapter 1. Mars rovers aim to find the shortest escape paths quickly to avoid damage after marsquake. For Mars rovers, we only know the damaged region after a quake, so the coverage of the original *TIN* stored in Mars rovers is large (e.g., the entire Mars surface), and similar to the updated *TIN*. NASA's Mars 2020 rover has 256MB memory and 2GB hard disk space [16], and it autonomously calculates paths [12]. Since the round trip signal delay between Earth and Mars is 40 minutes [17], it is time-consuming to send *TIN* information captured by a rover after a quake to Earth, ask human experts to find shortest escape paths, and then send the paths to Mars. Before a quake, a rover stores the temporary graph G (a complete graph that stores the pairwise P2P exact shortest paths passing on the original *TIN*) in the hard disk and transfers this to memory as needed. After a quake, it needs to update G (a complete graph that stores the updated pairwise P2P exact shortest paths passing on the updated *TIN*). It then needs to efficiently generate *UP-Oracle* output graph G′ (a sub-graph of G) for evacuation path calculation. Because our experi-

mental results show that for a *TIN* with 250k faces and 250 POIs, the original G consumes 127MB, the updated G occupies 126MB, and G′ occupies 10MB. When a rover starts to escape, it needs 210MB extra memory for different sensors to work [28]. Since 126MB + 210MB = 336MB $>$ 256MB and 10MB + 210MB = 220MB $<$ 256MB, we can only fit G′ in the memory of a rover for escaping. We have sufficient memory for path updating and subgraph generation since 127MB + 126MB = 253MB $<$ 256MB and 126MB + 10MB = 136MB $<$ 256MB, and we can store the updated G in the hard disk for subsequent changes since 126MB $<$ 2GB.

8) Summary Concerning the oracle update time, output size and shortest path query time, *UP-Oracle* is up to 88 times, 12 times and 3 times (resp. 15 times, 50 times and 100 times) better than the best-known oracle *SE-Oracle* for the P2P query (resp. *EAR-Oracle* for the AR2AR query) on *TIN*s. (i) For the P2P query on a *TIN* with 0.5M faces and 250 POIs, *UP-Oracle*'s oracle update time is 400s $\approx$ 6.7 min, while *SE-Oracle* and *RC-TIN-Oracle* take 35,100s $\approx$ 9.8 hours and 28,100s $\approx$ 7.5 hours. (ii) the shortest path query time for computing 100 paths is 0.1s for *UP-Oracle*, while the time is 8,600s $\approx$ 2.4 hours for *ESP-Fly-Algo*, 0.3s for *SE-Oracle*, and 0.1s for *RC-TIN-Oracle*. (iii) For the AR2AR query on a *TIN* with 20k faces, the oracle update time and shortest path query time for computing 100 shortest paths of *UP-Oracle-AR2AR* are 480s $\approx$ 7 min and 0.05s, while the values are 7,100s $\approx$ 2 hours and 5s for *EAR-Oracle*.

CHAPTER 5

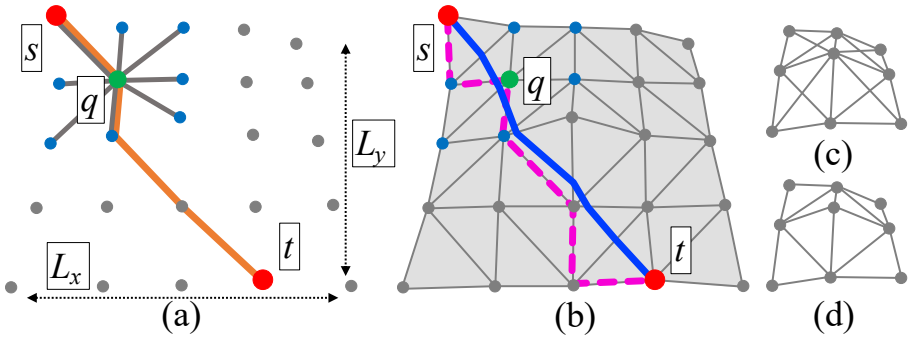
ORACLE FOR PROXIMITY QUERIES ON POINT CLOUDS

This chapter studies proximity queries on point clouds using an oracle. Chapter 5.1 provides the preliminary. Chapter 5.2 presents the methodology. Chapter 5.3 presents the experimental results.

5.1 Preliminary

5.1.1 Notation and Definitions

1) Point cloud and *TIN* Given a set of N points, we let C be a point cloud of these points. We let x_p, y_p and z_p be the three coordinate values of each point $p \in C$. We let x_{max} and x_{min} (resp. y_{max} and y_{min}) be the maximum and minimum x (resp. y) coordinate value for all points in C. We let $L_x = x_{max} - x_{min}$ (resp. $L_y = y_{max} - y_{min}$) be the x-axis (resp. y-axis) side length of C, and $L = \max(L_x, L_y)$. Figure 5.1 (a) shows a point cloud C with $L_x = L_y = 4$. In this paper, the point cloud C that we considered is a grid-based point cloud [23, 36], because a grid-based 3D object, e.g., a grid-based point cloud [23, 36] and a grid-based *TIN* [32, 51, 64, 71, 72], is commonly adopted in many papers. Given a point p in C, we let $N(p)$ be a set of neighbor points of p, which denotes the closest 8 surrounding points of p in the xy coordinate 2D plane. In Figure 5.1 (a), given a green point q, $N(q)$ is denoted as 7 blue points and 1 red point s. We can easily extend our problem to the non-grid-based point cloud. Given a point p in a non-grid-based point cloud, we just change $N(p)$ to be a set of neighbor points of p such that the Euclidean distance between p and all points in this non-grid-based point cloud is smaller than a given parameter. Let P be a set of n POIs, each of which is a point on the point cloud. Since a POI can only be a point on C, $n \leqslant N$, i.e., POIs are a subset of points in a point cloud. Let T be a *TIN* triangulated [37] by the points in C. Figure 5.1 (b) shows a *TIN* T. In this figure, given a green vertex q, the neighbor vertices of q are 6 blue vertices.



Remark: (a) A point cloud with orange $\Pi^*(s,t|C)$, (b) a *TIN* with blue $\Pi^*(s,t|T)$ and pink $\Pi_N(s,t|T)$, (c) a point cloud graph, and (d) a *TIN* graph of the *TIN*.

Figure 5.1. A small point cloud and a *TIN*

2) Point cloud graph We let G be a point cloud graph of C. G contains a set of vertices $G.V$ and edges $G.E$. Each point in C is denoted by a vertex in $G.V$. For each point $q \in C$, $G.E$ consists of a set of edges between q and $q' \in N(q)$, and each edge's weight is the Euclidean distance between these two vertices. Figure 5.1 (c) shows a point cloud graph. Given a pair of points p and p' in 3D space, we let $d_E(p,p')$ be the Euclidean distance between them. Given a pair of POIs s and t in P, let $\Pi^*(s,t|C)$ be the exact shortest path passing on (G of) C between s and t. Given a pair of POIs s and t in P, let $\Pi(s,t|C)$ be the shortest path returned by *RC-Oracle*. The shortest path passing on C from a source (POI) to a destination (POI) can contain different sub-paths where a sub-path starts from a point on C to another point on C, i.e., the differences between the points and POIs are that (i) we use points (from C) to construct G, and then calculate the shortest path passing on G, but (ii) we use POIs as sources and destinations to calculate the shortest path. G is stored as a data structure in the memory for internal processing and C can be cleared from the memory, so we do not need to construct G every time when we need to calculate the shortest path passing on C. Our experimental results show that it just needs 0.01s to construct G of C with 2.5M points. Figure 5.1 (a) shows $\Pi^*(s,t|C)$ in orange line. We let $|\cdot|$ be a path's distance (e.g., $|\Pi^*(s,t|C)|$ is $\Pi^*(s,t|C)$'s distance). *RC-Oracle* guarantees that $|\Pi(s,t|C)| \leqslant (1+\epsilon)|\Pi^*(s,t|C)|$ for any s and t in P.

Similar to G, we let G' be a *TIN* graph of T. G' contains a set of vertices $G'.V$ and edges $G'.E$, where each vertex in T is denoted by a vertex in $G'.V$, and each edge in T is denoted by an edge in $G'.E$ (the edge's weight is the same as in G). Figure 5.1 (d) shows a

TIN graph of the *TIN*. Given a pair of POIs s and t in P, let $\Pi^*(s,t|T)$ be the exact shortest surface path passing on T between s and t. Given a pair of POIs s and t in P, let $\Pi_N(s,t|T)$ be the shortest network path passing on (G' of) T between s and t. G' is also stored as a data structure in the memory for internal processing and T can be cleared from the memory. Figure 5.1 (b) shows $\Pi^*(s,t|T)$ in blue line and $\Pi_N(s,t|T)$ in pink line. Table 5.1 shows a notation table.

Table 5.1. Frequent used notations in the study of proximity queries on point clouds

Notation	**Meaning**
C	The point cloud with a set of points
N	The size of C
L	The maximum side length of C
$d_E(p,p')$	The Euclidean distance between point p and p'
P	The set of POI
n	The size of P
ϵ	The error parameter
T	The *TIN* converted from C
$\Pi^*(s,t\|C)$	The exact shortest path passing on C between s and t
$\|\Pi^*(s,t\|C)\|$	$\Pi^*(s,t\|C)$'s distance
$\Pi(s,t\|C)$	The shortest path passing on C between s and t returned by *RC-Oracle*
$\Pi^*(s,t\|T)$	The exact shortest surface path passing on T between s and t
$\Pi_N(s,t\|T)$	The shortest network path passing on T between s and t

3) Proximity queries We give the details of three proximity queries. (i) In the shortest path query, given a source s and a destination t, we answer the shortest path between s and t. (ii) In the *kNN* query, given a query object q and a user parameter k, we answer all shortest paths from q to its k nearest objects. (iii) In the range query, given a query object q and a user parameter r, we answer all shortest paths from q to objects whose distance to it is at most r.

4) P2P and A2A query By creating POIs that have the same coordinate values as all points in the point cloud, the A2A query can be regarded as one form of the P2P query. Furthermore, in the P2P query, there is no need to consider the case when a new POI is added or removed. In the case when a POI is added, we can create an oracle to answer the A2A query, which implies we have considered all possible POIs to be added. In the case when a POI is removed, we can still use the original oracle.

5.1.2 Problem

Given C and P, the problem is to (1) design an efficient $(1+\epsilon)$-approximate shortest path oracle on C with the best performance concerning the oracle construction time, oracle size and shortest path query time such that $|\Pi(s,t|C)| \leqslant (1+\epsilon)|\Pi^*(s,t|C)|$ for any pairs of s and t in P, and (2) use this oracle for efficiently answering the $(1+\epsilon)$-approximate *kNN* and range query.

5.2 Methodology

5.2.1 Overview of *RC-Oracle*

We first illustrate *RC-Oracle* with an example. In Figure 5.2 and Figure 5.3 (a), we have a point cloud, a set of POIs and an error parameter ϵ. In Figure 5.2 and Figures 5.3 (b) - (e), we construct *RC-Oracle* by calculating the shortest paths among these POIs. In Figure 5.2 and Figure 5.3 (f), we answer the shortest path query between a pair of POIs using *RC-Oracle*. Next, we introduce the two components and two phases of *RC-Oracle*.

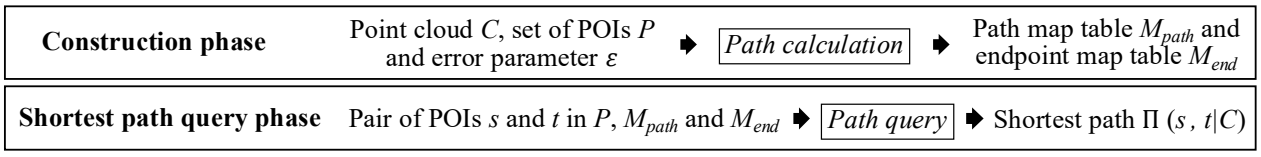


Figure 5.2. *RC-Oracle* framework overview description

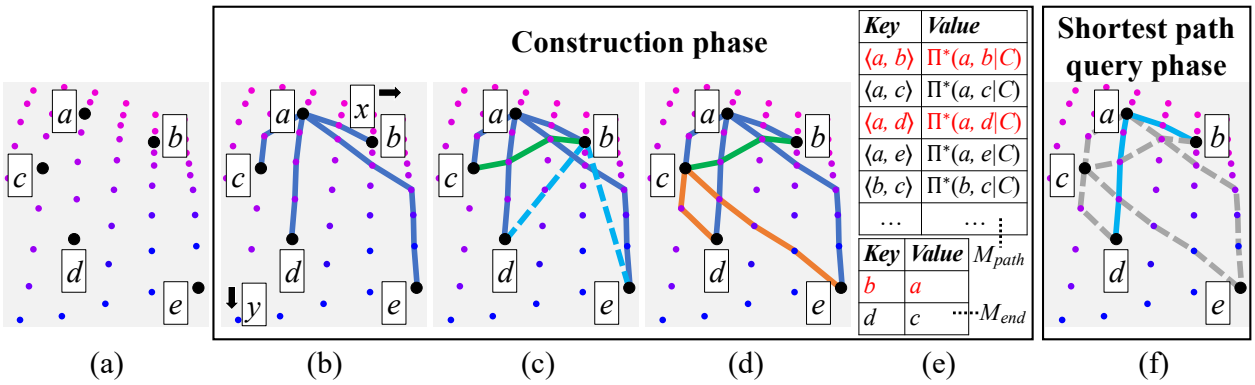


Figure 5.3. *RC-Oracle* framework overview

1) Components of *RC-Oracle* There are two components, i.e., the *path map table* and the *endpoint map table*.

(i) **The path map table** M_{path} is a *hash table* [27] that stores a set of key-value pairs. For each key-value pair, it stores a pair of endpoints (i.e., POIs) u and v, as a key $\langle u, v \rangle$ (the key here refers to a pair of endpoints), and the corresponding exact shortest path $\Pi^*(u, v|C)$ passing on C, as a value (the value here refers to a whole path). M_{path} needs linear space concerning the number of paths to be stored. Given a pair of endpoints (i.e., POIs) u and v, M_{path} can return the associated exact shortest path $\Pi^*(u, v|C)$ passing on C in $O(1)$ time. In Figure 5.3 (d), there are 7 exact shortest paths passing on C, and they are stored in M_{path} in Figure 5.3 (e). For the exact shortest paths passing on C between b and c, M_{path} stores $\langle b, c \rangle$ as a key and $\Pi^*(b, c|C)$ as a value.

(ii) **The endpoint map table** M_{end} is a *hash table* that stores a set of key-value pairs. For each key-value pair, it stores an endpoint (i.e., a POI) u as a key (such that we do not store all the exact shortest paths passing on C in M_{path} from u to other non-processed endpoints, and the key here refers to an endpoint), and another endpoint (i.e., a POI) v as a value (such that v is close to u, and we concatenate $\Pi^*(u, v|C)$ and the exact shortest paths passing on C with v as a source, to approximate the shortest paths passing on C with u as a source, and the value here refers to an endpoint). The space consumption and query time of M_{end} is similar to M_{path}. In Figure 5.3 (d), a is close to b, we concatenate $\Pi^*(b, a|C)$ and the exact shortest paths passing on C with a as a source, to approximate the shortest paths passing on C with b as a source, so we store b as a key and a as a value in M_{end} in Figure 5.3 (e).

2) Phases of *RC-Oracle* There are two phases, i.e., *construction phase* and *shortest path query phase* (see Figures 5.2 and 5.3). (i) In the construction phase, given a point cloud C, a set of POIs P and an error parameter ϵ, we pre-compute the exact shortest paths passing on C between some selected pairs of POIs, store them in M_{path}, and store the non-selected POIs and their corresponding selected POIs in M_{end}. (ii) In the shortest path query phase, given a pair of POIs s and t, M_{path} and M_{end}, we answer the path results between s and t efficiently.

5.2.2 Key Ideas of *RC-Oracle*

1) Small oracle construction time We give the reason why *RC-Oracle* has a small oracle construction time.

(i) **Rapid point cloud on-the-fly shortest path querying by algorithm *FastFly***: When constructing *RC-Oracle*, given a point cloud C and a pair of POIs s and t on C, we use algorithm *FastFly* (a Dijkstra's algorithm [34]) to directly calculate the *exact* shortest path passing on the point cloud graph of C between s and t, which is efficient.

(ii) **Rapid oracle construction**: When constructing *RC-Oracle*, we regard each POI as a source and use algorithm *FastFly*, i.e., a *SSAD* algorithm, for n times for oracle construction, and we assign a *tight* earlier termination criteria for each POI to terminate the *SSAD* algorithm earlier for time-saving. There are two versions of a *SSAD* algorithm. The first version is that given a source POI and a set of destination POIs, the *SSAD* algorithm can terminate earlier if it has visited all destination POIs. The second version is that given a source POI and a *termination distance* (denoted by D), the *SSAD* algorithm can terminate earlier if the searching distance from the source POI is larger than D. We use the first version. For each POI, by considering more geometry information of the point cloud, including the Euclidean distance and the distance of the previously calculated shortest paths, we use *tight* earlier termination criteria to calculate the corresponding destination POIs, such that the number of destination POIs is minimized, and these destination POIs are closer to the source POI compared with other POIs.

We use an example for illustration. In Figure 5.3 (a), we have a set of POIs a, b, c, d, e. In Figure 5.3 (b) - (d), we process these POIs based on their y-coordinate, i.e., we process them in the order of a, b, c, d, e. In Figure 5.3 (b), for a, we use the *SSAD* algorithm (i.e., *FastFly*) to calculate the shortest paths passing on C from a to all other POIs. We store the paths in M_{path}. In Figure 5.3 (c), for b, if b is close to a, more specifically $\frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)| < d_E(b, d)$ and $\frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)| < d_E(b, e)$, which is judged using the previously calculated $|\Pi^*(a, b|C)|$, and if b is far away from d (resp. e), more specifically $\frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)| < d_E(b, d)$ (resp. $\frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)| < d_E(b, e)$), which is judged using the Euclidean distance $d_E(b, d)$ (resp. $d_E(b, e)$), then we can use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ (resp. $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$) to approximate $\Pi^*(b, d|C)$ (resp. $\Pi^*(b, e|C)$). Thus, we just need to use the *SSAD* algorithm

with b as a source, and terminate earlier when it has visited c. We store the path in M_{path}, and b as key and a as value in M_{end}. In Figure 5.3 (d), for c, we repeat the process as of for a. We store the paths in M_{path}. Similarly, for d, we use $|\Pi^*(c, d|C)|$ and $d_E(c, e)$ to determine whether we can terminate the *SSAD* algorithm earlier with d as a source. We found that there is even no need to use the *SSAD* algorithm with d as the source. For *different* POIs b and d, we use *customized* termination criteria (i.e., $|\Pi^*(a, b|C)|$ and $d_E(b, d)$ for b, $|\Pi^*(c, d|C)|$ and $d_E(c, e)$ for d) to calculate a tight and different set of destination POIs for time-saving. We store d as key and c as value in M_{end}. In Figure 5.3 (e), we have M_{path} and M_{end}.

However, in *SE-Oracle-Adapt*, it has the *loose criterion for algorithm earlier termination* drawback. After the construction of the compressed partition tree, it pre-computes the shortest surface paths passing on T using the *SSAD* algorithm (i.e., *CH-Adapt*) with each POI as a source for n times, to construct the well-separated node pair sets. It uses the second version of the *SSAD* algorithm and sets termination distance $D = \frac{8r}{\epsilon} + 10r$, where r is the source POI's radius in the compressed partition tree. Given two POIs p and q in the same level of the compressed partition tree, their termination distances are the same (suppose that the value is d_1). However, for p, it is enough to terminate the *SSAD* algorithm when the searching distance from p is larger than d_2, where $d_2 < d_1$. This results in a large oracle construction time. In Figure 5.4, when processing d, suppose that b and d are in the same level of the compressed partition tree, and they use the *same* termination criteria to get the *same* termination distance D. Since $|\Pi^*(d, e|C)| < D$, for d, it cannot terminate the *SSAD* algorithm earlier until e is visited, which means its termination criteria is loose. The two versions of the *SSAD* algorithm have similar ideas, we achieve a small oracle construction time mainly by using *tight* and customized termination criteria for different POIs.

2) Small oracle size We introduce the reason why *RC-Oracle* has a small oracle size. We only store a small number of paths in *RC-Oracle*, i.e., we do not store the paths between any pairs of POIs. In Figure 5.3 (d), for a pair of POIs b and d, we use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ to approximate $\Pi^*(b, d|C)$, i.e., we will not store $\Pi^*(b, d|C)$ in M_{path} for memory saving.

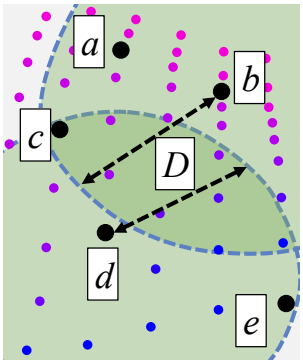


Figure 5.4. An illustration of *SE-Oracle-Adapt*

3) Small shortest path query time We use an example to introduce the reason why *RC-Oracle* has a small shortest path query time. In Figure 5.3 (f), in the shortest path query phase of *RC-Oracle*, we need to query the shortest path passing on C (i) between a source a and a destination d, and (2) between a source b and a destination d. (1) For a and d, since $\langle a, d\rangle \in M_{path}.key$, we can directly return $\Pi^*(a, d|C)$. (ii) For b and d, since $\langle b, d\rangle \notin M_{path}.key$, b and d are both keys in M_{end}, we use the key b with a smaller y-coordinate value to retrieve the value a in M_{end}, and then in M_{path}, we use $\langle b, a\rangle$ and $\langle a, d\rangle$ to retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, for approximating $\Pi^*(b, d|C)$.

5.2.3 Implementation Details of *RC-Oracle*

1) Construction phase Given a point cloud C and a set of POIs P, *RC-Oracle* pre-computes the exact shortest paths passing on C between some selected pairs of POIs, stores them in M_{path}, and stores the non-selected POIs and their corresponding POIs in M_{end}.

Notation: Let $P_{remain} = \{p_1, p_2, \dots\}$ be a set of remaining POIs of P that we have not used algorithm *FastFly* to calculate the exact shortest paths passing on C with $p_i \in P_{remain}$ as a source. P_{remain} is initialized to be P. Given a POI q, let $P_{dest}(q) = \{p_1, p_2, \dots\}$ be a set of POIs of P that we need to use *FastFly* to calculate the exact shortest paths passing on C from q to $p_i \in P_{dest}(q)$. $P_{dest}(q)$ is empty at the beginning. In Figure 5.3 (c), $P_{remain} = \{c, d, e\}$ since we have not used *FastFly* to calculate the exact shortest paths with c, d, e as source, $P_{dest}(b) = \{c\}$ since we need to use *FastFly* to calculate the exact shortest path from b to c.

Detail and example: Algorithm 7 shows the construction phase in detail, and the fol-

lowing illustrates it with an example.

Algorithm 7 *Construction* (C, P, ϵ)

Input: a point cloud C, a set of POIs P and and an error parameter ϵ
Output: a path map table M_{path} and an endpoint map table M_{end}

1: $P_{remain} \leftarrow P, M_{path} \leftarrow \emptyset, M_{end} \leftarrow \emptyset$
2: **if** $L_x \geqslant L_y$ (resp. $L_x < L_y$) **then**
3: sort POIs in P_{remain} in ascending order using x-coordinate (resp. y-coordinate)
4: **while** P_{remain} is not empty **do**
5: $u \leftarrow$ a POI in P_{remain} with the smallest x-coordinate / y-coordinate
6: $P_{remain} \leftarrow P_{remain} - \{u\}, P'_{remain} \leftarrow P_{remain}$
7: calculate the exact shortest paths passing on C from u to each POI in P'_{remain} simultaneously using algorithm *FastFly*
8: **for** each POI $v \in P'_{remain}$ **do**
9: $key \leftarrow \langle u, v \rangle$, $value \leftarrow \Pi^*(u, v|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
10: sort POIs in P'_{remain} in ascending order using the exact distance on C between u and each $v \in P_{remain}$, i.e., $|\Pi^*(u, v|C)|$
11: **for** each sorted POI $v \in P'_{remain}$ such that $d_E(u, v) \leqslant \epsilon L$ **do**
12: $P_{remain} \leftarrow P_{remain} - \{v\}, P'_{remain} \leftarrow P'_{remain} - \{v\}, P_{dest}(v) \leftarrow \emptyset$
13: **for** each POI $w \in P'_{remain}$ **do**
14: **if** $\frac{2}{\epsilon} \cdot |\Pi^*(u, v|C)| < d_E(v, w)$ and $v \notin M_{end}.key$ **then**
15: $key \leftarrow v$, $value \leftarrow u$, $M_{end} \leftarrow M_{end} \cup \{key, value\}$
16: **else if** $\frac{2}{\epsilon} \cdot |\Pi^*(u, v|C)| \geqslant d_E(v, w)$ **then**
17: $P_{dest}(v) \leftarrow P_{dest}(v) \cup \{w\}$
18: calculate the exact shortest paths passing on C from v to each POI in $P_{dest}(v)$ simultaneously using algorithm *FastFly*
19: **for** each POI $w \in P_{dest}(v)$ **do**
20: $key \leftarrow \langle v, w \rangle$, $value \leftarrow \Pi^*(v, w|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
21: **return** M_{path} and M_{end}

(i) *POIs sort* (lines 2-3): In Figure 5.3 (b), since $L_x < L_y$, the sorted POIs are a, b, c, d, e.

(ii) *Shortest paths calculation* (lines 4-20): There are two steps.

- *Exact shortest paths calculation* (lines 5-9): In Figure 5.3 (b), a has the smallest y-coordinate based on the sorted POIs in P_{remain}, we delete a from P_{remain} (so $P_{remain} = P'_{remain} = \{b, c, d, e\}$), calculate the exact shortest paths passing on C from a to b, c, d, e (in purple lines) using algorithm *FastFly*, and store each POIs pair as a key and the corresponding path as a value in M_{path}.

- *Shortest paths approximation* (lines 10-20): In Figure 5.3 (c), b is the POI in P'_{remain} closest to a, c is the POI in P'_{remain} second closest to a, so the following order is

$b, c, \ldots$. There are two cases:

- *Approximation loop start* (lines 11-20): In Figure 5.3 (c), we first select a's closest POI in P'_{remain}, i.e., b, since $d_E(a, b) \leqslant \epsilon L$, it means a and b are not far away, we start the approximation loop, delete b from P_{remain} and P'_{remain}, so $P_{remain} = P'_{remain} = \{c, d, e\}$. There are three steps:
  - *Far away POIs selection* (lines 13-15): In Figure 5.3 (c), $\frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)| < d_E(b, d)$, $\frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)| < d_E(b, e)$, $d \notin M_{end}.key$ and $e \notin M_{end}.key$, it means d and e are far away from b, we can use $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$ that we have already calculated before to approximate $\Pi^*(b, d|C)$, and use $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$ that we have already calculated before to approximate $\Pi^*(b, e|C)$, so we get $\Pi(b, d|C)$ by concatenating $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, and get $\Pi(b, e|C)$ by concatenating $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$, we store b as key and a as value in M_{end}.
  - *Close POIs selection* (line 13 and lines 16-17): In Figure 5.3 (c), $\frac{2}{\epsilon} \cdot |\Pi^*(a, b|C)| \geqslant d_E(b, c)$, it means c is close to b, so we cannot use any existing exact shortest paths passing on C to approximate $\Pi^*(b, c|C)$, and then we store c into $P_{dest}(b)$.
  - *Selected exact shortest paths calculation* (lines 18-20): In Figure 5.3 (c), when we have processed all POIs in P'_{remain} with b as a source, we have $P_{dest}(b) = \{c\}$, we use algorithm *FastFly* to calculate the exact shortest path passing on C between b and c, i.e., $\Pi^*(b, c|C)$ (in green line), and store $\langle b, c\rangle$ as a key and $\Pi^*(b, c|C)$ as a value in M_{path}. Note that we can terminate algorithm *FastFly* earlier since we just need to visit POIs that are close to b, and we do not need to visit d and e.
- *Approximation loop end* (line 11): In Figure 5.3 (c), since we have processed b, and $P'_{remain} = \{c, d, e\}$, we select a's closest POI in P'_{remain}, i.e., c, since $d_E(a, c) > \epsilon L$, it means a and c are far away, and it is unlikely to have a POI m that satisfies $\frac{2}{\epsilon} \cdot |\Pi^*(a, c|C)| < d_E(c, m)$, we end the approximation loop and terminate the iteration.

(ii) *Shortest paths calculation iteration* (lines 4-20): In Figure 5.3 (d), we repeat the iter-

ation, and calculate the exact shortest paths passing on C with c as a source (in orange lines).

2) Shortest path query phase Given a pair of POIs s and t in P, M_{path} and M_{end}, *RC-Oracle* efficiently answers the associated shortest path $\Pi(s, t|C)$ passing on C, which is a $(1+\epsilon)$-approximated exact shortest path of $\Pi^*(s, t|C)$ in $O(1)$ time. Given a pair of POIs s and t, there are two cases (s and t are interchangeable, i.e., $\langle s, t\rangle = \langle t, s\rangle$):

(i) *Exact shortest path retrieval*: If $\langle s, t\rangle \in M_{path}.key$, we use $\langle s, t\rangle$ to retrieve $\Pi^*(s, t|C)$ as $\Pi_{RC\text{-}Oracle}(s, t|C)$ in $O(1)$ time. In Figures 5.3 (d) and (e), given a and d, since $\langle a, d\rangle \in M_{path}.key$, we retrieve $\Pi^*(a, d|C)$.

(ii) *Approximate shortest path retrieval*: If $\langle s, t\rangle \notin M_{path}.key$, it means $\Pi^*(s, t|C)$ is approximated by two exact shortest paths passing on C in M_{path}, and (a) either s or t is a key in M_{end}, or (b) both s and t are keys in M_{end}. Without loss of generality, suppose that (a) s exists in M_{end} if either s or t is a key in M_{end}, or (b) the x- (resp. y-) coordinate of s is smaller than or equal to t when $L_x \geqslant L_y$ (resp. $L_x < L_y$) if both s and t are keys in M_{end}. For both of two cases, we use the key s to retrieve the value s' in M_{end} in $O(1)$ time, and then use $\langle s, s'\rangle$ and $\langle s', t\rangle$ to retrieve $\Pi^*(s, s'|C)$ and $\Pi^*(s', t|C)$ in M_{path} in $O(1)$ time. We then use $\Pi^*(s, s'|C)$ and $\Pi^*(s', t|C)$ as $\Pi_{RC\text{-}Oracle}(s, t|C)$ to approximate $\Pi^*(s, t|C)$. (a) In Figures 5.3 (d) and (e), given b and e, $\langle b, e\rangle \notin M_{path}.key$, and b is a key in M_{end}. So, we use the key b to retrieve the value a in M_{end}. Then, in M_{path}, we use $\langle b, a\rangle$ and $\langle a, e\rangle$ to retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, e|C)$, for approximating $\Pi^*(b, e|C)$. (b) In Figures 5.3 (d), (e) and (f), given b and d, $\langle b, d\rangle \notin M_{path}.key$, b and d are both keys in M_{end}, and $L_x < L_y$. So, we use the key b with a smaller y-coordinate value to retrieve the value a in M_{end}. Then, in M_{path}, we use $\langle b, a\rangle$ and $\langle a, d\rangle$ to retrieve $\Pi^*(b, a|C)$ and $\Pi^*(a, d|C)$, for approximating $\Pi^*(b, d|C)$).

5.2.4 Adaptation to *RC-Oracle-A2A*

We can adapt *RC-Oracle* (that answers the P2P query) to be *RC-Oracle-A2A* (that answers the A2A query) on a point cloud C, by simply creating POIs that have the same coordinate values as all points in C. We just need to pre-compute the exact shortest paths passing on C between *some* selected (not *all*) pairs of points on C, so *RC-Oracle-A2A* also has a good performance.

5.2.5 Proximity Query Algorithms

Given a point cloud C, a set of n' objects O on C, a query object $q \in O$, a value k in *kNN* query and a value r in range query, we can answer other proximity queries, i.e., the *kNN* and range query using *RC-Oracle* and *RC-Oracle-A2A*. In the P2P query (resp. A2A query), these objects are POIs in P (resp. any point on C). A naive algorithm is to perform a linear scan using the shortest path query results. But, our efficient algorithm can reduce the proximity query time. Intuitively, when constructing *RC-Oracle* or *RC-Oracle-A2A*, we have used the *SSAD* algorithm to calculate the shortest paths passing on C with q as a source and sorted these paths in ascending order based on their distance in M_{path} (we can use an additional table to store these sorted paths). For these paths, we do not need to perform linear scans over all of them in proximity queries for time-saving. Since the proximity query algorithms for *RC-Oracle* and *RC-Oracle-A2A* have similar ideas, we use *RC-Oracle* as an example for illustration. Then, we give the notation, detail and example (using *kNN* query with $k = 1$).

Notation: Let $list_u, list_v, \ldots$ be a set of lists, where each list stores a set of sorted paths calculated by *SSAD* algorithm with one POI $u, v, \ldots$ as the source. Let M_{list} be a hash table that stores a set of key-value pairs. For each key-value pair, it stores an endpoint u as a key, and the corresponding list $list_u$ as a value. u is the endpoint that we use as a source to calculate the exact shortest paths passing on C. $list_u$ is the corresponding sorted paths. In Figure 5.3 (d), we store $\Pi^*(a, b|C)$, $\Pi^*(a, c|C)$, $\Pi^*(a, d|C)$ and $\Pi^*(a, e|C)$ in order in $list_a$. We store $\Pi^*(b, c|C)$ in $list_b$. We store a as a key, and $list_a$ as a value in M_{list}. We also store b as a key, and $list_b$ as a value in M_{list}.

Detail and example: There are two cases.

(1) *Approximation needed in direct result return*: If $q \in M_{end}.key$, it means we need to use two paths in M_{path} to approximate some other paths in a later stage, we use the key q to retrieve the value q' in M_{end}, there are two more cases. In Figures 5.3 (d) and (e), given b as the query object, since $b \in M_{end}.key$, we use the key b to retrieve the value a in M_{end}, there are two more cases.

(i) *Linear scan*: For the target objects with a smaller or same x- (resp. y-) coordinate compared with q' when $L_x \geqslant L_y$ (resp. $L_x < L_y$), we perform a linear scan on the shortest

path query result between q and them. We maintain *kNN* or range query result. In Figures 5.3 (d) and (e), since $L_x < L_y$, the POI with a smaller or same y-coordinate compared with a is $\{a\}$, we perform a linear scan on the shortest path query result between b and $\{a\}$. The *kNN* result stores $\{\Pi^*(a, b|C)\}$.

(ii) *Direct result return*: For the target objects (not including q) with a larger x- (resp. y-) coordinate compared with q' when $L_x \geqslant L_y$ (resp. $L_x < L_y$), there are further more two cases. In Figures 5.3 (d) and (e), since $L_x < L_y$, the POIs with a larger y-coordinate compared with a are $\{c, d, e\}$, there are further more two cases.

- *Direct result return without approximation*: If the endpoint pairs of q and these target objects are keys in M_{path}, it means that we have used the *SSAD* algorithm with q as a source for such objects and already sorted such paths in order in $list_q$. We can use q to retrieve $list_q$ in M_{list}. We maintain *kNN* or range query result. In Figures 5.3 (d) and (e), since $\langle b, c\rangle \in M_{path}.key$, we know that $\Pi^*(b, c|C)$ is sorted in order in $list_b$. We use b to retrieve $list_b$ in M_{list}. The sorted path in $list_b$ is $\Pi^*(b, c|C)$. The *kNN* result stores $\{\Pi^*(a, b|C)\}$.

- *Direct result return with approximation*: If the endpoint pairs of q and these target objects are not keys in M_{path}, it means that we have used the *SSAD* algorithm with q' as a source for such objects and already sorted such paths in order in $list_{q'}$. We can use q' to retrieve $list_{q'}$ in M_{list}. We just need to use the exact distance between q' and these target objects plus $|\Pi^*(q', q|C)|$, to get the approximate distance between q and o. We maintain *kNN* or range query result. In Figures 5.3 (d) and (e), since $\langle b, d\rangle \notin M_{path}.key$ and $\langle b, e\rangle \notin M_{path}.key$, we know that $\Pi^*(a, d|C)$ and $\Pi^*(a, e|C)$ are sorted in order in $list_a$. We use a to retrieve $list_a$ in M_{list}. The sorted paths in $list_a$ are $\Pi^*(a, d|C)$ and $\Pi^*(a, e|C)$. So, $\Pi(b, d|C)$ and $\Pi(b, e|C)$ are also sorted in order. The *kNN* result stores $\{\Pi^*(a, b|C)\}$.

(2) *Approximation not needed in direct result return*: If $q \notin M_{end}.key$, it means we do need to use two paths in M_{path} to approximate all other paths in a later stage, there are two more cases. In Figures 5.3 (d) and (e), given c as the query object, since $c \notin M_{end}.key$, there are two more cases.

(i) *Linear scan*: For the target objects with a smaller or same x- (resp. y-) coordinate compared with q when $L_x \geqslant L_y$ (resp. $L_x < L_y$), we perform a linear scan on the shortest path query result between q and them. We maintain *kNN* or range query result. In Figures 5.3 (d) and (e), since $L_x < L_y$, the POIs with a smaller y-coordinate compared with c are $\{a, b\}$, we perform a linear scan on the shortest path query result between c and $\{a, b\}$. The *kNN* result stores $\{\Pi^*(a, c|C)\}$.

(ii) *Direct result return*: For the target objects with a larger x- (resp. y-) coordinate compared with q when $L_x \geqslant L_y$ (resp. $L_x < L_y$), we have used the *SSAD* algorithm with q as a source for such objects and already sorted such paths in order in $list_q$. We can use q to retrieve $list_q$ in M_{list}. We maintain *kNN* or range query result. In Figures 5.3 (d) and (e), since $L_x < L_y$, the POIs with a larger y-coordinate compared with c are $\{d, e\}$, we know that $\Pi^*(c, d|C)$ and $\Pi^*(c, e|C)$ are sorted in order in $list_c$. We use c to retrieve $list_c$ in M_{list}. The sorted paths in $list_c$ are $\Pi^*(c, d|C)$ and $\Pi^*(c, e|C)$. The *kNN* result stores $\{\Pi^*(c, d|C)\}$.

5.2.6 Theoretical Analysis

1) Algorithm *FastFly*, *RC-Oracle* and *RC-Oracle-A2A* The analysis of Algorithm *FastFly* is in Theorem 5.2.1, and the analysis of *RC-Oracle* and *RC-Oracle-A2A* are in Theorem 5.2.2.

Theorem 5.2.1 *The shortest path query time and memory consumption of algorithm FastFly are* $O(N \log N)$ *and* $O(N)$. *Algorithm FastFly returns the exact shortest path passing on the point cloud.*

Proof. Since algorithm *FastFly* is a Dijkstra's algorithm that returns the exact shortest path and there are total N points, we finish the proof. □

Theorem 5.2.2 *The oracle construction time, oracle size and shortest path query time of (i) RC-Oracle are* $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$, $O(1)$ *and (ii) RC-Oracle-A2A are* $O(\frac{N \log N}{\epsilon})$, $O(\frac{N}{\epsilon})$, $O(1)$, *respectively. RC-Oracle always have* $|\Pi(s, t|C)| \leqslant (1+\epsilon)|\Pi^*(s, t|C)|$ *for any pairs of POIs* s *and* t *in* P, *and RC-Oracle-A2A always have* $|\Pi_{RC\text{-}Oracle\text{-}A2A}(s, t|C)| \leqslant (1+\epsilon)|\Pi^*(s, t|C)|$ *for any*

pairs of points s *and* t *on* C*, where* $\Pi_{\textit{RC-Oracle-A2A}}(s, t|C)$ *is the shortest path of RC-Oracle-A2A passing on* C *between* s *and* t*.*

Proof. We give the proof for *RC-Oracle* as follows.

Firstly, we show the *oracle construction time*. (i) In *POIs sort* step, it needs $O(n \log n)$ time. Since there are n POIs, and we use the quick sort for sorting. (ii) In *shortest paths calculation* step, it needs $O(\frac{N \log N}{\epsilon} + n)$ time. Specifically, it needs to use $O(\frac{1}{\epsilon})$ POIs as a source to run algorithm *FastFly* for the exact shortest paths calculation according to standard packing property [40], and each algorithm *FastFly* needs $O(N \log N)$ time. For other $O(n)$ POIs that there is no need to use them as a source to run algorithm *FastFly*, we just calculate the Euclidean distance from these POIs to other POIs in $O(1)$ time for the shortest paths approximation. (iii) So the oracle construction time is $O(\frac{N \log N}{\epsilon} + n \log n)$.

Secondly, we show the *oracle size*. (i) For $M_{\textit{end}}$, its size is $O(n)$ since there are n POIs. (ii) For $M_{\textit{path}}$, its size is $O(\frac{n}{\epsilon})$. We store $O(\frac{n}{\epsilon})$ exact shortest paths passing on C from $O(\frac{1}{\epsilon})$ POIs (that uses algorithm *FastFly* as a source and cover all other POIs) to other $O(n)$ POIs, and $O(n)$ exact shortest paths passing on C from $O(n)$ POIs (that uses algorithm *FastFly* as a source and cover only some of POIs) to other $O(1)$ POIs. (iii) So the oracle size is $O(\frac{n}{\epsilon})$.

Thirdly, we show the *shortest path query time*. (i) If $\Pi^*(s, t|C) \in M_{\textit{path}}$, the shortest path query time is $O(1)$. (ii) If $\Pi^*(s, t|C) \notin M_{\textit{path}}$, we need to retrieve s' from $M_{\textit{end}}$ using s in $O(1)$ time, and retrieve $\Pi^*(s, s'|C)$ and $\Pi^*(s', t|C)$ from $M_{\textit{path}}$ using $\langle s, s' \rangle$ and $\langle s', t \rangle$ in $O(1)$ time, so the shortest path query time is still $O(1)$. Thus, the shortest path query time of *RC-Oracle* is $O(1)$.

Fourthly, we show the *error bound*. Given a pair of POIs s and t, if $\Pi^*(s, t|C)$ exists in $M_{\textit{path}}$, then there is no error. Thus, we only consider the case that $\Pi^*(s, t|C)$ does not exist in $M_{\textit{path}}$. Suppose that u is a POI close to s, such that $\Pi(s, t|C)$ is calculated by concatenating $\Pi^*(s, u|C)$ and $\Pi^*(u, t|C)$. This means that $\frac{2}{\epsilon} \cdot \Pi^*(u, s|C) < d_E(s, t)$. So we have $|\Pi^*(s, u|C)| + |\Pi^*(u, t|C)| < |\Pi^*(s, u|C)| + |\Pi^*(u, s|C)| + |\Pi^*(s, t|C)| = |\Pi^*(s, t|C)| + 2 \cdot |\Pi^*(u, s|C)| < |\Pi^*(s, t|C)| + \epsilon \cdot d_E(s, t) \leqslant |\Pi^*(s, t|C)| + \epsilon \cdot |\Pi^*(s, t|C)| = (1 + \epsilon)|\Pi^*(s, t|C)|$. The first inequality is because of triangle inequality. The second equality is because of $|\Pi^*(u, s|C)| = |\Pi^*(s, u|C)|$. The third inequality is because we have $\frac{2}{\epsilon} \cdot \Pi^*(u, s|C) < d_E(s, t)$.

The fourth inequality is because the Euclidean distance between two points is no larger than the distance of the shortest path passing on the point cloud between the same two points.

We give the proof for *RC-Oracle-A2A* as follows. We need to change (i) n to N in the oracle construction time and oracle size, and (ii) any pairs of POIs in P to any pairs of points on C in the error bound. □

2) The shortest path passing on a point cloud and the shortest surface or network path passing on a *TIN* We show the relationship of $|\Pi^*(s,t|C)|$ with $|\Pi_N(s,t|T)|$ and $|\Pi^*(s,t|T)|$ in Lemma 5.2.1.

Lemma 5.2.1 *Given a pair of points* s *and* t *on* C*, we have (i)* $|\Pi^*(s,t|C)| \leqslant |\Pi_N(s,t|T)|$ *and (ii)* $|\Pi^*(s,t|C)| \leqslant k' \cdot |\Pi^*(s,t|T)|$*, where* $k' = \max\{\frac{2}{\sin\theta}, \frac{1}{\sin\theta\cos\theta}\}$.

Proof. (i) In Figure 5.1 (a), given a green point q on C, it can connect with one of its 8 neighbor points (7 blue points and 1 red point s). In Figure 5.1 (b), given a green vertex q on T, it can only connect with one of its 6 blue neighbor vertices. So $|\Pi^*(s,t|C)| \leqslant |\Pi_N(s,t|T)|$. (ii) We let $\Pi_E(s,t|T)$ be the shortest path passing on the edges of T (where these edges belong to the faces that $\Pi^*(s,t|T)$ passes) between s and t. According to left hand side equation in Lemma 2 of [46], we have $|\Pi_E(s,t|T)| \leqslant k' \cdot |\Pi^*(s,t|T)|$. Since $\Pi_N(s,t|T)$ considers all the edges on T, $|\Pi_N(s,t|T)| \leqslant |\Pi_E(s,t|T)|$. Thus, we finish the proof by combining these inequalities. □

3) Proximity query algorithms We provide analysis on the proximity query algorithms using *RC-Oracle* and *RC-Oracle-A2A*. For the *kNN* and range query, both of them return a set of objects. Given a query object q, we let v_f (resp. v'_f) be the furthest object to q among the returned objects calculated using the exact distance on C (resp. the approximated distance on C returned by *RC-Oracle*). In Figure 1.1 (a), suppose that the exact k nearest POIs ($k = 2$) of a is b, c. And b is the furthest POI to a in these two POIs, i.e., $v_f = b$. Suppose that our *kNN* query algorithm finds the k nearest POIs ($k = 2$) of a is c, d. And d is the furthest POI to a in these two POIs, i.e., $v'_f = d$. We define the error ratio of the *kNN*

and range query to be $\frac{|\Pi^*(q,v'_f|C)|}{|\Pi^*(q,v_f|C)|}$, which is a real number no smaller than 1. In Figure 1.1 (a), the error ratio is $\frac{|\Pi^*(a,d|C)|}{|\Pi^*(a,b|C)|}$. Then, we show the query time and error ratio of *kNN* and range query using *RC-Oracle* and *RC-Oracle-A2A* in Theorem 5.2.3.

Theorem 5.2.3 *The query time and error ratio of both the kNN and range query by using RC-Oracle and RC-Oracle-A2A are both* $O(n')$ *and* $1+\epsilon$*, respectively.*

Proof. We give the proof for *RC-Oracle* as follows.

Firstly, we show the query time of both the *kNN* and range query algorithm. Given a query object, when we need to perform the *kNN* query or the range query, the worst case is that we need to perform a linear scan to check the distance between this query object to all other objects using the shortest path query phase of *RC-Oracle* in $O(1)$ time. Since there are total n' objects, the query time is $O(n')$. However, the real query time is smaller than $O(n')$. This is because for most of the cases, we do not need to perform a linear scan (since we have already sorted some distances in order during the construction phase of *RC-Oracle*).

Secondly, we show the error ratio of both the *kNN* and range query algorithm for *RC-Oracle*. We give some definitions first. For simplicity, given a query POI $q \in P$, (1) we let X be a set of POIs which contains the *exact* (i) k nearest POIs of q or (ii) POIs whose distance to q are at most r, calculated using the exact distance on C. Furthermore, given a query POI $q \in P$, (2) we let X' be a set of POIs which contains (i) k nearest POIs of q or (ii) POIs whose distance to q are at most r, calculated using the approximated distance on C returned by *RC-Oracle*. In Figure 1.1 (a), suppose that the exact k nearest POIs ($k = 2$) of a is b, c, i.e., $X = \{b, c\}$. Suppose that our *kNN* query algorithm finds the k nearest POIs ($k = 2$) of a is c, d, i.e., $X' = \{c, d\}$. Recall that we let v_f (resp. v'_f) be the furthest object to q in X (resp. X'), i.e., $|\Pi^*(q, v_f|C)| \leqslant \max_{\forall v \in X} |\Pi^*(q, v|C)|$ (resp. $|\Pi^*(q, v'_f|C)| \leqslant \max_{\forall v' \in X'} |\Pi^*(q, v'|C)|$). We further let w_f (resp. w'_f) be the furthest object to q in X (resp. X') based on the approximated distance on C returned by *RC-Oracle*, i.e., $|\Pi_1(q, w_f|C)| \leqslant \max_{\forall w \in X} |\Pi_1(q, w|C)|$ (resp. $|\Pi_1(q, w'_f|C)| \leqslant \max_{\forall w' \in X'} |\Pi_1(q, w'|C)|$). Then, we have $\alpha = \frac{|\Pi^*(q,v'_f|C)|}{|\Pi^*(q,v_f|C)|} \leqslant \frac{|\Pi_1(q,v'_f|C)|}{|\Pi^*(q,v_f|C)|} \leqslant \frac{|\Pi_1(q,v'_f|C)|}{|\Pi^*(q,w_f|C)|} \leqslant \frac{|\Pi_1(q,w'_f|C)|}{|\Pi^*(q,w_f|C)|} \leqslant \frac{|\Pi_1(q,w'_f|C)|(1+\epsilon)}{|\Pi_1(q,w_f|C)|} \leqslant 1+\epsilon$. The first equality is due to the error ratio of the *kNN* and range query, i.e., $\alpha = \frac{|\Pi^*(q,v'_f|C)|}{|\Pi^*(q,v_f|C)|}$.

The second inequality is because the approximated distance on C returned by *RC-Oracle* is always longer than the exact distance on C, i.e., $|\Pi_1(q, v'_f|C)| \geqslant |\Pi^*(q, v'_f|C)|$. The third inequality is because of the definition of v_f and w_f, i.e., $|\Pi^*(q, v_f|C)| \geqslant |\Pi^*(q, w_f|C)|$. The fourth inequality is because of the definition of v'_f and w'_f, i.e., $|\Pi_1(q, v'_f|C)| \leqslant |\Pi_1(q, w'_f|C)|$. The fifth inequality is because the error ratio of the approximated distance on C returned by *RC-Oracle* is $1+\epsilon$, i.e., $|\Pi_1(q, w_f|C)| \leqslant (1+\epsilon)|\Pi^*(q, w_f|C)|$. The sixth inequality is because of our *kNN* and range query algorithm, i.e., $|\Pi_1(q, w'_f|C)| \leqslant |\Pi_1(q, w_f|C)|$.

We give the proof for *RC-Oracle-A2A* as follows. Since the shortest path query time of *RC-Oracle-A2A* is the same as that of *RC-Oracle*, and *RC-Oracle-A2A* is also a $(1+\epsilon)$-approximate shortest path oracle, its query time and error ratio of both the *kNN* and range query algorithm is the same as that of *RC-Oracle*. □

5.3 Empirical Studies

5.3.1 Experimental Setup

We performed experiments on a Linux machine with 2.20 GHz CPU and 512GB memory. We implemented algorithms in C++. Our experimental setup generally follows the setups in the literature [45, 46, 51, 71, 72]. We conducted experiments with point clouds and *TIN*s as input, separately.

1) Datasets (i) Point cloud datasets: We conducted our experiment based on 34 ($= 5+5+24$) real point cloud datasets in Table 5.2, where the subscript p means a point cloud. (a) *5 Original datasets*: For BH_p and EP_p datasets, they are represented as a point cloud with 8km $\times$ 6km covered region. We can remove outliers by identifying points far from their neighbors. For GF_p, LM_p and RM_p, we first obtained the satellite map from Google Earth [3] with 8km $\times$ 6km covered region, and then used Blender [1] to generate the point cloud. These five original datasets have a resolution of 10m $\times$ 10m [32, 51, 64, 71, 72]. We extracted 500 POIs using OpenStreetMap [71, 72] for these datasets in the P2P query. (b) *5 Small-version datasets*: They are generated using the same region as the original datasets, with a (lower) 70m $\times$ 70m resolution, by following the dataset generation procedure (us-

ing distribution) in studies [51, 71, 72]. This procedure can be found in [81]. (c) *24 Multi-resolution datasets*: They are generated using the same procedure with different numbers of points. (ii) *TIN* datasets: Based on the 34 point cloud datasets, we triangulate [37] them and generate another 34 *TIN* datasets, and use t as the subscript. Storing a point cloud with 2.5M points needs 39MB, but storing a *TIN* converted from this point cloud needs 170MB.

Table 5.2. Point cloud datasets in the study of proximity queries on point clouds

Name	$\|N\|$
Original dataset	
BearHead (BH_{p}) [2, 71, 72]	0.5M
EaglePeak (EP_{p}) [2, 71, 72]	0.5M
GunnisonForest (GF_{p}) [5]	0.5M
LaramieMount (LM_{p}) [8]	0.5M
RobinsonMount (RM_{p}) [10]	0.5M
Small-version dataset	
BH_{p}*-small*	10k
EP_{p}*-small*	10k
GF_{p}*-small*	10k
LM_{p}*-small*	10k
RM_{p}*-small*	10k
Multi-resolution dataset	
BH_{p} multi-resolution	1M, 1.5M, 2M, 2.5M
EP_{p} multi-resolution	1M, 1.5M, 2M, 2.5M
GF_{p} multi-resolution	1M, 1.5M, 2M, 2.5M
LM_{p} multi-resolution	1M, 1.5M, 2M, 2.5M
RM_{p} multi-resolution	1M, 1.5M, 2M, 2.5M
EP_{p}*-small* multi-resolution	20k, 30k, 40k, 50k

2) Algorithms (i) Algorithms that support the shortest path query (and also other proximity queries) on a point cloud (i.e., algorithms for solving the problem studied in this paper): We adapted existing algorithms, originally designed for the problem on *TIN*s, for our problem on point clouds by converting the point cloud to a *TIN* (using the points of the point cloud as the vertices of the *TIN*, and triangulate [37] these vertices to create faces of the *TIN*) [59, 65, 86] so that existing algorithms could be used. Their algorithm names are appended by "*-Adapt*". We have four on-the-fly algorithms, i.e., (a) *CH-Adapt* [25]: the best-known adapted *TIN* exact shortest surface path query algorithm, (b) *Kaul-Adapt* [45]: the best-known adapted *TIN* approximate shortest surface path query algorithm, (c) *Dijk-*

Adapt [46]: the best-known adapted *TIN* approximate shortest network path query algorithm, and (d) *FastFly*: our algorithm. We have four oracles, i.e., (e) *SE-Oracle-Adapt*: the best-known adapted *TIN* oracle [71, 72] for the P2P query on a point cloud, (f) *EAR-Oracle-Adapt*: the best-known adapted *TIN* oracle [43] for the A2A query on a point cloud, (g) *RC-Oracle-Naive*: the naive version of our oracle *RC-Oracle* without shortest paths approximation step, and (h) *RC-Oracle*: our oracle. We compare them in Table 5.3. *RC-Oracle* is the best oracle and algorithm *FastFly* is the best on-the-fly algorithm. For on-the-fly algorithms, we cannot run them in parallel (i.e., using a distributed implementation), since we only execute them once for one source. For oracles, we can run them in parallel for faster processing, since we execute them multiple times for different sources. But, for the comparison fairness for on-the-fly algorithms, we do not run oracles in parallel.

Table 5.3. Comparison of algorithms (support the shortest path query) on a point cloud regarding *RC-Oracle*

Algorithm	**Oracle construction time**		**Oracle size**		**Shortest path query time**		**Error**
Oracle-based algorithm							
SE-Oracle-Adapt [71, 72]	$O(\frac{nN^2}{\epsilon^{2\beta}} + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small
EAR-Oracle-Adapt [43]	$O(\lambda\xi(mN)^2 + \frac{N^3}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi\log(\lambda\xi))$	Medium	Small
RC-Oracle-Naive	$O(nN\log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Tiny	Small
***RC-Oracle* (ours)**	$\boldsymbol{O(\frac{N\log N}{\epsilon} + n\log n)}$	**Small**	$\boldsymbol{O(\frac{n}{\epsilon})}$	**Small**	$\boldsymbol{O(1)}$	**Tiny**	**Small**
On-the-fly algorithm							
CH-Adapt [25]	-	N/A	-	N/A	$O(N^2)$	Large	Small
Kaul-Adapt [45]	-	N/A	-	N/A	$O(\gamma\log\gamma)$	Large	Small
Dijk-Adapt [46]	-	N/A	-	N/A	$O(N\log N)$	Medium	Medium
***FastFly* (ours)**	-	N/A	-	N/A	$\boldsymbol{O(N\log N)}$	**Medium**	**No error**

Remark: $n << N$, h is the compressed partition tree's height, β is the largest capacity dimension [71, 72], λ is the number of highway nodes in one square, ξ is the number of boxes under square root, m is the number of Steiner points on each face, θ is the minimum angle of the face in the *TIN*, l_{max} (resp. l_{min}) is the longest (resp. shortest) length of edges in *TIN*, and $\gamma = \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}$.

(ii) Algorithms that support the shortest path query (and also other proximity queries) on a *TIN* (i.e., algorithms for solving the problem studied by previous studies [43, 71, 72]): We adapted our algorithms on point cloud by converting the *TIN* to a point cloud (using the vertices of the *TIN* as the points of the point cloud) [52] so that our algorithms could be used. We store the point cloud as a data structure in the memory and clear the given *TIN* from the memory. Similarly, we have four on-the-fly algorithms, i.e., (a) *CH* [25], (b) *Kaul* [45], (c) *Dijk* [46], (d) *FastFly-Adapt*: our adapted algorithm. We have four oracles, i.e., (e) *SE-Oracle* [71, 72], (f) *EAR-Oracle* [43], (g) *RC-Oracle-Naive-Adapt*: the adapted naive

version of our oracle without shortest paths approximation step, and (h) *RC-Oracle-Adapt*: our adapted oracle.

3) Query generation We conducted all proximity queries, i.e., (i) shortest path query, (ii) *kNN* query and (iii) range query. (i) For the shortest path query, we issued 100 query instances where for each instance, we randomly chose two points (a) in P for the P2P query on a point cloud or a *TIN*, or (b) on the point cloud (resp. *TIN*) for the A2A query on a point cloud (resp. the AR2AR query on a *TIN*), one as a source and the other as a destination. The average, minimum and maximum results were reported. In the experimental result figures, the vertical bar and the points mean the minimum, maximum and average results. (ii & iii) For *kNN* query and range query, we perform the proximity query algorithm for *RC-Oracle* in Chapter 5.2.5 and a linear scan for other baselines (as described in [72]) using all objects as query objects. In the P2P query on a point cloud or a *TIN*, these objects are POIs in P. In the A2A query on a point cloud (resp. the AR2AR query on a *TIN*), we randomly select 2500 points on the point cloud (resp. *TIN*) as objects. Since we perform linear scans or use the sorted distance stored in M_{path} for proximity query algorithms, the value of k and r will not affect their query time, we set $\mathsf{k} = 3$ and $\mathsf{r} = 1$km.

4) Factors and measurements We studied three factors for the P2P query, namely (i) ϵ (i.e., the error parameter), (ii) n (i.e., the number of POIs), and (iii) N (i.e., the number of points or vertices in a point cloud or *TIN* dataset). We studied one factor ϵ for the A2A query. In addition, we used nine measurements to evaluate the algorithm performance, namely (i) *oracle construction time*, (ii) *memory consumption* (i.e., the space consumption when running the algorithm), (iii) *oracle size*, (iv) *query time* (i.e., the shortest path query time), (v & vi) *kNN* or *range query time*, (vii) *distance error ratio* (i.e., the distance error ratio of the algorithm compared with the exact algorithm), and (viii & xi) *kNN* or *range query error ratio* (i.e., the error ratio of the *kNN* or range query defined in Chapter 5.2.6 (3)).

5.3.2 Experimental Results for *TIN*s

We first study proximity queries on *TIN*s (studied by previous studies [43, 71, 72]) to justify why our proximity queries on *point clouds* are useful in practice. We have the following settings. (1) The distance of the path calculated by *CH* is used for distance error ratio calculation since the path is the exact shortest surface path passing on the *TIN*. (2) *SE-Oracle*,

EAR-Oracle and *RC-Oracle-Naive-Adapt* are impractical on large-version datasets due to their expensive oracle construction time (more than 24 hours), so we (i) compared *SE-Oracle*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt*, *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt* on small-version datasets (with default 50 POIs for the P2P query), and (ii) compared *RC-Oracle-Adapt*, *CH*, *Kaul*, *Dijk* and *FastFly-Adapt* on large-version datasets (with default 500 POIs for the P2P query). (3) The data conversion time from a *TIN* to a point cloud, and then to a point cloud graph of *FastFly-Adapt*, *RC-Oracle-Naive-Adapt* and *RC-Oracle-Adapt* is only counted once (i) in the shortest path query time, the *kNN* and range query time for *FastFly-Adapt*, and (ii) in the oracle construction time for *RC-Oracle-Adapt* and *RC-Oracle-Naive-Adapt*. (4) The data conversion time from a *TIN* to a *TIN* graph of *Dijk* is also only counted once in its shortest path query time, the *kNN* and range query time.

1) Baseline comparisons We study the effect of ϵ and n for the P2P query on a *TIN* in this subsection. We study the effect of N for the P2P query, and the comparisons for the AR2AR query on a *TIN* in [81].

Effect of ϵ for the P2P query on a *TIN*: In Figure 5.5, we tested 6 values of ϵ in $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_p-*small* dataset by setting N to be 10k and n to be 50 for baseline comparisons. Although a *TIN* is given as input, *RC-Oracle-Adapt* performs better than *SE-Oracle*, *EAR-Oracle* and *RC-Oracle-Naive-Adapt* concerning the oracle construction time, oracle size and shortest path query time. The shortest path query time of *FastFly-Adapt* is 100 times smaller than that of *CH* (although *FastFly-Adapt* needs to convert the given *TIN* to a point cloud, and there are no other additional steps for *CH*), since the query region of the path calculated by *FastFly-Adapt* is smaller than that of *CH*. The distance error ratio of *FastFly-Adapt* (i.e., 0.008) is very small compared with that of *CH* (i.e., without error), and much smaller than that of *Dijk* (i.e., 0.1). This means that the shortest distance on a point cloud (resp. the shortest network distance on a *TIN*) is 1.002 (resp. 1.1) times larger than the shortest surface distance on a *TIN*. This motivates us to conduct experiments on point clouds. The *kNN* query error ratio and range query error ratio are all equal to 1, meaning that there is no error (since the distance error ratio is small, $|\Pi^*(q, v'_f|C)| = |\Pi^*(q, v_f|C)|$ in the error ratio of the *kNN* and range query defined in Chapter 5.2.6 (3), although the theoretical error ratios are $1 + \epsilon$ in Theorem 5.2.3), so their results are omitted. We provided

some selected metrics performance figures, the full sets of metrics performance figures in [81].

Effect of n for the P2P query on a *TIN*: In Figure 5.6, we tested 5 values of n in {50, 100, 150, 200, 250} on EP_t dataset by setting N to be 10k and ϵ to be 0.1 for baseline comparisons. In Figure 5.6 (a), as n increases, the construction time of all oracles increases. In Figure 5.6 (b), as n increases, the memory consumption of *RC-Oracle-Adapt* exceeds that of *Dijk* and *FastFly-Adapt*. This is because (i) *RC-Oracle-Adapt* is an oracle which is affected by n, it needs more memory consumption during the oracle construction phase to calculate more shortest paths among these POIs as n increases, but (ii) *Dijk* and *FastFly-Adapt* are on-the-fly algorithms which are not affected by n, their memory consumption only measures the space consumption for calculating one shortest path.

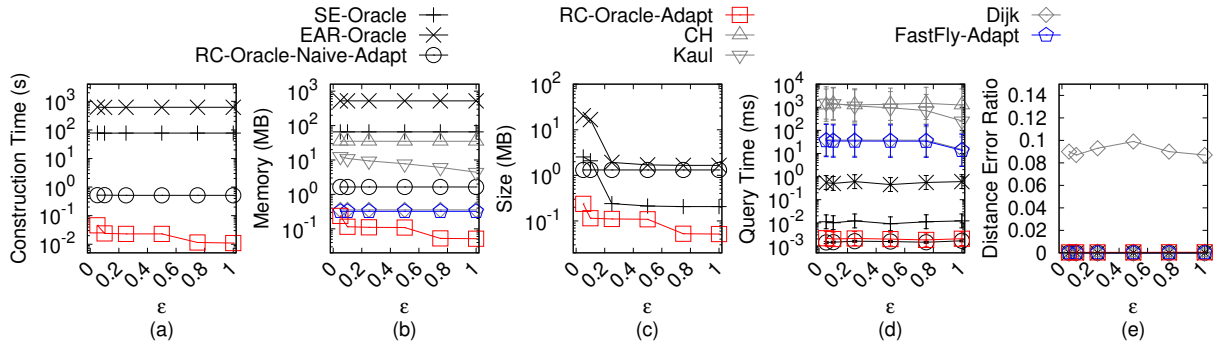


Figure 5.5. Baseline comparisons (effect of ϵ on BH_t-*small TIN* dataset for the P2P query) regarding *RC-Oracle-Adapt*

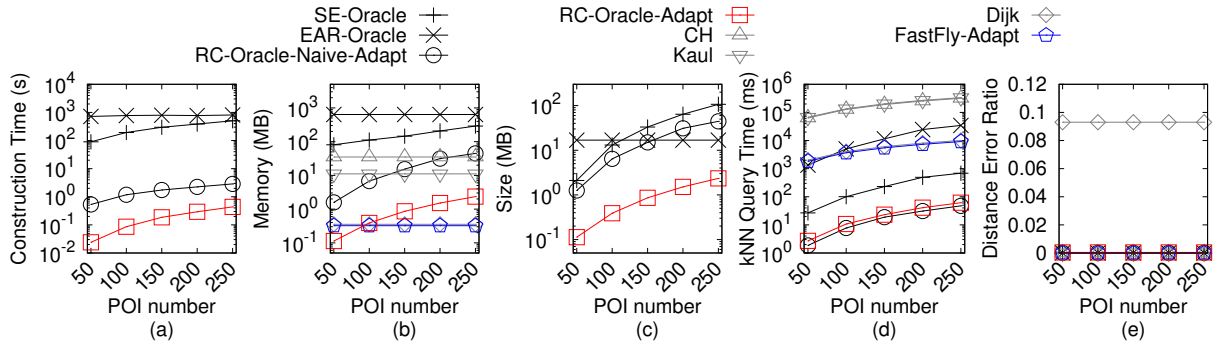


Figure 5.6. Baseline comparisons (effect of n on EP_t-*small TIN* dataset for the P2P query) regarding *RC-Oracle-Adapt*

5.3.3 Experimental Results for Point Clouds

Now, we understand the effectiveness of proximity queries on *point clouds*. In this section, we then study proximity queries on *point clouds* using the algorithms in Table 5.3. We have the following setting. (1) The distance of the path calculated by *FastFly* is used for distance error ratio calculation since the path is the exact shortest path passing on the point cloud. (2) *SE-Oracle-Adapt*, *EAR-Oracle-Adapt* and *RC-Oracle-Naive* are impractical on large-version datasets due to their expensive oracle construction time (more than 24 hours), so we (i) compared *SE-Oracle-Adapt*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive*, *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on small-version datasets (with default 50 POIs for the P2P query), and (ii) compared *RC-Oracle*, *CH-Adapt*, *Kaul-Adapt*, *Dijk-Adapt* and *FastFly* on large-version datasets (with default 500 POIs for the P2P query). (3) Since algorithm *FastFly* uses the Dijkstra algorithm on a point cloud graph, it is the same as the shortest path algorithm on a general graph (converted from the given point cloud), we do not (and there is no need to) compare them in the experiment. But, there is no existing study discussing how to build a point cloud graph from the point cloud. We fill this gap by proposing algorithm *FastFly*. (4) The data conversion time from a point cloud to a point cloud graph of *FastFly*, *RC-Oracle-Naive* and *RC-Oracle* is only counted once (i) in the shortest path query time, the *kNN* and range query time for *FastFly*, and (ii) in the oracle construction time for *RC-Oracle* and *RC-Oracle-Naive*. (5) The data conversion time from a point cloud to a *TIN* is also only counted once (i) in the shortest path query time, the *kNN* and range query time for *CH-Adapt* and *Kaul-Adapt*, and (ii) in the oracle construction time for *SE-Oracle-Adapt* and *EAR-Oracle-Adapt*. (6) The data conversion time from a point cloud to a *TIN*, and then to a *TIN* graph of *Dijk-Adapt* is also only counted once in its shortest path query time, the *kNN* and range query time.

1) Baseline comparisons We study the effect of ϵ, n and N for the P2P query on a point cloud, and the comparisons for the A2A query on a point cloud in this subsection.

Effect of ϵ for the P2P query on a point cloud: In Figure 5.7, we tested 6 values of ϵ in $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on EP_p*-small* dataset by setting N to be 10k and n to be 50 for baseline comparisons. (i) The oracle construction time and memory consumption of *RC-Oracle* is smaller than the best-known adapted *TIN* oracle *SE-Oracle-Adapt*, since *SE-Oracle-Adapt* has the *loose criterion for algorithm earlier termination* drawback, it cannot ter-

minate the *SSAD* algorithm earlier, so it requires more time and memory. The oracle size of *RC-Oracle* is smaller than *SE-Oracle-Adapt*, since *RC-Oracle* can terminate the *SSAD* algorithm earlier and store fewer paths. The shortest path query time of *RC-Oracle* is smaller than *SE-Oracle-Adapt*, since *RC-Oracle*'s shortest path query time is $O(1)$, while the time is $O(h^2)$ for *SE-Oracle-Adapt*. (ii) *RC-Oracle* performs better than other on-the-fly algorithms concerning the shortest path query time since it is an oracle. (iii) Algorithm *FastFly* performs better than other on-the-fly algorithms concerning the shortest path query time since it calculates the shortest path passing on a point cloud. (iv) In Figures 5.7 (a) and (b), regarding the oracle construction time and memory consumption, the variation of ϵ has a large effect on *RC-Oracle*, but due to the log scale used in the experimental figures, the effect is not obvious (e.g., the oracle construction time and memory consumption of *RC-Oracle* with $\epsilon = 1$ are both up to 5 times smaller than that of the case when $\epsilon = 0.05$). The variation of ϵ has a small effect on *SE-Oracle-Adapt* and *EAR-Oracle-Adapt*, because even when ϵ is large, they cannot terminate the *SSAD* algorithm earlier for most of the cases due to their *loose criterion for algorithm earlier termination* drawback. The variation of ϵ has no effect on *RC-Oracle-Naive* since it is independent of ϵ. (v) The *kNN* and range query time of *RC-Oracle* are very small. (vi) The distance error ratio of *RC-Oracle* is close to 0.

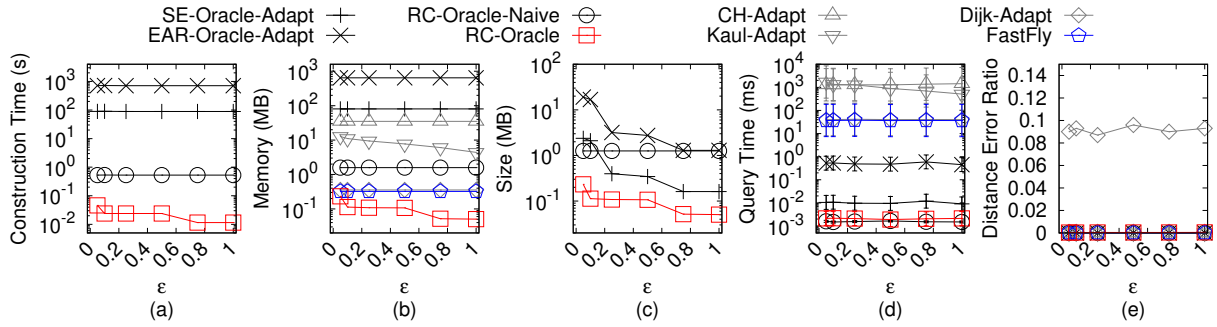


Figure 5.7. Baseline comparisons (effect of ϵ on EP_p-*small* point cloud dataset for the P2P query) regarding *RC-Oracle*

Effect of n for the P2P query on a point cloud: In Figure 5.8, we tested 5 values of n in {500, 1000, 1500, 2000, 2500} on GF_p dataset by setting N to be 0.5M and ϵ to be 0.25 for baseline comparisons. Since *RC-Oracle* is an oracle, its shortest path query time is small.

Effect of N (scalability test) for the P2P query on a point cloud: In Figure 5.9, we

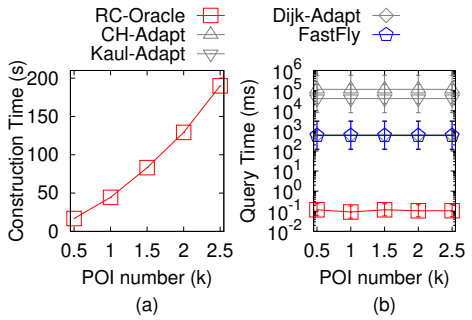


Figure 5.8. Baseline comparisons (effect of n on GF_p point cloud dataset for the P2P query) regarding *RC-Oracle*

tested 5 values of N in {0.5M, 1M, 1.5M, 2M, 2.5M} on LM_p dataset by setting n to be 500 and ϵ to be 0.25 for baseline comparisons. The oracle construction time of *RC-Oracle* is only 200s $\approx$ 3.2 min for a point cloud with 2.5M points and 500 POIs, this shows the scalable of *RC-Oracle*. The *kNN* query time of *RC-Oracle* ($O(n' \cdot n' = n'^2)$ in Theorem 5.2.3, since we use all objects as query objects) is the smallest.

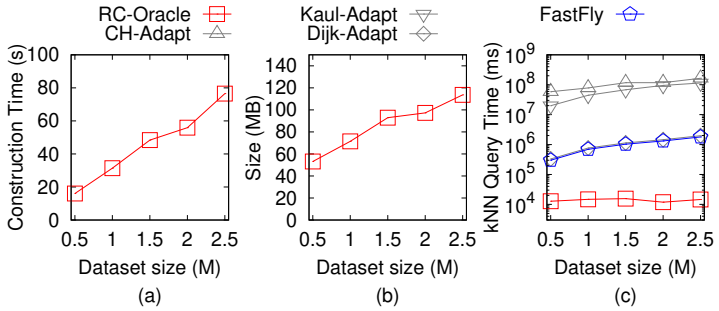


Figure 5.9. Baseline comparisons (effect of N on LM_p point cloud dataset for the P2P query) regarding *RC-Oracle*

A2A query on a point cloud: In Figure 5.10, we tested the A2A query by varying ϵ in {0.05, 0.1, 0.25, 0.5, 0.75, 1} and setting N to be 10k on a small-version of EP_p dataset. We adapt *SE-Oracle-Adapt* (resp. *RC-Oracle-Naive*) to be *SE-Oracle-Adapt-A2A* (resp. *RC-Oracle-Naive-A2A*) in a similar way to *RC-Oracle-A2A*. Although *EAR-Oracle-Adapt* is regarded as the best-known adapted *TIN* oracle in the A2A query on a point cloud, *RC-Oracle-A2A* still performs more efficiently than it due to the *loose criterion for algorithm earlier termination* drawback of *EAR-Oracle-Adapt*.

2) Ablation study for the P2P query on a point cloud We denote *SE-Oracle-FastFly-Adapt* (resp. *EAR-Oracle-FastFly-Adapt*) to be another adapted oracle of *SE-Oracle-Adapt*

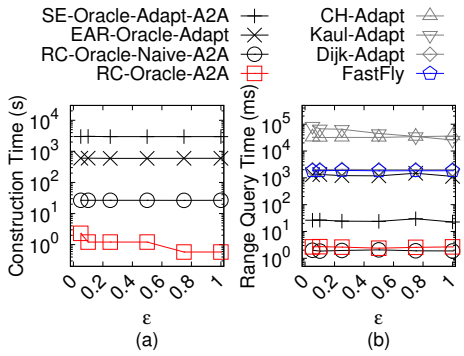


Figure 5.10. Baseline comparisons on EP_{p} point cloud dataset for the A2A query regarding *RC-Oracle-A2A*

(resp. *EAR-Oracle-Adapt*) that uses algorithm *FastFly* to directly calculate the shortest path passing on a point cloud without converting it to a *TIN*. In Figure 5.11, we tested 6 values of ϵ in $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on RM_{p} dataset by setting N to be 0.5M and n to be 500 for ablation study among *SE-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt* and *RC-Oracle*, such that they only differ by the oracle construction. The oracle construction time, oracle size and shortest path query time of *RC-Oracle* perform better than the two baselines.

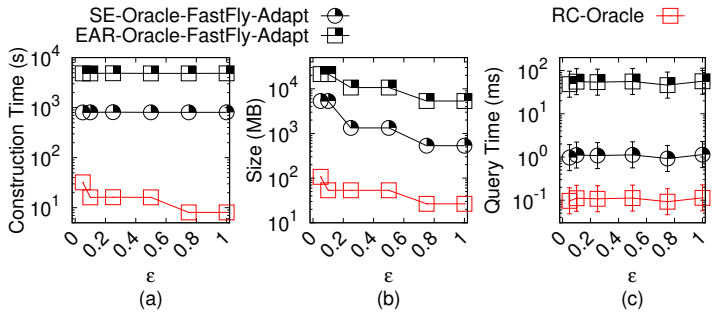


Figure 5.11. Ablation study on RM_{p} point cloud dataset for the P2P query regarding *RC-Oracle*

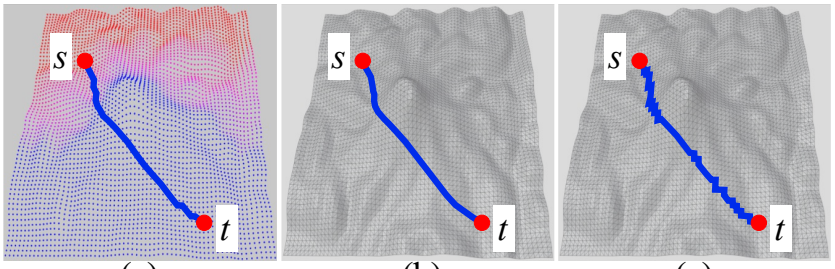
3) Comparisons with other proximity queries oracle and algorithm on a point cloud We compared *RC-Oracle* using our efficient proximity query algorithm with *SU-Oracle-Adapt* (i.e., the oracle designed for the *kNN* query) and *RC-Oracle* using the naive proximity query algorithm in [81]. For a point cloud with 2.5M points and 500 objects, the *kNN* query time of *RC-Oracle* using our efficient proximity query algorithm is 12.5s, but the time is 1520s $\approx$ 25 min for *SU-Oracle-Adapt*, and 25s for *RC-Oracle* using the naive proximity query algorithm (since the shortest path query time of *RC-Oracle* is $\text{O}(1)$, and we do not need to perform linear scans over all the objects in our efficient proximity query

algorithm).

4) Case study We performed a case study on the wildfire evacuation in Santa Monica Mountains National Recreation Area in Chapter 1. When the wildfire just starts, we capture the newest point cloud, and then we construct the oracle. In Figure 1.1 (a), when the number of tourists is large and the capacity of each shelter/hotel is limited, only finding the shortest path from each viewing platform to its nearest shelter/hotel is not enough, we need to find the shortest paths (in blue and yellow lines) from POI a (one of the viewing platforms) to its k-nearest POIs b to d (k-nearest shelters/hotels). In Figures 1.1 (d), POIs b and c are the k-nearest POIs to POI a where $k = 2$. The average distance between the viewing platforms and hotels in Santa Monica Mountains National Recreation Area is 11.2km [4], and the average car driving speed is 90km/h, so the evacuation can be finished in $7.4 (= \frac{11.2\text{km} \times 60\text{min/h}}{90\text{km/h}})$ min. Our experimental results show that for a point cloud with 2.5M points and 500 POIs (250 viewing platforms and 250 shelters/hotels), the oracle construction time for (i) *RC-Oracle* is 200s $\approx$ 3.2 min and (ii) the best-known adapted *TIN* oracle *SE-Oracle-Adapt* is 78,000s $\approx$ 21.7 hours. The query time for calculating 10 nearest shelters/hotels of each viewing platform for (i) *RC-Oracle* is 6s, (ii) *SE-Oracle-Adapt* is 75s, (iii) the best-known adapted *TIN* on-the-fly approximate shortest surface path query algorithm *Kaul-Adapt* is 80,500s $\approx$ 22.5 hours and (iv) *FastFly* is 2,000s $\approx$ 33 min. Thus, *RC-Oracle* is the best one in the evacuation since 3.2 min + 6s $<$ *7.4 min* $<$ 33 min $<$ 21.7 hours + 75s $<$ 22.5 hours. *RC-Oracle* also supports real-time responses, i.e., it can construct the oracle in 0.4s and answer the *kNN* query and range query in both 7 ms on a point cloud with 10k points and 250 POIs.

5) Path visualization Given a point cloud, Figure 5.12 (a) shows the shortest path passing on a point cloud calculated by algorithm *FastFly*, and Figure 5.12 (b) (resp. Figure 5.12 (c)) shows the shortest surface (resp. network) path passing on a *TIN* (constructed by the point cloud) calculated by algorithm *CH-Adapt* (resp. *Dijk-Adapt*). The paths in Figures 5.12 (a) and (b) are similar, but calculating the former path is much faster than the latter path, since the query region of the former path is smaller than the latter path. The path in Figure 5.12 (c) has a larger error than the path in Figure 5.12 (a). Given a *TIN*, Figure 5.12 (a) shows the shortest path passing on a point cloud (constructed by the *TIN*) calculated by algorithm *FastFly-Adapt*, and Figure 5.12 (b) (resp. Figure 5.12 (c)) shows the

shortest surface (resp. network) path passing on a *TIN* calculated by algorithm *CH* (resp. *Dijk*).



Remark: (a) The shortest path passing on a point cloud, the shortest (b) surface and (c) network path passing on a *TIN*.

Figure 5.12. The shortest path passing on a point cloud, the shortest surface and network path passing on a *TIN*

6) Summary Concerning the oracle construction time, oracle size and proximity (e.g., *kNN*) query time, *RC-Oracle* with the proximity query algorithm is up to 390 times, 30 times and 12 times better than the best-known adapted *TIN* oracle *SE-Oracle-Adapt* for the P2P query on a point cloud, respectively. *RC-Oracle-A2A* with the proximity query algorithm is up to 500 times, 140 times and 50 times better than the best-known adapted *TIN* oracle *EAR-Oracle-Adapt* for the A2A query on a point cloud, respectively. For the P2P query on a point cloud with 2.5M points and 500 POIs, these values for *RC-Oracle* are 200s $\approx$ 3.2 min, 50MB and 12.5s, but the values for *SE-Oracle-Adapt* are 78,000s $\approx$ 21.7 hours, 1.5GB and 150s, respectively. For the A2A query on a point cloud with 100k points and 5000 objects, these values for *RC-Oracle-A2A* are 100s $\approx$ 1.6 min, 150MB and 0.25s, but the values for *EAR-Oracle-Adapt* are 50,000s $\approx$ 13.9 hours, 21GB and 25s, respectively.

CHAPTER 6

CONCLUSION

In this thesis, we present three studies. Firstly, we study shortest path queries on a weighted *TIN* using an on-the-fly algorithm. Secondly, we study shortest path queries on an updated *TIN* using an oracle. Thirdly, we study proximity queries on a point cloud using an oracle.

For the first study, we propose an efficient $(1+\epsilon)$-approximate on-the-fly shortest path algorithm of two points on a weighted *TIN*, called algorithm *Roug-Ref*. Our experimental results show that algorithm *Roug-Ref* is up to 1,630 times and 40 times better than the best-known algorithm on a weighted *TIN* concerning the query time and memory usage, respectively. In our future work, we can investigate how to introduce a new pruning step (by considering other geometric information of the weighted *TIN*) to further reduce the algorithm's query time. Currently, we consider Snell's law, face weight, internal angle and edges' length of the weighted *TIN*. Some other geometric information about the weighted *TIN*, e.g., the shortest network path that passes on the edges of the weighted *TIN* may be useful for additional pruning since it is computationally efficient.

For the second study, we propose an efficient $(1+\epsilon)$-approximate shortest path oracle of a set of POIs on an updated *TIN*, called *UP-Oracle*. We adapt *UP-Oracle* to the case if POIs are not given as input. We also adapt *UP-Oracle* for handling subsequent changes and to a multi-layer structure. Our experimental results show that when POIs are given (resp. not given) as input, *UP-Oracle* is up to 88 times, 12 times and 3 times (resp. 15 times, 50 times and 100 times) better than the best-known oracle on a *TIN* concerning the oracle update time, output size and shortest path query, respectively. In our future work, we can introduce a new pruning step (by reducing the likelihood of using algorithm *SSAD* when updating *UP-Oracle* through reducing the disk radius in the *non-updated TIN shortest path intact* property) to further reduce the oracle update time. Currently, the disk radius is half of the distance between the two endpoints, and it is fixed for all cases. We may use a smaller value for different cases for efficient pruning.

For the third study, we propose an efficient $(1+\epsilon)$-approximate shortest path oracle of a set of POIs on a point cloud, called *RC-Oracle*. We adapt *RC-Oracle* to the case if POIs are not given as input. With the assistance of *RC-Oracle*, we also propose algorithms for answering other proximity queries, i.e., the *kNN* and range query. Our experimental results show that when POIs are given (resp. not given) as input, *RC-Oracle* with the proximity query algorithm is up to 390 times, 30 times and 12 times (resp. 500 times, 140 times and 50 times) better than the best-known adapted *TIN* oracle concerning the oracle construction time, oracle size and proximity (e.g., *kNN*) query time, respectively. In our future work, we can investigate how to build a novel oracle designed for the *kNN* and range query to avoid storing unnecessary shortest path information in the oracle. Currently, we first need to build the whole *RC-Oracle*, and then we can answer *kNN* and range queries based on *RC-Oracle*. We may only store the nearest neighbor *POI* information of each *POI* in the oracle, and then use this information to gradually expand the searching region to answer *kNN* and range queries.

Apart from the future work related to these studies, we can also use large language models for the *TIN* or the point cloud shortest path query as future work. To the best of our knowledge, there is no existing study about this. We may use large language models for faster path querying. Then, in the case of a disaster, we can perform the evacuation more efficiently.

PUBLICATION

- **Yinzhao Yan** and Raymond Chi-Wing Wong,

  "*Efficient Proximity Queries on Simplified Height Maps*",

  the 2026 ACM Conference on Management of Data (SIGMOD), Bengaluru, India on 31 May - 5 June, 2026

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

  "*An Efficiently Updatable Path Oracle for Terrain Surfaces*",

  IEEE Transactions on Knowledge and Data Engineering (TKDE), 2024

- **Yinzhao Yan** and Raymond Chi-Wing Wong,

  "*Efficient Shortest Path Queries on 3D Weighted Terrain Surfaces for Moving Objects*",

  the 25th IEEE International Conference on Mobile Data Management (MDM 2024), Brussels, Belgium on 24-27 June, 2024 (Acceptance 21/78 = 26.92%),

  Selected as the best paper (out of 21 accepted papers)

- **Yinzhao Yan** and Raymond Chi-Wing Wong,

  "*Proximity Queries on Point Clouds using Rapid Construction Path Oracle*",

  the 2024 ACM Conference on Management of Data (SIGMOD), Santiago, Chile on 9-15 June, 2024

- **Yinzhao Yan** and Raymond Chi-Wing Wong,

  "*Path Advisor: A Multi-Functional Campus Map Tool for Shortest Path*",

  the 47th International Conference on Very Large Data Bases (VLDB'21), Copenhagen, Denmark on 16-20 Aug, 2021

REFERENCES

[1] Blender. https://www.blender.org.

[2] Data geocomm. http://data.geocomm.com.

[3] Google earth. https://earth.google.com/web.

[4] Google map. https://www.google.com/maps.

[5] Gunnison national forest. https://gunnisoncrestedbutte.com/visit/places-to-go/parks-and-outdoors/gunnison-national-forest.

[6] January 2025 southern california wildfires. https://en.wikipedia.org/wiki/January_2025_Southern_California_wildfires.

[7] Landslides are among the hazards emerging as la-area wildfires scar terrain. https://abcnews.go.com/US/hazards-remain-southern-california-after-wildfires-subside/story?id=117506329.

[8] Laramie mountain. https://www.britannica.com/place/Laramie-Mountains.

[9] Metaverse. https://about.facebook.com/meta.

[10] Robinson mountain. https://www.mountaineers.org/activities/routes-places/robinson-mountain.

[11] Snell's law in vector form. https://physics.stackexchange.com/questions/435512/snells-law-in-vector-form.

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

[13] 2018 Anchorage earthquake, 2025. https://www.usgs.gov/news/featured-story/2018-anchorage-earthquake.

[14] Gujarat earthquake, 2001, 2025. https://www.actionaidindia.org/emergency/gujarat-earthquake-2001.

[15] Mar 11, 2011: Tohoku earthquake and Tsunami, 2025. https://education.nationalgeographic.org/resource/tohoku-earthquake-and-tsunami.

[16] Mars 2020 mission perseverance rover brains, 2025. https://mars.nasa.gov/mars2020/spacecraft/rover/brains.

[17] Mars 2020 mission perseverance rover communications, 2025. https://www.statista.com/chart/24232/life-cycle-costs-of-mars-missions.

[18] Moderate mag. 4.1 earthquake - 6.3 km northeast of sierre, valais, switzerland, 2025. https://www.volcanodiscovery.com/earthquakes/quake-info/1451397/mag4quake-Oct-24-2016-Leukerbad-VS.html.

[19] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM Conference on Management of Data (SIGMOD)*, pages 349–360, 2013.

[20] L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.-R. Sack. An ε-approximation algorithm for weighted shortest paths on polyhedral surfaces. In *Workshop on Algorithm Theory (WAT)*, 1998.

[21] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry (DGG)*, 9(1):81–100, 1993.

[22] C. Amante and B. W. Eakins. Etopo1 arc-minute global relief model: procedures, data sources and analysis. 2009.

[23] G. Antova. Application of areal change detection methods using point clouds data. In *IOP Conference Series: Earth and Environmental Science*, volume 221, 2019.

[24] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)*, 42(1):67–90, 1995.

[25] J. Chen and Y. Han. Shortest paths on a polyhedron. In *Symposium on Computational Geometry (SOCG)*, page 360–369, 1990.

[26] M. Chen, R. A. Chowdhury, V. Ramachandran, D. L. Roche, and L. Tong. Priority queues and dijkstra's algorithm. 2007.

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

[28] J. A. Crisp, M. Adler, J. R. Matijevic, S. W. Squyres, R. E. Arvidson, and D. M. Kass. Mars exploration rover mission. *Journal of Geophysical Research: Planets (JGRP)*, 108, 2003.

[29] G. Das and G. Narasimhan. A fast algorithm for constructing sparse Euclidean spanners. In *Symposium on Computational Geometry (SOCG)*, pages 132–139, 1994.

[30] J.-L. De Carufel, C. Grimm, A. Maheshwari, M. Owen, and M. Smid. A note on the unsolvability of the weighted region shortest path problem. *Computational Geometry*, 47(7):724–727, 2014.

[31] K. Deng, H. T. Shen, K. Xu, and X. Lin. Surface k-nn query processing. In *IEEE International Conference on Data Engineering (ICDE)*, pages 78–78, 2006.

[32] K. Deng and X. Zhou. Expansion-based algorithms for finding single pair shortest path on surface. In *International Workshop on Web and Wireless Geographical Information Systems (W2GIS)*, pages 151–166, 2004.

[33] K. Deng, X. Zhou, H. T. Shen, Q. Liu, K. Xu, and X. Lin. A multi-resolution surface distance model for k-nn query processing. *VLDB Journal (VLDBJ)*, 17(5):1101–1119, 2008.

[34] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[35] H. N. Djidjev and C. Sommer. Approximate distance queries for weighted polyhedral surfaces. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 579–590, 2011.

[36] D. Eriksson and E. Shellshear. Approximate distance queries for path-planning in massive point clouds. In *IEEE International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, volume 2, pages 20–28, 2014.

[37] H. Feng, Y. Song, and L. De Floriani. Critical features tracking on triangulated irregular networks by a scale-space method. In *ACM International Conference on Advances in Geographic Information Systems (ICAGIS)*, pages 54–66, 2024.

[38] C. Goerzen, Z. Kong, and B. Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems (JIRS)*, 57(1):65–100, 2010.

[39] A. V. Goldberg and C. Harrelson. Computing the shortest path: a search meets graph theory. In *Symposium on Discrete Algorithms (SODA)*, volume 5, pages 156–165, 2005.

[40] A. Gupta, R. Krauthgamer, and J. R. Lee. Bounded geometries, fractals, and low-distortion embeddings. In *IEEE Symposium on Foundations of Computer Science (SFCS)*, pages 534–543, 2003.

[41] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics (SSC)*, 4(2):100–107, 1968.

[42] T. Hayashi, T. Akiba, and K.-i. Kawarabayashi. Fully dynamic shortest-path distance query acceleration on massive networks. In *International Conference on Information and Knowledge Management (CIKM)*, pages 1533–1542, 2016.

[43] B. Huang, V. J. Wei, R. C.-W. Wong, and B. Tang. Ear-oracle: on efficient indexing for distance queries between arbitrary points on terrain surface. In *ACM Conference on Management of Data (SIGMOD)*, volume 1, pages 1–26, 2023.

[44] S. Kapoor. Efficient computation of geodesic shortest paths. In *ACM Symposium on Theory of Computing (STC)*, pages 770–779, 1999.

[45] M. Kaul, R. C.-W. Wong, and C. S. Jensen. New lower and upper bounds for shortest distance queries on terrains. In *International Conference on Very Large Data Bases (VLDB)*, volume 9, pages 168–179, 2015.

[46] M. Kaul, R. C.-W. Wong, B. Yang, and C. S. Jensen. Finding shortest paths on terrains by killing two birds with one stone. In *International Conference on Very Large Data Bases (VLDB)*, volume 7, pages 73–84, 2013.

[47] T. Kawamura, J. F. Clinton, G. Zenhäusern, S. Ceylan, A. C. Horleston, N. L. Dahmen, C. Duran, D. Kim, M. Plasman, S. C. Stähler, et al. S1222a—the largest marsquake detected by insight. *Geophysical Research Letters (GRL)*, 50(5), 2023.

[48] M. Lanthier. *Shortest path problems on polyhedral surfaces*. PhD thesis, Carleton University, 2000.

[49] M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica*, 30(4):527–562, 2001.

[50] S. H. Lee, S. M. Im, and I. S. Hwang. Quartic functional equations. *Journal of Mathematical Analysis and Applications (JMAA)*, 307(2):387–394, 2005.

[51] L. Liu and R. C.-W. Wong. Finding shortest path on land surface. In *ACM Conference on Management of Data (SIGMOD)*, pages 433–444, 2011.

[52] H. Masuda and J. He. Tin generation and point-cloud compression for vehicle-based mobile mapping systems. *Advanced Engineering Informatics (AEI)*, 29(4):841–850, 2015.

[53] B. Max and W. Emil. Principles of optics. 1959.

[54] J. S. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM Journal on Computing (JOC)*, 16(4):647–668, 1987.

[55] H. K. Ng, H. W. Leong, and N. L. Ho. Efficient algorithm for path-based range query in spatial databases. In *IEEE International Database Engineering and Applications Symposium (IDEAS)*, pages 334–343, 2004.

[56] B. Padlewska. Connected spaces. *Formalized Mathematics*, 1(1):239–244, 1990.

[57] R. Pallardy. 2010 Haiti earthquake, 2025. https://www.britannica.com/event/2010-Haiti-earthquake.

[58] K. Pletcher and J. P. Rafferty. Sichuan earthquake of 2008, 2025. https://www.britannica.com/event/Sichuan-earthquake-of-2008.

[59] S. Pütz, T. Wiemann, J. Sprickerhof, and J. Hertzberg. 3D navigation mesh generation for path planning in uneven terrain. *IFAC-PapersOnLine*, 49(15):212–217, 2016.

[60] Y. Rezaei and S. Lee. Sat2pc: Generating building roof's point cloud from a single 2d satellite images. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 221–230, 2023.

[61] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 652–663, 2009.

[62] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. In *International Conference on Very Large Data Bases (VLDB)*, volume 2, pages 1210–1221, 2009.

[63] N. P. Service. Santa monica mountains, 2025.

[64] C. Shahabi, L.-A. Tang, and S. Xing. Indexing land surface for efficient knn query. In *International Conference on Very Large Data Bases (VLDB)*, volume 1, pages 1020–1031, 2008.

[65] B. Sober, R. Ravier, and I. Daubechies. Approximating the riemannian metric from point clouds via manifold moving least squares. *arXiv preprint arXiv:2007.09885*, 2020.

[66] Spatial. Lidar scanning with spatial's ios app, 2022. https://support.spatial.io/hc/en-us/articles/360057387631-LiDAR-Scanning-with-Spatial-s-iOS-App.

[67] A. Suppasri, K. Pakoksung, I. Charvet, C. T. Chua, N. Takahashi, T. Ornthammarath, P. Latcharote, N. Leelawat, and F. Imamura. Load-resistance analysis: an alternative approach to tsunami damage assessment applied to the 2011 great east japan tsunami. *Natural Hazards and Earth System Sciences (NHESS)*, 19(8):1807–1822, 2019.

[68] F. Tauheed, L. Biveinis, T. Heinis, F. Schurmann, H. Markram, and A. Ailamaki. Accelerating range queries for brain simulations. In *IEEE International Conference on Data Engineering (ICDE)*, pages 941–952, 2012.

[69] N. Tran, M. J. Dinneen, and S. Linz. Close weighted shortest paths on 3d terrain surfaces. In *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 597–607, 2020.

[70] V. J. Wei, R. C.-W. Wong, and C. Long. Architecture-intact oracle for fastest path and time queries on dynamic spatial networks. In *ACM Conference on Management of Data (SIGMOD)*, pages 1841–1856, 2020.

[71] V. J. Wei, R. C.-W. Wong, C. Long, and D. M. Mount. Distance oracle on terrain surface. In *ACM Conference on Management of Data (SIGMOD)*, pages 1211–1226, 2017.

[72] V. J. Wei, R. C.-W. Wong, C. Long, D. M. Mount, and H. Samet. Proximity queries on terrain surface. *ACM Transactions on Databases Systems (TODS)*, 2022.

[73] V. J. Wei, R. C.-W. Wong, C. Long, D. M. Mount, and H. Samet. On efficient shortest path computation on terrain surface: A direction-oriented approach. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, (1):1–14, 2024.

[74] E. K. Xidias. On designing near-optimum paths on weighted regions for an intelligent vehicle. *International Journal of Intelligent Transportation Systems Research (IJITS)*, 17(2):89–101, 2019.

[75] S.-Q. Xin and G.-J. Wang. Improving chen and han's algorithm on the discrete geodesic problem. *ACM Transactions on Graphics (TOG)*, 28(4):1–8, 2009.

[76] S. Xing, C. Shahabi, and B. Pan. Continuous monitoring of nearest neighbors on land surface. In *International Conference on Very Large Data Bases (VLDB)*, volume 2, pages 1114–1125, 2009.

[77] D. Yan, Z. Zhao, and W. Ng. Monochromatic and bichromatic reverse nearest neighbor queries on land surfaces. In *International Conference on Information and Knowledge Management (CIKM)*, pages 942–951, 2012.

[78] Y. Yan and R. C.-W. Wong. Path advisor: a multi-functional campus map tool for shortest path. In *International Conference on Very Large Data Bases (VLDB)*, volume 14, pages 2683–2686, 2021.

[79] Y. Yan and R. C.-W. Wong. Efficient shortest path queries on 3D weighted terrain surfaces for moving objects. In *IEEE International Conference on Mobile Data Management (MDM)*, 2024.

[80] Y. Yan and R. C.-W. Wong. Efficient shortest path queries on 3D weighted terrain surfaces for moving objects (technical report). 2024. https://github.com/yanyinzhao/WeightedTerrainCode/blob/master/TechnicalReport.pdf.

[81] Y. Yan and R. C.-W. Wong. Proximity queries on point cloud using rapid construction path oracle (technical report). 2024. https://github.com/yanyinzhao/PointCloudPathCode/blob/master/TechnicalReport.pdf.

[82] Y. Yan and R. C.-W. Wong. Proximity queries on point clouds using rapid construction path oracle. In *ACM Conference on Management of Data (SIGMOD)*, volume 2, pages 1–26, 2024.

[83] Y. Yan and R. C.-W. Wong. Efficient proximity queries on simplified height maps. In *ACM Conference on Management of Data (SIGMOD)*, volume 3, pages 1–26, 2026.

[84] Y. Yan, R. C.-W. Wong, and C. S. Jensen. An efficiently updatable path oracle for terrain surfaces. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, (1):1–14, 2024.

[85] Y. Yan, R. C.-W. Wong, and C. S. Jensen. An efficiently updatable path oracle for terrain surfaces (technical report). 2024. https://github.com/yanyinzhao/UpdatedStructureTerrainCode/blob/master/TechnicalReport.pdf.

[86] H. Yu, J. J. Zhang, and Z. Jiao. Geodesics on point clouds. *Mathematical Problems in Engineering (MPE)*, 2014, 2014.