Enabling Advanced Path-Finding on Terrains and in Spatial Networks

Manohar Kaul

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark
Enabling Advanced Path-Finding on Terrains and in Spatial Networks

A Dissertation
Presented to the Faculty of Science and Technology of Aarhus University in Partial Fulfillment of the Requirements for the PhD Degree

by
Manohar Kaul
July 4, 2014
Abstract

Large scale collection of spatial data via mobile sensors allows us to better model the complex phenomena in our physical surroundings. This capability puts a strain on existing data storage infrastructures; and although the resulting models are much richer and closer to reality, these improvements come with the cost of added complexity. This thesis addresses needs to improve the accuracy, effectiveness, and efficiency of important spatial queries on new, information-rich, and complex models. Specifically, it addresses the need for speeding up existing spatial queries on 3D surfaces, for efficient spatial joins between 3D and 2D datasets, and for maintaining efficiently the up-to-date fastest paths for a large number of users in a dynamic spatial network.

First, we propose lower bounds for the shortest surface distance, which in turn improves the performance of many important queries to which shortest surface distance computation is inherent. We propose two approaches. In the first, the lower bound is a ratio of the shortest network distance computed on the graph representation of the 3D triangulation. The second approach proposes further segmentation of existing triangles/faces so that a new surface face-crossing path is used to compute an even tighter lower bound. Experiments assist us in better understanding the relationship between lower bound tightness and query performance.

Next, we present a framework that generates a 3D spatial network by conducting a spatial join between a massive 3D point-cloud (terabytes) and a 2D spatial network (gigabytes). In a filtering phase, the 3D points close to the 2D road network are identified. Then, the subsequent lifting phase triangulates these to obtain elevation values of points in the 2D road network. We study two filtering approaches. The first scans the 3D point-cloud in a single pass and constructs the 3D road network on the fly, while the second approach reads the 3D points as disk blocks in accordance to a locality-preserving space-filling curve. Results from empirical studies with real-world data offer insight into the effectiveness and efficiency of the two approaches.

Finally, we propose a scalable distributed system that maintains a large number of continuous fastest-path queries on a large spatial network, providing users with guaranteed approximate fastest paths to their destinations, while facing heavy real-time traffic updates. The solution assumes that the dynamic real-valued edge weights are bounded in a manner consistent with real-world data, which enables a theoretical guarantee on the travel time. Comprehensive experiments suggest that the proposed methods are accurate, efficient, robust, and scalable.
Resumé

Omfattende indsamling af geografiske data via mobile sensorer muliggør mere nøjagtig modellering af komplekse fænomen i vores omgivelser. Denne mulighed belaster imidlertid eksisterende infrastrukturer til håndtering af data. Afhandlingen gens bidrag understøtter behov for at forbedre nøjagtigheden, nytteeffekten og effektiviteten af vigtige geografiske forespørgsler på nye, informationsrike og komplekse modeller. Mere specifikt understøtter den behovet for at effektivisere eksisterende geografiske forespørgsler på 3D overflader, for effektiv korellering af 3D og 2D data og for effektivt at kunne vedligeholde information om de aktuelt hurtigste ruter for et stort antal brugere i dynamiske vejnetværk.


Dernæst beskriver afhandlingen teknikker der kan generere et 3D vejnetværk ved at korrelere en 3D punktsky (terabytes) og et 2D netværk (gigabytes). Først identificeres de 3D punkter, der er tæt på vejnetværket i 2D. Efterfølgende løftes 2D vejenetværket til en overflade, der er skabt ved at triangulere disse 3D punkter. De relevante 3D punkter kan findes på to måder. En måde er at scanne alle 3D punkter en enkelt gang og at bygge 3D vejenetværket undervejs. En anden måde er at tilgå 3D punkter i disk-blokke, der er ordnet efter en afstandsbevarende afbildning. Resultater fra empiriske studier med virkelige data giver indsigt i de to fremgangsmåders performanceegenskaber.

Endelig præsenterer afhandlingen et skalerbart distribueret system, der er i stand til at vedligeholde resultaterne et stort antal kontinueret hurtigste-route forespørgsler i et stort vejenetværk med mange løbende opdateringer af trafiksituationen, så brugerne hele tiden har de approksimativt hurtigste ruter til deres destinationer. Løsningen giver garantier for rejsetidens nøjagtighed under antagelse af, at de dynamiske køretider i vejenetværket opfører sig svarende til observationer baseret på virkelige data. Omfattende eksperimenter indikerer at de foreslåede teknikker er nøjagtige, effektive, robuste og skalerbare.
To my parents, my younger brother, my extended family, my wife Sofi, and my in-laws for their sacrifices, constant support and unconditional love.
Acknowledgments

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन ।
मा कर्मफलंहृत्युर्मा ते संगोऽस्तवकर्मणि ॥

– Karma Yoga, Bhagavad Gita

The English translation of the text in Sanskrit above is: “You have the power to act, you do not have the power to influence the result. Therefore, you must act selflessly without anticipation of the result and without succumbing to inaction.” I remember submitting papers to conferences and then trying to speculate and discuss the outcome of the reviews with my supervisor, Prof. Christian S. Jensen. He constantly reminded me how futile and debilitating this thought process was and how it was getting in my way of doing good research. If I was able to focus on completing my thesis, with minimal downtime, then major credit and my sincerest thanks go to Christian, who unknowingly explained the true meaning of the verse in Sanskrit. I appreciate him for nodding his head in agreement during all my talks ... it gave me “super-powers”! Every time I pitched an idea to him for a new paper, he would respond by “Try it out.” I am thankful for the freedom and space he allowed me. His almost excruciating attention to detail drove me to finally learn to punctuate academic prose. It is the little things that go a long way. Apart from the above, I have had the opportunity to learn a lot from Christian both from a research perspective and also on a personal level.

Next, I would like to thank Bin Yang. I learned some very valuable writing tips from Bin. His strict and exacting standards greatly helped me improve my manuscripts, before Christian got a chance to lay his eyes on my manuscript. So thanks for making me look better than I was.

I am very thankful to Raymond Chi-Wing Wong at HKUST, who accommodated me during my six-month stay abroad in Hong Kong. The hard work done during this period was eased by his sense of humor and graciousness as a host. This trip was also a great opportunity to learn from Raymond and brainstorm with him. I am also thankful to his friendly research group at HKUST. I would like to extend a special thanks to Cheng Long and Qi Wang, who taught me the ropes at HKUST and Hong Kong in general.
I thank my Ph.D. support group, consisting of Christian Storm Pedersen and Marianne Graves Petersen, for their kind words of encouragement and help in tracking my progress. I also thank them for their guidance to better organize this thesis. Ira Assent, thank you for the valuable and interesting discussions.

I am more than grateful to all my colleagues at the Data Intensive Systems Group, who are also now my close friends. This group includes, Yu Ma (Mike), Qiang Qu, Anders Skovsgaard, Laura Radaelli, Vaida Ceikute, Bin Yang, Chenjuan Guo, Darius Sidlauskas, Barbora Micenková, Michael Lind Mortensen, Peiman Barnaghi, Matteo Magnani, Xuan-Hong Dang, Sean Chester, Kenneth Bogh and Leon Derczynski. This tight-knit group provided a very friendly work atmosphere. We shared some great moments together and I wish them success in all their future endeavors.

I would like to extend my gratitude to the members of MADALGO for sharing their data for my first research paper and also for their discussions and design ideas. I would also like to thank Dominik Scheder and Navid Talebanfard for the interesting discussions we had that served as an inspiration to several of my ideas.

I thank the administrative staff, including Else Magard, Ann Molhave, and Annemette Hammer, who did things above and beyond the call of duty to help me out, when I expressed a complete lack of knowledge about paperwork, protocol, and procedures.

On a personal note, I thank my parents, whose love and sacrifices have been an inspiration and driving force in my life. My brother for his moral support and sense of humor. My grandparents who have always believed in me.

I would like to thank my wife Sofi, for her support during this arduous journey. I deeply thank her for tolerating me, even when I was irritable, frustrated and feeling low. She had to put on hold her romantic dreams of having a loving husband who would hold her hand and go for long walks on the beach on weekends. Instead, she had to settle for a “nerd” who spent his weekends stuck on a computer. Thus, huge credits go to Sofi for helping me throughout this journey and sharing both my joys and sorrows along the way.

I am indebted to my in-laws who have been extremely supportive and caring. Whenever the going got tough, I always knew I could count on them for help. Last, but not the least, I thank all my friends spread around the globe who have provided me with some great moments to reminisce upon.

Thank you all.

Manohar Kaul,
Aarhus, July 4, 2014.
Contents

Abstract i
Resume iii
Acknowledgments vii
Contents ix

I Overview 1

1 Introduction 3

1.1 Speeding up Surface Spatial Queries 4
  1.1.1 Background and Motivation 4
  1.1.2 Our Proposed Solution 6
  1.1.3 Key Findings from Experimental Study 9
  1.1.4 Future Work 10

1.2 Building a 3D Road Network 11
  1.2.1 Background and Motivation 11
  1.2.2 Our Proposed Solution 11
  1.2.3 Key Findings from Experimental Study 14
  1.2.4 Future Work 15

1.3 Scalable Real-time Continuous Fastest Route Planning 15
  1.3.1 Background and Motivation 15
  1.3.2 Our Proposed Solution 17
  1.3.3 Key Findings from Experimental Study 18
  1.3.4 Future Work 18

1.4 Dissertation Organization 18

II Publications 21

2 Finding Shortest Paths on Terrains 23

2.1 Introduction 23
# CONTENTS

2.2 Preliminaries .......................................................... 26
2.3 Upper and Lower Bounds ............................................. 27
2.4 Related Work ........................................................... 31
  2.4.1 Multi-Resolution Range Ranking Method .................. 31
  2.4.2 Voronoi Diagram Based Method ............................... 32
  2.4.3 Other Existing Approaches ................................. 34
2.5 Generating a Smaller Graph ......................................... 34
  2.5.1 Generating $G'$ .................................................. 35
  2.5.2 Analysis ........................................................ 41
2.6 Experiments ........................................................... 42
  2.6.1 Experimental Setup ............................................ 42
  2.6.2 Our Bounds and the Original Graph ........................ 43
  2.6.3 Existing Bounds and Our Smaller Graph .................. 45
  2.6.4 Our Bounds and Our Smaller Graph ........................ 48
  2.6.5 Scalability ...................................................... 51
  2.6.6 Summary ........................................................ 52
2.7 Conclusion ............................................................. 52
2.8 Acknowledgments ...................................................... 52

3 Constant-Factor Lower Bound for Shortest Distance Queries on Terrains ................. 55
  3.1 Introduction .......................................................... 55
  3.2 Related Work ........................................................ 58
  3.3 Preliminaries ........................................................ 60
  3.4 Upper and Lower Bounds ......................................... 61
  3.5 Bound Algorithm .................................................... 67
    3.5.1 Correctness Proof ............................................ 70
    3.5.2 Complexity Analysis .......................................... 72
  3.6 Experiments .......................................................... 73
    3.6.1 Experimental Setup .......................................... 73
    3.6.2 Impact of our bounds on Shortest Surface Path Query .... 74
    3.6.3 Impact of our bounds on Surface $k$-NN Query ........... 78
    3.6.4 Impact of our bounds on Reverse Surface NN Query ...... 79
    3.6.5 Impact of our bounds on Range Query ..................... 80
    3.6.6 Scalability .................................................... 80
    3.6.7 Summary ...................................................... 81
  3.7 Conclusion ........................................................... 82

4 Terrain-Toolkit: A Multi-Functional Tool for Terrain Data ................................. 83
  4.1 Introduction .......................................................... 83
  4.2 Design ................................................................. 85
    4.2.1 Background Knowledge ....................................... 85
    4.2.2 Architecture of Terrain-Toolkit ........................... 86
    4.2.3 Model/Format Conversion .................................... 87
## CONTENTS

4.2.4 Surface Simplification ........................................... 87
4.2.5 Shortest Surface Path Computation ............................ 88
4.3 Demonstration .......................................................... 89
  4.3.1 UI Demonstration ................................................... 89
  4.3.2 Use Case: Emergency Response ................................. 89
4.4 Conclusion .............................................................. 92

5  Building Accurate 3D Spatial Networks for Intelligent Transportation 93
  5.1 Introduction ............................................................ 93
  5.2 Related Work .......................................................... 96
  5.3 Preliminaries .......................................................... 96
    5.3.1 Data Modeling .................................................... 96
    5.3.2 Problem Formulation ............................................ 99
    5.3.3 Framework Overview ........................................... 100
  5.4 Spatial Network Lifting ............................................ 101
    5.4.1 Filtering .......................................................... 101
    5.4.2 Lifting ........................................................... 106
  5.5 Experiments ........................................................... 109
    5.5.1 Experimental Setup ............................................. 109
    5.5.2 Ground Truth Generation ...................................... 111
    5.5.3 Accuracy Studies ............................................... 111
    5.5.4 Storage Studies ................................................ 112
    5.5.5 Efficiency Analysis ............................................ 114
    5.5.6 Scalability Studies ............................................. 114
  5.6 Conclusion and Outlook ............................................ 115

6  Scalable Real-time Continuous Fastest Route Planning 117
  6.1 Introduction .......................................................... 117
  6.2 Problem Definition .................................................. 120
  6.3 Related Work ........................................................ 125
    6.3.1 Fastest Path Queries on Road Networks with Fixed Weight Edges ................................. 125
    6.3.2 Fastest Path Queries on Road Networks with Dynamic Weight Edges ............................... 126
  6.4 Exact Algorithm ....................................................... 128
    6.4.1 Algorithm Finding $P_{dh}(u, v)$ ............................... 128
    6.4.2 Algorithm Finding $P_{dr}(u, v)$ ............................... 129
  6.5 Approximate Algorithm .......................................... 130
    6.5.1 Finding a Portion of the Network ........................... 130
    6.5.2 Finding $\epsilon$-Approximate Path .......................... 132
  6.6 Discussion ............................................................. 132
  6.7 Experiments ........................................................... 134
    6.7.1 Experimental Setup ............................................. 134
CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7.2 Comparison with Existing Methods</td>
<td>137</td>
</tr>
<tr>
<td>6.7.3 Study on our Approximate Graphs</td>
<td>138</td>
</tr>
<tr>
<td>6.7.4 Abnormal Event Sensitivity Study</td>
<td>139</td>
</tr>
<tr>
<td>6.7.5 Effect of GPS Sampling Rates</td>
<td>142</td>
</tr>
<tr>
<td>6.7.6 Scalability Studies</td>
<td>143</td>
</tr>
<tr>
<td>6.7.7 Case Studies</td>
<td>144</td>
</tr>
<tr>
<td>6.7.8 Summary</td>
<td>145</td>
</tr>
<tr>
<td>6.8 Conclusion and Outlook</td>
<td>145</td>
</tr>
</tbody>
</table>

Bibliography 147
Chapter 1

Introduction

The widespread use of sensors, scanners, and location-aware communication devices, like mobile phones fitted with GPS capabilities, has enabled an increasing amount of spatial information to be captured. This in turn has allowed the generation of much richer models that are a step closer to representing the \textit{real-world} behavior observed in the physical world that is naturally complex. For instance, in the past few years, we have been able to collect enormous volumes of highly-accurate \textit{elevation} information about land surfaces that cover very large regions. Such data is then used to generate accurate 3D representations of the actual terrain. Another example that provides evidence of such advanced models is the recent availability of high frequency GPS updates that are collected from navigation devices fitted on vehicles driving along a road network. This real-time GPS data has enabled us to much better model, simulate, and study live traffic scenarios on the road network.

However, this phenomenon has also led to the following challenges. The excessive collection of data is causing a huge strain on traditional data management technologies, and the models generated are quite complex. In order to meet these two challenges, much attention in both industry and academia is being paid to the efficient storage and querying of these enormous and information-rich models. To better handle the massive volumes of spatial data, research is being conducted in the fields of \textit{distributed spatial indexing} \cite{5, 19, 20, 36} and \textit{parallel spatial query processing} \cite{52, 58, 59, 74} on both multi-core CPUs and GPUs. Simultaneously, the increased complexity in the newer models poses very pertinent research questions, since the spatial query processing techniques that served us well in the past on older models cannot trivially be extended to these newer, complex models. For example, algorithms that were used to compute the intersection between two 2D spatial objects cannot be extended to their 3D counterparts in a straightforward fashion.

Distance measurement between spatial objects is a fundamental operation in most kinds of spatial queries. Traditionally, 2D representations of spatial object locations have been used. Most of the existing work on spatial query processing has only considered Euclidean and spatial network spaces. Earlier works on spatial query processing assumed Euclidean distances between the spatial objects, i.e., the length
of the line segment connecting the two spatial objects. In opposition, subsequent research advocates the use of spatial network distances instead because in real-world applications, spatial objects move along an underlying spatial network, like a road or railway network, and thus it is more realistic and practical to use the shortest network distance between the objects as a distance metric. 3D models are the “closest to reality” when compared to 2D models because they also include elevation information. Currently, there is very limited research done on 3D spatial models, where the shortest surface distance is much more expensive to compute.

Also, when estimated travel time is used as a distance metric on a road network then distance computation, i.e., the total time to travel between two points, becomes a difficult task. When dealing with real-time traffic updates, the travel times constantly change, which in turn leads to stale travel time estimations. In order to know the most up-to-date travel times, the total travel times have to be recomputed when trafficupdates occur.

Motivated by the above observations, we identified the need to further investigate how to improve the effectiveness and efficiency of important spatial queries on complex models, including 3D terrains and dynamic spatial graphs whose estimated travel times are always in a state of flux. Particularly, the goal of this thesis is three-fold: 1) to speed up existing spatial queries on 3D surfaces, 2) to devise efficient spatial join techniques between 3D and 2D datasets, and 3) to compute the most up-to-date fastest path for a large number of users on a dynamic spatial network in an efficient manner.

The remainder of the chapter is organized as follows. Section 1.1 describes our work on speeding up surface spatial queries. Section 1.2 describes the creation of a 3D road network. Section 1.3 describes how we compute the fastest path for a large number of concurrent users on a dynamic spatial network. Finally, Section 1.4 shows the organization of the rest of the thesis chapters.

1.1 Speeding up Surface Spatial Queries

1.1.1 Background and Motivation

Three dimensional mesh representations of physical land surfaces are becoming quite commonplace due to the availability of massive and accurate terrain data collected via aerial laser scans that covers a large expanse.

A wide range of applications that are not constrained to road networks motivate the collection of such large volumes of accurate terrain data. Common applications that require to compute spatial queries on such mesh data include robotic path planning for unmanned vehicles on unknown terrains, geo-realistic computer games, and battlefield simulations for tactical path planning. Some other applications in the medical field also conduct spatial queries on 3D mesh models that are bio-realistic. For example, neuroscientists conduct spatial range queries on brain mesh simulations to study the neuron density and number of branches in a selected region of the brain [63].
1.1. SPEEDING UP SURFACE SPATIAL QUERIES

In academia, several recent studies [13, 44, 56, 68, 69] focus on the challenges presented by terrain data. For example, they consider extending versions of some well-known spatial queries such as shortest path queries, k-nearest neighbor queries, and reverse nearest neighbor queries, from the Euclidean space to terrains.

**Research Challenges:** The surface shortest path computation that underlies all these terrain spatial queries is identified to be extremely expensive and time consuming. The best known algorithm to compute the exact surface shortest path is proposed by Chen and Han [4] and has a time complexity of $O(N^2)$, where $N$ denotes the total number of vertices in the triangulated surface. For example, in our experiments, for a triangulation with $20K$ vertices, we show that the surface shortest path takes $7.2$ hours. Thus, the main challenge lies in attempting to find a solution so that surface spatial queries can be solved faster.

**Existing studies:** Several studies [13, 44, 56, 68] observed that it was more efficient to compute lower and upper bounds of the surface shortest distance, to prune as many unnecessary candidate spatial objects as possible before refinement of query results. This pruning avoids many expensive surface shortest path computations and hence results in faster query processing.

To better illustrate the need for distance bounds, consider a surface 1-NN query. Suppose that $q$ is the query point and there are two objects, $o_1$ and $o_2$. If the lower bound of the shortest surface distance between $o_1$ and $q$ exceeds the upper bound of the shortest surface distance between $o_2$ and $q$ then $o_1$ can be pruned. Thus, we do not consider $o_1$ and need not calculate the exact shortest surface distance between $o_1$ and $q$.

Given a source point $s$ and destination point $t$ on a terrain, the aforementioned works used the Euclidean distance $\Pi_e$ and the shortest network distance $\Pi_n$, as the...
lower and upper bounds for the surface shortest path $\Pi_s$, respectively. Figure [1] illustrates all three $(s,t)$-paths, where $\Pi_s$ is allowed to cut across the faces of the triangulation, while $\Pi_n$ is restricted to the edges of the faces. The motivation behind computing these bounds was their ease of computation in comparison to computing the exact surface shortest path $\Pi_s$.

1.1.2 Our Proposed Solution

Having considered the challenges and existing distance bounds pertaining this problem, we propose methods to compute tighter lower bounds than the existing ones. Furthermore, our experiments show that the shortest surface path can easily be up to 9 times longer than the Euclidean distance. This suggests that using the Euclidean distance as a lower bound distance is too loose and that there is large scope to discover a tighter lower bound that takes into account the geometry of the underlying terrain/surface.

We propose two very different approaches to improving the lower bound:

- **$\theta_m$ related approach ($T,M$)**, where $\theta_m$ is the minimum interior angle of any triangle in the terrain. In this approach the lower bound computed is a ratio of the shortest network distance computed on the graph representation of the triangulation. The quality of the lower bound is also dependent on the value of $\theta_m$, and

- **Cut-vertex related approach ($CV$)** where new vertices called cut-vertices are introduced to edges of relevant triangles in order to further segment the triangles into smaller triangles. In this method, we find a constant ratio lower bound that, unlike $T,M$, is not dependent on the quality of $\theta_m$.

Next, we will describe these solutions and our contributions in more detail.

$\theta_m$ related approach ($T,M$)

Initially, we compute a tighter lower bound by finding a relation between the shortest surface path $\Pi(s,t)$ and the shortest network path $\Pi_G(s,t)$ between a start vertex $s$ and end vertex $t$.

We break down the shortest surface path $\Pi(s,t)$ into smaller segments. A segment is defined to be the portion of $\Pi(s,t)$ that is confined to the boundaries of a single triangle in the triangulation. Then, we consider various scenarios in which a line segment belonging to $\Pi(s,t)$ can cut across a triangle of varying shapes and for each of these scenarios we computed a lower bound for the segment in terms of the edges of the triangle. The edges of the triangle belong to a network path, since network paths can only traverse across the edges of a triangle and not cut across a triangle’s face. Later, we collect the bounds on each segment to get the lower bound for the entire path $\Pi(s,t)$ in terms of $\Pi_G(s,t)$. 


1.1. SPEEDING UP SURFACE SPATIAL QUERIES

We find that the shortest surface path \( \Pi(s, t) \) obeys the following distance bound inequality:

\[
\lambda \cdot |\Pi_G(s, t)| \leq |\Pi(s, t)| \leq |\Pi_G(s, t)|,
\]

where \( \lambda = \min\{\sin \theta_m, \sin \theta_m \cos \theta_m\} \) and \( \theta_m \) is the minimum interior angle of a triangle in the terrain.

Next, we proposed to speed up the computation of the lower bound by proposing a new approximation algorithm to simplify the terrain/surface by removing vertices and edges in such a manner that our distance bounds were always being satisfied.

Surface simplification is the process of reducing the number of faces used in the surface while trying to keep the overall topology preserved as much as possible. Reducing the number of faces in a surface model can greatly speed up the visualization of the model and also computations based on the model, e.g., shortest surface path computations. Many surface simplification techniques have been proposed in the research literature (for a detailed survey, please read [29]). We propose a new surface simplification method that guarantees that the shortest surface distance based on the simplified surface is within a bounded distance from the shortest surface distance based on the original surface.

Contributions:

- To the best of our knowledge, we are the first to extensively study the improvement of the lower bound in shortest surface distance computations, an important component in many spatial queries.
- We propose to use the network distance for both upper and lower bounds, which yields new tighter bounds without introducing significant computational overhead.
- We study how the tightened bounds can be incorporated in standard as well as recent, complex algorithms in order to speed up spatial queries.
- We propose an approach to generate smaller graphs that yield faster lower and upper bound computations while providing guarantees for the tightness of the bounds.
- We present a comprehensive empirical study that offers insight into the accuracy, efficiency, and scalability properties of the framework.

Cut-vertex related approach (CV)

In \( TM \), the bounds proposed have a dependence on the minimum interior angle \( \theta_m \) of all faces in the terrain. Observe that a larger \( \theta_m \) improves the tightness of their lower bound, while decreasing \( \theta_m \) results in their lower bound loosening. Thus, for some low values of \( \theta_m \), the lower bound quality of \( TM \) deteriorates to be worse
than even the Euclidean distance $\Pi_e$. Thus, $\mathcal{T}M$ adopts a constrained Delaunay triangulation to ensure that $\theta_m$ is always greater than $45^\circ$.

To better illustrate, Figure 1.2 shows a triangulation with two degenerate triangles, $\triangle AsB$ and $\triangle AtB$, that share a common base and where $s$ and $t$ are the source and target vertices, respectively.

![Figure 1.2: Terrain with degenerate faces.](image)

This example shows that in the presence of such degenerate triangles, for source vertex $s$ and target vertex $t$, the surface shortest path $\Pi(s, t)$, which is allowed to cut across a face is much smaller than the network path $\Pi_G(s, t)$, that is restricted to only the edges of the faces. Thus, $\Pi_G(s, t)$ does not form a good approximation for $\Pi(s, t)$ in this case. Additionally, $\theta_m$ is also substantially decreased.

Motivated by these aforementioned observations, we propose a new technique $\mathcal{CV}$, that eliminates the dependency on the quality of $\theta_m$ and is not based on only the network distance $\Pi_G(s, t)$ which can be bad in cases where there are degenerate triangles (i.e., skinny triangles) in the triangulation.

To combat these problems, we define a new path called the surface face-crossing path $\Pi_{FC}(s, t)$, which unlike the network path $\Pi_G(s, t)$ is not constrained to the edges of the faces, but can also cut across the face of a triangle.

We propose the introduction of more vertices on the edges of the triangles and then add edges between all pair of vertices on the triangles. We refer to these newly introduced vertices as cut-vertices, since now the surface face-crossing path $\Pi_{FC}(s, t)$ can cut-across the faces of triangles by also using the newly introduced edges connecting cut-vertices.

Figure 1.3 shows $\triangle ABC$ with corner vertices $A, B, C \in V$ and cut vertices $\{A', A'', B', B'', C', C''\} \in V'$. The dotted lines in $\triangle ABC$ denote the newly introduced edges in $E'$ (e.g, $C'B'$). Note that there are no new edges introduced between any pair of vertices placed on the same edge, except the original edge between the corner vertices of that edge. Hence, edges $(C', C'')$ or $(B', C)$ are non-existent. These additional edges in $E'$ form a nearly complete graph and allow additional possibilities for path $\Pi_{FC}(s, t)$ to cut across the face of $\triangle ABC$ via the cut-vertices introduced on the edges.

Additionally, we propose an $A^*$-like approach that introduces these cut-vertices on only the triangles that lie on the way to the target vertex $t$. Using $\Pi_{FC}(s, t)$, which was a tighter approximation of the shortest surface path $\Pi(s, t)$, we were able to propose a tighter lower bound in terms of $\Pi_{FC}(s, t)$ that was always tighter than the lower bound proposed in $\mathcal{T}M$. 

\[ \text{Figure } 1.2: \text{ Terrain with degenerate faces.} \]
1.1. SPEEDING UP SURFACE SPATIAL QUERIES

Contributions:

• To the best of our knowledge, we are the first to propose a tighter constant-factor lower bound for the surface shortest path, that is always tighter than the lower bound proposed in $TM$.

• We theoretically prove the tightness of our lower bound and show how it is always tighter than the lower bound proposed in $TM$ for all possible values of $\theta_m$.

• We introduce a user-defined error parameter $\epsilon$ in our experiments, to better comprehend the trade-off between the tightness of our lower bound and the bound computation time. Studying the effect of our bounds on existing surface spatial queries gives us a better understanding of how our new bounds affect the performance of such fundamental queries on the surface.

• We present a comprehensive empirical study that offers insight into the accuracy, efficiency, and scalability properties of our proposed lower bound.

1.1.3 Key Findings from Experimental Study

Here, we present brief summaries of the results obtained from experiments for both methods, i.e., for $TM$ and $CV$.

**Experimental Summary of $TM$:** Various experiments were conducted to study the effect our tighter lower bounds and surface simplification method had on the performance of existing surface spatial queries like $k$-NN, range and reverse-NN queries on surfaces. We introduced a user defined error parameter $\omega \in [1, 2]$ to study the trade-off between lower bound tightness and query performance. Simply put,
CHAPTER 1. INTRODUCTION

<table>
<thead>
<tr>
<th>Query Type</th>
<th>$\theta_m = 45^\circ$</th>
<th>$\theta_m = 30^\circ$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface k-NN (VOR)</td>
<td>11.6</td>
<td>90</td>
</tr>
<tr>
<td>Surface Range</td>
<td>10.1</td>
<td>82.2</td>
</tr>
<tr>
<td>Surface Reverse k-NN</td>
<td>10.2</td>
<td>85</td>
</tr>
</tbody>
</table>

Table 1.1: Speedup Comparison (With Default Parameters)

when error $\omega$ is increased, the lower bounds are loosened, resulting in slower query performance, since more candidates have to now be considered for refinement.

In our experimental studies, our lower bound is up to 2.8 times larger (or better) than the Euclidean distance, the popular lower bound adopted in the literature. At $\omega = 1$, the computation of our upper bound computed on the simplified and reduced graph $G'$ is up to 10 times faster than that of the original upper bound computed on the original graph $G$ (with the same upper bound value). Importantly, all existing approaches relying on lower and upper bounds experience considerable speedups with our new bounds. In particular, the speedup experienced by VOR, i.e., the state-of-the-art algorithm, is up to 43 times for surface $k$-NN queries on the largest dataset (1M vertices, $k = 5$), which is quite significant.

In general, the best speedups are achieved when using our bounds on the smaller graph. A smaller graph can give a positive effect on faster bound computations but it can also introduce a negative effect on looser bounds (resulting in more candidates explored in some spatial queries). In our experiments, we find that $\omega$ should be set to a value smaller than 1.4 (which means a 40% error, a large error). When $\omega$ is set to a value smaller than 1.4, the positive effect outweighs the negative effect. When $\omega$ is set to a value larger than 1.4, in some cases, the negative effect may dominate the positive effect.

Experimental Summary of CV: Our experimental studies show that for $\theta_m = 45^\circ$ (default) and $\theta_m = 30^\circ$, our lower bound achieves an improvement ratio over the previous bound proposed in TM of 1.75 and 3.1, respectively.

In spite of a slower bound computation when compared to the state-of-the-art, our bounds are much tighter, which in turn results in far fewer expensive surface distance computations. More importantly, we observe a significant speedup in all the surface spatial queries tested. More specifically, the state-of-the-art surface $k$-NN query, i.e., VOR experiences a significant speedup of nearly 46 times on the largest dataset, which has 1 million vertices and $k = 5$.

Table 1.1 outlines the speedups that were experienced for various surface spatial queries for different settings of $\theta_m$.

1.1.4 Future Work

There are a lot of promising research directions. First, it is of interest to derive the lower bound and upper bounds of the shortest surface path when the slope constraint is considered [8]. Second, it is of interest to study real time spatial queries such as continuous $k$-nearest neighbors using our bounds. Third, with the speed of bound
computations, we can use the bounds to compute continuous $k$-nearest neighbors in more advanced situations where both the query object and the spatial objects are in motion on the surface. Furthermore, studies can be performed on efficient path-planning when meeting static or dynamic obstacles in the paths on the surface.

1.2 Building a 3D Road Network

1.2.1 Background and Motivation

The future generations of vehicle routing services and Advanced Driver Assistance Systems (ADAS) will need 3D spatial network models that accurately capture slope and elevation information.

Applications that target fuel savings and reduced greenhouse gas emissions, immensely benefit from the availability of an accurate 3D map. A transportation study finds that eco-routing that uses a 3D spatial network model can yield fuel cost savings of 8–12%, when compared to standard routing based on a 2D model [64]. Another study reports that the use of a 3D model built from aerial laser scan data for vehicle routing will yield annual fuel savings of approximately USD 6 billion in USA [60]. This figure stems from TomTom, a worldwide leading manufacturer of navigation systems. A study of models of vehicular environmental impact shows how increased accuracy of road slopes yields more accurate estimations of fuel consumption and greenhouse gas (GHG) emissions from vehicles [27].

ADAS applications provide critical information to the driver about the vehicle’s surroundings. Here, a 3D map can enhance information obtained from vehicle sensors and can also serve as a failsafe mechanism when sensors fail [18]. For example, adaptive headlights take into account the slopes of the road ahead and intelligently steer the headlights in order to offer maximum night-time visibility. Under adverse weather conditions such as excessive fog or rain, the sensors that are used under normal conditions can fail to operate optimally, and a 3D map can be used instead. ADAS specifications require a 3D road model with an accuracy of at least $\pm$ 2 meters.

**Research Challenges:** There is currently no existing study that details the creation of a 3D spatial network where aerial laser scan data is applied to the lifting of a 2D spatial network. Another key challenge is to achieve a suitable balance between the accuracy of a model, as dictated by the intended applications, and the storage space required by the model.

1.2.2 Our Proposed Solution

Motivated by our observations about possible applications and the research challenges presented, we propose two spatial network lifting techniques that use aerial laser-scan data to augment 2D maps with elevation information. We begin by describing the general framework of our solution followed by an overview for each module in the filtering and lifting phases.
CHAPTER 1. INTRODUCTION

Framework: Figure 1.4 depicts the overview of spatial network lifting, which consists of two major phases: filtering and lifting.

The filtering phase takes as input a 2D spatial network and a massive laser scan point cloud, and prunes irrelevant laser points in the point cloud in order to obtain an appropriate region surrounding every road segment and road intersection in the 2D spatial network. Two alternative filtering methods, one pass filtering and external memory based filtering, are provided in order to exploit situations where the 2D networks fits in main memory and to also provide a more general solution that does not make this assumption, respectively.

The lifting phase consists of two steps, where interpolation follows triangulation. In the triangulation step, laser points in the region surrounding each road segment are transformed into a Triangulated Irregular Network (TIN). After projecting the 2D spatial network onto the TINs, the interpolation step computes the corresponding elevation information, thus providing a 3D spatial network as the final output.

Filtering: The main goal of this phase is to obtain the surrounding regions of the 2D road network.

- One pass filtering (OPF): The first step in this filtering method is to construct a grid index \( H_g \) on the 2D road network. The width of each cell \( c \) in \( H_g \) is a user configurable parameter. This grid index \( H_g \) holds vital information about the cells that contain or intersect with the road network. Thus, \( H_g(c) \) contains all the 2D model elements that intersect with cell \( c \).
1.2. BUILDING A 3D ROAD NETWORK

Next, we stream through the laser-scan 3D points in a single pass and only collect the 3D points whose 2D projections lie within a non-empty cell of previously constructed $H_0$. This yields the 3D points of interest that lie in the vicinity of the road network.

As OPF scans the point cloud, when $H_p(c)$ contains more than $\alpha \cdot \delta \cdot \delta$ laser points, $H_p(c)$ can be passed to the lifting phase immediately. Here, the fill factor $\alpha \in (0, 1]$ is a parameter that represents a trade off between efficiency and accuracy: the higher the fill factor is, the more laser points must be contained in the $\epsilon$-Neighborhoods, thus making the final 3D spatial network more accurate.

This method takes advantage of an observation made by Isenburg and Lindstrom [32] that the laser points are stored in an order that is an artifact of how they were collected by the planes flying over the covered land surfaces. In other words, the laser points in the point cloud are not stored randomly.

The advantages of one pass filtering are: (i) OPF does not incur any pre-processing cost of sorting or indexing the massive point cloud $P_c$; and it only scans the point cloud once; (ii) OPF is able to output parts of the resulting 3D spatial network with different accuracy requirements (by configuring $\alpha$) as the laser points stream in. (iii) OPF can easily be parallelized to take advantage of either new hardware architectures like GPUs or cloud infrastructures like MapReduce.

- **External memory filtering (EMF):** The basic goal of EMF is to efficiently filter large data sets of arbitrary sizes given a limited and fixed main memory budget.

In order to generate a TIN of higher accuracy, EMF reads not only the cell $c$ which intersects with the road’s geometry, but also the eight neighboring cells of cell $c$. This eight-cell neighborhood is referred to as the Moore Neighborhood of cell $c$.

EMF reads road blocks into main memory according to a locality preserving space filling curve. We studied two space filling curves: Row-major and Z-curve. After reading a new road block, EMF reads in its corresponding 9-cell laser blocks and overwrites laser blocks using the least recently used (LRU) [30] policy.

**Lifting:** Upon successful filtering, the laser points in the surrounding regions of the road network are triangulated into a TIN. Then the elevation information in the TIN is assigned to the 2D road segments by projecting these onto the TINs and performing interpolation.

- **Triangulation:** transforms a set of laser points, which represent discrete measurements on a surface, into a set of non-overlapping triangles where the vertices of the triangles are the laser points. We use Delaunay Triangulation [7]
for triangulation. This is a specialized triangulation method where, in the resulting triangles, no triangle vertex is inside the circumscribed circles of any other triangle.

- **Interpolation:** begins by sampling points from the 2D road segment where either (i) the direction of the 2D road network changes or (ii) the slope of the 2D road network derived from the TIN model changes. Once these points are sampled, they are projected back onto the TIN to get the elevation values. For points that fall within the face of a triangle, *plane-based* interpolation is used to compute their elevations.

**Contributions:**

- We propose a novel filtering and lifting framework that uses an aerial laser scan point cloud for lifting a spatial network.

- Two alternative filtering techniques, a *one pass filter* and an *external memory based filter*, are proposed for obtaining the particular points from an aerial laser scan data set that are needed for the lifting.

- Techniques for spatial lifting, consisting of triangulation and interpolation, are proposed to augment the 2D spatial network with elevation information using the remaining laser points.

- We present a comprehensive and large-scale empirical study that offers insight into the accuracy, efficiency, and scalability properties of the framework.

### 1.2.3 Key Findings from Experimental Study

Here, we present a brief summary of the results obtained from experiments.

In our *accuracy analysis*, we noticed that for *OPF* increasing the fill factor $\alpha$ resulted in more points being collected in each cell and hence more accurate triangulations, which in turn resulted in more accurate elevation values. There were still some elevation values which were found in the 1.5–2 meters error range in the *OPF* method. On the other hand, *EMF* was much more accurate in comparison to *OPF*, with less than 10 points in the 10–20 centimeters range.

In our *storage analysis*, The biggest memory budget in *EMF*, with cache size set to $4 \cdot c$ and the largest grid cell width of 128m, does not exceed 80 MB of main memory. Additionally, given the maximum fill factor and grid cell size, *OPF* has a memory upper bound of approximately 600 MB. Hence, both *EMF* and *OPF* can function with very limited memory.

In our *runtime analysis*, *OPF*, which employs no preprocessing at all, is nearly an order of magnitude faster than *EMF*. However, this superior run-time performance comes at the cost of an accuracy degradation in *OPF* as compared to the accuracy achieved by *EMF*. 
1.3 Scalable Real-time Continuous Fastest Route Planning

1.3.1 Background and Motivation

Societies rely on vehicular transportation infrastructures for efficient and predictable transportation. Such infrastructures are complex and time-varying. The travel time from a source to a destination may vary considerably between off-peak periods and peak-periods (rush hour). Small and large unscheduled and unpredictable events occur regularly. For example, a malfunction in a traffic light can slow down traffic considerably, an accident may block a highway for a period of time, which may affect travel times in the surrounding infrastructure very considerably, and a downpour may slow down traffic.

With the proliferation of mobile and location-aware communication devices such as smart-phones and navigation devices, it is becoming possible to monitor the state of a road network in real time. We may view a spatial network as a large-scale sensor system where each road segment, or spatial network edge, emits travel-time information when a vehicle with one of these devices traverses the edge. Specifically, when such a vehicle traverses an edge, it is possible to determine how long it took for the vehicle to do so. Thus, we have a large system of sensors that emit data.

This development makes it possible to not only capture the expected, time-varying travel times of routes that take into account historical information about rush hour, etc. Rather, it becomes possible to also take into account the up-to-date state of the infrastructure.

Research Challenges: Most existing fastest-path queries are one-time queries. They rely on historical information available when they are computed and can neither take into account the user’s subsequent actual progress in the road network nor more recent traffic information. This results in two drawbacks.

The first, called the unpunctual arrival drawback, is that the estimated arrival time at each vertex along the path found by the existing methods is not equal to the user’s actual arrival time due to the current traffic, which may deviate from the historical traffic. Consider a user that traverses an edge \((v_1, v_2)\) followed by edge \((v_2, v_3)\). When a user traverses \((v_1, v_2)\), the estimated travel time is calculated based on historical information, and thus the estimated arrival time at vertex \(v_2\) is also calculated based on the historical information. Next, the the mapping function used for estimating the travel time of traversing edge \((v_2, v_3)\) depends on the arrival time at \(v_2\), which is the time when the traversal starts. The estimated time of arrival at
$v_2$ may be different from the actual arrival time. Thus, the estimated travel time of traversing edge $(v_2, v_3)$ becomes inaccurate.

The second drawback, called the im
curate travel time drawback, is that the estimated travel time of an edge at a time point is not equal to the actual travel time. Thus, the path found based on the estimated travel time may not be the actual fastest path.

The two aforementioned drawbacks form the crux of our research challenge.

Existing Studies: To the best of our knowledge, there is only one closely related study by Malviya et al. [47], which proposed the current traffic information-driven approach. Specifically, the proposed model considers the current traffic information for finding the fastest paths on the network. In this model, the travel time of each edge changes over time based on the current traffic on the road network. The problem is to find the so-called “fastest” path for each user registering for the system. This work is considered as the state-of-the-art method for the current traffic information-driven approach.

Malviya et al. [47] proposed a model as follows. When a user registers in our system with the source vertex $u$ and the destination vertex $v$, the system finds a set $X$ of $k$ edge-disjoint fastest paths from $u$ to $v$ where $k$ is a user parameter. Note that no two paths in $X$ contain the same edge. Additionally, the $k$ paths in $X$ have an ordering in ascending order of their estimated travel times. The $i$-th path in $X$ based on this ordering is called the $i$-th candidate path in $X$ for each $i \in [1, k]$.

Initially, the server sends the first candidate path $\pi$ in $X$ to the client. Then, the client starts to follow this path $\pi$. Whenever the client is still traversing along this path, the system checks the two pre-defined conditions regularly. The first condition is whether there exists an edge in the path $\pi$ such that its current travel time differs from the previously recorded travel time by a threshold value $\alpha$ which is a user parameter. The second condition is whether there are at least $\beta$ edges in $\pi$ such that their actual travel times are updated where $\beta$ is another user parameter. If one of the conditions is satisfied, then the system will send the next candidate path $\pi'$ in $X$ to the client so that the client will follow this updated path $\pi'$ instead of the original path. The above steps continue until the user reaches the destination vertex.

The approach [47] suffers from the following drawbacks. First, the requirement of “edge-disjoint” paths in the output set $X$ results in undesirable paths when a non-first candidate path in $X$ is used. To elaborate, we consider the following scenario. When a user receives the first candidate path $\pi$ from the system, it travels along this path $\pi$. When it reaches some points in the middle part of $\pi$ and receives another candidate path $\pi'$ from the system (since one of the two conditions is satisfied), it has to travel back to the original source vertex and follows the updated path $\pi'$. This scenario can be easily explained by the “edge-disjoint” requirement in the output set $X$ since two paths in $X$ are connected via either a source vertex or a destination vertex. Since it has to travel back to the source vertex, the actual travel time of this user is very large. Second, since there is no position tracking of the user in the system, there are some unnecessary changes from one path to another part. Consider that the user passes some edges in the first candidate path $\pi$ suggested by the system,
1.3. SCALABLE REAL-TIME CONTINUOUS FASTEST ROUTE PLANNING

says $e_1$ and $e_2$. It is obvious that the current traffic of $e_1$ does not affect the remaining route of the user along $\pi$. However, since the system has no information about the position of this user, whenever it knows that there is a traffic update on $e_1$ and one of the two conditions is satisfied based on this traffic, the system has to notify the user to change to another candidate path in the output set. Third, Malviya et al. [47] restrict the greatest distance between a source and destination vertex to 50 miles, due to scalability issues.

1.3.2 Our Proposed Solution

With the objective of achieving a more realistic and reactive dynamic path-planning service, we propose a scalable distributed system that maintains a large number of user queries on a large spatial network, providing users with guaranteed approximate fastest paths to their respective destinations, while facing heavy real-time traffic updates. Our solution assumes that the dynamic real-valued edge weights are bounded in a manner consistent with real-world data, which allows us to provide a theoretical guarantee on the travel time.

In our distributed model, the server stores the entire road network in main memory and clients register their fastest path query with the server. The registration process involves the client/user sending a triplet $Q = (u, v, t_0)$, where $u$ is the start vertex, $v$ the destination vertex and $t_0$ is the time the user wants to begin the journey. In return, the user expects a dynamic fastest path that is capable of providing the fastest path to the destination in a continuous fashion, reacting accordingly to the changes in traffic conditions.

When a user registers $Q$, the server computes a guaranteed region $G_0'$, i.e., a subgraph of the main graph $G$. The rationale behind computing a subgraph $G_0'$, where $|G_0'| \ll |G|$, is that we can now ship $G_0'$ to the user’s navigation device. The server then must only track the edge weight changes in the user’s corresponding $G_0'$ and communicate those changes to the user. Upon receiving an edge weight change, the user can recompute the fastest path to the destination. Observe that the re-computations in reaction to traffic changes is much faster on the smaller $G_0'$ when compared to re-computing on $G$.

Furthermore, we also introduce a user-defined error parameter $\epsilon$ by means of which we can construct an ever sparser subgraph $G_\epsilon'$ to ship to the user. Initially, we construct a $(1 + \epsilon)$-subgraph from $G_0'$ by removing some edges. The property of a $(1 + \epsilon)$-subgraph is as follows. For any given pair of vertices $u$ and $v$ in $G_\epsilon'$ the total travel time of the fastest $(u, v)$-path is at most $(1 + \epsilon)$ times the total travel time of the fastest $(u, v)$-path computed in the original graph $G$.

Contributions:

- We are the first to propose two continuous fastest path queries, namely the fastest dynamic historical path query and the fastest dynamic real-time path query, which give more accurate results compared with the traditional one-time fastest path query considered in most existing studies [6, 12, 17, 35, 49].
We see this as an initial but important move away from the simple one-time fastest path computation based on historical information only studied in the literature [6,12,17,35,49].

- We propose to construct a sub-network $G'_0$, smaller than the original entire network $G$, so that the fastest dynamic real-time path computed based on $G'_0$ is exactly equal to the fastest dynamic real-time path computed based on the original entire network $G$. Furthermore, we propose a $(1 + \epsilon)$-approximate algorithm that is able to use a smaller sub-network $G'_\epsilon$ such that the fastest paths computed in $G'_\epsilon$ have their greatest actual travel time to be at most $(1+\epsilon)$ times the greatest actual travel time computed in the original graph $G$.

- We report on comprehensive experiments, showing that our proposed methods are accurate, efficient, robust, and scalable.

1.3.3 Key Findings from Experimental Study

The actual travel time of our algorithm ourAlg-dr is the smallest among all algorithms. In addition, the difference between the actual travel time and the estimated travel time of ourAlg-dr is the smallest among all algorithms. In the largest dataset, the memory consumption of OurAlg-dr at the client side is at most 650 MB which is affordable to many electronic devices (e.g., the minimum memory specification of iPhone 5 (i.e., 1GB)). A baseline method, $k$-DJPath, has the greatest actual travel time in all experiments. Although the other baseline methods, Corridor and A-Corridor, have a similar actual travel time as OurAlg-dr, the memory consumption of these algorithms at the client side is very large (e.g., at least 3.85 GB in the largest dataset) and the preprocessing times of these algorithms are very large (e.g., at least 2 hours in the largest dataset). Since the preprocessing time can be regarded as the waiting time to start the system and issue the shortest query, a large preprocessing time is not affordable which means that Corridor and A-Corridor are not desirable. Furthermore, the path returned by $k$-DJPath and the path returned by Corridor are nearly 475% and 157% slower than the path returned by OurAlg-dr, respectively.

1.3.4 Future Work

The first direction is to propose a model estimating the travel time of traversing an edge by considering both the historical information and the real-time traffic. The second possible direction is to study whether we can use a MapReduce like approach for this problem when the current traffic information changes frequently over time.

1.4 Dissertation Organization

The remaining chapters of the thesis in Part II: Publications correspond to self-contained and unedited papers. The content of the papers remains unedited with the
1.4. DISSERTATION ORGANIZATION

slight exception of formatting changes done in order to adhere to the thesis template. Therefore, each chapter corresponds to a paper and can be read in isolation.

The chapters are logically organized into three groups: \((G1)\) Speeding up surface spatial queries, \((G2)\) Building a 3D Road Network, and \((G3)\) Scalable Real-time Continuous Fastest Route Planning. Figure 1.5 shows the grouping of the rest of chapters into groups \(G1–G3\).

Figure 1.5: Organization of chapters in Part II.

- Chapter 2 corresponds to publication [38]. It presents an approach to speeding up existing surface spatial queries by computing a tighter lower distance bound for the surface shortest path.
- Chapter 3 corresponds to manuscript [40] (to be submitted). It presents an alternative methodology to the problem proposed in [38] by computing a tighter distance bound that has fewer constraints.
- Chapter 4 corresponds to publication [66] (accepted). It demonstrates a toolkit which accepts a comprehensive set of data formats, supports terrain simplification and provides the surface distance operator.
- Chapter 5 corresponds to publication [39]. This publication won the best paper award at the conference. It presents two spatial network lifting techniques that use aerial laser-scan data to augment 2D maps with elevation information.
- Chapter 6 corresponds to manuscript [41] (to be submitted). It presents a scalable distributed system that maintains a large number of user queries on a large spatial network, providing users with guaranteed approximate fastest paths to their respective destinations, while facing heavy real-time traffic updates.

Apart from the publications included in my thesis, I also co-authored and contributed to the following publications during my Ph.D. studies.


Part II

Publications
Chapter 2

Finding Shortest Paths on Terrains

Abstract

With the increasing availability of terrain data, e.g., from aerial laser scans, the management of such data is attracting increasing attention in both industry and academia. In particular, spatial queries, e.g., $k$-nearest neighbor and reverse nearest neighbor queries, in Euclidean and spatial network spaces are being extended to terrains. Such queries all rely on an important operation, that of finding shortest surface distances. However, shortest surface distance computation is very time consuming. We propose techniques that enable efficient computation of lower and upper bounds of the shortest surface distance, which enable faster query processing by eliminating expensive distance computations. Empirical studies show that our bounds are much tighter than the best-known bounds in many cases and that they enable speedups of up to 43 times for some well-known spatial queries.

2.1 Introduction

We are witnessing an increasing availability of terrain data: More regions are being covered, and the coverage is becoming increasingly accurate and up-to-date. This acquisition of terrain data is motivated by a plethora of applications that are not constrained by road networks.

The defense industry was amongst the earliest to recognize the importance of terrain models to simulate battlefield landscapes to allow tactical path planning \[51\] through valleys (or across ridges) of the terrain via shortest terrain paths. Other applications include robot path planning for unmanned vehicles on terrains and georealistic computer games.

In academia, several recent studies \[13, 44, 56, 68, 69\] focus on the challenges presented by terrain data. For example, they consider “terrain” versions of some well-known spatial queries such as shortest path queries, $k$-nearest neighbor queries, and reverse nearest neighbor queries.
CHAPTER 2. FINDING SHORTEST PATHS ON TERRAINS

Surface shortest path queries \[44\] are fundamental in their own right and occur as an aspect of many other spatial queries (e.g., surface k-nearest neighbor (k-NN) queries \[13 56 68\], surface range queries, and surface reverse nearest neighbor queries \[69\]). Given a source point \(s\) and a destination point \(t\) on a terrain, a shortest surface path query returns the shortest surface path from \(s\) to \(t\) on the surface. Figure 2.1 shows a surface path \(P\) from \(s\) to \(t\), the shortest surface path \(\Pi_s\) from \(s\) to \(t\) and the direct Euclidean distance \(\Pi_E\) from \(s\) to \(t\). Given a set of objects and a query point \(q\) on the surface, a surface k-NN query returns \(k\) objects on the surface such that no other objects are closer to \(q\), where the “closeness” is computed by a surface shortest path query.

Computing surface shortest paths is much more challenging and more expensive than computing network shortest paths on a road network. Specifically, the best-known algorithm for finding the surface shortest path is the Chen-and-Han algorithm \[4\] that is recognized as the state-of-the-art algorithm in the literature \[13 44 56 68\]. Its time complexity is \(O(N^2)\), where \(N\) is the number of vertices used to represent the terrain.

In our experiments, when there are 20K vertices, this algorithm takes 7.2 hours
to find a surface shortest path, which is extremely time-consuming. In contrast, Dijkstra’s algorithm \cite{23} takes only 0.04 seconds to find the corresponding shortest network path when the dataset used has the same number of vertices.

Motivated by this observation, several existing studies \cite{13, 44, 56, 68, 69} propose efficient methods to find lower and upper bounds of the shortest surface distance. These can then be used to avoid some of the expensive surface distance computations inherent in spatial queries.

To illustrate, consider a surface 1-NN query. Suppose that \( q \) is the query point and there are two objects, \( o_1 \) and \( o_2 \). If the lower bound of the shortest surface distance between \( o_1 \) and \( q \) exceeds the upper bound of the shortest surface distance between \( o_2 \) and \( q \) then \( o_1 \) can be pruned. Thus, we do not consider \( o_1 \) and need not calculate the exact shortest surface distance between \( o_1 \) and \( q \).

A popular method for finding the upper bound of the shortest surface path distance from a source \( s \) to a destination \( t \) is to find the shortest network distance from \( s \) to \( t \) based on a Delaunay graph of the terrain \cite{7}. Figure 2.2 shows the Delaunay graph of the terrain in Figure 2.1. In Figure 2.2, \( \Pi_n \) is a network path from \( s \) to \( t \), and \( \Pi_n \) is the shortest network path from \( s \) to \( t \).

We can map each network path in the Delaunay graph to a path on the surface of the terrain (by mapping the vertices and edges used in the network path). Intuitively, each mapped path is a surface path on the terrain with the constraint that the path must pass through only the vertices and the edges of the Delaunay graph. Figure 2.3 shows the surface path \( p \) that is obtained from the Delaunay graph network path \( \Pi_n \).

The reason why the shortest network distance is commonly used as the upper bound of the shortest surface distance \cite{13, 44, 56, 69} is that this bound is tight and is computationally cheap compared to the shortest surface distance computation.

Next, it is popular to lower bound the shortest surface path distance by the Euclidean distance, which is cheap to compute. However, this lower bound can be very loose. To illustrate, recall Figure 2.1, where \( \Pi_E \) is the Euclidean distance. This distance does not capture any information about the surface of the terrain, and thus it is significantly shorter than the shortest surface distance. In our experiments, the shortest surface distance is up to 9 times the Euclidean distance.

Motivated by the above observations, we propose another method to find the lower bound of the shortest surface distance that is tight and computationally-cheap. The major feature of this method is to “kill two birds with one stone.” Specifically, whenever we want to find the lower and the upper bounds of the shortest surface distance, we only need to find the shortest network distance in the Delaunay graph. This distance serves as an upper bound, as discussed previously; and when multiplied by a constant factor derived from the terrain, it also serves as a lower bound.

In our experiments, this lower bound is generally tighter than the previous lower bound. For example, our lower bound is up to 2.8 times larger than the previous lower bound for the largest dataset size. Importantly, this lower bound method does not introduce a significant overhead of finding both the upper and the lower bounds of the shortest surface distance. It thus does not adversely affect the overall performance of the algorithms that use lower and upper bounds.
Although computing shortest network distances on the Delaunay graph is fast, it can be attractive to compute lower and upper bounds even more quickly. This can be achieved by sacrificing the tightness of the bounds. Thus, we propose an algorithm to generate a smaller graph $G'$ from the Delaunay graph $G$ so that the bounds can be computed faster on $G'$.

A key challenge is to generate $G'$ so that the bounds do not become overly loose. With this in mind, we introduce an input parameter $\omega$ that controls the looseness of the bounds. When $\omega$ is set to its minimum value 1, $G'$ yields the same bounds as the original graph $G$. If $\omega$ is set to a larger value, the tightness of the bounds is sacrificed. However, we ensure that lower bound $\tilde{L}$ calculated on $G'$ is not significantly smaller than the lower bound $L$ calculated on $G$; and we ensure that the upper bound $\tilde{U}$ calculated on $G'$ is not significantly larger than the upper bound $U$ calculated on $G$. Specifically, we maintain the following two inequalities: $\frac{1}{\omega} \cdot L \leq \tilde{L} \leq L$ and $U \leq \tilde{U} \leq \omega \cdot U$. Note that the inequality for $\tilde{L}$ is different from the inequality for $\tilde{U}$ because a looser lower bound corresponds to a smaller lower bound and a looser upper bound corresponds to a larger upper bound.

Our contributions can be summarized as follows. First, to the best of our knowledge, we are the first to extensively study the improvement of the lower bound in shortest surface distance computations, an important component in many spatial queries. Second, we propose to use the network distance for both upper and lower bounds, which yields new tighter bounds without introducing significant computational overhead. Third, we study how the tightened bounds can be incorporated in standard as well as recent, complex algorithms in order to speed up spatial queries. Fourth, we propose an approach to generate smaller graphs that yield faster lower and upper bound computations while providing guarantees for the tightness of the bounds. Fifth, we present a comprehensive empirical study that offers insight into the accuracy, efficiency, and scalability properties of the framework.

The remainder of the paper is organized as follows. Section 2.2 formulates the problem. Section 2.3 discusses the lower and upper bounds. Section 2.4 describes related work. Section 2.5 outlines our proposed algorithm for generating smaller graphs. Section 2.6 covers the empirical study of the proposed algorithms and framework. Finally, Section 2.7 concludes the paper.

### 2.2 Preliminaries

Consider a three-dimensional space. Each point $p$ is represented by an $x$-coordinate, a $y$-coordinate, and a $z$-coordinate. Usually, the $z$-coordinate of a point is said to be the elevation of this point.

A terrain is the graph of a continuous function that assigns every point on a horizontal plane to an elevation. In the literature, a terrain is typically represented by a triangulated irregular network (TIN) model that consists of a set $\mathcal{F}$ of faces each of which is represented by a triangle. Each triangle has three corners called vertices and three edges, each connecting two of its corners. Each vertex is a point in the
three-dimensional space. We assume that each interior angle of every triangle/face is non-zero. To ensure this, we replace any triangle violating this assumption by a single edge. Figure 2.1 shows an example of a terrain. Two distinct triangles are said to be adjacent if they share an edge $e$. In this model, there are two types of triangles in $T$, namely normal triangles and boundary triangles. Each normal triangle is adjacent to three other triangles in $T$, and each boundary triangle is adjacent to one or two other triangles in $T$. A point $p$ is said to be on the terrain if there exists a triangle in $T$ such that $p$ is on the plane containing this triangle and $p$ is inside this triangle.

We denote $V$ and $E$ to be the set of all vertices and the set of all edges in the model. The Delaunay graph $G$ of the terrain is defined to be a weighted graph where the set of vertices and the set of edges in this graph are $V$ and $E$, respectively, and the weight of each edge is the Euclidean distance between the two end-points of the edge. In the following, for brevity, we simply write “graph” for “Delaunay graph.”

We denote the line segment connection between two points $p$ and $p'$ to be $(p, p')$. We define the length of line segment $(p, p')$, denoted by $|((p, p'))|$, to be the Euclidean distance between $p$ and $p'$.

Given two vertices $s$ and $t$ in $V$, a surface path from $s$ to $t$, denoted by $\Pi(s, t)$, is a sequence $\langle p_1, p_2, \ldots, p_n \rangle$ where (1) $p_1 = s$, (2) $p_n = t$, and (3) each $p_i$ is a point along an edge in $E$. The surface path is thus composed of $n-1$ line segments, $(p_1, p_2), (p_2, p_3), \ldots, (p_{n-1}, p_n)$. Figure 2.1 shows an example of a surface path $P$ from $s$ to $t$. The length of a surface path $\Pi(s, t)$, denoted by $|\Pi(s, t)|$, is defined to be $\sum_{i=1}^{n-1} |(p_i, p_{i+1})|$. The shortest surface path from $s$ to $t$ is the surface path from $s$ to $t$ with the smallest length.

Further, a network path from $s$ to $t$, denoted by $\Pi_G(s, t)$, is represented by a sequence $\langle v_1, v_2, \ldots, v_n \rangle$ where (1) $v_1 = s$, (2) $v_n = t$, (3) each $v_i$ is a vertex in $V$, and (4) each $(v_i, v_{i+1})$ is an edge in $E$. Figure 2.2 shows an example of a network path from $s$ to $t$. The length of a network path $\Pi_G(s, t)$, denoted by $|\Pi_G(s, t)|$, is defined to be $\sum_{i=1}^{n-1} |(v_i, v_{i+1})|$. Finally, the shortest network path from $s$ to $t$ is defined to be the network path from $s$ to $t$ with the smallest length.

From the literature, we have the following lemma.

**Lemma 1** ([13, 56, 69]) For any two vertices $v$ and $v'$ in $V$, $|\Pi(v, v')| \leq |\Pi_G(v, v')|$. 

### 2.3 Upper and Lower Bounds

In this section, we give the theoretical property that the shortest network distance can be used as the lower bound and the upper bound of the shortest surface distance.

**Lemma 2** (Distance Bound) Let $\Pi(s, t)$ and $\Pi_G(s, t)$ be the shortest surface path and the shortest network path between source $s$ and destination $t$ on terrain $T$, respectively. Then,

$$\lambda \cdot |\Pi_G(s, t)| \leq |\Pi(s, t)| \leq |\Pi_G(s, t)|,$$

where $\lambda$ is a constant.
where \( \lambda = \min \{ \frac{\sin \theta_m}{2}, \sin \theta_m \cos \theta_m \} \) and \( \theta_m \) is the minimum interior angle of a triangle in the terrain.

**Proof:** We need to show two inequalities: \( \lambda \cdot |\Pi_G(s, t)| \leq |\Pi(s, t)| \) and \( |\Pi(s, t)| \leq |\Pi_G(s, t)| \). The second inequality is derived from Lemma \(^1\) In the following, we focus on showing the correctness of the first inequality.

Suppose that \( \Pi(s, t) \) is a sequence \( < p_1, p_2, ..., p_n > \), where \( p_i \) is a point along an edge. Note that \( p_1 = s \) and \( p_n = t \). Each line connecting \( p_i \) and \( p_{i+1} \) is on face \( f_i \).

In the following, we define that each point \( p_i \) has its *owner*, denoted by \( o_i \), which is one of the corners of the face containing \( p_i \). Specifically, we set \( o_1 = s \) and \( o_n = t \). Consider a point \( p_i \) along an edge \( e \) of a face \( f \) where \( i \in [2, n-1] \). (If \( p_i \) is a vertex so that multiple edges contain \( p_i \), then we arbitrarily pick one of the edges as \( e \).) Let \( A \) and \( B \) be the two end-points of the edge \( e \). Note that \( A \) and \( B \) are the two vertices on the terrain. If \( |[A, p_i]| \neq |[B, p_i]| \), we set \( o_i \) to be the end-point with the smaller Euclidean distance to \( p_i \). Otherwise, we set \( o_i \) to the end-point of \( e \) shared with the edge that \( p_{i-1} \) is along.

Consider a sequence \( O : \langle o_1, o_2, ..., o_n \rangle \). Each \( o_i \) is a vertex on the terrain. In addition, for each \( i \in [2, n] \), we know that either (i) \( o_i \) is equal to \( o_{i-1} \) or (ii) \( (o_i, o_{i-1}) \) is an edge of a face on the terrain. We deduce that sequence \( O \) forms a network path from \( s \) to \( t \) since \( o_1 = s \) and \( o_n = t \). Let \( |O| \) be the distance of \( O \). Since \( \Pi_G(s, t) \) is the shortest network path from \( s \) to \( t \), we have that \( |\Pi_G(s, t)| \leq |O| \). In the remaining part of the proof, we show that \( \lambda \cdot |O| \leq |\Pi(s, t)| \). With these two inequalities, we derive that \( \lambda \cdot |\Pi_G(s, t)| \leq |\Pi(s, t)| \), which completes the proof.

To show that \( \lambda \cdot |O| \leq |\Pi(s, t)| \), consider a pair \( (o_i, o_{i+1}) \) on a single face \( f_i \). We want to show that \( \lambda \cdot |(o_i, o_{i+1})| \leq |(p_i, p_{i+1})| \) for each \( i \in [1, n-1] \).

Consider two cases. **Case 1:** \( o_i = o_{i+1} \). In this case, illustrated in Figure 2.4, we know that \( |(o_i, o_{i+1})| = 0 \). Thus, the inequality holds.

**Case 2:** \( o_i \neq o_{i+1} \). Face \( f_i \) has two corners/vertices, \( o_i \) and \( o_{i+1} \). Let \( F \) be the remaining corner of \( f_i \). Let \( C \) be the mid-point of edge \( (o_i, F) \), let \( D \) be the mid-point of edge \( (o_{i+1}, F) \), and let \( E \) be the mid-point of edge \( (o_i, o_{i+1}) \). We denote the interior angles of \( f_i \) at vertices \( o_i, o_{i+1} \), and \( F \) by \( \alpha, \beta, \) and \( \gamma \), respectively.

We further consider four sub-cases. **Case 2(a):** Both \( p_i \) and \( p_{i+1} \) are not along edge \( (o_i, o_{i+1}) \). This case is illustrated in Figure 2.5(a). First, by the mid-point theorem, we have

\[
|\langle C, D \rangle| = \frac{1}{2} \cdot |(o_i, o_{i+1})|.
\] (2.1)

Consider four sub-cases. **Case (i):** \( \alpha < \frac{\pi}{2} \) and \( \beta < \frac{\pi}{2} \). In addition, \( p_i \) is on \( (o_i, C) \) only, and \( p_{i+1} \) is on \( (o_{i+1}, D) \) only because \( o_i \) is the owner of \( p_i \) and \( o_{i+1} \) is the owner of \( p_{i+1} \). Thus, we know that \( |(p_i, p_{i+1})| \geq |\langle C, D \rangle| \). From Equation 2.1 we obtain \( \frac{1}{2} \cdot |(o_i, o_{i+1})| \leq |(p_i, p_{i+1})| \). Also, since \( \sin \theta_m \cos \theta_m = \frac{\sin \theta_m}{2} \) and \( \sin 2\theta_m \leq 1 \), we have \( \sin \theta_m \cos \theta_m \leq \frac{1}{2} \). Since \( \sin \theta_m \cos \theta_m \leq \frac{1}{2} \) and \( \sin \theta_m \cos \theta_m \leq \frac{1}{2} \), we have \( \lambda \leq \frac{1}{2} \). Thus, the inequality holds.
2.3. UPPER AND LOWER BOUNDS

Figure 2.4: Case 1 in the Proof of Lemma 2

Figure 2.5: Case 2(a) in the Proof of Lemma 2

Figure 2.6: Case 2(b) in the Proof of Lemma 2

Case (ii): $\alpha \geq \frac{\pi}{2}$ and $\beta < \frac{\pi}{2}$ (illustrated in Figure 2.5(b)).

Since $\alpha \geq \frac{\pi}{2}$ and $\alpha + \beta + \gamma = \pi$, we have $\beta + \gamma < \frac{\pi}{2}$. Since $\beta \geq \theta_m$ and $\gamma \geq \theta_m$, we have $\beta + \gamma \geq 2\theta_m$. We derive that

$$2\theta_m \leq \beta + \gamma < \frac{\pi}{2}.$$  \hspace{1cm} (2.2)

We draw a perpendicular line from $D$ to edge $(o_i, F)$. Let $G$ be the end-point of this line which is along edge $(o_i, F)$. Similarly, $p_i$ is on $(o_i, C)$ only and $p_{i+1}$ is on $(o_{i+1}, D)$ only. Thus, we have

$$|\langle p_i, p_{i+1} \rangle| \geq |\langle D, G \rangle|.$$  \hspace{1cm} (2.3)

Consider the triangle with corners $C$, $D$, and $G$. By the mid-point theorem, we know that the line connecting $C$ and $D$ is parallel to edge $(o_i, o_{i+1})$. We deduce that the interior angle of this triangle at corner $C$ is equal to $\pi - \alpha$. Note that $|\langle D, G \rangle| =
Thus, from Equation 2.1 and Inequalities 2.3 and 2.4, we obtain

\[
\sin(\theta) \cos(\theta) \leq \frac{1}{2} \sin \theta \cdot \cos \theta.
\]

Since \( \theta_m \cos \theta_m = \sin \theta_m \), \( \sin \theta_m \leq 1 \), \( \cos \theta_m \leq 1 \), and \( \theta_m \leq 1 \), we have \( \sin \theta_m \cos \theta_m \leq 1 \). Therefore, \( \sin \theta_m \) satisfies the inequality.

\[
\sin \theta_m \cdot \cos \theta_m \leq \frac{1}{2} \sin \theta_m \cdot \cos \theta_m.
\]

(2.4)

Thus, from Equation 2.1 and Inequalities 2.3 and 2.4, we obtain

\[
\sin \left( \frac{\theta_i + \gamma}{2} \right) \leq \sin \left( \frac{\theta_i + \pi}{2} \right).
\]

Case (ii): \( \alpha < \frac{\pi}{2} \) and \( \beta \geq \frac{\pi}{2} \). This case is illustrated in Figure 2.3(c) and is similar to Case (ii).

Case (iii): \( \alpha \geq \frac{\pi}{2} \) and \( \beta < \frac{\pi}{2} \). This case is impossible because the third angle must be greater than 0 and the sum must be equal to \( \pi \). We conclude that the inequality holds for Case 2(a).

Case 2(b): \( p_i \) is along edge \((o_i, o_{i+1})\) but \( p_{i+1} \) is not.

We consider four sub-cases. Case (i): \( \beta < \frac{\pi}{2} \) and \( \gamma < \frac{\pi}{2} \). This case is illustrated in Figure 2.6(a).

We draw a line from \( E \) to edge \((F, o_{i+1})\) such that this line is perpendicular to edge \((E, o_{i+1})\). Let \( G \) be the end-point of this line which is along edge \((F, o_{i+1})\). Since \( p_i \) is on \((o_i, E)\) and \( p_{i+1} \) is on \((o_{i+1}, D)\) only, we have \(|(p_i, p_{i+1})| \geq |(E, G)|\). Consider the triangle with corners \( E, G \) and \( o_{i+1} \). We have \(|(E, o_{i+1})| \sin \beta = |(E, G)|\). Since \( \frac{1}{2} \cdot |(o_i, o_{i+1})| = |(E, o_{i+1})|, \) we have \( \frac{1}{2} \cdot |(o_i, o_{i+1})| \sin \beta = |(E, G)|\). We deduce that \( \frac{\sin \beta}{2} \cdot |(o_i, o_{i+1})| \leq |(p_i, p_{i+1})|\). Since \( \lambda \leq \frac{\sin \beta}{2} \), the inequality holds.

Case (ii): \( \beta < \frac{\pi}{2} \) and \( \gamma \geq \frac{\pi}{2} \). This case is illustrated in Figure 2.6(b).

By the mid-point theorem, we have \(|(D, E)| = \frac{1}{2} \cdot |(o_i, F)|\). By the sine rule, we have \( \sin \beta = \frac{\sin \gamma}{|o_i, F|} \). Thus, we derive \(|(o_i, o_{i+1})| = \frac{\sin \gamma}{\sin \beta} \cdot |(o_i, F)|\). Since \( \sin \gamma \leq 1 \), we have \(|(o_i, o_{i+1})| \leq \frac{1}{\sin \beta} \cdot |(o_i, F)|\). We deduce that \( \frac{\sin \theta_m}{2} \cdot |(o_i, o_{i+1})| \leq |(D, E)|\). Since \( \theta_m \leq \beta < \frac{\pi}{2} \), we have \( \frac{\sin \theta_m}{2} \cdot |(o_i, o_{i+1})| \leq |(D, E)|\). In this case, since \( p_i \) is on \((o_i, E)\) and \( p_{i+1} \) is on \((o_{i+1}, D)\) only, we know that \(|(p_i, p_{i+1})| \geq |(D, E)|\). We derive that \( \frac{\sin \theta_m}{2} \cdot |(o_i, o_{i+1})| \leq |(D, E)|\). Since \( \lambda \leq \frac{\sin \theta_m}{2} \), we have \( \lambda \cdot |(o_i, o_{i+1})| \leq |(p_i, p_{i+1})|\).

Case (iii): \( \beta \geq \frac{\pi}{2} \) and \( \gamma < \frac{\pi}{2} \). This case is illustrated in Figure 2.6(c). Since \( p_i \) is on \((o_i, E)\) only and \( p_{i+1} \) is on \((o_{i+1}, D)\) only, we have \(|(p_i, p_{i+1})| \geq |(E, o_{i+1})|\). Since \(|(E, o_{i+1})| = \frac{1}{2} \cdot |(o_i, o_{i+1})|, \) we have \( \frac{1}{2} \cdot |(o_i, o_{i+1})| \leq |(p_i, p_{i+1})|\). Besides, since \( \sin \theta_m \cos \theta_m = \frac{\sin \theta_m}{2} \) and \( \sin 2\theta_m \leq 1 \), we have \( \sin \theta_m \cos \theta_m \leq \frac{1}{2} \). Since \( \sin \theta_m \cos \theta_m \leq \frac{1}{2} \) and \( \sin \theta_m \leq \frac{1}{2} \), we have \( \lambda \leq \frac{1}{2} \). We deduce that \( \lambda \cdot |(o_i, o_{i+1})| \leq |(p_i, p_{i+1})|\).

Case (iv): \( \beta \geq \frac{\pi}{2} \) and \( \gamma \geq \frac{\pi}{2} \). As Case (iv) of Case 2(a), this case is impossible.

Case 2(c): \( p_{i+1} \) is along edge \((o_i, o_{i+1})\) but \( p_i \) is not. This case is similar to Case 2(b).

Case 2(d): Both \( p_i \) and \( p_{i+1} \) are along edge \((o_i, o_{i+1})\). In this case, we know that \( o_i = p_i \) and \( o_{i+1} = p_{i+1} \). This can be explained by the observation that the shortest
2.4. RELATED WORK

We review existing approaches using lower and upper bounds.

2.4.1 Multi-Resolution Range Ranking Method

Deng et al. [13] propose a multi-resolution range ranking (MR3) method for a surface $k$-NN query. This method uses lower and upper bounds for pruning some objects that
are far away from the query point $q$.

Specifically, it involves the following four steps. Let $H$ be the (conceptual) horizontal plane at the sea level of the terrain. Let $O$ be a set of objects in the dataset for the surface $k$-NN query.

**Step 1 (2D $k$-NN Query):** Let $q'$ be the query point projected on $H$. Let $O'$ be the set of objects in $O$ projected on $H$. The MR3 method finds a set $S'$ of $k$ objects in $O'$ nearest to $q'$ on $H$. Let $S$ be the set of all objects whose projections are in $S'$.

**Step 2 (Surface Distance Computation):** It finds the $k$-th nearest object $o$ in $S$ from $q$ according to their surface distances.

**Step 3 (2D Range Query):** It computes the upper bound $U$ of the shortest surface distance between $q$ and $o$. It performs a range query from $q'$ with its radius equal to $U$ and obtain a set $T'$ of objects from the range query. Let $T$ be the set of all objects whose projections are in $T'$.

**Step 4 (Surface Distance Ranking):** It finds the $k$ objects in $T$ whose surface distances are at most the surface distance between $q$ and the $k$-th nearest object in $T$.

Since Step 4 finds surface distances, the MR3 method makes use of the upper and lower bounds for pruning. Even though it is equipped with the lower and upper bound computations for pruning, it is found in [56, 69] that the MR3 method does not return accurate $k$-NN results, especially as $k$ gets larger.

### 2.4.2 Voronoi Diagram Based Method

Shahabi et al.[56] propose a Voronoi diagram-based approach for computing a surface $k$-NN query, which is the state-of-the-art method for this query. Unlike the MR3 method, it returns accurate results. This approach exploits so-called *tight cells* and *loose cells*. Let $O$ be a set of objects on a surface.
2.4. RELATED WORK

Given an object $o \in O$, the tight cell of $o$, $TC(o)$, is the region on the surface such that each point $p$ in this region has a network distance to $o$ that is at most the Euclidean distance between $p$ and each other object in $O$. In Figure 2.7 showing two objects $o_1$ and $o_2$, the region enclosed by the solid line containing $o_1$ corresponds to $TC(o_1)$ and the region enclosed by the solid line containing $o_2$ corresponds to $TC(o_2)$. Consider a point $q$ along the boundary of $TC(o_1)$. Note that the network distance between $q$ and $o_1$ ($d_N(o_1, q)$) is equal to the Euclidean distance between $q$ and $o_2$ ($d_E(q, o_2)$). Given an object $o \in O$, the loose cell of $o$, $LC(o)$, is the region on the surface such that each point $p$ in this region has a Euclidean distance to $o$ that is at most the network distance between $p$ and each other object in $O$. In Figure 2.7, the region enclosed by the dashed line containing $o_1$ corresponds to $LC(o_1)$ and the region enclosed by the dashed line containing $o_2$ corresponds to $LC(o_2)$.

[56] found the following three properties related to tight cells and loose cells. (1) It is possible that the loose cell of an object can overlap with the loose cell of another object. (2) The loose cell of an object does not overlap with the tight cell of another object. (3) The tight cell of an object is completely inside its loose cell.

If a query point $q$ is inside the tight cell of an object $o$, we know that there is only one such tight cell containing $q$. Thus, $o$ is the nearest neighbor of $q$ according to the surface distance. Otherwise, $q$ is inside a number of loose cells of objects, but is outside the tight cells of these objects. Let $O'$ be these objects. We are not sure which object in $O'$ is nearest to $q$, and we need to run the algorithm for finding the shortest surface distance $|O'|$ times to find the answer. According to the above observation, if the area of the tight cell is larger, then it will be more likely that $q$ is in a tight cell and we do not need to issue the time-consuming algorithm for finding the shortest surface distance. Besides, if the area of the loose cell is smaller then it will be more likely that the size of $O'$ is smaller, and thus we will run the algorithm for finding the shortest surface distance fewer times.

Shahabi et al. [56] propose an algorithm that first finds the tight cells of all objects and the loose cells of all objects. Then it finds the $k$ nearest neighbors on the surface from a query point $q$ according to these cells. Specifically, the algorithm first finds all tight and loose cells covering $q$ in order to determine the 1-NN $o$. Then, it expands all the neighbors of the loose cell of $o$ to incrementally find the next nearest neighbor from $q$. During the expansion, it finds the shortest surface distance between $q$ and each object accessed.

Interestingly, the above algorithm can be improved significantly when our bounds are used. Specifically, as we described before, the lower bound we proposed is at least the Euclidean distance studied by [56]. We modify the tight cell definition and the loose cell definition by replacing the Euclidean distance calculation by our lower bound. Since our lower bound is at least the Euclidean distance, it is easy to verify that if the area of the tight cell of each object is larger and the area of the loose cell of each object is smaller. As we explained, a larger tight cell and a smaller loose cell can avoid a lot of shortest surface path computation and thus the algorithm can be speeded up.
2.4.3 Other Existing Approaches

Other approaches exist that use lower and upper bounds. Examples include continuous surface $k$-NN queries [68] and reverse nearest neighbor queries [69].

Xing et al. [68] propose an algorithm for a continuous surface $k$-NN query that relies on the concept of the expansion area of a query point $q$. Specifically, given a query point $q$, the expansion area of $q$ is a region containing all objects such that the Euclidean distance between each object in this region and $q$ is at most a threshold being updated during the execution of the algorithm. The algorithm considers all objects in the expansion area of $q$ only and calculates the shortest surface distance between $q$ and each of these objects. Similarly, when our bounds are used, we can replace the Euclidean distance by our lower bound. In this case, fewer objects remain in the expansion area. Thus, the algorithm can be speeded up.

Yan et al. [69] propose algorithms to find bichromatic and monochromatic reverse nearest neighbors on terrains. The algorithms proposed by [69], the only best-known algorithm for these queries, rely on the tight and loose cells proposed by [56]. Similarly, with our new bound adoption, it is expected that these algorithms can run more quickly.

2.5 Generating a Smaller Graph

As we described in Section 2.1, in order to efficiently compute the network distance (which is used in our upper/lower bound computation), we propose to generate a smaller graph $G'$ from $G$. We introduce a parameter $\omega$ such that the bounds computed from $G'$ are not quite different from the bounds computed from $G$. We give a formal definition for $\omega$ as follows. Let $L_G(s, t)$ and $U_G(s, t)$ be the lower and upper bounds of a surface distance from a source $s$ to a destination $t$ computed based on a graph $G$, respectively.

**Property 1 (Network Distance Property)** Given an original graph $G = (V, E)$ and another graph $G' = (V', E')$, we say that $G'$ satisfies the network distance property if and only if for any two vertices $s$ and $t$ in $V$, the following two inequalities are satisfied.

\[
\frac{1}{\omega} \cdot L_G(s, t) \leq L_{G'}(s, t) \leq L_G(s, t) \tag{2.5}
\]

\[
U_G(s, t) \leq U_{G'}(s, t) \leq \omega \cdot U_G(s, t) \tag{2.6}
\]

As described in Section 2.1, we would like to maintain Property 1. The major idea behind generating a smaller graph $G'$ from $G$ is to remove some vertices from $G$ and re-adjust the edges in $G$ such that $G'$ satisfies the network distance property.
2.5. GENERATING A SMALLER GRAPH

2.5.1 Generating $G'$

We propose an iterative approach that removes a vertex $v$ iteratively from $G$ to generate $G'$. Consider an iteration of this approach. Before we execute this iteration, we have a graph $G'$ where some vertices have been removed. After the execution, we remove a vertex $v$ from $G'$ and form a smaller graph $G''$. Let $V'$ and $E'$ be the set of vertices and the set of edges in $G'$, respectively. Let $V'' = V' - \{v\}$ and $E''$ be the vertices and edges in $G''$, respectively. We denote the operation of removing vertex $v$ from $G'$ by $o(G',v)$; thus $G'' = o(G',v)$.

Algorithm 1 shows the algorithm for generating $G'$. In this algorithm, we have to check whether $G'$ satisfies the network distance property (Property 1).

Algorithm 1: Algorithm for Generating $G'$

1: $G' ← G$
2: while there exists a vertex $v$ in $G'$ such that a graph $G''$ obtained after the removal operation of $v$ from $G'$ satisfies the network distance property do
3: $G'' ← o(G',v)$
4: $G' ← G''$
5: end while
6: return $G'$

In Property 1 we have to maintain Inequalities 2.5 and 2.6. Note that we detailed how to compute $L_G(s,t)$ and $U_G(s,t)$ (based on $G$) in Section 2.3. That is,

$$L_G(s,t) = \lambda \cdot |\Pi_G(s,t)|$$
$$U_G(s,t) = |\Pi_G(s,t)|.$$  

However, how to compute $L_G'(s,t)$ and $U_G'(s,t)$ (based on $G'$) has not been specified yet. Next, we describe a method to compute $L_G'(s,t)$ and $U_G'(s,t)$. Formally, given distinct vertices $s$ and $t$ in $V$ (from the original graph $G$), we denote the estimated distance between $s$ and $t$ on a smaller graph $G'$ by $\tilde{d}_{G'}(s,t)$. In Section 2.5.1 we describe a method to compute this estimated distance such that it satisfies the following property.

Property 2 (Estimated Distance Property) Given an original graph $G$ and another graph $G'$, we say that $G'$ satisfies the estimated distance property if and only if for any two vertices $s$ and $t$ in $V$,

$$|\Pi_G(s,t)| \leq \tilde{d}_{G'}(s,t) \leq \omega \cdot |\Pi_G(s,t)|.$$  (2.7)

After we describe how to compute $\tilde{d}_{G'}(s,t)$, we will argue in Section 2.5.2 that the above property holds.
Proof: First, we show that Inequality 2.6 holds. Since $G'$ satisfies the estimated distance property, for any two vertices $s$ and $t$ in $V$, $|\Pi_G(s, t)| \leq \tilde{d}_{G'}(s, t) \leq \omega \cdot |\Pi_{G'}(s, t)|$. Since $U_G(s, t) = |\Pi_G(s, t)|$ and $U_{G'}(s, t) = \tilde{d}_{G'}(s, t)$, we derive that $U_G(s, t) \leq U_{G'}(s, t) \leq \omega \cdot U_{G'}(s, t)$. Thus, Inequality 2.6 holds.

Next, we show that Inequality 2.5 also holds. We first show that $L_{G'}(s, t) \leq L_G(s, t)$. Since $L_{G'}(s, t) = \lambda \cdot \tilde{d}_{G'}(s, t)$ and $\tilde{d}_{G'}(s, t) \leq \omega \cdot |\Pi_{G'}(s, t)|$, we derive that $L_{G'}(s, t) \leq \lambda \cdot |\Pi_{G'}(s, t)|$. Since $L_G(s, t) = \lambda \cdot |\Pi_G(s, t)|$, we conclude that $L_{G'}(s, t) \leq L_G(s, t)$.

In the following, we show that $\frac{1}{\omega} \cdot L_G(s, t) \leq L_{G'}(s, t)$. Note that $L_{G'}(s, t) = \lambda \cdot |\Pi_{G'}(s, t)|$. Since $L_G(s, t) = \lambda \cdot |\Pi_G(s, t)|$, we have $L_{G'}(s, t) \geq \frac{\lambda}{\omega} \cdot L_G(s, t)$.

Thus, Inequality 2.5 also holds. In conclusion, $G'$ satisfies the network distance property. \[\square\]

With the above lemma, if $G'$ satisfies the estimated distance property (Property 2), then $G'$ satisfies the network distance property (Property 1). In the following, we focus on checking whether $G'$ satisfies the estimated distance property.

There are four remaining issues in Algorithm 1 described in Sections 2.5.1 – 2.5.1.
2.5. GENERATING A SMALLER GRAPH

How to Compute $\tilde{d}_{G'}(s,t)$

As the first step in describing how to compute $\tilde{d}_{G'}(s,t)$, we define the concept of “guest” and “host” vertices by drawing an analogy to a familiar occurrence where guests leave, while hosts remain at a residence.

Consider graph $G'$. Let $V' = V - V''$, where $V''$ and $V'$ denote the set of removed and remaining vertices, respectively. Each removed vertex $\tau$ in $V''$ is associated with a non-empty set of some remaining vertices in $V'$. Each (remaining) vertex in this set is called a host of $\tau$ and the set of hosts is denoted by $H(\tau)$. If $v$ is a host of a removed vertex $\tau$, then $\tau$ is said to be a guest of $v$. Note that each guest is in the removed vertex set ($V''$) and each host is in the remaining vertex set $V'$. Given a vertex $v \in V'$, we maintain not only the guest set of $v$, $G(v)$, but also the information about the estimated distance between $v$ and each $v$ of its guests in $G(v)$ (i.e., $\tilde{d}_{G'}(\tau, v)$). Specifically, we define the guest information set of $v$, denoted by $GL(v)$, to be the set of entries where each entry is in the form of $(\tau, \tilde{d}_{G'}(\tau, v))$ for each $\tau \in G(v)$. Here, each entry contains the second component in the form of $\tilde{d}_{G'}(\tau, v)$ which can be computed when we create or update this entry. Details will be described later in Section 2.5.1. At this moment, we assume that this component is given.

Example 1 (Host and Guest) Figure 2.8(a) shows the original graph $G$. Suppose that we remove vertex $v_6$ and vertex $v_{10}$ from $G$, and a smaller graph $G'$ is generated as shown in Figure 2.8(b). According to some methods which will be described later, we generate the guest set and the guest information set of each remaining vertex in $G'$, as shown in Figure 2.8(c). Similarly, the host set of each vertex removed can be found in Figure 2.8(d).

With the concepts of “guest” and “host,” we are ready to describe how we compute $\tilde{d}_{G'}(s,t)$. Given any two vertices $s$ and $t \in V$, we want to estimate the network distance between $s$ and $t$ on the smaller graph $G' = (V', E')$. Since some vertices are removed from $V$ and cannot be found in $V'$, $s$ may occur in $V'$ or not, and $t$ may also occur in $V'$ or not.

This yields three cases to be considered when we compute the shortest network distance between $s$ and $t$. Case 1: Both $s$ and $t$ are not found in $V'$, Case 2: Only one of $s$ and $t$ is found in $V'$, and Case 3: Both $s$ and $t$ are found in $V'$.

Consider Case 1. Note that each vertex not in $V'$ is associated with at least one host that can be found in $V'$. The major idea of estimating the distance between $s$ and $t$ involves three components, namely the guest-to-host distance, the host-to-host distance, and the host-to-guest distance.
We first estimate the distance between $s$ which is not in $V'$ and one of its hosts, says $u$. We call this distance the guest-to-host distance from $s$ to $u$. Second, we estimate the distance between this host and one of the hosts of $t$, says $u'$. We call this distance the host-to-host distance from $u$ to $u'$. Third, we estimate the distance between $u'$ and $t$. We call this distance the host-to-guest distance from $u'$ to $t$.

Consider the guest-to-host distance. Given a host $u \in \mathcal{H}(s)$, we have $s \in \mathcal{G}(u)$. We define the estimated guest-to-host distance from $s$ to $u$, denoted by $\hat{d}_{\mathcal{G}}(s,u)$, to be $\hat{d}$, where $\hat{d}$ is the second component of the entry $(s,\hat{d}) \in \mathcal{GL}(u)$.

Consider the host-to-host distance. Given a vertex $u$ and a vertex $u'$, we define the estimated host-to-host distance from $u$ to $u'$, denoted by $\hat{d}_{\mathcal{H}}(u,u')$, to be $|\Pi_{\mathcal{G}}(u,u')|$.

Consider the host-to-guest distance. Given a host $u' \in \mathcal{H}(t)$, similarly, we define the estimated host-to-guest distance from $u'$ to $t$, denoted by $\hat{d}_{\mathcal{H}}(u',t)$, to be $\hat{d}$ where $\hat{d}$ is the second component of the entry $(t,\hat{d}) \in \mathcal{GL}(u')$.

In the above discussion, we just consider a particular host $u$ of $s$ and a particular host $u'$ of $t$. In general, there are multiple hosts of $s$ and multiple hosts of $t$. Thus, the set of all possible pairs of hosts of $s$ and hosts of $t$ can be denoted by $\mathcal{H}(s) \times \mathcal{H}(t)$. Among all possible pairs, we want to find the best host pair yielding the smallest estimated distance. Formally, we define the best host pair of $(s,t)$, denoted by $h_{\hat{d}}(s,t)$, to be $\arg \min_{(u,u') \in \mathcal{H}(s) \times \mathcal{H}(t)} |\Pi_{\mathcal{G}}(u,u')|$

where $(u,u')$ is the best host pair of $(s,t)$, we say that $u$ is the best host of $s$ and $u'$ is the best host of $t$.

We define $\hat{d}_{\mathcal{G}}(s,t) = dist_{\mathcal{G}}(s,u) + dist_{\mathcal{H}}(u,u') + dist_{\mathcal{G}}(u',t)$, where $(u,u') = h_{\hat{d}}(s,t)$.

Consider Cases 2 and 3, which are simpler than Case 1. If $s$ is a vertex in $V'$, we set the guest-to-host distance to be 0. In this case, we define the best host of $s$ to be itself. If $t$ is a vertex in $V'$, we set the host-to-guest distance to be 0. Then, we define the best host of $t$ to be itself.

Now, we know how to compute $\hat{d}_{\mathcal{G}}(s,t)$. In Section 2.5.2, we will show that Property 2 (based on $\hat{d}_{\mathcal{G}}(s,t)$) holds.

**Example 2 (Estimated Distance)** Consider Example 1 (as shown in Figure 2.8). Suppose that we want to find the estimated distance between $v_6$ and $v_{10}$ (i.e., $\hat{d}_{\mathcal{G}}(v_6, v_{10})$). Note that vertex $v_6$ has three hosts in $\mathcal{H}(v_6)$, namely $v_1$, $v_2$, and $v_7$, and vertex $v_{10}$ has three hosts in $\mathcal{H}(v_{10})$, namely $v_4$, $v_5$, and $v_{11}$.

Consider the host $v_2$ of $v_6$. There exists an entry $(v_6, 2.4)$ in $\mathcal{G}(v_2)$. The estimated guest-to-host distance is 2.4. Consider the host $v_4$ of $v_{10}$. There exists an entry $(v_{10}, 1.5)$ in $\mathcal{G}(v_4)$. The estimated host-to-host distance is 1.5. In this case, the estimated host-to-host distance is equal to $|\Pi_{\mathcal{G}}(v_2, v_4)| = 2.1 + 2.5 = 4.6$.

All pairs of hosts of $v_6$ and hosts of $v_{10}$ correspond to $\mathcal{H}(v_6) \times \mathcal{H}(v_{10})$. It is easy to verify that $(v_2, v_4)$ is the best host pair of $(v_6, v_{10})$. Finally, the estimated distance between $v_6$ and $v_{10}$ is equal to $2.4 + 4.6 + 1.5 = 8.5$.\[\Box\]
2.5. GENERATING A SMALLER GRAPH

Algorithm 2: Algorithm for Removing a Vertex $v$ from $G'$ (i.e., $o(G', v)$)

1: $N \leftarrow N_{G'}(v)$
2: for each $v' \in N$ do
3: \hspace{1em} $\tilde{D}_{v'} \leftarrow \tilde{d}_{G'}(v, v')$
4: end for
5: remove $v$ from $V'$
6: remove all edges containing $v$ from $E'$
7: $P \leftarrow$ a polygon formed from a set of all the remaining edges which are adjacent to any two vertices in $N$
8: triangulate $P$
9: $H(v) \leftarrow N$
10: for each $v' \in N$ do
11: \hspace{1em} $\mathcal{G}(v') \leftarrow \mathcal{G}(v') \cup \{v\} \cup \mathcal{G}(v)$
12: \hspace{1em} $\mathcal{G}L(v') \leftarrow GLInsert(\mathcal{G}L(v'), (v, \tilde{D}_{v'}))$
13: \hspace{1em} for each $(\pi, \tilde{d}) \in \mathcal{G}L(v)$ do
14: \hspace{2em} \hspace{1em} $\mathcal{G}L(v') \leftarrow GLInsert(\mathcal{G}L(v'), (\pi, \tilde{d} + \tilde{D}_{v'}))$
15: \hspace{1em} $H(\pi) \leftarrow H(\pi) - \{v\} \cup \{v'\}$
16: end for
17: end for
18: return $G'$

How to Perform $o(G', v)$

The next issue is the details of the operation $o(G', v)$. Algorithm 2 shows the algorithm for the operation $o(G', v)$. In this algorithm, line 1 corresponds to finding all neighbors of the vertex $v$ to be removed. Given a vertex $v$ and a vertex $v' \in V'$, $v$ is a neighbor of $v'$ if and only if $(v, v') \in E'$ (or $(v', v) \in E'$). Given a vertex $v$ in $V'$ (from $G'$), we denote the set of all neighbors of $v$ to be $N_{G'}(v)$. In this algorithm, line 7 corresponds to a standard triangulation method in the TIN model which is a process that partitions a given polygon into a number of triangles. We adopted the Delaunay triangulation method [7].

Moreover, in this algorithm, we introduce a function called $GLInsert$ which takes the guest information list of a vertex $v'$, $\mathcal{G}L(v')$, and an entry in the form of $(v, \tilde{d})$ as inputs, and outputs the updated guest information list of $v'$. The function returns the updated guest information list according to the following cases. 

Case 1: There does not exist any entry $(v, \tilde{d}) \in \mathcal{G}L(v')$. In this case, the updated guest information list to be returned is set to $\mathcal{G}L(v') \cup \{(v, \tilde{d})\}$. 

Case 2: There exists an entry $(v, \tilde{d}) \in \mathcal{G}L(v')$. In this case, we further consider two sub-cases. 

Case 2(a): $\tilde{d} \leq \tilde{d}'. \hspace{1em}$ In this sub-case, the updated guest information list to be returned is set to $\mathcal{G}L(v')$. 

Case 2(b): $\tilde{d}' > \tilde{d}$. In this sub-case, the updated guest information list to be returned is set to $\mathcal{G}L(v') - \{(v, \tilde{d}')\} \cup \{(v, \tilde{d})\}$. 


CHAPTER 2. FINDING SHORTEST PATHS ON TERRAINS

Checking Whether $G'$ Satisfies Estimated Distance Property

The third issue is how to check whether $G'$ satisfies the estimated distance property. Specifically, in Algorithm 1 whenever we remove a vertex $v$ from $V'$, we have to check whether the resulting graph after the vertex removal operation satisfies the estimated distance property (Property 2) (and thus the network distance property (Property 1)). This property requires that for any two vertices in $V$, Inequality 2.7 is satisfied. Checking this property naively is time-consuming. Fortunately, we just need to check two properties which are related to the neighbors of $v$ only, instead of all vertices in $V$. In Section 2.5.2 we will show that if these two properties are satisfied, then Property 2 holds.

Before we introduce the two properties, let us give an intuition of these two properties. Consider an iteration of Algorithm 1. Just before this iteration, suppose that $G'$ denotes the current graph with some vertices removed. Let $v$ be the vertex to be removed in this iteration. It generates $G''$ which is equal to $o(G', v)$. Roughly speaking, we want to maintain two kinds of distance information stored in the graph after $v$ is removed.

- The first kind of distance information is the intra-distance information. We want to make sure that the pairwise (estimated) distance between any two neighbors of $v$ does not change too much after $v$ is removed from the graph.
- The second kind of distance information is the inter-distance information. We want to make sure that the pairwise (estimated) distance between each neighbor of $v$ and each guest of $v$ does not change too much after $v$ is removed from the graph.

The two properties are formally given as follows. The first property is called the neighborhood error bound property which is used to maintain the intra-distance information. The second property is called the host-guest error bound property which is used to maintain the inter-distance information.

**Property 3 (Neighborhood Error Bound Property)** Let $G'$ be a graph. Given a vertex $v$ in $V'$, $v$ is said to satisfy the neighborhood error bound property in $G'$ if and only if for any two vertices $v_i$ and $v_j$ in $N_{G'}(v)$,

$$|\Pi_{G'}(v_i, v_j)| \leq \tilde{d}_{G''}(v_i, v_j) \leq \omega \cdot |\Pi_G(v_i, v_j)|,$$

where $G'' = o(G', v)$.

**Property 4 (Host-Guest Error Bound Property)** Let $G'$ be a graph. Given a vertex $v$ in $V'$, $v$ is said to satisfy the host-guest error bound property in $G'$ if and only if for each vertex $\overline{v} \in \mathcal{G}(v)$ and each vertex $v' \in N_{G'}(v)$,

$$|\Pi_G(v', \overline{v})| \leq \tilde{d}_{G''}(v', \overline{v}) \leq \omega \cdot |\Pi_G(v', \overline{v})|,$$

where $G'' = o(G', v)$.
2.5. GENERATING A SMALLER GRAPH

Given a graph $G'$ and a vertex $v \in V$, $v$ is said to satisfy the removal property in $G'$ if and only if $v$ satisfies both the neighborhood error property and the host-guest error property in $G'$.

With this removal property, in Algorithm, we change the checking condition in line 2 to that “there exists a vertex $v$ in $G'$ such that $v$ satisfies the removal property in $G'$.”

How to Find Vertex to be Removed

The last issue is how to find a vertex to be removed. As we described in the previous section, we need to find a vertex $v$ in $G'$ such that $v$ satisfies the removal property. In our implementation, we find this vertex $v$ by processing all vertices in a particular order based on the regularity of the polygon $P$ formed from a set of all the remaining edges which are adjacent to any two vertices in $N_{G'}(v)$. We define a function $f(v)$ which takes a vertex $v$ as an input and returns a non-negative real number as an output denoting how regular the shape of $N_{G'}(v)$ is. Specifically, we define $f(v)$ to be the average difference between an interior angle and the average interior angle within the polygon $P$. If $f(v)$ is smaller, then $P$ is more regular. Here, we would like to choose a vertex $v$ whose polygon is more regular for processing first. Triangulating the polygon can result in many triangles with more regular shapes. It is more likely that more triangles with more regular shapes increases the opportunity of simplifying the graph in the later process. This is because an irregular triangle containing one long side and one short side has two extreme scenarios for simplifying the graph, resulting in a lesser opportunity to remove vertices in this triangle.

2.5.2 Analysis

In this section, we show that the smaller graph $G'$ satisfies the estimated distance property (Property 2).

**Lemma 4** Let $G'$ be the graph generated by Algorithm. Then, $G'$ satisfies the estimated distance property.

**Proof Sketch:** According to Property, we want to show that for any two vertices $s$ and $t$ in $V$, $|\Pi_{G}(s, t)| \leq \tilde{d}_{G'}(s, t) \leq \omega \cdot |\Pi_{G}(s, t)|$.

In this proof, we focus on Case 1 mentioned in Section. Cases 2 and 3 can be shown similarly.

Let $u$ be the best host of $s$ and $\tilde{d}_1$ be the corresponding guest-to-host distance. Let $u'$ be the best host of $t$ and $\tilde{d}_2$ be the corresponding host-to-guest distance. We have

$$\tilde{d}_{G'}(s, t) = \tilde{d}_1 + |\Pi_{G'}(u, u')| + \tilde{d}_2. \quad (2.8)$$

By Property, we deduce that for any two remaining vertices $v$ and $v'$ in $V'$, we have

$$|\Pi_{G}(v, v')| \leq \tilde{d}_{G'}(v, v') \leq \omega \cdot |\Pi_{G}(v, v')|. \quad (2.9)$$
Inequalities 2.9 and 2.10, we derive that
\[ \tilde{d}(v, \overline{v}) \geq D \]  

Equation 2.8, we derive that
\[ d \leq |\Pi_G(s, u)| + |\Pi_G(w, u')| + |\Pi_G(u', t)| \geq |\Pi_G(s, t)|. \]  

Consider the distance \( D \) from \( s \) to \( t \) in \( G' \) which is equal to the sum of the guest-to-host distance (i.e., \( \tilde{d}_{G'}(s, w) \)), the shortest distance from \( w \) to \( w' \) and the host-to-guest distance (i.e., \( \tilde{d}_{G'}(w', t) \)). Note that
\[ D = \tilde{d}_{G'}(s, w) + |\Pi_{G'}(w, w')| + \tilde{d}_{G'}(w', t). \]  

By Inequalities 2.9 and 2.10, we derive that
\[ D \leq \omega \cdot |\Pi_G(s, w)| + \omega \cdot |\Pi_G(w, w')| + \omega \cdot |\Pi_G(w', t)|. \]  

Thus, from Equation 2.11 we obtain
\[ D \leq \omega \cdot |\Pi_G(s, t)|. \]  

Since \( D \geq \tilde{d}_{G'}(s, t) \), we have
\[ d \leq \omega \cdot |\Pi_G(s, t)|. \]  

A detailed proof can be found in [37].

\[ 2.6 \quad \text{Experiments} \]

\[ 2.6.1 \quad \text{Experimental Setup} \]

\textbf{Data Sets and Parameter Settings:} Experiments were conducted on the Eagle Peak (EP) dataset (http://data.geocomm.com/). This widely used dataset is from Wyoming, USA, covers an area of 10.7 x 14 km², and has 1.3 million data points [13, 44, 56, 68, 69]. We used sub-regions of varying sizes to obtain robust results.

The experiments were conducted by varying several parameters to study the effect of the trade-offs among accuracy, efficiency, and memory usage. Table 2.1 shows the parameters with their default values shown in bold. The default value of \( \omega \) is set to 1.2, which means that there is a 20% error for generating a smaller graph. Experiments were conducted with default parameter values unless explicitly stated.

\textbf{Implementation:} The core algorithms were implemented in C and C++, and some auxiliary tasks were implemented in Perl. A terrain tool, developed by CMU, called
Triangle (http://www.cs.cmu.edu/~quake/triangle.html), was employed for generating the TIN model with a minimum interior angle quality. In all datasets, with this tool, \( \theta_m \) generated is at least 45°.

The Chen-and-Han implementation\cite{34} was used to compute shortest surface paths. All experiments were carried out on a Fedora 18 Linux machine with an Intel Xeon E5 CPU (20MB cache, hyper-threading, 8 cores) and 32 GB internal memory.

All experiments were conducted 100 times. Average values were reported in our final results. For each spatial query with a query location, following \cite{56, 69}, we generate a query location randomly and select 10% of the vertices in the TIN model randomly as objects.

Note that there are two contributions in this paper. The first contribution is the proposed tighter bounds, and the second contribution is the proposed smaller graph. In order to highlight the significance of our contributions, we study them both individually and combined. In Section 2.6.2, we study the effect of our tighter bounds based on the original graph. Section 2.6.3 shows how a compressed smaller graph affects existing bounds. In Section 2.6.4, we show how our bounds based on our smaller graph improve existing results. Finally, Section 2.6.5 depicts the scalability of our bound computation and algorithms using our bounds.

### 2.6.2 Our Bounds and the Original Graph

In Section 2.6.2, we compare our distance bounds with existing distance bounds based on an original graph. In Section 2.6.2, we study how the performance of some existing algorithms (described in Section 2.4) are improved when our bounds are used.

#### Distance Bound Comparison

Based on the original graph, on average our lower bound is 5,075 meters, while the existing lower bound is 2,671 meters only. The improvement ratio for the lower bound on EP is 1.9. Conducting the same experiment with different source and destination points, and also over different data set sizes (20K–1M vertices), we get an overall average improvement ratio of 2.8.

#### Impact on Existing Methods

We study how our bounds can be used for three popular spatial queries, namely (1) surface \( k \)-NN queries, (2) surface range queries, and (3) reverse surface NN queries. (1) Surface \( k \)-NN Queries: The impact of our bounds for surface \( k \)-NN queries based on the original graph \( G \) are studied. We compared three algorithms, namely the straightforward approach (\( SF \)), the \( MR3 \) approach (\( MR3 \) \cite{13}) and the Voronoi diagram-based approach (\( VOR \)) \cite{56}.

\( SF \) is an algorithm containing two steps. In the filtering step, it finds all objects whose lower bounds of their shortest surface distances to a given query point \( q \) are
at most the $k$-th smallest upper bound of the shortest surface distance from an object to $q$. In the refinement step, it then computes the shortest surface distances of all objects found in the filtering step and returns $k$ objects with the least shortest surface distances.

For the rest of the paper, we denote the various combinations of algorithms, bound types, and graph types as $A$-$O$Bound($G$). $A$ is a placeholder for the implemented algorithms, with possible values \{SF, MR3, VOR, MSRNN\} (where MSRNN is an algorithm which will be used later in our experiments). $O$ can be original/existing (Org) or new (Our) bounds, and $G$ can be the original graph ($G$) or the smaller graph ($G'$). For example, SF$\cdot$OrgBound($G$) denotes the SF algorithm using existing original bounds on the original graph $G$.

In this section, we are studying the performance of $A$-OurBound($G$) compared with $A$-OrgBound($G$) for each existing algorithm $A$.

Figure 2.9(a) shows that every algorithm $A$ using our bounds (i.e., $A$-OurBound($G$)) is faster than its counterpart using the original existing bounds (i.e., $A$-OrgBound($G$)) on graph $G$. Since $k$ increases, fewer candidates (Figure 2.9(b)) need refinement due to our tighter lower bounds, resulting in an order of magnitude speedup in VOR/SF. Specifically, when $k$ increases from 2 to 20, there is an increase from 9 to 25.2 times, respectively. Although VOR has a larger candidate set, its query time is the lowest. This is because it has the lowest cost of processing per candidate compared with other algorithms. Specifically, VOR precomputes the tight/loose cells and computes shortest surface distances incrementally by expanding cells. Since other algorithms lack such an incremental cell expansion, they have larger query times.

(2) Surface Range Queries: We conducted experiments for surface range queries with a fixed range of 500m. Since there are no existing algorithms for these queries, we conducted experiments with a straightforward (SF) algorithm only. Similar to surface $k$NN queries, in the context of surface range queries, SF contains two steps. In the filtering step, it finds all objects whose lower bounds of their shortest surface
2.6. EXPERIMENTS

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Our Bound Only (Section 2.6.2)</th>
<th>Small Graph Only (Section 2.6.3)</th>
<th>Our Bound and Small Graph (Section 2.6.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface k-NN (VOR)</td>
<td>9</td>
<td>15</td>
<td>38</td>
</tr>
<tr>
<td>Surface Range</td>
<td>10</td>
<td>1</td>
<td>23.4</td>
</tr>
<tr>
<td>Surface Reverse k-NN</td>
<td>18.6</td>
<td>10</td>
<td>31.7</td>
</tr>
</tbody>
</table>

Table 2.2: Speedup Comparison (With Default Parameters)

distances to a given query point \( q \) are at most a given range value \( r \). In the \textit{refinement} step, it computes the shortest surface distances of all objects found in the filtering step and finally returns all objects whose shortest surface distances to \( q \) are at most \( r \).

Similar to surface \( k \)-NN queries, we implemented it with two variations, namely \( SF-\text{OrgBound}(G) \) and \( SF-\text{OurBound}(G) \). The query time and the number of candidates of \( SF-\text{OrgBound}(G) \) are 1, 400 seconds and 52, respectively. But, the query time and the number of candidates of \( SF-\text{OurBound}(G) \) are 107 seconds and 7, respectively, showing a speedup of nearly an order of magnitude.

(3) Reverse Surface NN Queries: We implemented the algorithm for monochromatic reverse surface NN queries in [69], namely \( MSRNN \). In the following, we focus on reverse surface 1-NN queries. Similar to surface \( k \)-NN queries, we implemented it with two variations, namely \( MSRNN-\text{OrgBound}(G) \) and \( MSRNN-\text{OurBound}(G) \). The query time and the number of candidates for \( MSRNN-\text{OrgBound}(G) \) is 2.8 second and 28, respectively. However, the query time and the number of candidates for \( MSRNN-\text{OurBound}(G) \) is 0.15 second and 10, respectively, which means that the algorithm using our bounds is 18.6 times faster and explores fewer candidates when compared to the one using the existing bounds.

\textit{Conclusion:} We find that the algorithms using our bounds on the original graph perform more efficiently than the algorithms using existing bounds on the same original graph. Refer to the first contribution in the column with header “Our Bound Only” in Table 2.2 for speedups.

2.6.3 Existing Bounds and Our Smaller Graph

Distance Bound Comparison

Existing lower bound computation uses the Euclidean distance which is independent of the underlying graph and hence is unaffected by our graph compression. On the other hand, the upper bound which is the network distance on the smaller graph is affected. When \( \omega = 1 \), the existing upper bound calculated based on a smaller graph is 10, 150 meters (same as the network distance on the original graph) and takes 24.4 milliseconds to compute. However, it takes 2.8 seconds to find the existing upper bound calculated based on the original graph. Note that when \( \omega = 1 \), the bounds calculated based on the original graph are exactly the same as those calculated based on the smaller graph. Besides, there are some vertices which can be removed even
when $\omega = 1$, resulting in a faster time. When $\omega$ increases to 2, the reduction in the number of vertices causes the upper bound to reach 10, 290 meters with a smaller runtime of 8.1 milliseconds. When bounds are computed based on $G'$, substantial speedup is achieved.

**Impact on Existing Methods**

For each algorithm $A$, we study the effect of $A$ using the original bounds on the smaller graph $G'$ ($A$-OrgBound($G'$)) to $A$ using original bounds on $G$ ($A$-OrgBound($G$)).

1) **Surface k-NN Queries:** We varied $\omega$ and $k$ to study the effects. Figure 2.10 shows the results when $\omega$ is varied. In Figure 2.10a, when $\omega$ lies between 1 and 1.2, graph compression causes speedier bound computations. At $\omega = 1.1$, we achieve speedups of nearly 15 and 28 for VOR and SF, respectively.
However, for $\omega > 1.4$, the query times of $SF$-$OrgBound(G')$ and $VOR$-$OrgBound(G')$ get larger than $SF$-$OrgBound(G)$ and $VOR$-$OrgBound(G)$ respectively, because the bounds are looser, resulting in more exact surface distance computations. Figure 2.10(b) shows the number of candidates explored in each algorithm.

Figure 2.11 shows the results when $k$ is varied. We observe a similar trend as before.

(2) Surface Range Queries: Improvement in surface range queries can only be found when the lower bound is improved, which in turn improves the pruning capacity in the filtering step. Since the original lower bound is the Euclidean distance, which is inert to changes in the underlying graph structure, we notice that varying $\omega$ does not change the lower bounds and hence does not affect the query time and the number of candidates to refine. In our experiments, the query time and the number of candidates to be refined are 1, 400 seconds and 52, respectively.

(3) Reverse Surface NN Queries: Since, in Figure 2.12, we observe a similar behavior as the Surface $k$-NN Queries in Section 2.6.3(1), the same explanations hold true in this case too. Note that in Figure 2.12(a), when $\omega = 1.4$ (i.e., 40% error), the query time of $MSRNN$-$OrgBound(G')$ is slightly larger than that of $MSRNN$-$OrgBound(G)$. This is because when $\omega$ is a large value (in this case, $\omega = 1.4$), the error is already large (40%). Then, the bounds calculated based on $G'$ are larger, resulting in more candidates which need more surface shortest path queries. We argue that 40% is already a large error and thus it is not recommended to set $\omega$ to a large value (e.g., 1.4).

Conclusion: We find that the algorithms using existing bounds on the smaller graph perform more efficiently than those using the same existing bounds on the original graph when $\omega$ is set to a value smaller than 1.4 (40% error). Refer to the second contribution in column with header “Small Graph Only” in Table 2.2 for speedups.
Figure 2.13: Impact of Our Bounds on Our Smaller Graph - Distance Bounds: Effect of $\omega$

### 2.6.4 Our Bounds and Our Smaller Graph

#### Distance Bound Comparison

In Figure 2.13, we denote $OrgLB(G)$ to be the original lower bound on $G$ and $OurLB(G')$ to be our lower bound on $G'$. We also denote $UB(G)$ and $UB(G')$ to be the upper bounds on $G$ and $G'$, respectively. We denote $Ds$ to be the surface distance.

Figure 2.13(a) shows that our lower bound $OurLB(G')$ is larger than the existing lower bound $OrgLB(G)$. Figure 2.13(b) shows that the computation time of $Ds$ is the greatest and it took 504 hours to compute $Ds$. Computing the network distance on a smaller graph $G'$ (i.e., $UB(G')$ and $OurLB(G')$) is nearly an order of magnitude faster than computing the network distance on an original graph (i.e., $UB(G)$). Even when $\omega = 1$, the computation of the network distance on a smaller graph $G'$ is faster. Figure 2.13(c) shows that the number of remaining vertices in $G'$ decreases when $\omega$ increases. Furthermore, Figure 2.13(d) shows that the time of generating $G'$ increases when $\omega$ increases because more vertices are removed.
2.6. EXPERIMENTS

Figure 2.14: Impact of Our Bounds on Our Smaller Graph - Surface $k$-NN Queries: Effect of $\omega$

Figure 2.15: Impact of Our Bounds on Our Smaller Graph - Surface $k$-NN Queries: Effect of $k$

Impact on Existing Methods

Similar to previous sections, for each algorithm $A$, we study the effect of our bounds on our smaller graph, denoted as $A$-OurBound($G'$).

1) Surface $k$-NN Queries:

Figure 2.14 shows the performance of algorithms when $\omega$ changes. Figure 2.14(a) shows that the query times of SF-OurBound($G'$) and VOR-OurBound($G'$) increase with $\omega$ because the upper/lower bounds calculated are looser and thus more candidates are generated for computing the exact shortest surface distances (as illustrated in Figure 2.14(b)). At $\omega = 1.1$, SF and VOR show speedups of 96 and 38 times, respectively.

Figure 2.15 shows the results of varying $k$. Figures 2.15(a) and (b) show that the query times and the number of candidates of all algorithms increase with $k$.

2) Surface Range Queries: Figure 2.16(a) shows that the query time of SF-OurBound
(G') is smaller than that of SF-OrgBound(G) since the computation time of SF-OurBound(G') is based on a smaller graph G' compared with SF-OrgBound(G) which is based on G. Specifically, our tightest bound (ω = 1) produces a speedup of 23.4 times. When ω increases, the computation time of SF-OurBound(G') increases, since more candidates objects have to be explored (as shown in Figure 2.16(b)) due to looser bounds.

(3) Reverse Surface NN Queries: Figure 2.17(a) shows that the query time of MSRNN-OurBound(G') is smaller than that of MSRNN-Orgbound(G) when ω is smaller than 1.8. At ω = 1, a speedup of 31.7 times is achieved. Figure 2.17(b) shows fewer candidates explored by the algorithm using our new bounds.

Conclusion: We find that the algorithms using our new tighter bounds on the smaller graph perform more efficiently than those using existing bounds on the original graph when ω is set to a value smaller than 1.4 (40% error). Table 2.2 compares the speedups of our individual contributions, i.e., “Our Bound Only” and “Small Graph
2.6. EXPERIMENTS

Here, we study the scalability of the existing algorithms described in Section 2.6.4 by varying the dataset size which is defined to be the total number of vertices used in the model.

Consider the scalability of surface $k$-NN queries with $k$ set to 5. The $SF$, $MR3$ and $VOR$ approaches using our bounds (i.e., $SF$-$OurBound(G')$, $MR3$-$OurBound(G')$ and $VOR$-$OurBound(G')$) have shorter query times compared with these approaches using the original bounds, as shown in Figure 2.18(b). In particular, the speedups of $SF$, $MR3$ and $VOR$ using our bounds are up to 68, 6.2 and 43 times, respectively, which is quite significant. Figure 2.18(a) shows the corresponding preprocessing times of these algorithms.

Consider the scalability for surface range queries (Figure 2.19). Similar results can be found in the figure. In particular, the speedup of $SF$ using our bounds is at

![Figure 2.18: Scalability: Surface k-NN Queries](image)

![Figure 2.19: Scalability: Surface Range Queries](image)
least 32.5 times.

### 2.6.6 Summary

In our experimental studies, our lower bound is up to 2.8 times larger (or better) than the Euclidean distance, the popular lower bound adopted in the literature. At $\omega = 1$, the computation of our upper bound computed on $G'$ is 10 times faster than that of the original upper bound computed on $G$ (with the same upper bound value). Importantly, all existing approaches relying on lower and upper bounds experience considerable speedups with our new bounds. In particular, the speedup experienced by VOR, i.e., the state-of-the-art algorithm, is up to 43 times for surface $k$-NN queries on the largest dataset (1M vertices, $k = 5$), which is quite significant.

In general, the best speedups are achieved when using our bounds on the smaller graph. A smaller graph can give a positive effect on faster bound computations but it can also introduce a negative effect on looser bounds (resulting in more candidates explored in some spatial queries). In our experiments, we find that $\omega$ should be set to a value smaller than 1.4 (which means a 40% error, a large error). When $\omega$ is set to a value smaller than 1.4, the positive effect outweighs the negative effect. When $\omega$ is set to a value larger than 1.4, in some cases, the negative effect may dominate the positive effect.

### 2.7 Conclusion

In this paper, we study a fundamental operation, i.e., shortest surface path computation, which is used widely in spatial queries. We find that we can compute the shortest network distance once and then use this distance for both the upper bound and lower bound of the shortest surface distance, which incurs only little overhead. In addition, when we need to compute the bounds quicker, we propose a method to generate a smaller graph from the Delaunay graph of the terrain such that the bound computation can be faster. Our experiments show that our lower bound is much tighter than the best-known lower bound. They also show that the existing state-of-art surface $k$-NN algorithm, i.e., VOR, can be speeded up nearly 43 times in the best case on the largest dataset.

There are a lot of promising research directions. First, it is of interest to derive the lower bound and upper bounds of the shortest surface path when the slope constraint is considered [44]. Second, it is of interest to study real time spatial queries such as continuous $k$ nearest neighbors using our bounds.

### 2.8 Acknowledgments

The work conducted by the co-authors from Aarhus University was supported by the Reduction project, funded by the European Comission as FP7-ICT-2011-7 STREP project number 288254. The work done by Raymond Chi-Wing Wong was supported
by grant DAG11EG05G. The authors would also like to thank Dominik Scheder and Boris Aronov for valuable discussions and the reviewers for valuable comments.
Chapter 3

Constant-Factor Lower Bound for Shortest Distance Queries on Terrains

Abstract

The widespread use and availability of massive and accurate laser data that covers a large expanse, have enabled the formulation of well-known spatial queries on terrain data. Computing the exact surface shortest path between two points on a terrain forms the crux of this new class of terrain spatial queries. However, the surface shortest distance computation on large terrains has been found to be extremely time consuming.

In order to speedup these queries, we reduce the number of expensive surface shortest distance computations by employing our new lower and upper distance bounds that are tighter than the recent state-of-the-art bounds proposed. Unlike the state of the art, our bounds do not rely on the quality of the triangulation, i.e., its minimum interior angle $\theta_{\text{min}}$.

Our proposed bounds are theoretically proven to be tighter and extensive empirical studies show evidence of speedups nearly an order of magnitude higher for some of the well-known terrain spatial queries.

3.1 Introduction

Three dimensional mesh representations of actual terrain surfaces to simulate natural phenomena is gaining traction and becoming commonplace due to the availability of massive and accurate laser data that covers a large expanse.

Common applications that require to compute spatial queries on such mesh data include robotic path planning for unmanned vehicles on unknown terrains, georealistic computer games and battlefield simulations for tactical path planning. Some other applications in the medical field also conduct spatial queries on 3D mesh models that are bio-realistic. E.g, neuroscientists conduct spatial range queries on brain mesh simulations to study the neuron density and number of branches in an area [62].
CHAPTER 3. IMPROVED BOUNDS FOR SURFACE SHORTEST PATH

Several recent works in academia [13, 44, 56, 68, 69] have focused on extending well-known spatial queries, from the Euclidean space to terrains. The surface shortest path computation that underlies all these terrain spatial queries is identified to be extremely expensive and time consuming. The best known algorithm to compute the exact surface shortest path is proposed by Chen and Han [4] and has a time complexity of $O(N^2)$, where $N$ denotes the total number of vertices in the triangulated surface. Therefore, in order for the spatial queries on terrains to be effective, they employ distance bounds in order to prune as many unnecessary candidates as possible before refinement, which results in avoiding expensive surface shortest path computations.

Given a source point $s$ and destination point $t$ on a terrain, the aforementioned works used the Euclidean distance $\Pi_e$ and the shortest network distance $\Pi_n$, as the lower and upper bounds for the surface shortest path $\Pi_s$, respectively. Figure 3.1 illustrates all three $(s, t)$-paths, where $\Pi_s$ is allowed to cut across the faces of the triangulation, while $\Pi_n$ is restricted to the edges of the faces. The motivation behind computing these bounds was their ease of computation in comparison to computing the exact surface shortest path $\Pi_s$. Kaul et al. [38] show that a surface shortest path computation on a triangulation with 20K vertices, takes 7.2 hours, while computing the shortest network path on the same triangulation takes only 0.04 seconds in comparison. However, the lower bound was shown to be nearly 9 times smaller than $\Pi_s$ [38], which was a very loose bound, as it greatly underestimated $\Pi_s$.

Kaul et al. [38] proposed a much tighter lower bound that captured the information about the surface of the underlying terrain. Both lower and upper bounds are derived from the shortest network distance $\Pi_n$. More specifically, they compute the lower bound as $\lambda \cdot \Pi_n$, where $\lambda = \min\left\{ \frac{\sin \theta_{\min}}{2}, \sin \theta_{\min} \cos \theta_{\min} \right\}$ and $\theta_{\min}$ is the minimum interior angle of all faces in the terrain. The application of these bounds to all the existing surface spatial queries, namely $k$-NN, reverse $k$-NN and range queries, showed a substantial improvement in execution times.

![Figure 3.1: Paths illustrated on surface.](image)
3.1. INTRODUCTION

The quality of the state-of-the-art bounds proposed in [38] has a dependence on the minimum interior angle $\theta_{\text{min}}$ of all faces in the terrain. Observe that a larger $\theta_{\text{min}}$ improves the tightness of their lower bound, while decreasing $\theta_{\text{min}}$ results in their lower bound loosening. Thus, for some low values of $\theta_{\text{min}}$, their lower bound quality deteriorates to be worse than even the Euclidean distance $\Pi_c$. Thus, they use a constrained Delaunay triangulation to ensure that $\theta_{\text{min}}$ is always greater than 45°.

To better illustrate, Figure 3.2 shows a triangulation with two degenerate triangles, $\Delta AsB$ and $\Delta AtB$, that share a common base and where $s$ and $t$ are the source and target vertices, respectively.

![Figure 3.2: Terrain with degenerate faces.](image)

This example shows that in the presence of such degenerate triangles, for source vertex $s$ and target vertex $t$, the surface shortest path $\Pi_s$, which is allowed to cut across a face is much smaller than the network path $\Pi_n$, that is restricted to only the edges of the faces. Thus, $\Pi_n$ does not form a good approximation for $\Pi_s$ in this case. Additionally, $\theta_m$ is also substantially decreased.

Motivated by the above observations, we propose a tighter lower bound for the surface shortest path $\Pi_s$, that does not rely on $\theta_{\text{min}}$. Our lower bound is unaffected by the introduction of degenerate skinny triangles (smaller $\theta_m$) in the triangulation. In comparison, the quality of the state-of-the-art lower bound algorithm deteriorates when $\theta_{\text{min}}$ is lowered.

Our approach allows our approximation path to cut-across a face, like $\Pi_s$ does, and is thus also a tighter approximation to $\Pi_s$. We term this new approximation as a face-crossing path $\Pi_{fc}$. Further, $\Pi_{fc}$ cuts across a face, via additional cut vertices that are placed strategically on the edges of the face. Our lower bound is then computed as $\lambda' \cdot |\Pi_{fc}|$, where $\lambda'$ is a constant fraction, whose value is 0.9. In our experiments, we study the effect on the tightness of our lower bound, by varying the amount of cut-vertices introduced on each edge. Doing so, causes the values of $\lambda'$ and $|\Pi_{fc}|$ to be adjusted accordingly.

Our contributions can be summarized as follows. First, to the best of our knowledge, we are the first to propose a tighter constant-factor lower bound for the surface shortest path, that is always tighter than the lower bound proposed in [38]. Second, we theoretically prove the tightness of our lower bound and show how it is always tighter than the lower bound proposed in [38] for all possible values of $\theta_{\text{min}}$. Third, we introduce a user-defined error parameter $\epsilon$ in our experiments, to better comprehend the trade-off between the tightness of our lower bound and the bound computa-
tion time. Studying the effect of our bounds on existing surface spatial queries gives us a better understanding of how our new bounds affect the performance of such fundamental queries on the surface. Fourth, we present a comprehensive empirical study that offers insight into the accuracy, efficiency, and scalability properties of our proposed lower bound.

The remainder of the paper is organized as follows. Section 3.2 describes related work. Section 3.3 formulates the problem. Section 3.4 describes our lower and upper bounds. Section 3.5 shows our algorithm. Section 3.6 covers the empirical study of our bounds and proposed algorithm. Finally, Section 3.7 concludes the paper.

### 3.2 Related Work

In this section, we review some of the existing works that explore spatial queries on terrains, that make use of distance bounds. These works propose models to combat the major underlying challenge, which is the massive computational cost associated with the exact shortest surface path calculation on a terrain.

Further, we discuss the state-of-the-art lower and upper distance bounds computation that enables speedups in the spatial queries on terrains by eliminating expensive surface shortest distance computations, by proposing much tighter lower bounds.

**Surface Spatial Queries** Deng et al. [13] reduce the cost by simplifying the surface data to various resolutions and storing them in a hierarchical structure. Based on this multi-resolution model, they begin by computing $k$-NN results on simpler surfaces, and move to higher resolutions when the $k$-NN results are ambiguous and require more refinement. Although, later works [56, 69] find that the multi-resolution model does not give accurate $k$-NN results, especially when $k$ gets larger. Additionally, this method incurs a large storage overhead.

Shahabi et al. [56] propose an extension to the Voronoi-diagram to surfaces to compute the surface $k$-NN query in an incremental fashion. This method is the state-of-the-art method for this query. This scheme computes so-called tight and loose cells and stores them together in an R-tree like structure, they term as the SIR-tree. They incrementally expand the query search region using the SIR-tree and report $k$-NN results as they come available.

Xing et al. [68] extend the work by Shahabi et al. [56] by allowing continuous surface $k$-NN queries in in a highly dynamic environment which allows for arbitrary movements of data objects.

Yan et al. [69] propose an algorithm to find the bichromatic and monochromatic reverse nearest neighbors on terrains. Their algorithm makes use of the tight and loose cells proposed in [56].

**State-of-the-art Distance Bounds for Surface Queries** All the aforementioned works used the Euclidean distance between two points on the surface as the lower bound for the shortest surface path between the two points. Kaul et al. [38] proposed a much tighter lower bound than the Euclidean distance, which when applied to the existing works on surface spatial queries showed a marked improvement in query
execution times. More specifically, in [56] when the lower bounds proposed in [38] are used, the tight cells increase, while their surrounding loose cells shrink, causing a speedup due to being able to prune candidates more effectively.

To the best of our knowledge, the lower bound proposed by Kaul et al. [38] is the first study that captures information about the surface of the terrain by computing both the lower and upper bounds from the shortest network path computed on the Delaunay Graph representation of the surface. They further propose a graph compression technique to speedup the bound computation.

This method proposed a lower bound that depended on $\theta_{\text{min}}$, i.e., the minimum interior angle of any triangle in the triangulation. They proposed their distance bounds on realistic terrain models [8], because the tightness of their lower bound depends on $\theta_{\text{min}}$. The Computational Geometry community avoids very complicated, hypothetical inputs (such as degenerate skinny triangles, as shown in Figure 3.2) to create algorithms that are provably efficient in realistic situations and hence uses constrained Delaunay triangulations, where the minimum interior angle can be forced to exceed a specified minimum threshold.

**Surface Shortest Path Approximation Algorithms** In the field of Computational Geometry the problem of computing an approximate shortest path on a non-convex polyhedral surface is still considered a challenging open problem. Varadarajan et al. [65], partition the faces of the polyhedron using the well-known planar separator theorem [43] and compute the shortest path on the resulting graph. They present an algorithm with quadratic time complexity which produces an approximate shortest path that is at most $7$ times the optimal shortest surface path.

[42, 48] consider the problem where each face of the given polyhedron has a weight associated with it and the cost of traversing a face is the distance traveled on the face times the weight of the face. They employ a strategy of introducing new points on the edges of the polyhedron and connecting these points with new edges to be treated as a graph. This work is closest to our work as we too employ a similar point placement strategy. The runtime complexity of their approach is $O(n^3)$, where $n$ is the total number of vertices in the graph. While, the goal of their work is to compute an upper bound on the surface shortest path length, our work focuses on computing a lower bound, which presents us with a different set of challenges.

[33] propose another method that also places points on edges, but uses a selective-refinement strategy to iteratively use Dijkstra’s algorithm on the discrete graph of the polyhedron to reduce the region in which the shortest path can exist. Unlike our method, their method does not provide any theoretical bounds on the approximate shortest surface paths they compute.

The aforementioned approximation algorithms require a pre-processing step to place new vertices on all edges of the triangulation, which does not scale well on larger datasets. In comparison our algorithm generates these points on-the-fly during the expansion of our search frontier, which produces much quicker results that also scale well on large terrains.
3.3 Preliminaries

Consider a three-dimensional space. Each point \( p \) is represented by an \( x \)-coordinate, a \( y \)-coordinate, and a \( z \)-coordinate. Usually, the \( z \)-coordinate of a point is said to be the elevation of this point.

A terrain is the graph of a continuous function that assigns every point on a horizontal plane to an elevation. In the literature, a terrain is typically represented by a triangulated irregular network (TIN) model that consists of a set \( T \) of faces each of which is represented by a triangle. Each triangle has three corners called vertices and three edges, each connecting two of its corners. Each vertex is a point in the three-dimensional space. We assume that each interior angle of every triangle/face is non-zero. To ensure this, we replace any triangle violating this assumption by a single edge. Figure 3.1 shows an example of a terrain. Two distinct triangles are said to be adjacent if they share an edge \( e \). In this model, there are two types of triangles in \( T \), namely normal triangles and boundary triangles. Each normal triangle is adjacent to three other triangles in \( T \), and each boundary triangle is adjacent to one or two other triangles in \( T \). A point \( p \) is said to be on the terrain if there exists a triangle in \( T \) such that \( p \) is on the plane containing this triangle and \( p \) is inside this triangle.

We denote \( V \) and \( E \) to be the set of all vertices and the set of all edges in the model. The Delaunay graph \( G \) of the terrain is defined to be a weighted graph where the set of vertices and the set of edges in this graph are \( V \) and \( E \), respectively, and the weight of each edge is the Euclidean distance between the two end-points of the edge. In the following, for brevity, we simply write “graph” for “Delaunay graph.” We also use the terms “surface”, “terrain” and “triangulation” interchangeably.

We denote the line segment connection between two points \( p \) and \( p' \) to be \((p, p')\). We define the length of line segment \((p, p')\), denoted by \(|(p, p')|\), to be the Euclidean distance between \( p \) and \( p' \).
Given two vertices $s$ and $t$ in $V$, a surface path from $s$ to $t$, denoted by $\Pi(s,t)$, is a sequence $\langle p_1, p_2, ..., p_n \rangle$ where (1) $p_1 = s$, (2) $p_n = t$, and (3) each $p_i$ is a point along an edge in $E$. The surface path is thus composed of $n-1$ line segments, $(p_1, p_2), (p_2, p_3), ..., (p_{n-1}, p_n)$. Figure 3.1 shows an example of a surface path $P$ from $s$ to $t$. The length of a surface path $\Pi(s,t)$, denoted by $|\Pi(s,t)|$, is defined to be $\sum_{i=1}^{n-1} (p_i, p_{i+1})$. The shortest surface path from $s$ to $t$ is the surface path from $s$ to $t$ with the smallest length.

Further, a network path from $s$ to $t$, denoted by $\Pi_G(s,t)$, is represented by a sequence $\langle v_1, v_2, ..., v_n \rangle$ where (1) $v_1 = s$, (2) $v_n = t$, (3) each $v_i$ is a vertex in $V$, and (4) each $(v_i, v_{i+1})$ is an edge in $E$. Figure 3.1 shows an example of a network path from $s$ to $t$. The length of a network path $\Pi_G(s,t)$, denoted by $|\Pi_G(s,t)|$, is defined to be $\sum_{i=1}^{n-1} |(v_i, v_{i+1})|$. Finally, the shortest network path from $s$ to $t$ is defined to be the network path from $s$ to $t$ with the smallest length.

From the literature, we have the following lemma.

**Lemma 5** ([13, 56, 69]) For any two vertices $v$ and $v'$ in $V$, $|\Pi(v, v')| \leq |\Pi_G(v, v')|$. 

Further, we define $V'$ to be the set of newly introduced vertices located between the end-points of each edge in $E$. We refer to these new vertices as cut-vertices. We introduce an edge $(v_a, v_b)$ between cut-vertices $v_a$ and $v_b$ if they are located on adjacent edges of a face. This new set of edges is denoted as $E'$. Given the definitions of $V, V', E$ and $E'$, we proceed to define a surface face-crossing path $\Pi_{FC}(s,t)$, which is represented by a sequence $\langle v_1, v_2, ..., v_n \rangle$ where (1) $v_1 = s$, (2) $v_n = t$, (3) each $v_i$ is a vertex in $V \cup V'$, and (4) each $(v_i, v_{i+1})$, except when $v_i$ and $v_{i+1}$ are located on the same edge, is a newly introduced edge in $E \cup E'$. The path $\Pi_{FC}(s,t)$, unlike the network path $\Pi_G(s,t)$, is not restricted to the edges of the face, but can also cut across the face of a triangle.

For better illustration, Figure 3.3 shows $\Delta ABC$ with corner vertices $A, B, C \in V$ and cut vertices $\{A', A'', B', B'', C', C''\} \in V'$. The dotted lines in $\Delta ABC$ denote the newly introduced edges in $E'$ (e.g., $C'B'$). Note that there are no new edges introduced between any pair of vertices placed on the same edge, except the original edge between the corner vertices of that edge. Hence, edges $(C', C'')$ or $(B', C)$ are non-existent. These additional edges in $E'$ form a nearly complete graph and allow additional possibilities for path $\Pi_{FC}(s,t)$ to cut across the face of $\Delta ABC$ via the cut-vertices introduced on the edges.

### 3.4 Upper and Lower Bounds

In this section, we give the theoretical property that a lower bound for the shortest surface path $\Pi(s,t)$ between vertices $s$ and $t$ can be obtained as a constant ratio of the surface face-crossing path $\Pi_{FC}(s,t)$.

Suppose that $\Pi(s,t)$ is a sequence $< p_1, p_2, ..., p_n >$, where $p_i$ is a point along an edge in $E$. Note that $p_1 = s$ and $p_n = t$. Each line connecting $p_i$ and $p_{i+1}$ is on face...
Additionally, we define that each point $p_i$ has its owner, denoted by $o_i$, which is one of the closest corners in $V$ or cut-vertices in $V'$, of the face containing $p_i$. Thus, line segment $(o_i, o_{i+1})$ can be viewed as the closest approximation to line segment $(p_i, p_{i+1})$ and $(o_i, o_{i+1})$ is termed as the owner segment of line segment $(p_i, p_{i+1})$. Figure 3.4(c) shows the closest vertices to $p_i$ and $p_{i+1}$ chosen as their owners, $o_i$ and $o_{i+1}$, respectively. Additionally, Figure 3.4(c) also illustrates how segment $(p_i, p_{i+1})$ is approximated by segment $(o_i, o_{i+1})$. Furthermore, we set $o_1 = s$ and $o_n = t$.

In the vicinity of corner vertices, line segment $(p_i, p_{i+1})$ can have an infinitesimal length and hence cannot be approximated by a corresponding $(o_i, o_{i+1})$ line segment. Thus, we introduce a minimum length threshold $|(o_i, o_{i+1})|_{\text{min}}$, below which any $(p_i, p_{i+1})$ has its corresponding $o_i = o_{i+1}$, i.e., they are approximated by the corner vertex itself and $|(o_i, o_{i+1})| = 0$.

For illustration, in Figure 3.4, the corner vertex is $A$ and the length of line segment $(X, Y)$ denotes our $|(o_i, o_{i+1})|_{\text{min}}$. Since, $|(p_i, p_{i+1})| \leq |(o_i, o_{i+1})|_{\text{min}}$, both $p_i$ and $p_{i+1}$ have the same owner, i.e., corner vertex $A$ and $|(o_i, o_{i+1})| = 0$.

![Figure 3.4](image)

Furthermore, our cut-vertex placement strategy must ensure that every possible owner segment $(o_i, o_{i+1})$ satisfies our constant bound property. In order to do this, we must first ascertain the minimum possible length $|(o_i, o_{i+1})|_{\text{min}}$ of any owner segment $(o_i, o_{i+1})$ in face $f_i$.

Let $\theta_m$ denote the minimum interior angle of a single face, not to be confused with the earlier $\theta_{\text{min}}$, which is the minimum interior angle amongst all the faces belonging to a triangulation $\mathcal{T}$. Depending on the shape of the face, $|(o_i, o_{i+1})|_{\text{min}}$ varies accordingly. For example, $|(o_i, o_{i+1})|_{\text{min}}$ is shorter for faces with a smaller $\theta_m$, than faces with larger $\theta_m$.

In order to compute $|(o_i, o_{i+1})|_{\text{min}}$ for a face $f_i$, we focus on the corner vertex $v_{\text{cor}}$ to which $\theta_m$ belongs. Let $l_{\text{min}}$ denote the length of the shortest edge in all the faces of $\mathcal{T}$. Then, an initial distance of $l_{\text{min}}/2$ from corner vertex $v_{\text{cor}}$ is chosen on both the edges that are adjacent to $v_{\text{cor}}$. Figure 3.4 shows an example where $v_{\text{cor}} = A$ and $|AX| = |AY| = l_{\text{min}}/2$, forming an isosceles triangle $\Delta AXY$, where line segments $AX$ and $AY$ subtend the angle $\theta_m$ of the triangle.

Applying the law of cosines, we have $|(o_i, o_{i+1})|_{\text{min}} = l_{\text{min}}/2 \cdot \sqrt{2(1 - \cos \theta_m)}$. We chose $l_{\text{min}}/2$ as an initial start value, to improve the time complexity of our
3.4. UPPER AND LOWER BOUNDS

After having computed the $|\overline{o_i, o_{i+1}}|_{min}$ for a face, cut-vertices are placed on the edge with a gap $\Delta I$ that is at most $1/K \cdot |\overline{o_i, o_{i+1}}|_{min}$, where $K$ is a constant set to 10. Setting $K = 10$, we have $\Delta I \leq \frac{|\overline{o_i, o_{i+1}}|_{min}}{10}$, which also ensures that our lower bound is always tighter than the bound proposed in [38] for all possible values of $\theta_{min}$. Note that $K$ can be replaced by a user defined variable, but this results in added time-complexity, since many more cut-vertices get introduced on the edges.

More intuitively, for lower values of $\theta_{min}$ (skinny faces), $|\overline{o_i, o_{i+1}}|_{min}$ is lowered and so is $\Delta I$, thus many more cut-vertices and new edges connecting these cut-vertices are introduced on this face, improving how closely segment $\overline{o_i, o_{i+1}}$ approximates segment $\overline{p_i, p_{i+1}}$. This property allows our lower bound to achieve a constant-factor bound even for degenerate skinny faces.

Consider a sequence $\Pi_{FC}(s, t) : \overline{o_1, o_2, ..., o_n}$. Each $o_i$ is a vertex in $V \cup V'$, on the terrain. Each line segment connecting $o_i$ and $o_{i+1}$ is on face $f_i$. Additionally, for each $\overline{o_i, o_{i+1}}$ segment, we have a minimum length $|\overline{o_i, o_{i+1}}|_{min}$, as shown in Figure 3.4. Thus, $\Pi_{FC}(s, t)$ is made up of line segments that either have (i) $o_i = o_{i+1}$ and zero-length or (ii) $\overline{o_i, o_{i+1}}$ is an edge with non-zero length in $E \cup E'$.

With these definitions, we derive the lower and upper distance bounds of the shortest surface path. We begin by breaking down the $(s, t)$-paths into line segments that must traverse over faces and provide distance bounds over these individual line segments. Later, we accumulate the results to provide the final bounds for the entire shortest surface path $\Pi(s, t)$.

**Lemma 6** Let $(p_i, p_{i+1})$ denote the segment portion of the shortest surface path $\Pi(s, t)$ on face $f_i$, and $(o_i, o_{i+1})$ denote the segment portion of the surface face-crossing path $\Pi_{FC}(s, t)$ on face $f_i$. Then,

$$\lambda \cdot |\overline{o_i, o_{i+1}}| \leq |\overline{p_i, p_{i+1}}|$$

where $\lambda \cdot |\overline{o_i, o_{i+1}}|$ is our lower bound, $\lambda = 0.9$ and $\Delta I \leq \frac{|\overline{o_i, o_{i+1}}|_{min}}{10}$.
Proof: This proof can be broken down into several cases.

Case 1: \( o_i = o_{i+1} \)  This case is illustrated in Figure 3.5(a). Here, \(|(o_i, o_{i+1})| = 0\). The inequality holds.

Case 2: Both \( p_i \) and \( p_{i+1} \) lie along the same edge \((v_a, v_b)\) and \( o_i \neq o_{i+1} \).

This case is illustrated in Figure 3.5(b), where bold lines indicate the scenario depicted in this case, and the dotted lines indicate the actual path that should have been followed. This case never arises because it violates a fundamental observation in [57] which states that a surface shortest path must become a straight line segment when the faces crossed by the path are unfolded onto a plane. It is easy to note that when unfolding the faces over which P crosses, path \( \langle p_{i-1}, p_i, p_{i+2} \rangle \) unfolds into a straight line, which is a shorter path than path \( \langle p_{i-1}, p_i, p_{i+1}, p_{i+2} \rangle \).

Case 3: \( p_i \) and \( p_{i+1} \) do not lie on the same edge and \( o_i \neq o_{i+1} \). We begin proving this sub-case by making a claim as follows.

Claim 1 Let \( (o_i, o_{i+1}) \) denote the segment of path \( \Pi_{FC}(s, t) \) that crosses face \( f_i \) and \((p_i, p_{i+1})\) denote the segment of path \( \Pi(s, t) \) that crosses face \( f_i \). Then,

\[
|\|(o_i, o_{i+1})| - \Delta I| \leq |(p_i, p_{i+1})|\
\]
3.4. UPPER AND LOWER BOUNDS

![Diagram](image)

Figure 3.7: Subcases 2(a) and 2(b) in Claim Case 2.

Proof: We provide two figures for each sub-case, to illustrate how faces with smaller \(p_m\) have more finer intervals, i.e., lower \(\Delta I\), and hence result in more cut-vertices being placed on each edge. Also note that there are edges joining every pair of vertices placed on different edges of the face, thus forming a nearly complete graph (corner vertices placed on the same edge, do not have an edge between them). These new edges have been omitted from Figure 3.6 for clarity of images. Two sub-cases arise.

**Case 1:** \(p_i\) is located on a corner vertex and \(p_{i+1}\) is located on a cut-vertex of the opposite edge This case is illustrated in Figures 3.6(a) and (b). In this case, since \(p_i\) is on a corner-vertex, we choose the closest vertex \(o_i\) to be the same as \(p_i\). \(p_{i+1}\) is closest to \(o_{i+1}\) and the length of segment \((o_{i+1}, p_{i+1})\), is at most \(\frac{1}{2}\Delta I\). By the triangle inequality, we have that
\[
|p_i, o_{i+1}| \leq |(p_i, p_{i+1})| + \frac{1}{2}\Delta I.
\]
Re-arranging this we get
\[
|p_i, o_{i+1}| - \frac{1}{2}\Delta I \leq |(p_i, p_{i+1})|.
\]

**Case 2:** \(p_i\) and \(p_{i+1}\) are located on cut-vertices of adjacent edges. This case is illustrated in Figures 3.6(c) and (d).

When \(p_i\) and \(p_{i+1}\) are located on cut-vertices of adjacent edges, then their corresponding owners, i.e., \(o_i\) and \(o_{i+1}\), are cut-vertices that are closest to \(p_i\) and \(p_{i+1}\), respectively. In order to prove the inequality, we study the relation between edge \((p_i, p_{i+1})\) and edge \((o_i, o_{i+1})\). In Case 2, we notice 3 more subcases as follows.

**Case (a) Edges \((p_i, p_{i+1})\) and \((o_i, o_{i+1})\) cut across each other** This case is illustrated in Figure 3.7(a). \((p_i, p_{i+1})\) is indicated with a solid line, while \((o_i, o_{i+1})\) is shown with a dashed line. Let \(X\) denote the point at which both edges cross each other.

In \(\Delta p_i o_i X\), using the triangle-inequality we have
\[
|o_i, X| \leq |p_i, X| + |(p_i, o_i)|.
\]
Applying the same in \(\Delta X p_{i+1} o_{i+1}\), we arrive at the inequality:
\[
|o_{i+1}, X| \leq |X, p_{i+1}| + |p_{i+1}, o_{i+1}|.
\]
Adding both inequalities, we have
\[
|o_i, X| + |o_{i+1}, X| \leq |p_i, X| + |X, p_{i+1}| + |p_{i+1}, o_{i+1}| + |o_i, o_{i+1}|.
\]
We observe that
\[
|p_i, X| + |X, p_{i+1}| = |p_i, p_{i+1}| \quad \text{and} \quad |o_i, X| + |X, o_{i+1}| = |o_i, o_{i+1}|.
\]
Also, we ob-
serve that \(|(p_i, q)| \leq \Delta I/2\) and \(|(p_{i+1}, q)| \leq \Delta I/2\). Introducing these two observations into the previous inequality, we have \(|(o, o_{i+1})| \leq |(p_i, p_{i+1})| + \Delta I\), which then re-arranges to \(|(o, o_{i+1})| - \Delta I \leq |(p_i, p_{i+1})|\).

**Case (b) Edges \((o, o_{i+1})\) and \((p_i, p_{i+1})\) do not cross over and \(|(o, o_{i+1})| \geq |(p_i, p_{i+1})|\).** This case is illustrated in Figure 3.7b. \((p_i, p_{i+1})\) is indicated with a solid line, while \((o, o_{i+1})\) is shown with a dashed line. We also have a dotted line joining \(o\) and \(p_{i+1}\).

Consider \(\Delta p_i p_{i+1} o_i\). Applying the triangle inequality, we get our first inequality: \(|(o, p_i)| \leq |(p_i, p_{i+1})| + |(p_i, o_i)|\). In \(\Delta o_i p_{i+1} o_{i+1}\), we get the second inequality as \(|(o, o_{i+1})| \leq |(p_i, o_i)| + |(p_{i+1}, o_{i+1})|\). Using the RHS of our first inequality, we substitute \(|(o, p_i)|\) in the second inequality. We get, \(|(o, o_{i+1})| \leq |(p_i, p_{i+1})| + |(p_i, o_i)| + |(p_{i+1}, o_{i+1})|\). The same reduction steps as in Case (a) can be applied here to finally reduce to \(|(o, o_{i+1})| - \Delta I \leq |(p_i, p_{i+1})|\).

**Case (c) Edges \((o, o_{i+1})\) and \((p_i, p_{i+1})\) do not cross over and \(|(o, o_{i+1})| \leq |(p_i, p_{i+1})|\).** This subcase is proven trivially since when \(|(o, o_{i+1})| \leq |(p_i, p_{i+1})|\), then \(|(o, o_{i+1})| \leq |(p_i, p_{i+1})| + \Delta I\) also holds, which finally then re-arranges to \(|(o, o_{i+1})| - \Delta I \leq |(p_i, p_{i+1})|\).

Finally, combining the results from Cases 1 and 2, we get \(|(o, o_{i+1})| - \Delta I \leq |(p_i, p_{i+1})|\). This completes the proof for our claim.

Following our proven Claim, we know that the inequality \(|(p_i, p_{i+1})| \geq |(o, o_{i+1})| - \Delta I\) holds. Thus, substituting our chosen value of \(\Delta I = \frac{|(o, o_{i+1})|_{\text{min}}}{10}\), we have \(|(p_i, p_{i+1})| \geq (o, o_{i+1}) - \frac{|(o, o_{i+1})|_{\text{min}}}{10}\).

We also know that \(|(o, o_{i+1})|_{\text{min}} \geq |(o, o_{i+1})|\). Thus, the inequality \(|(p_i, p_{i+1})| \geq \frac{|(o, o_{i+1})| - \frac{|(o, o_{i+1})|_{\text{min}}}{10}}{10}\) also holds because \(\frac{|(o, o_{i+1})|_{\text{min}}}{10}\) is a larger value reduced from \(|(o, o_{i+1})|\).

Finally, we get 0.9 \(|(o, o_{i+1})| \leq |(p_i, p_{i+1})|). This completes our proof for the lower bound where \(\lambda = 0.9\), which is a constant bound.

**Theorem 1 (Distance Bound)** Let \(\Pi_{FC}(s, t)\), \(\Pi_G(s, t)\) and \(\Pi(s, t)\) be a surface face-crossing path, the shortest surface path and the shortest network path between source \(s\) and destination \(t\) on terrain \(P\), respectively. Then,

\[ \lambda \cdot |\Pi_{FC}(s, t)| \leq |\Pi(s, t)| \leq |\Pi_G(s, t)| \]

where \(\lambda = 0.9\).

**Proof:** There are two inequalities: \(\lambda \cdot |\Pi_{FC}(s, t)| \leq |\Pi(s, t)|\) and \(|\Pi(s, t)| \leq |\Pi_G(s, t)|\). The second inequality is derived from Lemma. In the following, we focus on the first inequality.

Recall that \(\Pi_{FC}(s, t) = \langle o_1, o_2, ..., o_{k+1} \rangle\), with \(k\) segments. Also, observe that consecutive edges \((o_i, o_{i+1})\) and \((o_{i+1}, o_{i+2})\) always share a common vertex \(o_{i+1}\), thus making sure that \(\Pi_{FC}(s, t)\) will always be a connected path. This holds even
when some edges are approximated as a single vertex, e.g., Case 1 in Lemma 6 (Figure 3.5(a)).

Applying Lemma 6, we know that \(\lambda \cdot |(o_i, o_{i+1})| \leq |(p_i, p_{i+1})|\), for each \(i \in [1, k+1]\). Combining the inequalities for each segment constituting paths \(\Pi_{FC}(s, t)\) and \(\Pi(s, t)\) we get, \(\lambda \cdot \sum_{i=1}^{k} |(o_i, o_{i+1})| \leq \sum_{i=1}^{k} |(p_i, p_{i+1})|\). This further simplified gives, \(\lambda \cdot |\Pi_{FC}(s, t)| \leq |\Pi(s, t)|\), which completes our proof.

### 3.5 Bound Algorithm

Our bound computation algorithm computes the lower and upper bound distances for the shortest surface path \(\Pi(s, t)\) between a source vertex \(s\) and target vertex \(t\) on the graph \(G\). While Lemma 6 assumed knowledge of path \(\Pi(s, t)\), our algorithm to compute the bounds cannot make such an assumption.

Some noteworthy properties of the shortest surface path \(\Pi(s, t)\) are as follows.

1. \(\Pi(s, t)\) is composed of a series of straight line segments, where each line segment, touching the face, connects the points on adjacent edges of single face.

2. \(\Pi(s, t)\) traverses any given face at most once and does not bend on the interior of a face.

3. Two shortest surface paths \(\Pi^{(1)}(s, t)\) and \(\Pi^{(2)}(s, t)\), originating from a common source \(s\) never cross each other.

Based on these aforementioned properties, we observe that surface paths originating from the source vertex \(s\), start by cutting across faces adjacent to vertex \(s\) and propagate outwards like a wavefront from the edges of the adjacent faces that are opposite to vertex \(s\). Further, when a surface path emanates from an edge, it exits the face attached to the edge, from either of the adjacent edges to the entry edge.

Similar to the shortest surface path \(\Pi(s, t)\), we follow the surface face-crossing path \(\Pi_{FC}(s, t)\) to compute our bounds, by taking into account the faces that might be traversed by all possible surface paths \(\Pi^{(1)}(s, t), \Pi^{(2)}(s, t),...,\Pi^{(n)}(s, t)\) that originate from source \(s\).

We consider the surface paths crossing from one face to another, via an edge, as an extension and we represent this with two pairs called events. An event \(E\) is a pair \(\langle u, D(u) \rangle\), where \(u\) is the closest vertex from which the extension occurs and \(D(u)\) our calculated shortest network distance of \(u\) from source vertex \(s\).

Algorithm 3 stores the events described above in a priority queue \(Q\) and processes them in a fashion similar to the Dijkstra-algorithm for computing shortest paths. Algorithm 3 then applies the cases of Lemma 6 to each triangle. Additionally, to improve the performance of our algorithm, we use an \(A^*\)-like heuristic, where we add new cut-vertices to only edges of faces, that are closer to our destination vertex \(t\). We add another speedup optimization that checks whether an edge has
already had cut-vertices placed on them from previous rounds, in which case no new cut-vertices are introduced to this edge. The 3-D Euclidean distance from the vertex being processed to the target vertex $t$ is used. The pseudo-code for the algorithm is outlined in Algorithm 3.

**Example 3** Consider the example terrain with two adjacent faces $f_1$ and $f_2$ illustrated in Figure 3.8. In this example, we set the source $s = 1$ and the target vertex as $t = 4$. Here, face $f_1$ has $\theta_m = 45^\circ$ and $i = 0.8$, while face $f_2$ has $\theta_m = 60^\circ$ and $i = 1$. We purposely chose higher interval $i$ values, in order to reduce the number of intermediate cut-vertices that our algorithm must place and consider when con-
3.5. BOUND ALGORITHM

Algorithm 3: Compute Distance Bound

```plaintext
get.bounds(G, s, t)
1: for each v ∈ G do
2: Δ(v) ← ∞
3: end for
4: Δ(s) ← 0
5: Initialize event priority queue Q
6: while Q not empty do
7: ⟨u, Δ(u)⟩ ← Extract vertex of event with least Δ(u)
8: if u = t then
9: Compute (lb, ub) = (0.9 × Δ(u), 1.1 × Δ(u))
10: output (lb, ub)
11: end if
12: S ← S ∪ {u}
13: d(u, t) ← Compute vertex u’s distance to t
14: for each adjacent face f of vertex u do
15: Compute for face f:
16: θ_m and interval i
17: if u is a corner-vertex then
18: e′ ← get the edge opposite u in face f
19: L ← L ∪ e′
20: else if u is a cut-vertex then
21: e ← get edge on which u is located
22: {e′, e″} ← get both adjacent edges to e
23: L ← L ∪ {e′, e″}
24: end if
25: for each edge (v_a, v_b) in list L do
26: d((v_a, v_b), t) ← Compute edge (v_a, v_b)’s distance to t
27: if d(u, t) ≥ d((v_a, v_b), t) then
28: j ← 1
29: while j ≤ ⌊(v_a, v_b)⌋ do
30: Place cut-vertex v_c at i · j from v_a on edge (v_a, v_b)
31: if u and v_c not in δ-neighborhood(u) and
32: |(u, v_c)| ≥ 10i then
33: Add edge (u, v_c)
34: end if
35: Insert v_c into Q.
36: end while
37: end if
38: end for
39: Δ(v) ← min {Δ(v), Δ(u) + |(u, v)|}
40: Update ⟨v, Δ(v)⟩ in Q
41: end for
42: end while
```
structuring the shortest path. \{1, 2, 3, 4\} is the set of corner vertices and \{5, 6, 7\} are cut-vertices introduced by the algorithm. Also, note that there are no edges connecting cut-vertices that lie on the same edge, e.g., no edge connects vertices 2 and 5, or vertices 3 and 5. The grayed dashed line joining source and target vertices 1 and 4, is the shortest surface path \(\Pi(s, t)\) and in our example, \(|\Pi(s, t)| = 2.125\).

We illustrate the step-by-step working of the bound computation algorithm outlined in Algorithm 3 in the table shown in Figure 3.8(b). The columns in gray show: 1) The vertex \(u\) extracted for processing by the algorithm, 2) the distance of the vertex from source vertex \(s\), 3) the faces adjacent to \(u\), and 4) the elements in \(Q\) prior to extraction of \(u\). The boxes in white following the gray boxes, indicate the actions taken by the algorithm to add new cut-vertices and edges.

In our example, \(\Pi_{FC}(1, 4) = (1, 5, 4)\) and \(|\Pi_{FC}(1, 4)| = 0.85 + 1.3 = 2.15\). Thus, giving a lower bound of 0.9 \times 2.15 = 1.935 and upper bound of 1.1 \times 2.15 = 2.365.

3.5.1 Correctness Proof

**Theorem 2** The bound computation algorithm \[\text{3}\] takes as input the Delaunay Graph \(G = (V, E)\), the source vertex \(s\) and target vertex \(t\). Computing the shortest \(\Pi_{FC}(s, t)\) after placement of cut-vertices and new edges, produces the lower and upper bounds of the shortest surface path \(\Pi(s, t)\) between \(s\) and \(t\).

**Proof:** This proof assumes no knowledge of the actual shortest surface path \(\Pi(s, t)\), unlike in Theorem 1. We use the fact that the shortest surface path \(\Pi(s, t)\) is also a face-crossing path, which is lower and upper bounded by ratios of the shortest \(\Pi_{FC}(s, t)\).

Recollect, that we denote \(\Pi(s, t)\) as a sequence \((p_1, p_2, \ldots, p_n)\) where (1) \(p_1 = s\), (2) \(p_n = t\), and (3) each \(p_i\) is a point along an edge in \(E\). The surface path is thus composed of \(n - 1\) line segments, \((p_1, p_2), (p_2, p_3), \ldots, (p_{n-1}, p_n)\).

We provide a proof by induction on \(|S|\) (size of the solution set). The base case arises when \(|S| = 1\). This implies that both \(s\) and \(t\) are one and the same, and hence \(|\Pi(s, t)| = |\Pi_{FC}(s, t)| = 0\). So, the base case is trivially true.

The inductive hypothesis is to assume that our claim is true for \(|S| = k \geq 1\), which implies that we have found a portion of the shortest \(\Pi_{FC}(s, t)\) that is valid.

Let \(u\) denote the vertex extracted from \(Q\) that is currently being processed. Then, path \(\Pi_{FC}(s, u)\) correctly bounds the shortest surface path \(\Pi(s, u)\) and is a part of the path \(\Pi_{FC}(s, t)\). From \(u\) onwards, we consider the faces adjacent to \(u\) and add to the correct edge constituting the face, depending on whether \(u\) is a cut-vertex or a corner vertex and also depending on whether the edge satisfies our heuristic of having a shorter distance to the target vertex \(t\). Each edge \((v_a, v_b)\) that satisfies the previously mentioned predicates, has a set of cut-vertices \(V_c\) added to it, at an interval of at most \(i\) apart from each other. For each vertex \(v \in V_c \cup \{v_a\} \cup \{v_b\}\), a new edge \((u, v)\) is added to \(G\). Each such edge is a possible extension to \(\Pi_{FC}(s, u)\), where each \((u, v)\) segment obeys the bounds in Lemma 6 for any possible extension of \(\Pi(s, u)\). Thus, extending in Dijkstra’s fashion using a priority queue \(Q\), must relax
3.5. BOUND ALGORITHM

one of the edges that all obey the bounds shown in Lemma 6. Thus, solving the case for \(|S| = k + 1\). The correctness of Dijkstra’s shortest path is well-known and this completes our proof.

**Proof:** We consider several cases presented by us and compare them, where possible, to the state-of-the-art\([38]\) on a case by case basis.

**Case 1:** \(o_i = o_{i+1}\) This case is illustrated in our Case 1 (Figure 3.5(a)) and their Case 1. Here, trivially \(|\Pi_{FC}(s, t)| = |\Pi_G(s, t)| = 0\). In this case, \(\mathcal{L}_n = \mathcal{L}_o\).

**Case 2(d)\([38]\) and Case 2 in our proof \(p_i\) and \(p_{i+1}\) lie on the same edge** This case is shown in Figures 3.5(b) and (c).

In both studies, \(o_i = p_i\) and \(o_{i+1} = p_{i+1}\), thus, \((o_i, o_{i+1}) = (p_i, p_{i+1})\). In [38], \(\lambda'\) achieves its maximum value of approximately 0.43 (i.e. tightest lower bound value) at \(\theta_{min} = 60^\circ\) and \((o_i, o_{i+1})\) is approximated as an edge of the triangle. In our proof, \((o_i, o_{i+1})\) is also an edge of the triangle, but since there is no dependency on \(\theta_{min}\), we have \(\lambda = 0.9\). Thus in this case, \(\mathcal{L}_n > \mathcal{L}_o\) holds, since \(\lambda \cdot |(o_i, o_{i+1})| > \lambda' \cdot |(o_i, o_{i+1})|\).

**Case 2(a)(b)(c)\([38]\) and Case 3 in our proof \(p_i\) and \(p_{i+1}\) do not lie on the same edge** This case is shown in Figure 3.6.

Let the \((o_i, o_{i+1})\) in our proof be denoted by \(O\) and the equivalent segment in the previous work [38] as \(O' = (o_i', o_{i+1}')\).

Two sub-cases arise here when comparing this situation in both works. And our proof is based on the following observation.

**Figure 3.9:** Subcases 1 and 2.
CHAPTER 3. IMPROVED BOUNDS FOR SURFACE SHORTEST PATH

Observation 1 In our proof, $O$ closely mimics the segment $(p_i, p_{i+1})$ by also cutting across faces in the triangulation, and hence is variable in length. In comparison, $O'$ can only approximate $(p_i, p_{i+1})$ by either an edge of the face or by a corner vertex, thus having only 2 possible values.

Subcase 1 This subcase is illustrated in Figure 3.9(a). Here, both $p_i$ and $p_{i+1}$ are above the mid-point line (e.g. $(C', D')$ in Figure 3.9(a)). In this case, $|O'| = 0$ always. In our proof, when both $p_i$ and $p_{i+1}$ are not present within the $\delta$-neighborhood of a corner vertex, then $|O| \geq 0$. Thus, our proof presents a better lower bound in this situation, while the old method always gives a lower bound of $0$ in this subcase.

Subcase 2 This subcase is illustrated in Figure 3.9(b). Here, both $p_i$ and $p_{i+1}$ are below the mid-point line (e.g. $(C', D')$ in Figure 3.9(b)). Trivially, we have the inequality $|O'| \leq |O|$, due to the mid-point theorem. To make the RHS our lower bound, we multiply both sides by $0.9$, to get, $0.9 \cdot |O'| \leq 0.9 \cdot |O|$. This simplifies to $0.45 \cdot |O'| \leq \mathcal{L}_n$.

In [38], the best lower bound of $0.43 \cdot |O'|$ can only be achieved when $\theta_{\min} = 60^\circ$. Thus, our lower bound is always better than the lower bound proposed in [38] for $0^\circ < \theta_{\min} \leq 60^\circ$.

It is important to note that in practice it is extremely difficult to have $\theta_{\min} = 60^\circ$, thus [38] uses $\theta_{\min} = 45^\circ$, which gives a looser lower bound of $0.3|O'|$. This completes our proof.

3.5.2 Complexity Analysis

Let $V'$ denote the cut-vertices introduced on the edges by Algorithm 3 and thus $V'$ is the total number of cut-vertices. The total vertices in the graph is thus, $|V + V'|$. Therefore, in Algorithm 3 the main while loop (Lines 6–42) iterates over a priority queue $Q$, implemented as a Fibonacci Heap [21] with a maximum size of $|V + V'|$.

We begin by analyzing a single iteration of the while loop. Inside the while loop, on Line 7 a vertex $u$ is extracted from $Q$. The time complexity of this operation is $O(\log(|V + V'|))$.

Within the main loop, Lines 14–37 (for each face loop), loops and operates on all the faces that are adjacent to vertex $u$. Let $\Delta(G)$ denote the maximum degree of a vertex in $G$, then the number of faces adjacent to any vertex can be upper bounded by $\binom{\Delta(G)}{2}$. In reality, the number of faces can easily be upper bounded by a constant $K$, because there is a finite number of faces that can be adjacent to a vertex in a triangulation. Therefore, this loop is executed $\frac{\binom{\Delta(G)}{2}}{2}$ times.

Delving deeper into the faces loop, Lines 14–37, compute the edges of a single face that need cut-vertices introduced on. This is a maximum of 2 such edges at any time, per face, and are added to the edge-list $L$. Lines 24–36 (for each edge loop), operate on each of the edges in edge-list $L$. Here, we add new vertices and edges to the edges of the face. Let $l_{\max}$ denote the longest possible edge length of all faces in the triangulation and $i_{\min}$ denote the lowest possible granularity for interval lengths
3.6. Experiments

(gaps between vertices on an edge). Let $v_e^c$ denote the cut-vertices introduced on an edge $e$ and $|v_e^c|$ is the total number of cut-vertices added. Then, in the worse case, $|v_e^c| = \frac{l_{\text{max}}}{l_{\text{min}}} - 1$. We require a time complexity of $O(|v_e^c|)$ to add new vertices and $O(|v_e^c|^2)$ to add edges between a pair of new vertices. Line 33, inserts cut-vertices into $Q$, each requiring amortized time $O(1)$. Thus, the total time complexity for Lines 24–36 is $O(2 \cdot (|v_e^c| + |v_e^c|^2 + |v_e^c| \cdot 1))$, which further reduces to $O(|v_e^c|^2)$. Calculating the time-complexity for Lines 14–37, for all adjacent faces, becomes $O((\Delta(G))^2 \cdot |v_e^c|^2$).

Lines 38–41, update $Q$ for each adjacent vertex of $u$. The update operation for $Q$ has time $O(1)$. The maximum number of vertices to $u$ is bounded by the maximum degree of $u$, i.e., $\Delta(G)$. Thus, giving a time complexity of $O(\Delta(G))$.

Combining the overall time complexity for one iteration of the main while loop (Lines 6–42), we have $O(\log |V + V'| + (\Delta(G))^2 \cdot |v_e^c|^2 + \Delta(G))$. Since, $\Delta(G)$ has a constant upper bound in a triangulation, we can further reduce the time complexity to be $O(\log |V + V'| + |v_e^c|^2)$.

The worse case complexity is arrived at when the while loop iterates over all the elements in $Q$, which gives a final time complexity of $O(|V + V'| \cdot (\log |V + V'| + |v_e^c|^2))$.

In practise, as is evident from our experimental results, our lower bound computation algorithm performs much faster because it employs an $A^*$-like heuristic and also includes other speedup optimizations.

3.6 Experiments

In this section, we empirically study the performance of our proposed bounds, comparing it with the state-of-the-art bounds proposed by Kaul et al. [38].

3.6.1 Experimental Setup

Data Sets and Parameter Settings: Experiments were conducted on the Eagle Peak (EP) dataset (http://data.geocomm.com/). This widely used dataset is from Wyoming, USA, covers an area of 10.7 x 14 km$^2$, and has 1.3 million data points[13, 44, 56, 68, 69]. We generated sub-regions of sizes similar to [38] to compare our results.

The experiments were conducted by varying several parameters to study the effect of the trade-offs among accuracy, efficiency, and memory usage. Table 3.1 shows the parameters with their default values shown in bold. Experiments were conducted with default parameter values unless explicitly stated.

Error Parameter ($\epsilon$): In order to better evaluate the tradeoff between the tightness of our lower bound and the bound computation time, we introduce a new error parameter ($\epsilon$). At $\epsilon = 1$, our bound with constant-factor $\lambda = 0.9$ is achieved. For higher values of $\epsilon$, a larger $|(o_i, o_{i+1})|_{\text{min}}$ is chosen, which in turn increases the gap between cut-vertices, i.e. $\Delta I$. This increase results in fewer cut-vertices and edges
being introduced on each face, so that our face-crossing path $\Pi_{FC}(s, t)$ gets longer, but the constant-factor $\lambda$ drops, so that we get an overall looser lower bound. The opposite effect is achieved for values of $\epsilon < 1$.

**Implementation:** The core algorithms were implemented in C and C++, and some auxiliary tasks were implemented in Perl. A terrain tool, developed by CMU, called Triangle (http://www.cs.cmu.edu/~quake/triangle.html), was employed for generating the TIN model with a minimum interior angle quality. In addition to generating the constrained Delaunay triangulation of the terrain with $\theta_{min} = 45^\circ$, to compare with [38], we also generated a synthetic dataset, with this tool, setting $\theta_{min}$ to at least $30^\circ$.

The Chen-and-Han implementation [34] was used to compute shortest surface paths. All experiments were carried out on a Fedora 18 Linux machine with an Intel Xeon E5 CPU (20MB cache, hyper-threading, 8 cores) and 32 GB internal memory.

All experiments were conducted 100 times. Average values were reported in our final results. Following [38, 56], for each spatial query that required an initial query point, we generate a query location randomly and select 10% of the vertices in the TIN model randomly as objects.

In Section 3.6.2, we study the tightness of our lower bound by varying the error, i.e., $\epsilon$. In Section 3.6.3, we study the effect of our new tighter lower bounds on the state-of-the-art surface $k$-NN query algorithm [56]. In Sections 3.6.4 and 3.6.5, we show how the use of our newly proposed bounds improve the state-of-the-art algorithms for reverse surface NN query [69] and surface range query [38], respectively. Section 3.6.6 depicts the scalability of our bound computation and algorithms using our bounds. Finally, Section 3.6.7 summarizes our experimental findings.

### Table 3.1: Parameter Settings

<table>
<thead>
<tr>
<th>Dataset Sizes $D$ (points)</th>
<th>20K, 200K, 400K, 800K, 1000K</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Error Parameter $\epsilon$</td>
<td>0.1, 0.5, 1, 2, 5, 10</td>
</tr>
<tr>
<td>$\theta_{min}$</td>
<td>$30^\circ$, $45^\circ$</td>
</tr>
<tr>
<td>$k$</td>
<td>2, 5, 10, 15, 20</td>
</tr>
</tbody>
</table>

In Figures 3.10(a) and 3.10(b), we denote $D_s$ and $D_e$ to be the surface shortest path distance and Euclidean distance, respectively. We denote our upper bound, i.e., the network shortest path as $D_n$. Additionally, $PrevLB$ and $OurLB$ denote the state-of-the-art and our new tighter lower bounds, respectively.

Figure 3.10(a) shows that for the least error setting, i.e. $\epsilon = 0.25$, our lower bound ($OurLB$) is much larger than $PrevLB$ and much closer to $D_s$ (9,075 meters). Based on this error setting, on average $PrevLB$ for EP dataset is 5,075 meters, while $OurLB$ for EP is 8,862 meters, which gives an improvement ratio for the lower bound as nearly 1.75. As expected, when error is increased fewer cut-vertices are placed on the edges, thus loosening our lower bound distance that is based on the shortest
3.6. EXPERIMENTS

Figure 3.10: Distance Bounds ($\theta_{\text{min}} = 45^\circ$): Effect of $\epsilon$

Figure 3.11: Distance Bounds ($\theta_{\text{min}} = 30^\circ$): Effect of $\epsilon$
network path computation after new cut-vertices and the corresponding edges are added to the graph representation of the surface triangulation.

Figure 3.10(b) displays a reduction in runtime as error increases. Increasing the error results in fewer cut-vertices and new edges being introduced by our algorithm, which in turn, results in faster network path computation as the graph has significantly fewer edges to process. Also, note that even for the lowest error setting, our algorithm is nearly 3 orders of magnitude faster than the optimal surface shortest path computation, which takes nearly 504 hours to run. The PrevLB is always computed faster than our lower bound (OurLB), because it is based on computing the network shortest path on the original graph, which is much quicker. While, PrevLB beats OurLB in bound computation time, it results in a looser lower bound than ours, which later results in longer query times when the lower bound is used in other surface spatial queries.

Figure 3.10(c) shows that the average number of cut-vertices and corresponding edges introduced per face also decreases as ε increases. Figure 3.10(d) shows a slight variation in the average number of faces that are accessed during our algorithm’s search. It is important to note that due to our A*-like heuristic many faces are not processed, because their edges lie outside the ellipse shaped search frontier of our algorithm.

Figure 3.11(a) shows, for $\theta_{\min} = 30^\circ$, PrevLB becomes looser as their bound depends on the quality of $\theta_{\min}$, causing a loosening of their bound. PrevLB becomes nearly as loose as using the Euclidean distance $De$ as a lower bound. Thus, OurLB outperforms PrevLB with a larger extent for this setting of $\theta_{\min}$. For $\epsilon = 0.25$ (least error), we achieve an improvement ratio of nearly 3.1.

Figure 3.11(b) shows that the runtimes are unaffected. Figure 3.11(c) shows a larger number of cut-vertices and edges introduced per face because $\Delta I$ depends on $\theta_m$, where by reducing $\theta_m$ results in a finer $\Delta I$ and hence, more cut-vertices per edge. Note that reducing $\theta_{\min}$ for the triangulation $T$ also results in many triangles having their $\theta_m$ lowered. Due to an increase in the number of skinny triangles, we notice an increase in faces traversed too (Figure 3.11(d)).
3.6. EXPERIMENTS

Figure 3.13: Surface $k$-NN Queries ($\theta_{\text{min}} = 30^\circ$): Effect of $k$

Figure 3.14: Surface $k$-NN Queries ($\theta_{\text{min}} = 45^\circ$): Effect of $\epsilon$
3.6.3 Impact of our bounds on Surface $k$-NN Query

Here, we study the impact of our new bound on the existing state-of-the-art surface $k$-NN algorithm (VOR), that provides a Voronoi Diagram-based approach [56]. We briefly describe their technique as it is also used by the reverse surface NN state-of-the-art [69].

The VOR algorithm proposes a pre-processing task to generate tight and loose cells around each object $o$, based on similar techniques used to generate Voronoi cells. A tight cell is enclosed by a loose cell and both these cells were built by using the Euclidean distance as the lower bound and the shortest network distance as the upper bound. Their motivation being that a query point $q$ found in a tight cell of an object $o$ can immediately report $o$ as the 1-NN of $q$, thus avoiding an expensive surface shortest path computation. Further, they propose an algorithm, that begins by finding the tight cell and loose cells (loose cells can overlap each other, but tight cells cannot), and then they incrementally expand their neighboring loose cells to search for the other neighbors and report till they exhaust $k$ neighbors.

Kaul et al [38] showed that using their lower and upper bounds, the VOR algorithm’s tight cell area increased, while their loose cell area decreased. This resulted in faster computation of the surface $k$-NN because it avoided many expensive exact surface shortest path computations.

For the rest of the paper, we denote the combinations of algorithms and bound types as $A$-$O$Bound. $A$ is a placeholder for the implemented algorithms, with possible values $\{VOR,MSRNN,SF\}$, where MSRNN and SF are the reverse NN and range query algorithms, respectively, shown later in our experiments. $O$ can be either Euclidean distance bound ($Euc$), the state-of-the-art lower bound [38] ($Prev$) or our new tighter lower bound ($Our$). For example, $VOR$-$OurBound$ denotes the VOR algorithm using our proposed lower bound.

Figure 3.12(a) shows that algorithm VOR using our bounds ($VOR$-$OurBound$) is significantly faster than its counterpart using the state-of-the-art bounds, i.e., $VOR$-$PrevBound$.

As $k$ increases, we notice in Figure 3.12(b), that fewer candidates are needed to be further refined, in comparison to when using $VOR$-$PrevBound$, due to our tighter bounds that allow pruning many objects and hence resulting in an order of magnitude speedup in VOR. Recall that our tighter lower bounds result in larger tight-cells, which in turn increase the chances that the query point $q$ lies inside the tight-cell of an object $o$, which can immediately be returned as the nearest neighbor to $q$. When $k$ is increased from 2 to 20 for VOR, we notice speedup increases from 8 to 56 times, respectively.

When $\theta_{min} = 30^\circ$, in Figure 3.13(b), we notice that since $VOR$-$PrevBound$ is as loose as $VOR$-$EucBound$, they process nearly the same number of candidates. Figure 3.13(a) shows how $VOR$-$PrevBound$ is outperformed by $VOR$-$OurBound$. Specifically, when $k$ is increased from 2 to 20 for VOR, we notice speedup increases from 9 to 840 times, respectively.

Figure 3.14(a) shows the results when $\epsilon$ is varied. For the lowest error setting, $\epsilon =$
0.25, a speedup of nearly 23 times is achieved, in comparison to VOR-PrevBound. For the same reasons as explained in Section 3.6.2, increasing the error results in a reduction in the number of cut-vertices and edges, which loosens our lower bound, causing the VOR tight-cells to shrink and loose-cells to get larger, which means there is more overlap between loose-cells and more candidates need to be refined. For the default setting of $k = 2$, Figure 3.14(b) shows an increase in the number of candidates that need refinement as error goes up.

During VOR cell generation we use a 2-step filtering strategy, we first use the faster VOR-PrevBound algorithm to reduce the number of candidates, upon which we run our VOR-OurBound algorithm to further tighten the lower bound of these candidates. Figure 3.14(c) shows a decrease in the average number of cut-vertices and new edges added to each face, as the error is increased. Figure 3.14(d) shows the average number of faces that are traversed during the refinement of a candidate.

### 3.6.4 Impact of our bounds on Reverse Surface NN Query

We implemented the algorithm for monochromatic reverse surface NN queries in [69], namely MSRNN. In the following, we focus on reverse surface 1-NN queries only. This algorithm also uses the tight/loose cells proposed in VOR and hence is also affected in a similar fashion.

Figures 3.15(a) and (b) show a similar trend to what we observed earlier in Section 3.6.3 (Figures 3.14(a) and (b)), for the same reasons as explained earlier.

Figure 3.15(a) shows that at $\epsilon = 0.25$, where the tightest lower bound occurs, the query time for MSRNN-OurBound is 5 milliseconds, in comparison to 0.1 seconds for MSRNN-PrevBound, which results in an overall speedup of 20 times on average. Additionally, Figure 3.15(b) shows far fewer candidates explored by MSRNN-OurBound, when compared to MSRNN-PrevBound.
3.6.5 Impact of our bounds on Range Query

The surface range query is a straightforward (SF) algorithm that was implemented in [38]. SF is an algorithm with two steps. In the filtering step, the algorithm finds all objects whose lower bounds of their shortest surface distances to a given query point \( q \) are at most a given range value \( r \). The candidate objects found in the filtering step are then processed in the refinement step by computing each of their exact shortest surface distances to \( q \) and reporting the objects whose distances are at most \( r \).

In algorithm SF-OurBound, we use a 2-step approach in the filtering step of SF. First, the set of candidates are got by using the state-of-the-art faster and cheaper lower bound computation algorithm, i.e. PrevBound, upon which we later employ our new lower bound, that is more expensive to compute, to further refine the set of candidates. The advantage of this approach is that our new bound, which is comparatively more expensive to compute, is used on a reduced set of candidate objects, which saves a lot of computation time.

In Figure 3.16(a), as the error increases, our lower bound loosen's for the surface range query and thus the query time goes up. Since, the range query has no pre-processing step, the query time also includes the lower bound computation times. Thus, due to our 2-step approach in the filtering method we incur the bound computation time of PrevBound on all objects and the bound computation time of OurBound on the reduced set of candidates. Figure 3.16(b) shows that up to \( \epsilon = 2 \), the number of candidates to refine are lower for SF-OurBound and hence, we notice that the query times for SF-OurBound beat those of SF-PrevBound for values of \( \epsilon \) less than 2, after which our bounds get too loose and SF-PrevBound is the winner. Additionally, at \( \epsilon = 0.25 \), we achieve our tightest lower bound that gives a speedup of nearly 10 times when compared to the previous bound.

3.6.6 Scalability

The scalability of the existing algorithms using the state-of-the-art and our new bounds are studied by varying the dataset size. The size is varied by changing the
3.6. EXPERIMENTS

For the scalability study, we set $k = 5$, similar to [38]. Figure 3.17(a) shows the preprocessing times involved in generation of the VOR cells. We notice that VOR-OurBound is nearly 2 orders of magnitude slower than VOR-PrevBound, due to its expensive lower bound computation. This gap was reduced by employing the 2-step filtering strategy outlined in earlier sections.

Figure 3.17(b) shows that VOR has a much lower query time when using our new bounds. In particular, for the largest dataset, for the default settings, a speedup of nearly 18 times is achieved.

The preprocessing time of the surface range query, as shown in Figure 3.18(a), follows a similar trend to the surface $k$-NN query shown in Figure 3.17(a) for the same reasons. Figure 3.18(b), shows that for the largest dataset, a speedup of nearly 15.5 times is achieved.

3.6.7 Summary

Our experimental studies show that for $\theta_{\text{min}} = 45^\circ$ (default) and $\theta_{\text{min}} = 30^\circ$, our lower bound achieves an improvement ratio over the previous bound of 1.75 and 3.1,
respectively. In spite of a slower bound computation when compared to the state-of-the-art, our bounds are much tighter, which in turn results in far fewer expensive surface distance computations. More importantly, we observe a significant speedup in all the surface spatial queries tested. More specifically, the state-of-the-art surface $k$-NN query, i.e. VOR experiences a significant speedup of nearly 46 times on the largest dataset, which has 1 million vertices and $k = 5$. Table 3.2 outlines the speedups that were experienced for various surface spatial queries for different settings of $\theta_{\text{min}}$.

### 3.7 Conclusion

In this paper, we propose a new constant-factor lower bound for the surface shortest path, that is tighter than the previous state-of-the-art lower bound proposed in [38]. We achieve this by approximating the shortest surface path with a face-crossing path that closely mimics the shortest surface path, by also being able to cut across the faces of the surface triangulation, and hence produce a much closer approximation to the shortest surface path. Additionally, we ensure that the quality of our bound has no dependence on the minimum interior angle ($\theta_{\text{min}}$), thus our lower bound is unaffected by the introduction of degenerate skinny triangles in the triangulation, while the quality of the state-of-the-art lower bound algorithm deteriorates when $\theta_{\text{min}}$ is lowered. Furthermore, we provide theoretical proofs for the tightness of our lower bound and why our bounds always outperform the state-of-the-art bounds. Our experiments provide further insight into the tightness of our new bounds and also provide evidence of substantial speedups for some of the well-known terrain spatial queries.

Future research directions include studying the effect of our new bounds on continuous surface spatial queries and studying how our bounds can be modified to improve constrained surface shortest path queries, with constraints such as static and dynamic obstacles in the path between a source and destination.
Chapter 4

Terrain-Toolkit: A Multi-Functional Tool for Terrain Data

Abstract

Terrain data is becoming increasingly popular both in industry and in academia. Many tools have been developed for visualizing terrain data. However, we find that (1) they usually accept very few data formats of terrain data only; (2) they do not support terrain simplification well which, as will be shown, is used heavily for query processing in spatial databases; and (3) they do not provide the surface distance operator which is fundamental for many applications based on terrain data. Motivated by this, we developed a tool called Terrain-Toolkit for terrain data which accepts a comprehensive set of data formats, supports terrain simplification and provides the surface distance operator.

4.1 Introduction

Terrain which usually refers to the land surface (simply surface\(^1\)) is increasingly popular in industry. For example, Google (specifically, Google Maps) and Microsoft (specifically, Bing Maps for Enterprise) maintain a huge database of terrain data and use them for better user experience of exploring the maps. Terrain data also attracts extensive attention in academia. For example, researchers have devoted considerable efforts to kNN queries, range queries and shortest path queries (called shortest surface path queries in the context of terrain data) based on terrain data [13, 44, 56, 68, 69].

The most intuitive way for using and analyzing terrain data is to visualize it. In fact, many tools have been developed for visualizing terrain data. Some examples include MeshMan, CityGML, Terraserver, Google Earth, and Spaceye3D (A comprehensive survey could be found at http://vterrain.org/). Different tools have their

\(^1\) In the following, we use “terrain” and “surface” interchangeably.
different major purposes. For example, Google Earth puts its focus usually on viewing the terrain data freely in many ways (multi-layers, rotation, zoom-in and zoom-out) while some others put their focuses on the visual effect, rendering process and so on.

However, we observe that these existing visualization tools have several insufficiencies, especially from the perspective of computing with terrain data. We show them one by one as follows.

First, one existing visualization tool alone usually does not cover a rich enough set of models (and file formats) for the terrain data. These file formats can be mainly classified as Raster (e.g. USGSDEM, GTiff and SDTS) and Non-Raster (e.g. XYZ, OFF, PLY) formats. Raster formats require points with elevation values to be positioned on a uniform-grid, while non-raster formats are free to position their points anywhere. We classify collections of 2D/3D polygons or triangles (i.e. mesh models) as non-raster files too. To the best of our knowledge, there does not exist a single tool that supports all the previously mentioned data formats and allows for conversions between the formats.

Second, to the best of our knowledge, no existing visualization tool supports surface simplification [38] which refers to the process of simplifying a given surface to a simpler one. Surface simplification is used as a core component in many computations based on terrain data [38, 44], and the reason behind this is that computations based on terrain data are usually rather expensive (for example, the fastest algorithm for computing the shortest surface path between two points has its time complexity of $O(n^2)$ where $n$ is the number of vertices involved in the model representing the surface) and the surface simplification process helps to reduce the complexity of the surface considerably such that the computations based on the simpler surface become acceptably efficient.

Third, as far as we know, no existing visualization tool supports the surface distance operator which computes the length of the shortest surface path between two given points. Note that surface distance is usually not equal to the Euclidean distance, and in fact, Euclidean distance corresponds to a lower bound of the surface distance. Based on the surface distance, many queries have been defined on terrain data, e.g., $k$NN queries, range queries and shortest surface path queries [13, 44, 56, 68, 69].

**Contributions.** Motivated by the aforementioned insufficiencies of the existing visualization tools for terrain data, in this paper, we develop a new tool called Terrain-Toolkit which accepts a rich set of models and file formats for terrain data, provides the surface simplification functionality and supports the surface distance operator.

In the remainder of this paper, we first introduce our design of Terrain-Toolkit in Section 4.2. Afterwards, we describe the demonstration setup in Section 4.3 and conclude with some future directions in Section 4.4.
4.2  Design

In this part, we give some background knowledge on terrain in Section 4.2.1, present the architecture of Terrain-Toolkit in Section 4.2.2, and then introduce three major components of Terrain-Toolkit, namely model/format conversion, surface simplification and shortest surface path computation in Sections 4.2.3, 4.2.4 and 4.2.5 respectively.

4.2.1  Background Knowledge

A point in a three-dimensional space is represented by an \((x, y, z)\) coordinate where \(z\) represents the elevation of the point \((x, y)\). A Digital Elevation Model (DEM) is a 3D geometric representation of the terrain. A DEM can be denoted by either a raster grid or a Triangular Irregular Network (TIN). In a raster grid, for each point, its \(x\)-coordinate and its \(y\)-coordinate are restricted to a grid on a horizontal plane, but its \(z\)-coordinate is not restricted and corresponds to its elevation. A TIN is a vector-based representation of the terrain as a collection of non-overlapping triangles (also referred to as faces). Each triangle/face is comprised of three corner points called vertices that are connected to each other by line segments called edges. Two faces are adjacent to each other if they share a common edge. Figure 4.1 illustrates a simple TIN model with a few faces, where face \(F\) consists of vertices \(a, b\) and \(c\). For larger and more realistic TIN models, refer to Figure 4.3.

TIN representations are usually triangulated from raster grids. Delaunay triangulation [9] is one of the most popular triangulation methods used. In comparison to a raster grid, a TIN model consumes more storage space but provides a superior ability to describe the underlying surface at different levels of resolution.

The vertices and edges in a TIN already form a graph \(G\), where each vertex is associated with a 3D coordinate and the weight on an edge is the 3D Euclidean distance between the two endpoint vertices of the edge.

For an arbitrary pair of vertices \(s\) and \(t\) on a TIN, a surface path is a path that traces along the surface of the TIN, and a network path is a sequence of edges in the graph representation \(G\) of the TIN. Figure 4.1 shows the two paths. Note that a surface path can cut-across any face in the TIN, and a network path must be confined...
4.2 Architecture of Terrain-Toolkit

The architecture of Terrain-Toolkit is shown in Figure 4.2. Terrain-Toolkit accepts terrain data, stored either in a mesh file or in a point file, as input. A mesh file stores terrain data by using a terrain model (such as TIN) and a point file stores a set of raw points which are usually sampled from the surface of terrain. With the Model/Format Conversion utility, terrain data stored in a file format can be converted to terrain data stored in another file format.

The input of terrain data stored in a mesh file can be directly processed via the Preprocessing utility to construct an Internal Terrain Model which maintains the terrain data in our own way for better manipulation (e.g., visualization, simplification, and shortest surface path computation) while the input of terrain data stored in a point file has to be triangulated first via the Triangulation utility and then could be processed via the Preprocessing utility to construct an Internal Terrain Model.
4.2. DESIGN

The Internal Terrain Model based on terrain data consists of two parts. The first part is a triangulation model (TIN) which maintains the vertices, edges and faces of the terrain and the second part is a graph which is another structure for capturing the vertices and edges of the terrain and is used for the purpose of computing the shortest surface path.

The Internal Terrain Model can be visualized via the Terrain Visualization utility and simplified via the Surface Simplification utility (we used OpenGL for our visualization purpose), whose results can either be visualized or used for further computation. Given two points on the surface of a terrain represented by either the Internal Terrain Model or the result of the Surface Simplification utility which also corresponds to an Internal Terrain Model, the shortest surface path can be computed via the Shortest Surface Path Computation utility.

Next, we explain three key utilities of Terrain-Toolkit in detail.

4.2.3 Model/Format Conversion

The surface file formats can broadly be classified as Raster and Non-Raster formats. In the context of 3D surfaces, a raster consists of points laid out in a uniform grid-like fashion and each point has a height (z-value) associated with it. In contrast, in a non-raster format, the position of each point is not constrained to be in a uniform grid. Next, we briefly describe each file format.

- **USGSDEM**: is a raster format with elevation values per grid cell. We adopt the widely-used United States Geological Survey (USGS) DEM format in this paper.
- **GeoTiff (GTiff)**: uses a TIFF file format that stores raster graphics and embeds geo-spatial information as metadata.
- **Spatial Data Transfer Standard (SDTS)**: is a raster format and is primarily intended to be used for distribution and archiving of spatial data in the form of rasters that are generated in adherence to an open standard.
- **XYZ**: is a non-raster format and is a list of raw \((x, y, z)\) co-ordinates generated from measurements of the actual land surface.
- **Geomview Object File Format (OFF)**: is a non-raster format and stores a description of 2D/3D surfaces comprising of polygons.
- **Polygon File Format (PLY)**: is a non-raster format and stores 2D/3D surfaces and is a widely used and well-known file format that is similar to the OFF file format.

Terrain-Toolkit allows convenient conversions among various file formats.

4.2.4 Surface Simplification

Surface simplification is the process of reducing the number of faces used in the surface while trying to keep the overall topology preserved as much as possible. Reducing the number of faces in a surface model can greatly speedup the visualization...
of the model and also computations based on the model, e.g., shortest surface path computations. Many surface simplification techniques have been proposed in the research literature (for a detailed survey, please read [29]). We adopt the surface simplification method proposed in [38] since it guarantees that the shortest surface distance based on the simplified surface is within a bounded distance from the shortest surface distance based on the original surface.

The method [38] is based on an iterative approach of removing vertices from $G$. In each iteration, a chosen vertex $\bar{v}$ together with all edges connected to it are removed. After the removal, the set of all remaining vertices adjacent to this $\bar{v}$ form a polygonal hollow. This polygonal hollow is re-triangulated to complete the original triangulation [9]. On completion of re-triangulation, two properties are tested against to ensure that the removal of vertex $\bar{v}$ does not break our distance guarantee on the shortest network distance. The properties are as following.

- **Intra-distance property**: The estimated network distance between any two adjacent vertices in the neighborhood of $\bar{v}$ should satisfy our distance guarantee.
- **Inter-distance property**: The estimated network distance between any adjacent vertex of $\bar{v}$ and a previously removed neighbor of $\bar{v}$ should satisfy our distance guarantee.

If removing a vertex and re-triangulating its polygonal neighborhood satisfies both the intra-distance property and the inter-distance property, then the vertex is removed and the simplified surface is ready for the next iteration of vertex removal. Otherwise, vertex $\bar{v}$ along with all edges originally connected to it are reinstated. For more detailed information about this surface simplification algorithm, please refer to [38].

Terrain-Toolkit allows a user to input a real-valued error parameter, whose value has the range $[1, 2]$. The toolkit provides the user with statistics about the number of vertices, faces and edges in the surface prior to and after simplification. In addition, the toolkit also provides the user a side-by-side visual comparison of the surface before the simplification process and after. Figures 4.3(a) and 4.3(b) show the before and after views, respectively.

### 4.2.5 Shortest Surface Path Computation

The computation of the shortest surface path between two points $s$ and $t$ on 3D surfaces is a fundamental operation in applications such as robotics, motion planning, geographic information systems (GIS), and is an important topic in fields like computational geometry and computer graphics.

The state-of-the-art exact shortest surface path computation algorithm was proposed by Chen and Han (CH) [4]. This algorithm has a $O(n^2)$ time complexity, where $n$ is the number of vertices in the polyhedral surface. The CH algorithm uses a technique called planar unfolding. This time-consuming method rotates the faces of the surface, so that all faces end up on the same level plane. Similar to the computation of visibility graphs, the CH algorithm computes shadows/projections of the
source vertex $s$ on each edge in the triangulation and stores the results in a sequence tree that encodes information about the shortest paths from the source $s$ to points on the edges. More recently, an improved CH algorithm was proposed by Xin et al. [67] which has the same theoretical time complexity but outperforms the original CH algorithm by orders of magnitude because it prunes a majority of unnecessary nodes from the sequence tree. We adopt this improved CH algorithm in our system.

Terrain-Toolkit allows a user to pick vertices $s$ and $t$ from the displayed surface model by clicking on the vertices. On completion of the shortest surface path computation, the path is displayed to the user on the visualized surface model along with additional statistics like the total path length and the number of vertices along the path. The user also has the ability to click on faces that the path cuts across to view additional information about the faces (e.g., face IDs and vertices on the faces). Figures 4.4(a) and (b) show the shortest surface paths computed on the simplified surface, while Figures 4.4(c) and (d) show the shortest surface path computed on the original surface.

4.3 Demonstration

In this section, we demonstrate our Terrain-Toolkit. In subsection 4.3.1 we briefly describe our user-interface (UI), followed by a use-case that highlights the main features of Terrain-Toolkit in Section 4.3.2.

4.3.1 UI Demonstration

Terrain-Toolkit provides an easy-to-use and intuitive user interface by allowing the user to interact with the system at various stages. For example, the user is allowed to better visualize a 3D surface by rotating, scaling and translating the surface. When doing Surface Simplification, the user is also allowed to fine-tune the error parameter ($\beta$) to control the desired resolution of the simplified surface. In addition, to further improve the user experience, we provide the user with clear instructions about how to use our system. Besides, meaningful statistics are displayed after completion of procedures such as Surface Simplification and Shortest Surface Path Computation. We recorded a video to demonstrate the UI of Terrain-Toolkit which can be found at http://www.cse.ust.hk/~raywong/paper/Terrain-Toolkit.mp4.

In the following, we illustrate a use-case of Terrain-Toolkit.

4.3.2 Use Case: Emergency Response

The use-case outlined in this part highlights the flexibility and ease-of-use of our prototype in a real-life emergency response situation where computing the shortest surface path is critical. Consider the following scenario.

The Search and Rescue team in Montana receives a distress call from a hiker scaling the Bearhead mountain (BH) (located in the “Glacier National Park, Montana,
Figure 4.3: Effect of Surface Simplification and Shortest Surface Path Computation
Sylvester, the team-leader of this rescue mission, instantly opens up the 3D surface model of BH in Terrain-Toolkit with the intent of finding the shortest and fastest path to reach the distressed hiker.

However, Sylvester realizes that the model has a very high resolution (i.e., with a large number of faces) and path computation is slow. Hence, he decides to load a previously simplified surface with much fewer faces to speed up this computation.

In the past, he had simplified the BH surface by loading the surface model, choosing an error parameter ($\beta$) in the pop-up dialog box and then clicking the Simplify Terrain button (with the black-blue spinning wheel icon). Since he chose larger values of $\beta$, he got simplified models with fewer faces. He was able to compare the before and after simplification terrains visually and then he chose to either revert or save the simplified terrain based on his satisfaction. Figures 4.3(a) and Figure 4.4(b) show this comparative view.

Equipped with this knowledge, he browses his file system for a simplified surface with the desired number of faces, loads it and marks the last known location of the hiker on the terrain by double-clicking and choosing one of the vertices on the terrain. He then visually locates the closest peak to this marked point, where the rescue helicopter can land safely, from where the rescue team and medics must navigate the terrain on foot to get to the hiker as soon as possible.
Sylvester marks the nearest peak by picking another vertex and clicks the \textit{Find Shortest Surface Path} button (with the walking man icon). He receives an immediate response within 1-2 seconds, with the shortest path clearly highlighted on the surface visualization. Furthermore, he is also given the total distance of the path and he can click on the faces that the path cuts across to extract further information. Figures 4.3(c) and 4.3(d) show the shortest surface path on the simplified surface, viewed from two different angles.

Sylvester immediately dispatches his helicopter rescue team. The rescue team on the helicopter decides to utilize the journey time to load a higher resolution model of BH and recompute the shortest path from their landing location to the hiker’s last known location. This computation takes longer (nearly 30-40 seconds), but provides them with a much more accurate path and distance to locate, stabilize and rescue the hiker in distress. Figures 4.3(e) and 4.3(f) show the shortest surface path on the original surface, viewed from two different angles.

\section{4.4 Conclusion}

Motivated by the fact that there are very few toolkits available for terrain/surface manipulation, we developed Terrain-Toolkit. We demonstrate the techniques used in Terrain-Toolkit to efficiently simplify surfaces \cite{38} and compute shortest surface paths \cite{67} on them. There are several interesting features that can be added to our demonstration proposal. First, the user can choose a rectangular region on a large terrain for simplification and also shortest path computation. Second, the user can choose multiple source points to reach a destination on the terrain. Third, various other simplification algorithms can be implemented and visualized using Terrain-Toolkit.
Chapter 5

Building Accurate 3D Spatial Networks for Intelligent Transportation Systems

Abstract

The use of accurate 3D spatial network models can enable substantial improvements in vehicle routing. Notably, such models enable eco-routing, which reduces the environmental impact of transportation. We propose a novel filtering and lifting framework that augments a standard 2D spatial network model with elevation information extracted from massive aerial laser scan data and thus yields an accurate 3D model. We present a filtering technique that is capable of pruning irrelevant laser scan points in a single pass, but assumes that the 2D network fits in internal memory and that the points are appropriately sorted. We also provide an external-memory filtering technique that makes no such assumptions. During lifting, a triangulated irregular network (TIN) surface is constructed from the remaining points. The 2D network is projected onto the TIN, and a 3D network is constructed by means of interpolation. We report on a large-scale empirical study that offers insight into the accuracy, efficiency, and scalability properties of the framework.

5.1 Introduction

While today’s vehicle routing services rely on 2D spatial networks, future generations of such systems and Advanced Driver Assistance Systems (ADAS) require 3D models that accurately capture elevation and slope. Different applications pose different accuracy and resolution requirements to such 3D models.

Applications that target fuel savings and reduced greenhouse gas emissions, benefit from the availability of an accurate 3D map. A transportation study finds that eco-routing that uses a 3D spatial network model can yield fuel cost savings of 8–12%, when compared to standard routing based on a 2D model [64]. Another study reports that the use of a 3D model built from aerial laser scan data for vehicle routing
CHAPTER 5. BUILDING ACCURATE 3D SPATIAL NETWORKS

will yield annual fuel savings of approximately USD 6 billion in USA [60]. This figure stems from TomTom, a worldwide leading manufacturer of navigation systems. A study of models of vehicular environmental impact shows how increased accuracy of road slopes yields more accurate estimations of fuel consumption and greenhouse gas (GHG) emissions from vehicles [27].

ADAS applications provide critical information to the driver about the vehicle’s surroundings. Here, a 3D map can enhance information obtained from vehicle sensors and can also serve as a failsafe mechanism when sensors fail [18]. For example, adaptive headlights take into account the slopes of the road ahead and intelligently steer the headlights in order to offer maximum night-time visibility. Under adverse weather conditions such as excessive fog or rain, the sensors that are used under normal conditions can fail to operate optimally, and a 3D map can be used instead. ADAS specifications require a 3D road model with an accuracy of at least ±2 meters.

Three well-known and accepted methods of 3D map generation exist. First, a vehicle fitted with differential GPS and an Inertial Navigation System (INS) is driven on roads to capture their 3D road geometries [53]. Although this approach is well tested, it is expensive and cumbersome because it entails driving all existing roads to form a comprehensive 3D spatial network. Second, some major map providers use digital elevation models (DEM) generated from aerial images collected by interferometric synthetic aperture radar (IfSAR) to generate 3D maps [72, 73]. The accuracy and generation time of such models are unknown. Third, crowd-sourced data from Personal Navigation Devices (PND)s and vehicular GPS traces are used for extraction of elevation information in order to generate 3D spatial networks. The accuracy of such a model is also unknown and has not been compared to models created using the previous two methods. Also, the method relies on the availability of data that covers an entire transportation network, which is problematic.

Motivated by these observations, we propose two methods that use aerial laser scan data (LiDAR), illustrated in Figure 5.1, to augment 2D maps with elevation and slope information. External Memory Filtering (EMF) is an external memory algorithm that produces a 3D model of very high accuracy and resolution, and One Pass Filtering (OPF) is a streaming in-memory based solution that sacrifices some of the accuracy to generate a lower-resolution model, while reducing the map processing load.

In our experiments on a spatial network that spans North Jutland, Denmark, covering a region of 185km × 130km, we find that EMF can generate a 3D model with an accuracy of ±20 cm in 21 hours with a memory use of just 230 MB, while OPF can generate the same model with an accuracy of ±2 meters in approximately 2 hours, which is an order of magnitude faster than EMF, using 2.4 GB of main memory.

The two methods can be combined such that a very large 3D spatial network with relatively low accuracy can be generated quickly using OPF, upon which EMF can be used to zoom in on regions of interest, generating high-accuracy maps for these.

A key challenge is to achieve a suitable balance between the accuracy of a model, as dictated by the intended applications, and the storage space required by the model. This paper addresses that challenge by proposing techniques that are capable of us-
5.1. INTRODUCTION

To the best of the authors’ knowledge, this is the first work that investigates the application of aerial laser scan data to the lifting of a 2D spatial network in a fast and efficient manner, to obtain a 3D spatial network. Specifically, our contributions are four-fold.

- We propose a novel filtering and lifting framework that uses an aerial laser scan point cloud for lifting a spatial network.

- Two alternative filtering techniques, a one pass filter and an external memory based filter, are proposed for obtaining the particular points from an aerial laser scan data set that are needed for the lifting.

- Techniques for spatial lifting, consisting of triangulation and interpolation, are proposed to augment the 2D spatial network with elevation information using the remaining laser points.

- We present a comprehensive and large-scale empirical study that offers insight into the accuracy, efficiency, and scalability properties of the framework.

The remainder of this paper is organized as follows. In Section 5.2, we survey related work. Section 5.3 presents a formal problem definition, including the proposed filtering and lifting framework. Section 5.4 details the filtering and lifting phases. Section 5.5 covers the empirical analyses of the proposed techniques. Finally, Section 5.6 concludes the paper.
5.2 Related Work

Currently there are three main methods for constructing 3D road network models. First, some map vendors use vehicles instrumented with differential GPS and INS to capture 3D road geometries \[53\]. Using this method, NAVTEQ and TeleAtlas have spent several years on covering the major roads in nearly 50 major cities across U.S.A, Asia and Western Europe, but not secondary and tertiary roads \[18\]. Second, the remote sensing community has explored the creation of 3D models using elevation data from IfSAR and aerial images \[72, 73\]. Third, Waze\[1\] and TomTom’s MapShare\[2\] allow users to upload GPS locations with altitude information whose accuracy is \(\pm 3\) meters \[61\].

This paper adopts a different approach and, to the best of the authors’ knowledge, is the first to provide a method for obtaining a 3D spatial network by lifting a 2D spatial network using aerial laser scan data in a fast, efficient, and accurate fashion.

Tavares et al. \[64\] study eco-routing based on a 3D network and find that eco-routes are 1.8\% longer than the corresponding shortest routes and that fuel cost savings are in the range 8–12\%. Their 3D spatial network is developed based on contour lines, which are of much lower resolution than is the laser scan data we use. The EcoMark \[27\] framework for the evaluation of models of vehicular environmental impact illustrates how the use of an accurate 3D model yields better estimations of fuel and greenhouse gas emissions from transportation.

Several methods have been proposed that use statistical or machine learning techniques to classify aerial laser scan points into different categories \[1, 45\]. We only consider a particular category of points, namely ground points, which capture the actual land surface, and we exclude other points representing, e.g., vegetation, water, buildings, and noise. In the remainder of the paper, when mentioning laser scan points, we refer only to ground points.

5.3 Preliminaries

We introduce definitions that underly the proposed problem, formalize the spatial network lifting problem, and provide an overview of the filtering and lifting framework. An overview of the notation used in the paper is provided in Table \[5.1\].

5.3.1 Data Modeling

A spatial network captures both the topology and the embedding into geographical space of a transportation network. We define 2D and 3D spatial networks next.

**Definition 1** A 2D spatial network is modeled as an undirected graph \(G_{2D} = (V, E, F_{2D}, H_{2D})\), where \(V\) and \(E\) is the vertex set and edge set, respectively, and \(F_{2D}\)

\[1\]http://www.waze.com
\[2\]http://www.tomtom.com/en_gb/maps/map-share/
Table 5.1: Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_{2D}$</td>
<td>A 2D spatial network.</td>
</tr>
<tr>
<td>$G_{3D}$</td>
<td>A 3D spatial network.</td>
</tr>
<tr>
<td>$P_c$</td>
<td>A 3D laser scan point cloud.</td>
</tr>
<tr>
<td>$p_i$</td>
<td>A 3D laser scan point in $P_c$.</td>
</tr>
<tr>
<td>$\text{prj}^\top(p_i)$</td>
<td>The projection of a 3D point $p_i$ onto the 2D plane.</td>
</tr>
<tr>
<td>$\text{prj}^\bot(g, \triangle)$</td>
<td>The projection of a 2D model element $g$ onto a TIN $\triangle$.</td>
</tr>
<tr>
<td>$\epsilon N(g)$</td>
<td>The $\epsilon$-neighborhood of a 2D model element $g$.</td>
</tr>
</tbody>
</table>

and $H_{2D}$ are the functions recording the embedding of vertices and edges into the 2D plane, respectively.

A vertex $v_i \in V$ indicates either a road intersection or the end of a road. An edge $e_k \in E \subset 2^V$ is defined as a set of two vertices, and represents a road segment connecting the two vertices. For example, edge $e_k = \{v_i, v_j\}$ represents a road segment that connects vertex $v_i$ and vertex $v_j$. Function $F_{2D} : V \rightarrow \mathbb{R}^2$ takes as input a vertex and returns its coordinates in the 2D plane. Function $H_{2D} : E \rightarrow \mathbb{R}^2 \times \ldots \times \mathbb{R}^2$ takes as input an edge $e$, and outputs a 2D polyline represented by a sequence of 2D points. For example, edge $H_{2D}(\{v_i, v_j\}) = (v_i, a_1, a_2, a_3, a_4, v_j)$, as shown in Figure 5.2.

**Definition 2** A **3D spatial network** is also modeled as an undirected graph $G_{3D} = (V', E', F_{3D}, H_{3D})$. The definitions of $V'$ and $E'$ in a 3D spatial network are identical to the counterparts in a 2D spatial network. Note that for higher resolutions, our method introduces more vertices and edges into $V'$ and $E'$ respectively. Functions $F_{3D}$ and $H_{3D}$ record the geometric information of vertices and edges in 3D space. Function $F_{3D} : V' \rightarrow \mathbb{R}^3$ takes as input a vertex and returns its coordinates in 3D space; and function $H_{3D} : E' \rightarrow \mathbb{R}^3 \times \ldots \times \mathbb{R}^3$ takes as input an edge and outputs a 3D polyline, represented as a sequence of 3D points. Figure 5.3 shows the 3D polyline of a road segment. Calculating slopes of each edge $E'$ in $G_{3D}$ becomes a trivial exercise.

**Definition 3** A **Laser Scan Point Cloud** ($P_c$) is a set of laser points $p_i$, which is formalized as

$$P_c = \{ p_i = (x_i, y_i, z_i) \in \mathbb{R}^3 \mid 1 \leq i \leq N \},$$

where $N$ is the total number of laser points in the point cloud.
CHAPTER 5. BUILDING ACCURATE 3D SPATIAL NETWORKS

Figure 5.2: A 2D Road Segment With its $\epsilon$-Neighborhood

Figure 5.3: A 3D Road Segment
5.3. PRELIMINARIES

5.3.2 Problem Formulation

Spatial network lifting augments a 2D spatial network $G_{2D}$ with elevation information extracted from a laser scan point cloud $P_e$ and returns $G_{2D}$'s corresponding 3D representation $G_{3D}$ as the result.

Intuitively, to lift a 2D model element (e.g., either a 2D polyline $H_{2D}(e_k)$ indicating a road segment or a 2D point $F_{2D}(v_i)$ indicating a road intersection), the 3D laser points locating nearby the 2D model element are more useful than the laser points that are further away. We take into account the laser points belonging to the $\epsilon$-Neighborhoods of the 2D elements while lifting them.

**Definition 4** Given a point cloud $P_e$, the $\epsilon$-Neighborhood of a 2D model element $g$, denoted as $\epsilon N(g)$, is a set of laser points in $P_e$ that satisfy a given spatial predicate $\epsilon$.

$$\epsilon N(g) = \{ p_i \in P_e \mid \epsilon(p_i, g) \},$$

where $\epsilon(p_i, g)$ denotes a spatial predicate defined on a laser point $p_i$ and a 2D model element $g$.

Since a 2D point can be regarded as a special case of a 2D polyline, the following discussion focuses on the lifting of road segments (i.e., 2D polylines) instead of the lifting of road intersections (i.e., 2D points).

An example of an $\epsilon$-Neighborhood of a road segment is shown in Figure 5.2, where the solid polyline in the center is a road segment $\{v_i, v_j\}$. The projection of a laser point $p_i \in P_e$ onto the 2D plane is defined as $prj^T(p_i) = (x_i, y_i, 0)$. We let the spatial predicate $\epsilon$ be defined as $\text{dist}(prj^T(p_i), H_{2D}(e)) \leq d$, which means that if a laser point satisfies the predicate, the shortest distance between its 2D projection $prj^T(p_i)$ to the 2D polyline of the edge should not exceed $d$. The two dotted lines indicate the boundary of points that satisfy the predicate. Since the projected points $prj^T(p_1)$, $prj^T(p_2)$, $prj^T(p_3)$ and $prj^T(p_4)$ lie in the range bounded by the two dotted lines, $\epsilon N(H_{2D}(\{v_i, v_j\})) = \{p_1, p_2, p_4, p_5\}$.

The laser points in $\epsilon$-Neighborhood $\epsilon N(g)$ can be transformed into a Triangulated Irregular Network (TIN), denoted as $\Delta(g)$, to approximate the surface around the road segment $g$. Projecting the 2D model element $g$ onto its corresponding TIN $\Delta(g)$, its 3D polyline representation becomes available. For example, the 3D representation of road segment $\{v_i, v_j\}$ is shown in Figure 5.3. Assuming $prj^+(g, \Delta)$ indicates the projection of a 2D model element $g$ onto a TIN surface $\Delta$, spatial network lifting is formalized as follows.

**Definition 5** Spatial network lifting takes as input a 2D spatial network $G_{2D}$ and a laser scan point cloud $P_e$, and it returns the corresponding 3D spatial network $G_{3D}$ in which $F_{3D}(v) = prj^+(F_{2D}(v), \Delta(v))$ for every $v \in \mathbb{V}$, and $H_{3D}(e) = prj^+(H_{2D}(e), \Delta(e))$ for every $e \in \mathbb{E}$. 
5.3.3 Framework Overview

Figure 5.4 depicts the overview of spatial network lifting, which consists of two major phases: filtering and lifting.

The filtering phase takes as input a 2D spatial network and a massive laser scan point cloud, and prunes irrelevant laser points in the point cloud in order to obtain an appropriate \( \epsilon \)-Neighborhood of every road segment and road intersection in the 2D spatial network. Two alternative filtering methods, one pass filtering and external memory based filtering, are provided in order to exploit situations where the 2D networks fits in main memory and to also provide a general solution that does not make this assumption.

The lifting phase consists of two steps, where interpolation follows triangulation. In the triangulation step, laser points in \( \epsilon \)-Neighborhoods are transformed into TINs. After projecting the 2D spatial network onto the TINs, the interpolation step computes the corresponding elevation information, thus providing a 3D spatial network as the final output.
5.4 Spatial Network Lifting

Spatial network lifting focuses on generating an accurate and compact 3D spatial network in an efficient and scalable manner.

5.4.1 Filtering

Overview

The main task of the filtering phase is to obtain $\epsilon$-neighborhoods for the 2D model elements in $G_{2D}$. Instead of proposing yet another index to filter the laser points, we explore the opportunities of applying existing indexing techniques. A priori knowledge of the laser points being almost uniformly distributed with a guaranteed minimum resolution (e.g., at least one laser point per square meter), significantly influences our decision to choose a space-driven indexing technique, in particular, the grid index, instead of a data-driven indexing technique, e.g., a tree-based index.

As an aside, we chose not to use an approach where we first build spatial indexes on the point cloud $P_c$ and on the 2D spatial network (e.g., using R-trees) and then join the two by synchronized traversing the two indices [3, 31, 46]. This is because $P_c$ is massive in size (for Denmark, on the order of terabytes) and is collected rarely. Thus, the join operation is not carried out repeatedly, and the filtering is merely an intermediate step in solving the lifting problem.

By utilizing grid based indices, the filtering becomes parallelizable and can be processed easily on powerful machines with large main memories. However, we also consider the setting where the available main memory is limited, as this renders the paper’s proposal applicable to commodity hardware.

We provide two filtering approaches that differ primarily in how they manage the smaller data set, i.e., the 2D spatial network $G_{2D}$.

1. One pass filtering, described in Section 5.4.1, assumes that there is enough internal memory to accommodate a grid index on $G_{2D}$ and filters the point cloud $P_c$ in a single pass by checking whether a laser point belongs to the $\epsilon$-Neighborhoods of all road segments in $G_{2D}$;

2. External memory based filtering, described in Section 11, works without any assumption on main memory size, so neither $G_{2D}$ nor $P_c$ are assumed to fit in internal memory. Two different traversal strategies, row-major and z-curve order, are used for loading disk blocks into memory for filtering.

One Pass Filtering

One pass filtering (or OPF for simplicity) utilizes a uniform grid to index both the 2D spatial network and the laser points. Cells in the grid are squares, and the width of a cell is governed by a user specified parameter $\delta$ that needs to be given before generating the grid. Figure 5.5 shows an example of how grid cells map to both road points (points contained in 2D polylines) and laser points. For ease of illustration,
only a small portion of laser scan points mapped to cells are shown, while in reality, the cells have much more laser scan points due to the high density of the laser scan point cloud.

The spatial predicate used in OPP, denoted as \( \epsilon_{OPP}(g, p_i) \), returns true if a 2D model element \( g \) and the 2D projection of a laser point \( p_i \), i.e., \( prj^\top(p_i) \), fall into the same grid cell. Given the same 2D model element \( g \), the bigger the cell width \( \delta \) is, the more laser points are contained in its \( \epsilon \)-Neighborhood \( \epsilon N(g) \).

OPF assumes that the grid index on the 2D spatial network, i.e., the mapping \( H_g \) given below, can fit fully into internal memory.

\[
H_g : C \rightarrow \bigcup_{\epsilon \in \mathbb{R}} H_{2D}(\epsilon).
\]

For each cell \( c \) in grid cells set \( C \), mapping \( H_g \) maintains a set, denoted as \( H_g(c) \), containing all the 2D model elements that intersect with the cell.

Upon creation, \( H_g \) acts as a seed for generating the grid index on the point cloud \( P_c \). OPP sequentially scans \( P_c \) only once to generate another mapping \( H_p \) from seed cells (those cells having 2D model elements in \( H_g \)) to laser points whose 2D projections are within the cells. \( H_p \) is formally defined as follows.

\[
H_p : SC \rightarrow P_c; \quad \text{where } H_g(c) \neq \emptyset \text{ if } c \in SC \subseteq C.
\]

Note that \( H_p(c) \) records the \( \epsilon \)-Neighborhoods of the 2D model elements in \( H_g(c) \).
Isenburg and Lindstrom [32] observe that laser points are inherently topologically-coherent, which implies that the laser points are stored in an order that is an artifact of how they were collected by the planes flying over the covered land surfaces. In other words, the laser points in the point cloud are not stored randomly. Rather, laser points that are geographically close are also stored close to each other and hence the point cloud is stored in a manner that to some extent is locality preserving. OPF exploits this property and thus avoids performing several passes to sort the point cloud \( P_c \).

Based on the above observation, the mapping \( H_p \) does not need to be maintained for every cell in memory at all times. Once \( H_p(c) \) contains a sufficient number of laser points for cell \( c \), artificial 2D road points are inserted at all intersections of the road edges and grid cell boundaries. Finally, the 2D model elements in \( H_g(c) \) are lifted. Recall that the laser point cloud we use guarantees one laser point per square meter, meaning that a cell with width \( \delta \) is expected to contain \( \delta \cdot \delta \) laser points. As OPF scans the point cloud, when \( H_p(c) \) contains more than \( \alpha \cdot \delta \cdot \delta \) laser points, \( H_p(c) \) can be passed to the lifting phase immediately. Here, the fill factor \( \alpha \in (0, 1) \) is a parameter that represents a trade off between efficiency and accuracy: the higher the fill factor is, the more laser points must be contained in the \( \epsilon \)-Neighborhoods, thus making the final 3D spatial network more accurate.

Algorithm 4 describes OPF. Function \( \text{GetIntersectedCells}(ls, \delta) \) (in line 3) returns the cells that intersect with line segment \( ls \) according to cell width \( \delta \), and function \( \text{GetContainedCells}(p_i, \delta) \) (in line 6) returns the cell that contains the 2D projection of laser point \( p_i \) according to cell width \( \delta \).

The advantages of one pass filtering are: (i) OPF does not incur any pre-processing cost of sorting or indexing the massive point cloud \( P_c \); and it only scans the point cloud once; (ii) OPF is able to output parts of the resulting 3D spatial network with different accuracy requirements (by configuring \( \alpha \)) as the laser points stream in. (iii) OPF can easily be parallelized to take advantage of either new hardware architectures like GPUs or cloud infrastructures like MapReduce.

**External Memory Based Filtering**

In contrast to OPF, external memory based filtering (abbreviated as EMF) works even if neither \( G_{2D} \) nor \( P_c \) is able to fit in internal memory. The basic goal of EMF is to efficiently filter large data sets of arbitrary sizes given a limited and fixed main memory budget.

In order to achieve an accurate TIN, EMF employs a spatial predicate \( \epsilon_{EMF}(g, p_i) \) that returns true if the cell containing \( prj^T(p_i) \) is the cell that contains \( g \) or is one of the eight neighboring cells of the cell containing \( g \) (also called Moore neighbor cells of \( g \)). In the following discussion, we use 9-cell to indicate a cell and its Moore neighbor cells. For example, the 9-cell of cell \((1, 1)\) is shown in the bottom left of Figure 5.5.

EMF scans both \( G_{2D} \) and \( P_c \), organizing them into road blocks (lines 1–4 in Algorithm 5) and laser blocks (lines 5–7 in Algorithm 5), where each road (laser)
Algorithm 4: OnePassFilter

Input: 2D spatial network $G_{2D}$, grid cell width $\delta$, point cloud $P_c$, fill factor $\alpha$.

/* Initialize mapping $H_g$ */
1 \textbf{for} each edge $e \in G_{2D}.E$ \textbf{do}
2 \hspace{1em} \textbf{for} each line segment $ls \in H_{2D}(e)$ \textbf{do}
3 \hspace{2em} \textbf{for} each cell $c \in \text{GetIntersectedCells}(ls, \delta)$ \textbf{do}
4 \hspace{3em} $H_g(c) \leftarrow H_g(c) \cup ls$;

/* One pass scan on the point cloud $P_c$ */
5 \textbf{for} each laser point $p_i \in P_c$ \textbf{do}
6 \hspace{1em} Cell $c \leftarrow \text{GetContainedCell}(p_i, \delta)$
7 \hspace{2em} /* Use $H_g$ as a seed */
8 \hspace{3em} if $H_g(c) \neq \emptyset$ then
9 \hspace{4em} $H_p(c) \leftarrow H_p(c) \cup p_i$;
10 \hspace{4em} if $|H_p(c)| \geq \alpha \cdot \delta \cdot \delta$ then
11 \hspace{5em} /* Lift the 2D model elements in cell $c$ */
12 \hspace{6em} Lifting($H_g(c), H_p(c)$);
13 \hspace{5em} Release $H_g(c)$ and $H_p(c)$;

block contains the road segments that intersect with (laser points that are in) a cell. The size of a laser block is typically decided by the cell width $\delta$. Assuming each laser point takes 20 bytes (two doubles and one float), a laser block needs $20 \cdot \delta \cdot \delta$ bytes. (Table 5.3 in Section 5.5.1 details the block sizes.)

A road block typically takes up much less space than a laser block because it is uncommon to have very dense road segments (e.g., with a point for every one meter). We therefore assume that a road block has at most the same size as a laser block. After block reorganization, EMF reads road blocks into main memory according to a locality preserving space filling curve. After reading a new road block, EMF reads in its corresponding 9-cell laser blocks and overwrites laser blocks using the least recently used (LRU) [30] policy. A road block along with its 9-cell laser blocks are fed into the lifting phase, as described in lines 9–13 in Algorithm 5.

Given a limited memory budget, the order in which road blocks are read has a significant impact on the performance of EMF. In order to avoid frequent re-reading of the same laser blocks, locality preserving space filling curves are considered when EMF loads the road blocks. In particular, we consider two space-filling curves, namely the row-major curve and the z-order curve. Figure 5.6 shows the orders in which road blocks are loaded in memory starting from the bottom left cell, according to the two row-major and z-order curves, respectively.

When a road block is being processed, its corresponding 9 laser blocks must be available in memory for processing. As EMF reads road blocks and moves along a
Algorithm 5: ExternalMemoryFilter

Input: 2D spatial network $G_{2D}$, grid cell size $\delta$, point cloud $P_c$, Curve Tag $TAG$; Memory Budget $B$.

/* Scan and sort $G_{2D}$ into road blocks */
for each edge $e \in G_{2D}$ do
  for each line segment $ls \in H_{2D}(e)$ do
    for each cell $c \in \text{GetIntersectedCells}(ls, \delta)$ do
      writeBlock($c$, $ls$, $FILE_G$);
  /* Scan and sort $P_c$ into laser blocks */
  for each point $p_i \in P_c$ do
    Cell $c \leftarrow \text{GetContainedCell}(p_i, \delta)$;
    writeBlock($c$, $p_i$, $FILE_P$);
  Buffer $A \leftarrow \emptyset$;
while Decide next cell $c$ according to $TAG$ curve do
  $A \leftarrow \text{readBlock}(c, FILE_G)$;
  for each cell $c' \in \text{MooreNeighbour}(c) \cup c$ do
    $B \leftarrow \text{LRU}(B, \text{readBlock}(c', FILE_P))$;
    /* Lift the 2D model elements in cell $c$ */
    Lifting($A$, $B$);

curve, there are moments where memory is full and old laser blocks are overwritten with new ones. The dotted boxes in Figure 5.6 show the points where there is a high likelihood that new laser blocks must be read from disk.

Analysis: Given a grid with $n \cdot n$ cells and a fixed-size memory (a multiple of $n$) that employs the LRU policy [30], we investigate the effects of using Z-order and row-major curves. Two grids with sizes $16 \cdot 16$ and $64 \cdot 64$ are considered, and we vary the memory size in multiples of $n$ with $0.5 \cdot n$ being the finest granularity. We then report the number of laser block replacements in Figure 5.8. The results show that for a memory budget ranging from $0.5 \cdot n$ to $2.5 \cdot n$, the Z-order yields the fewest reads. The benefit of using the Z-order increases as the grid size $n$ increases. Starting at $3 \cdot n$ and onwards, the row-major ordering begins to outperform Z-order and performs stably. This is because the 9 laser blocks that must be considered for a road block belong to 3 grid rows.

To illustrate, consider again Figure 5.6. Assuming that the memory budget is $3 \cdot n$, when the first two rows of road blocks have been processed according to the row-major curve, the memory budget is fully occupied by the first three rows of laser blocks. In order to process the next road block, i.e., $(2, 0)$ (surrounded with a dashed box), laser blocks $(3, 0)$ and $(3, 1)$ need to be read, and laser blocks $(0, 0)$ and $(0, 1)$ are over-written. The over-written blocks will never again be needed and hence with
memory budget size $3 \cdot n$, no laser block needs to be read more than once. Therefore, with memory budget size no less than $3 \cdot n$, row-major behaves in a stable fashion, while with the Z-order some blocks that are overwritten may be needed at a later time. Thus, if the memory budget is less than $3 \cdot n$, Z-order is preferable, while row-major wins and performs stably if the memory budget is no less than $3 \cdot n$.

5.4.2 Lifting

Upon successful filtering, the laser points in the $\epsilon$-Neighborhoods are triangulated into a TIN. Then the elevation information in the TIN is assigned to the 2D road segments by projecting these onto the TINs and performing interpolation.

Triangulation

The elevation values in a given region are only available for the points where measurements were taken (e.g., the laser points in the region). To get the elevation for other points in the region, some form of approximation must be applied. A naive approach assigns an elevation to a point that is equal to the elevation of the point’s nearest neighbor in the laser point cloud or that is equal to the average elevation of its $k$ nearest laser points. However, these approaches are unable to produce accurate results.

We adopt a different tack: given a region, all the pertinent laser points, e.g., those in $\epsilon$-Neighborhoods, are triangulated into a TIN to approximate the surface of the region. The elevation of any point in the region can then be interpolated from the TIN.

Triangulation transforms a set of laser points, which represent discrete measurements on a surface, into a set of non-overlapping triangles where the vertices of the triangles are the laser points. We use Delaunay Triangulation \[7\] for triangulation.
This is a specialized triangulation method where, in the resulting triangles, no triangle vertex is inside the circumscribed circles of any other triangle.

Recall that OPF utilizes a fill factor $\alpha$ to act as a parameter when determining how many laser points must be present when moving to the triangulation step. The higher the fill factor is, the more accurate the TIN approximates the real surface, thus yielding a more accurate 3D spatial network. For example, with a low fill factor of $\alpha = 0.1$, some road segments (road points $a_1$, $b_1$, and $c_1$) are not covered by the resulting TIN (shown in Figure 5.7(a)). Increasing $\alpha$ to 0.4 improves the coverage (Figure 5.7(b)).
EMF passes all the laser points in the relevant 9-cell to the triangulation step, as shown in Figure 5.7(c), which typically guarantees that all the road segments in the center cell are fully covered by the resulting TIN, thus achieving higher accuracy.

Interpolation

Figure 5.9 shows a 2D polyline representing a road segment and its TIN. Intuitively, projecting the 2D polyline to the TIN, the road segment’s 3D representation becomes available. However, directly projecting a polyline to a TIN is computationally expensive. Instead, we sample a set of 2D road points on the 2D polyline and then project them onto the TIN in order to obtain their 3D counterparts. By connecting these 3D road points, the 3D polyline representation of the road segment is obtained, e.g., the 3D polyline \( (a'_1, \ldots, a'_5, \ldots, a'_{11}) \) shown in Figure 5.9.

**Exact sampling:** ES selects the points where either the direction or the grade of the road segment change. ES starts by projecting the TIN to the 2D plane by ignoring the \( z \) coordinates of all vertices. To illustrate, this yields the dashed triangles on the 2D plane shown in Figure 5.9. Next, points on the road polyline are sampled by picking the points in the original 2D polyline representation (i.e., the points where the direction of the road changes) and all the intersections of the edges of the 2D triangles and the original 2D polylines (i.e., the points where the grade of the road segment may change). The grade of a road segment may change only when the road segment crosses from one triangle to another.

Since the intersections obtained by ES are always on TIN triangle edges, the elevations at the vertices of the triangle edges are used in linear interpolation to compute the intersections’ elevations. Figure 5.9 shows an example of a triangle edge in bold; its two corresponding vertices are used for computing the elevation of road point \( a'_5 \).

The remaining points in ES may fall in a triangle, and the elevations of these...
5.5. EXPERIMENTS

points are interpolated by taking into account the elevations of all three vertices of
the triangle, i.e., using plane-based interpolation.

5.5 Experiments

We detail the data sets, parameter settings, and implementation. Then we cover the
empirical study of the the efficiency, accuracy, and scalability properties of the pro-
posed filtering and lifting framework. Comparisons to existing methods, covered in
Section 5.2 were not performed because the data necessary to do so was not available
to us.

5.5.1 Experimental Setup

Data Sets: Two spatial networks in Denmark are lifted in the experiments.

_Aalborg (AA)_ covers the Aalborg region (North Jutland, Denmark) which is of
size approximately \(7\text{km} \cdot 5\text{km}\). The laser point cloud of AA occupies 2.84 GB.
The spatial network of Aalborg, obtained from OpenStreetMap, has a total length of
approximately \(4 \cdot 10^7\text{m}\), and its 2D polyline representation is 1 MB in storage size,
consisting of 13,366 points.

_North Jutland (NJ)_ , the northern part of Jutland, Denmark, covers a region of
\(185\text{km} \cdot 130\text{km}\). The laser point cloud of NJ occupies 342 GB. NJ contains a spatial
network with a total length of \(1.17 \cdot 10^7\text{m}\), whose 2D polyline representation is 28
MB in storage size, containing 414,363 points.

Figure 5.9: ES With Intersection Road Points
We report on experiments were carried out on AA, unless stated explicitly otherwise.

**Parameter Settings:** The experiments are conducted by varying several parameters to study the effect of the trade-offs between accuracy, efficiency, and memory usage. Table 5.2 shows the parameters with default values shown in bold. The equidistant parameter \( ed \) used in approximate sampling is set to \( 10 \) m. Experiments are conducted using default parameter values unless explicitly stated otherwise.

**Table 5.2: Parameter Settings**

<table>
<thead>
<tr>
<th>Grid Cell Width ( \delta ) (m)</th>
<th>16, 64, 128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filling Factor ( \alpha )</td>
<td>0.4, 0.6, 1.0</td>
</tr>
<tr>
<td>Memory Size</td>
<td>( c, 2 \cdot c, 3 \cdot c, 4 \cdot c )</td>
</tr>
</tbody>
</table>

Table 5.3 shows details on the grid indices for different cell widths. In the table, \( r \cdot c \) indicate the size of the grid, where \( r \) and \( c \) denotes the number of rows and columns, respectively.

**Table 5.3: Detail of Grid Indices**

<table>
<thead>
<tr>
<th>( \delta ) (m)</th>
<th>AA ( r \cdot c )</th>
<th>( NJ ) ( r \cdot c )</th>
<th>BS (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>316 · 465</td>
<td>8082 · 11526</td>
<td>5</td>
</tr>
<tr>
<td>64</td>
<td>79 · 117</td>
<td>2021 · 2882</td>
<td>80</td>
</tr>
<tr>
<td>128</td>
<td>40 · 59</td>
<td>1011 · 1441</td>
<td>320</td>
</tr>
</tbody>
</table>

We compare variants of EMF that use row-major ordering and Z-curve ordering in terms of disk block reads by varying the available memory budget. The available memory is set as a multiple of \( c \) blocks (where \( c \) is the width of the grid). Note that as \( \delta \) varies, the block size, which relates to how many laser points or 2D polylines fall in a cell in the grid, also varies accordingly. The size of a (laser or road) block is shown in the last column of Table 5.3.

**Implementation:** The filtering and lifting framework is implemented in C and C++, and Perl is used to to perform auxiliary tasks. SCALGO Terrastream\(^3\) is used to generate the TIN used in our ground-truth method (in Section 5.5.2). Triangle\(^4\) is employed for Delaunay triangulation in the lifting phase. An R-tree library\(^5\) is applied to facilitate the interpolation step in the lifting phase. All the experiments are carried out on an Ubuntu 11.04 LINUX machine with an Intel Xeon W3565 @3.2 GHz CPU (8MB cache, hyper-threading, 4 cores), 8 GB internal memory and 16.5 TB hard disk.

---

\(^3\)http://madalgo.au.dk/Trac-TerraSTREAM

\(^4\)http://www.cs.cmu.edu/~quake/triangle.html

\(^5\)http://superliminal.com/sources/RTree.zip
5.5. EXPERIMENTS

5.5.2 Ground Truth Generation

The ground truth is generated by triangulating all the laser points in the NJ point cloud into a huge TIN and then interpolate the road points obtained by exact sampling using this TIN. The statistics of the huge TIN are listed in Table 5.4, where ▽ indicates triangles in a TIN.

<table>
<thead>
<tr>
<th>Data</th>
<th>Laser</th>
<th>TIN</th>
<th>TIN-to-▽</th>
<th>Number of ▽</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>2.84 GB</td>
<td>3.1 GB</td>
<td>6.3 GB</td>
<td>3.7 · 10^7</td>
</tr>
<tr>
<td>NJ</td>
<td>342 GB</td>
<td>372 GB</td>
<td>742 GB</td>
<td>9 · 10^9</td>
</tr>
</tbody>
</table>

We use sum of squared errors (SSE), which is defined in Equation 5.1, for quantifying the differences between the 3D spatial networks generated by the filtering and lifting framework and the ground truth.

\[
SSE = \sum_{i=1}^{|R_p|} (z_i - g_t_i)^2
\]

In the equation, \(|R_p|\) is the total number of road points based on a sampling strategy (as described in Section 5.4.2); \(z_i\) is the elevation reported by the proposed method, and \(g_t_i\) is the elevation obtained from the ground truth. The absolute error, \(A_i = |z_i - g_t_i|\), is also introduced as another accuracy measure in the subsequent discussion.

5.5.3 Accuracy Studies

We analyze the accuracy of the proposed approaches against the ground truth.

**Accuracy Analysis of OPF:** Figure 5.10(a) illustrates the effect of varying the fill factor (\(\alpha\)), while fixing the grid size to its default size (64m). As we increase \(\alpha\), more laser points are passed into Delaunay triangulation, which yields more accurate elevation values.
Setting $\alpha$ to its default value and increasing the grid cell width, as shown in Figure 5.10(b), we notice that the SSE increases.

Although $\alpha$ acts as a threshold value for deciding how many laser points should be processed in triangulation, it is not able to control which laser points are chosen for triangulation. Since a grid cell with larger width should contain more laser points, the $\alpha$ fraction of laser points are more likely to be non-uniformly distributed in the cell, causing a deterioration in the triangulation and hence in the computed elevation values.

Accuracy Analysis of EMF: Unlike OPF, EMF employs triangulation on laser points in 9-cells, which results in a much better triangulation quality. As we increase the grid cell width, the SSE drops and the elevation values computed become much more accurate—see Figure 5.11(a). This occurs because the 9-cells contain increasingly more laser points, which results in higher quality triangulations.

Figures 5.10(c) and 5.11(b) for OPF and EMF respectively, display the numbers of absolute error $A_i$ within different error ranges, thus showing the distribution of the errors. The leftmost bar in each figure indicates the road points whose elevation are computed with no error. Note that the points in the subsequent buckets drop significantly in both methods, especially in EMF.

5.5.4 Storage Studies

We quantify the storage requirements as the maximum amount of main memory required at any moment in the case of OPF; and as the number of laser and road block reads from disk in the case of EMF.

Memory Usage of OPF: Figure 5.12(a) shows that the maximum memory required grows as $\alpha$ increases. The larger $\alpha$ gets, the longer the wait is until there are enough laser points to proceed to the lifting phase. Hence, there is a greater demand on mem-
ory. Likewise, increasing the grid cell width yields a greater memory requirement to hold laser points as the area of the cell increases (shown in Figure 5.12(b)).

![Figure 5.12: OPF Memory Measurements](image)

**Memory Usage of EMF:** Figure 5.13 indicates that the disk block read performance of the Z-curve and row-major orders adhere to our earlier analytical results (cf. Figure 5.8). The biggest memory budget in EMF, with size of $4 \cdot c$ and the largest grid cell width of $128 m$, does not exceed 80 MB of main memory. Additionally, given the maximum $\alpha$ and grid cell size, OPF has a memory upper bound of approximately 600 MB. Hence, both EMF and OPF can function with very limited memory for the AA spatial network.

![Figure 5.13: EMF Block Reads](image)  
![Figure 5.14: EMF Runtime](image)


5.5.5 Efficiency Analysis

We analyze the performance of our proposed methods in terms of runtime. The goal is to gain insight into the practical feasibility of OPF and EMF.

For OPF, Figures 5.15(a) and 5.15(b) exhibit the effects of varying grid cell width and fill factor, where both end up having to pass increasing numbers of laser points to the lifting phase where triangulation and interpolation take longer.

![Figure 5.15: One-Pass Runtime Measurements](image)

For EMF, Figure 5.14 shows the effect on overall runtime when the available memory is varied. A very large proportion of time is spent on disk reads, which are much more time consuming than in-memory operations. We see a strong correlation between the disk reads and the overall runtime of EMF, which is also highlighted by the similarity to the I/O measurements shown in Figure 5.13.

Although results show that OPF, which employs no pre-processing at all, is nearly an order of magnitude faster than EMF, this superior run-time performance comes at the cost of an accuracy degradation (as shown in Section 5.5.3). Note that both OPF and EMF take much less time than the ground truth computation.

5.5.6 Scalability Studies

We conduct experiments on both AA and NJ to observe the scalability of the proposed methods. Recall that the point cloud of NJ is almost 120 times larger than the point cloud of AA, as shown in Table 5.4. We apply OPF on both data sets with default parameters. The corresponding runtime and maximum memory requirement suggest good scalability, as shown in Table 5.5. For EMF, we use Z-order and a memory budget with $c$ blocks (i.e., around 230 MB); all the other parameters are set to default values. The results in Table 5.5 also suggest that the runtime and number of block reads are proportional to the numbers of laser points in both data sets, which in turn suggests that EMF is scalability in the point cloud size. EMF outputs a 3D spatial
5.6 Conclusion and Outlook

We study a spatial network lifting problem that augments a standard 2D spatial network with elevation values extracted from a laser scan point cloud. We propose a novel filtering and lifting framework that aims to produce accurate 3D spatial network models that occupy limited storage in an efficient and scalable manner using commodity hardware. The results of extensive empirical studies offer insight into the design properties of the framework and suggest that the framework is practical and is indeed capable of meeting the design goals.

Our future work aims at exploring higher accuracy estimations of eco-routes using our 3D spatial network.

Acknowledgements

The authors wish to thank the companies COWI and SCALGO for allowing us to use the aerial laser scan data and for their help with the pre-processing of this data. The work was supported by the Reduction project that is funded by the European Commission as FP7-ICT-2011-7 STREP project number 288254.

Table 5.5: Scalability Analysis

<table>
<thead>
<tr>
<th>Filtering</th>
<th>Dataset</th>
<th>Runtime (mins)</th>
<th>Max Mem (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPF (Default δ, α)</td>
<td>AA</td>
<td>2.9</td>
<td>327</td>
</tr>
<tr>
<td></td>
<td>NJ</td>
<td>122.3</td>
<td>2,457.6</td>
</tr>
<tr>
<td>Filtering</td>
<td>Dataset</td>
<td>Runtime (mins)</td>
<td>Blocks Read</td>
</tr>
<tr>
<td>EMF (Z-Order, c)</td>
<td>AA</td>
<td>24.2</td>
<td>12,092</td>
</tr>
<tr>
<td></td>
<td>NJ</td>
<td>1270.8</td>
<td>126,895</td>
</tr>
</tbody>
</table>

network that is an order of magnitude larger for NJ spatial network, with a total of 4,640,865 3D points.
Chapter 6

Scalable Real-time Continuous Fastest Route Planning

Abstract

Fastest path queries have become popular due to the proliferation of mobile and location-aware communication devices such as smartphones and navigation devices. A traditional fastest path query finds the fastest path from a source vertex to a destination vertex based on historical information.

Motivated by this, we propose two new kinds of fastest paths, namely the fastest dynamic historical path and the fastest dynamic real-time path. Both paths are computed based on the current arrival time of the vehicle to each vertex in their respective paths. The difference is that the estimated travel time of the first path is based on historical traffic information, while the estimated travel time of the second path is based on current traffic information.

We propose algorithms for finding the fastest dynamic historical/real-time paths. In addition, we generate a sub-network that is significantly smaller than the original entire network so that we just need to send the current traffic information based on this smaller sub-network to the client. Specifically, given an error parameter $\epsilon \in [0, 1]$, we propose a $(1 + \epsilon)$-approximate algorithm that takes as input the whole graph $G$ and outputs a reduced sub-network $G'_\epsilon$. Fastest paths computed in $G'_\epsilon$ have their greatest actual travel time guaranteed to be at most $(1 + \epsilon)$ times the greatest actual travel time computed in the original graph $G$.

Our experiments conducted on three road networks in Denmark and real-world GPS data offer valuable insights into the accuracy, efficiency, robustness and scalability of our proposed methods.

6.1 Introduction

Fastest path queries are gaining in prominence due to the proliferation of mobile and location-aware communication devices such as smartphones and navigation devices. Given a source vertex $u$ and a destination vertex $v$ in a road network, a fastest path query is to find a path
from \( u \) to \( v \) such that the time to travel from \( u \) to \( v \) is minimized. The fastest path query is more challenging than the traditional \textit{shortest path query} because the \textit{travel time} of a path varies with time, which is the not the case for the \textit{distance} of a path.

Fastest-path queries have been studied previously. Since the time of traversing an edge varies with time, existing studies \cite{6, 11, 12, 17, 35, 47, 49} generally derive travel times of network edges from \textit{historical information} about the traffic patterns in the road network. Most of these existing studies assume that the traffic patterns are \textit{periodic}. For example, the travel time of traversing an edge at 8 a.m. this Monday is assumed to be similar to that of traversing the edge at 8 a.m. every other Monday. With this assumption, they define a \textit{travel time mapping function} that takes an edge and a time point as arguments and returns an estimated travel time for traversing the edge when starting the traversal at that time point. Different proposals exist for how to construct the travel-time mapping function based on historical traffic information\cite{6, 12, 17, 35, 49}.

Most existing fastest-path queries are \textit{one-time queries}. They rely on historical information available when they are computed and can neither take into account the user’s subsequent actual progress in the road network nor more recent traffic information. This results in two drawbacks. The first, called the \textit{unpunctual arrival drawback}, is that the \textit{estimated} arrival time at each vertex along the path found by the existing methods is not equal to the user’s \textit{actual} arrival time due to the current traffic, which may deviate from the historical traffic. Consider a user that traverses an edge \((v_1, v_2)\) followed by edge \((v_2, v_3)\). When a user traverses \((v_1, v_2)\), the estimated travel time is calculated based on historical information, and thus the estimated arrival time at vertex \(v_2\) is also calculated based on the historical information. Next, the mapping function used for estimating the travel time of traversing edge \((v_2, v_3)\) depends on the arrival time at \(v_2\), which is the time when the traversal starts. The estimated time of arrival at \(v_2\) may be different from the actual arrival time. Thus, the estimated travel time of traversing edge \((v_2, v_3)\) becomes inaccurate.

The second drawback, called the \textit{inaccurate travel time drawback}, is that the \textit{estimated} travel time of an edge at a time point is not equal to the \textit{actual} travel time. Thus, the path found based on the estimated travel time may not be the actual fastest path. Traffic can vary substantially with time, and thus we say that this information is \textit{dynamic}. On the other hand, the historical information is \textit{static}. Thus, we call the fastest paths found by existing one-time methods \cite{6, 12, 17, 35, 49} \textit{fastest static historical paths}.

Motivated by this, we propose two new kinds of continuous fastest path queries. The first one is the \textit{fastest dynamic historical paths} query. Given a source vertex \(u\) and a destination vertex \(v\), the fastest dynamic historical path from \(u\) to \(v\) is the path computed based on the \textit{actual} arrival times at vertices instead of the \textit{estimated} arrival times. Thus, such a path eliminates the unpunctual arrival drawback. Initially, the user is at the source vertex. Whenever the user reaches a vertex, the current (or actual) arrival time at the vertex is used to estimate the travel time of a path from this vertex to the destination. In effect, this means that the path to the destination is reconsidered and potentially recomputed at each vertex.
The second kind of continuous fastest path query is the *fastest dynamic real-time path* query. Given again a source vertex \( u \) and a destination vertex \( v \), the fastest dynamic real-time path from \( u \) to \( v \) is the path that is not only computed based on the *actual* arrival times at vertices, but also based on *current* traffic information. Thus, such a path eliminates both of the identified drawbacks.

Computing fastest dynamic real-time paths is challenging. From time to time, the current traffic information gets updated. A naive approach to computing fastest dynamic real-time paths is to update each user (called a client) with the latest traffic information from the server based on the *entire* road network \( G \). This results in clients being inundated with traffic updates that do not affect their paths.

Instead, we propose to construct a relevant sub-network \( G'_0 \) with the property that the original fastest dynamic real-time path remains correct if the current traffic information for \( G'_0 \) is “consistent” with the historical information.

Although \( G'_0 \) is much smaller than the original network \( G \) and thus the current traffic information based on \( G'_0 \) sent from the server to the client is reduced, in some cases, the client would like to have a much smaller sub-network so that the cost of the communication between the client and the server is smaller. Although using the smaller sub-network may reduce the accuracy of the result, meaning that the path computed based on this smaller sub-network may not be the true fastest dynamic real-time path. To enable this, we introduce an error parameter \( \epsilon \in [0, 1] \) so that a smaller sub-network \( G'_\epsilon \) can be constructed with the property that, the path computed based on \( G'_\epsilon \) has its greatest actual travel time be at most \((1 + \epsilon)\) times the greatest actual travel time of the fastest dynamic real-time path. We propose an algorithm for generating such a sub-network \( G'_\epsilon \).

It is possible for the current traffic information to be *inconsistent* with the historical traffic information. This happens when abnormal traffic patterns, e.g., accidents, occur, which are not found in the historical information, and this causes a deterioration in travel time estimation. In such cases, we recompute the sub-graph \( G'_0 \) and resend it to the client in order to satisfy the travel time error guarantee and obtain a more accurate path.

There are two closely related studies \([11, 47]\) that consider current traffic information. However, the algorithms proposed are inaccurate for the problems we are studying. We cover these studies in the paper’s experimental study. For example, in a real-world case study with an actual unplanned event occurring due to an accident, we found that the path returned by one algorithm \([47]\) and the path returned by another algorithm \([11]\) are nearly 47.5% and 15.7% slower than the path returned by our algorithm on average, respectively.

The paper’s contribution is threefold. First, we are the first to propose two continuous fastest path queries, namely the *fastest dynamic historical path query* and the *fastest dynamic real-time path query*, which give more accurate results compared with the traditional one-time fastest path query considered in most existing studies \([6, 12, 17, 35, 49]\). We see this as an initial but important move away from the simple one-time fastest path computation based on historical information only studied in the literature \([6, 12, 17, 35, 49]\). Second, we propose to construct a sub-
network \( G'_0 \), smaller than the original entire network \( G \), so that the fastest dynamic real-time path computed based on \( G'_0 \) is exactly equal to the fastest dynamic real-time path computed based on the original entire network \( G \). Furthermore, we propose a \((1 + \epsilon)\)-approximate algorithm that is able to use a smaller sub-network \( G'_\epsilon \) such that the fastest paths computed in \( G'_\epsilon \) have their greatest actual travel time to be at most \((1 + \epsilon)\) times the greatest actual travel time computed in the original graph \( G \). Third, we report on comprehensive experiments, showing that our proposed methods are accurate, efficient, robust, and scalable.

The remainder of the paper is organized as follows. Section 6.2 formulates the problem. Section 6.3 describes related studies. Sections 6.4 and 6.5 give the exact algorithms and the approximate algorithms for finding fastest dynamic paths, respectively. Section 6.6 provides a discussion. Section 6.7 covers the empirical study of the proposed algorithms. Finally, Section 6.8 concludes the paper.

### 6.2 Problem Definition

We are given a road network \( G = (V, E) \) where \( V \) is a set of vertices representing intersections on roads in the network and \( E \) is a set of undirected edges representing road segments in the network.

For each edge \( e \in E \) and each time point \( t \), the travel time of a user on \( e \) starting at \( t \) is denoted by \( f(t, e) \). Before a user starts to traverse \( e \) starting at \( t \), we can estimate \( f(t, e) \), and its estimated value is denoted by \( \tilde{f}(t, e) \). There are many ways to estimate \( f(t, e) \). One such way is to use historical information. Specifically, most existing studies [6, 17, 35, 49] assume that traffic patterns are usually periodic. For example, the traffic pattern at 8 a.m. on this Monday is similar to that at 8 a.m. last Monday. Thus, we say that “8 a.m. on this Monday” and “8 a.m. last Monday” have the same traffic pattern during this period. Based on this assumption, \( \tilde{f}(t, e) \) is equal to the average travel time of a user on \( e \) starting at a time point in the past with the same period as \( t \). Similarly, the maximum (minimum) travel time of a user on \( e \) starting at \( t \), denoted by \( \tilde{f}_{\text{max}}(t, e) \) (\( \tilde{f}_{\text{min}}(t, e) \)), can be estimated based on the historical data. That is, \( \tilde{f}_{\text{max}}(t, e) \) (\( \tilde{f}_{\text{min}}(t, e) \)) is equal to the maximum (minimum) travel time of a user on \( e \) starting at a time point in the past with the same period as \( t \).

A second way of estimating \( \tilde{f}(t, e) \) is based on the current traffic information. Specifically, let \( T_0 \) be the current time point. For a time point \( t \geq T_0 \) and an edge \( e \), \( \tilde{f}(t, e) \) is equal to the travel time of a user on \( e \) recorded at the time point \( T_0 \).

In this problem setting, we consider a client-server model. Specifically, each client represents a user and a server is responsible for receiving requests from clients and sending updates to clients. Each user/client is equipped with a GPS device used to keep track of its location.

In the following, we define three types of paths, namely the fastest static historical path, the fastest dynamic historical path, and the fastest dynamic real-time path.
First, we define fastest static historical paths as follows. Consider a path $\pi$ of the form $\langle v_1, v_2, \ldots, v_l \rangle$, where $v_i$ is a vertex in $V$ for each $i \in [1, l]$. Suppose that the starting time point is $T_0$. The estimated time point of reaching $v_i$ is denoted by $\tilde{T}_{v_i}$ for $i \in [1, l]$. For vertex $v_1$, we know that $\tilde{T}_{v_1} = T_0$. For each vertex $v_i$ where $i \in [2, l]$, $\tilde{T}_{v_i}$ is defined to be $\tilde{T}_{v_{i-1}} + \sum_{i=2}^{l} f(\tilde{T}_{v_{i-1}}, (v_{i-1}, v_i))$. The estimated travel time of $\pi$ is defined to be $\tilde{T}_{v_l} - \tilde{T}_{v_1} = \sum_{i=2}^{l} f(\tilde{T}_{v_{i-1}}, (v_{i-1}, v_i))$. The estimated travel time is calculated based on historical data.

![Figure 6.1: Spatial Network](image_url)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Start Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_1, v_2)</td>
<td>5 15 15 25 35</td>
</tr>
<tr>
<td>(v_2, v_3)</td>
<td>5 10 15 15 25</td>
</tr>
<tr>
<td>(v_3, v_4)</td>
<td>25 25 20 2 5</td>
</tr>
<tr>
<td>(v_2, v_5)</td>
<td>95 100 85 80 80</td>
</tr>
<tr>
<td>(v_4, v_5)</td>
<td>80 80 80 80 80</td>
</tr>
<tr>
<td>(v_1, v_6)</td>
<td>20 15 15 15 15</td>
</tr>
<tr>
<td>(v_4, v_6)</td>
<td>20 25 15 10 15</td>
</tr>
</tbody>
</table>

Figure 6.2: $\tilde{f}(t, e)$ calculated based on the historical information

(At $t = 0$) **ATT**: Actual Travel Time

<table>
<thead>
<tr>
<th>Edge</th>
<th>ATT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_2, v_3)</td>
<td>20</td>
</tr>
<tr>
<td>(v_3, v_4)</td>
<td>2</td>
</tr>
</tbody>
</table>

(At $t = 2$) **ATT**: Actual Travel Time

<table>
<thead>
<tr>
<th>Edge</th>
<th>ATT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_2, v_3)</td>
<td>50</td>
</tr>
<tr>
<td>(v_2, v_5)</td>
<td>5</td>
</tr>
<tr>
<td>(v_3, v_4)</td>
<td>20</td>
</tr>
<tr>
<td>(v_4, v_5)</td>
<td>10</td>
</tr>
<tr>
<td>(v_4, v_6)</td>
<td>50</td>
</tr>
<tr>
<td>(v_1, v_6)</td>
<td>30</td>
</tr>
</tbody>
</table>

Figure 6.3: Real-time Edge Actual Travel Time Updates
Example 4 (Estimated Travel Time) Consider a road network $G$ as shown in Figure 6.1 containing 6 vertices, namely $v_1, v_2, \ldots, v_6$. Figure 6.2 shows the estimated travel time of a user via each edge in $G$ starting at each time point $t \in [0, 200)$. Consider the first row in the table representing edge $(v_1, v_2)$. The estimated travel time of a user via edge $(v_1, v_2)$ starting at time point $t \in [0, 5)$, denoted by $\tilde{f}(t, (v_1, v_2))$, is 5. Similarly, if $t \in [5, 10)$, $\tilde{f}(t, (v_1, v_2))$ is 15.

Consider that a user travels the path $(v_1, v_2, v_3, v_4)$ starting at time point 0. The estimated travel time of this path is calculated as follows. The user first traverses edge $(v_1, v_2)$. According to Figure 6.2, since the starting time point is 0, the estimated travel time via edge $(v_1, v_2)$ is 5. After traversing this edge, the user is at $v_2$ at the estimated time point equal to 5. Next, it will traverse edge $(v_2, v_3)$. Since the (estimated) time point is 5, the estimated travel time via edge $(v_2, v_3)$ is 10. The estimated time point of reaching $v_3$ is equal to 15 ($= 5 + 10$). Next, it traverses edge $(v_3, v_4)$. Similarly, we can obtain the estimated travel time of traversing edge $(v_3, v_4)$ starting from the time point 15 is 2. Thus, the total estimated travel time is $5 + 10 + 2 = 17$.

\[ \square \]

Definition 6 (Fastest Static Historical Path) Given a source vertex $u$ and a destination vertex $v$, the fastest static historical path from $u$ to $v$, denoted by $P_{sh}(u, v)$, is the path from $u$ to $v$ whose estimated travel time is the smallest.

Example 5 (Fastest Static Historical Path) Consider our running example in Example 4. Suppose that a user would like to travel from $v_1$ to $v_4$.

The fastest static historical path from $v_1$ to $v_4$ ($P_{sh}(v_1, v_4)$) is equal to $(v_1, v_2, v_3, v_4)$ since its estimated travel time is the smallest (which is 17).

Next, we define the fastest dynamic historical path as follows. We observe that the fastest static historical path is obtained based on the estimated time point to arrive at each vertex along the path, and is independent of the actual time point to arrive at a vertex. Motivated by this observation, we define the fastest dynamic historical path as follows by considering the actual arrival time. Specifically, a user starts at the source vertex. Whenever a user reaches a vertex at a time point $t$, if the actual arrival time $t$ at this vertex is not exactly equal to the estimated arrival time, then we have to compute the fastest static historical path from this vertex to the destination vertex. After that, the user follows the first edge along this path reaching another vertex and performs a similar step until it reaches the destination vertex.

Let the path found in the above process be $\pi_1 : \langle v_1, v_2, \ldots, v_m \rangle$ where $m$ is the total number of vertices traversed in the above process. This path is called the fastest dynamic historical path from $v_1 (= u)$ to $v_m (= v)$, denoted by $P_{dh}(u, v)$.

Example 6 (Fastest Dynamic Historical Path) Consider our running example (Example 4). Since computing the fastest dynamic historical path requires to know the actual travel time, we need to know the information about the actual travel time of traversing each edge starting at a given time point. Figure 6.3 shows the actual travel
6.2. PROBLEM DEFINITION

Consider the former table containing the information about edges \((v_2, v_3)\) and \((v_3, v_4)\) at time point 0. Consider edge \((v_2, v_3)\) (which is in the former table). According to Figure 6.3, at time point 0, the actual travel time of traversing \((v_2, v_3)\) starting from time point 0 is 20. Consider edge \((v_1, v_2)\) (which is not in the former table). The actual travel time of traversing edge \((v_1, v_2)\) starting from time point 0 is equal to 5 (See Figure 6.2).

Consider the latter table containing the information about edges \((v_2, v_3)\), \((v_3, v_4)\), \((v_4, v_5)\), \((v_2, v_5)\), \((v_4, v_5)\), and \((v_1, v_6)\) at time point 2. Similarly, the actual travel time of traversing \((v_2, v_3)\) starting no earlier than time point 2 (e.g., 2 and 3) is 50. Similarly, the actual travel time of traversing \((v_1, v_2)\) at time point 2 (or 3) is still 5.

The fastest dynamic historical path from \(v_1\) to \(v_4\) \((P_{dh}(v_1, v_4))\) is still equal to \(\langle v_1, v_2, v_3, v_4 \rangle\), which is the same as \(P_{sh}(v_1, v_4)\). But how to obtain path \(P_{dh}(v_1, v_4)\) is different from how to obtain \(P_{sh}(v_1, v_4)\).

Suppose that a user wishes to travel from \(v_1\) to \(v_4\). Initially, the user is at \(v_1\). According to the historical information (Figure 6.2), the user computes the fastest static historical path from \(v_1\) to \(v_4\), which is \(\langle v_1, v_2, v_3, v_4 \rangle\). Note that the estimated travel time of this path is 20. Then, the user traverses the first edge of this path (i.e., \((v_1, v_2)\)). In Example 4, we know that the estimated time point of reaching \(v_2\) is 5. According to Figure 6.3, we know that the actual time point of reaching \(v_2\) is still 5.

At \(v_2\), we can also compute the fastest static historical path from \(v_2\) to \(v_4\) (based on the historical information (Figure 6.2)). Similar to the estimation method illustrated in Example 4, we obtain the path as \(\langle v_2, v_3, v_4 \rangle\). The user then traverses edge \((v_2, v_3)\). Due to the current traffic (Figure 6.3), unfortunately, the actual travel time from \(v_2\) to \(v_3\) when starting at time point 5 is 50 (instead of 10, an estimated travel time based on the historical information). Thus, it reaches \(v_3\) at the time point equal to 55 (= 5 + 50).

Then, we compute the fastest static historical path from \(v_3\) to \(v_4\) (based on the historical information) and obtain the path as \(\langle v_3, v_4 \rangle\). The user then traverses edge \((v_3, v_4)\). Similarly, based on the current traffic (Figure 6.3), the actual travel time of traversing \((v_3, v_4)\) starting at time point 55 is 20. Thus, the actual time point that the user reaches \(v_4\) is 75 (= 55 + 20).

Now, we define the fastest dynamic real-time path. We observe that the fastest static/dynamic historical path is obtained based on the historical data, and is independent of the current traffic information. Motivated by this observation, we define the fastest dynamic real-time path, based on the current traffic information, as follows. Specifically, a user starts at the source vertex. Whenever the user reaches a vertex, the client computes the fastest path from the vertex to the destination vertex based on the current traffic information by running a shortest-path algorithm on the road network.
where the weight of each edge denotes the travel time of traversing this edge at the
current time point. Then, it proceeds to the next vertex along the edge and continues
the above process until the destination vertex is reached.

Let the path found in the above process be \( \pi : (v_1, v_2, ..., v_m) \), where \( m \) is the
total number of vertices traversed in the above process. This path is called the fastest
dynamic real-time path from \( v_1(= u) \) to \( v_m(= v) \), denoted by \( P_{dr}(u, v) \).

**Example 7 (Fastest Dynamic Real-time Path)** Consider our running example (Example 4). Suppose that a user would like to travel from \( v_1 \) to \( v_4 \).

The fastest dynamic real-time path \( (P_{dr}(v_1, v_4)) \) from \( v_1 \) to \( v_4 \) is \( (v_1, v_2, v_3, v_4) \).

To elaborate, consider the user at the source vertex \( v_1 \). Initially, we compute the
estimated travel time from \( v_1 \) to \( v_4 \) based on the current traffic information instead of the historical information. Consider the path \( (v_1, v_2, v_3, v_4) \). According to the
current traffic information at the time point \( 0 \) (Figure 6.2 and Figure 6.3), the travel
times of traversing \( (v_1, v_2), (v_2, v_3) \) and \( (v_3, v_4) \) are \( 5 \), \( 20 \) and \( 2 \), respectively. Thus, the total travel time of the path \( (v_1, v_2, v_3, v_4) \) is \( 27 \) \((= 5 + 20 + 2)\). It is found that this path is the path from \( v_1 \) to \( v_4 \) starting at time point \( 0 \) with the shortest travel time. Thus, the user moves along the first edge of this path (i.e., \( (v_1, v_2) \)).

Next, the user reaches \( v_2 \) via \( (v_1, v_2) \) at time point \( 5 \) based on the current traffic.
At vertex \( v_2 \), it computes the estimated travel time from \( v_2 \) to \( v_4 \) based on the current
traffic information. According to Figure 6.3, the travel times of traversing \( (v_2, v_3) \) and \( (v_3, v_4) \) become \( 50 \) and \( 20 \), respectively. At the current time point (i.e., \( 5 \)), it is found that the path \( (v_2, v_5, v_4) \) is the path from \( v_2 \) to \( v_4 \) starting at the time point \( 5 \)
with the shortest travel time. Thus, the user moves along the first edge of this path
(i.e., \( (v_2, v_5) \)).

Then, the user reaches \( v_5 \) via \( (v_2, v_5) \) at the time point \( 10 \) based on the current
traffic (since the time of traversing \( (v_2, v_5) \) starting at time point \( 5 \) based on the
current traffic is equal to \( 5 \)). At vertex \( v_5 \), we keep finding the path from \( v_5 \) to \( v_4 \)
starting at the time point \( 10 \) with the shortest travel time and find that this path is
equal to \( (v_5, v_4) \). Finally, the user reaches \( v_4 \).

\[ \]

<table>
<thead>
<tr>
<th>Path Type</th>
<th>Final Path Traversed</th>
<th>Travel Time Estimated at ( v_1 )</th>
<th>Final Travel Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_{sh}(v_1, v_4) )</td>
<td>( (v_1, v_2, v_3, v_4) )</td>
<td>17</td>
<td>75</td>
</tr>
<tr>
<td>( P_{dh}(v_1, v_4) )</td>
<td>( (v_1, v_2, v_3, v_4) )</td>
<td>17</td>
<td>75</td>
</tr>
<tr>
<td>( P_{dr}(v_1, v_4) )</td>
<td>( (v_1, v_2, v_3, v_4) )</td>
<td>27</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 6.1: Path Comparisons

Table 6.1 shows a summary of the three paths we described based on our running example. For each path, we show the final path traversed, the travel time estimated at
the source vertex \( v_1 \) and the final travel time. Obviously, \( P_{dr}(v_1, v_4) \) is better
than \( P_{sh}(v_1, v_4) \) and \( P_{dh}(v_1, v_4) \). It can be easily explained with the reason that the
6.3. RELATED WORK

computation of $P_{dr}(v_1,v_4)$ involves the current traffic information but $P_{sh}(v_1,v_4)$ and $P_{dh}(v_1,v_4)$ do not.

Since finding $P_{sh}(u,v)$ (i.e., static paths) has been studied in the literature [6, 17, 35, 49], in this paper, we focus on how to find $P_{dh}(u,v)$ and $P_{dr}(u,v)$ (i.e., dynamic paths), which has not been studied before in the literature. We are studying the following problem.

**Problem 1 (Finding Fastest Dynamic Paths)** Given a source vertex $u$ and a destination vertex $v$, we want to find $P_{dh}(u,v)$ and $P_{dr}(u,v)$.

---

6.3 Related Work

In this section, we first categorize the related studies into two categories. The first category is **fastest path queries on road networks with fixed weight edges** where the travel time of traversing each edge does not change over time (Section 6.3.1). The second category is **fastest path queries on road networks with dynamic weight edges** where the travel time of traversing each edge can change over time (Section 6.3.2).

### 6.3.1 Fastest Path Queries on Road Networks with Fixed Weight Edges

There are a lot of existing studies [10, 14, 16, 22, 25, 26, 28, 54, 55] for fastest path queries on road networks with fixed weight edges. Specifically, given a graph with static travel times (i.e., the travel times do not change over time) as edge weights, traditionally, Dijkstra’s algorithm [16] has been employed to efficiently compute the fastest paths between two vertices (i.e., the path between two vertices with the shortest travel time). Dijkstra’s algorithm uses a dynamic programming approach that uniformly expands its search frontier from the source vertex and ends when the destination vertex is encountered. The A* search [28] is an extension of Dijkstra’s algorithm that tries to reduce the intermediate vertices explored during the search, by incorporating a heuristic estimate of the cost to reach the destination vertex from any given vertex.

Despite the availability of many efficient Dijkstra and A* implementations [14, 22], commercial systems dealing with large real-world road networks require **speedup techniques** that take advantage of various precomputation strategies to avoid costly path computations. Hierarchical speedup approaches [10, 25, 26, 54, 55] incrementally prune low-degree vertices in the road network to form a **multi-level highway hierarchy**, accelerating Dijkstra-like searches. In comparison, goal-based approaches like ALT [24] pre-compute fastest paths from all nodes to a small number of landmarks; these paths are then used as lower bounds to improve the A* search. In summary, speedup techniques were proposed to study the trade-off among precomputation, storage space and query time, in order to speed up fastest path queries by several orders of magnitude.
In our problem setting, where edge weights fluctuate in real-time which is more realistic, precomputations must be kept fresh which in turn triggers many costly path recomputations. Thus, the methods described so far are valid when edge weights are static and thus their methods cannot be used in our problem setting.

### 6.3.2 Fastest Path Queries on Road Networks with Dynamic Weight Edges

There are two branches for fastest path queries on road networks with dynamic weight edges. The first branch is the queries based on the historical information only (Section 6.3.2). All algorithms under this branch, though can handle dynamic weight edges, do not consider the real-time traffic, which is very important to estimate a more accurate travel time. The second branch is the queries based on both the historical information and the current traffic information (Section 6.3.2). In this paper, we focus on the latter, which is more realistic and more accurate.

**Historical Information-Driven Approach**

There are some historical information-driven approaches in the literature \[6, 12, 17, 35, 49\]. Cooke and Halsey \[6\] first proposed fastest static historical path queries, originally named as time-dependent fastest path queries (TDFPQ), considering the historical information and solved with a discrete-time approach. This approach discretizes the time intervals into evenly distributed time points and then constructs an instance of the entire graph for each time point. Then, \[6\] makes use of the techniques described in Section 6.3.1 for the fastest path queries on the road network with fixed weight edges based on the instances constructed in order to find the fastest static historical path. However, this approach has two major drawbacks. First, it gives an unbounded approximate solution since it does not handle the traffic between any two consecutive time points. Second, storing graph copies per time point incurs a substantial storage overhead.

Orda and Rom \[49\] proposed a generalized Bellman-Ford based algorithm for this problem. Later, Kanouhas et al. \[35\] proposed a path-enumeration technique and Ding et al. \[17\] proposed an extended Dijkstra’s algorithm. Specifically, in \[17\], the weight of each edge $e$ in the network changes over time and is captured by $f(t, e)$ where $t$ is a timestamp. Whenever the extended algorithm needs to use the weight of an edge $e$, it computes $f(t, e)$ by taking the estimated best-known arrival time at $e$ as the input parameter $t$. Based on the weights computed from function $f(\cdot, \cdot)$, the algorithm finds the static historical path.

Recently, Demiryurek et al. \[12\] proposed a spatial partitioning technique which pre-computes the fastest paths between some pairs of spatial regions for this problem.

Bader et al. \[2\] proposed a comprehensive study of giving alternative routes between a source vertex and a destination vertex based on the historical information. Some examples of alternative paths are $k$-shortest paths, Pareto paths, plateau paths and penalty paths. Details can be found in \[2\].
Delling et al [11] proposed an algorithm called Corridor. Specifically, given a source vertex \( u \) and a destination vertex \( v \), Corridor finds a path \( \pi \) with the shortest distance based on the graph. Then, for each vertex \( x \) along \( \pi \), it finds all vertices which are at most \( k \) edges from \( x \). Finally, all these vertices form the set of vertices and the induced subgraph is based on this set of vertices. This induced subgraph is called the Corridor. Finally, the algorithm finds the shortest path within the subgraph and instructs the user to traverse from \( u \) to \( v \). Every time the user traverses 10 turns along the path, the algorithm instructs the user to deviate from the original path given by the algorithm (because [11] wants to probabilistically simulate the wrong driving pattern which can be found in real-life applications).

All the above algorithms only rely on the historical information in order to estimate the travel time of each edge. They do not consider the current traffic information considered by us in this paper.

**Current Traffic Information-Driven Approach**

To the best of our knowledge, there is only one closely related work [47] which proposed the current traffic information-driven approach. Specifically, the model proposed by [47] considers the current traffic information for finding the fastest paths on the network. In this model, the travel time of each edge changes over time based on the current traffic on the road network. The problem is to find the so-called “fastest” path for each user registering for the system. This work is considered as the state-of-the-art method for the current traffic information-driven approach.

[47] proposed a model as follows. When a user registers in our system with the source vertex \( u \) and the destination vertex \( v \), the system finds a set \( X \) of \( k \) edge-disjoint fastest paths from \( u \) to \( v \) where \( k \) is a user parameter. Note that no two paths in \( X \) contain the same edge. Besides, the \( k \) paths in \( X \) have an ordering in ascending order of their estimated travel times. The \( i \)-th path in \( X \) based on this ordering is called the \( i \)-th candidate path in \( X \) for each \( i \in [1, k] \).

Initially, the server sends the first candidate path \( \pi \) in \( X \) to the client. Then, the client starts to follow this path \( \pi \). Whenever the client is still traversing along this path, the system checks the two pre-defined conditions regularly. The first condition is whether there exists an edge in the path \( \pi \) such that its current travel time differs from the previously recorded travel time by a threshold value \( \alpha \) which is a user parameter. The second condition is whether there are at least \( \beta \) edges in \( \pi \) such that their actual travel times are updated where \( \beta \) is another user parameter. If one of the conditions is satisfied, then the system will send the next candidate path \( \pi' \) in \( X \) to the client so that the client will follow this updated path \( \pi' \) instead of the original path. The above steps continue until the user reaches the destination vertex.

The approach [47] suffers from the following drawbacks. First, the requirement of “edge-disjoint” paths in the output set \( X \) results in undesirable paths when a non-first candidate path in \( X \) is used. To elaborate, we consider the following scenario. When a user receives the first candidate path \( \pi \) from the system, it travels along this path \( \pi \). When it reaches some points in the middle part of \( \pi \) and receives another
candidate path \( \pi' \) from the system (since one of the two conditions is satisfied), it has to travel back to the original source vertex and follows the updated path \( \pi' \). This scenario can be easily explained by the “edge-disjoint” requirement in the output set \( X \) since two paths in \( X \) are connected via either a source vertex or a destination vertex. Since it has to travel back to the source vertex, the actual travel time of this user is very large. Second, since there is no position tracking of the user in the system, there are some unnecessary changes from one path to another part. Consider that the user passes some edges in the first candidate path \( \pi \) suggested by the system, says \( e_1 \) and \( e_2 \). It is obvious that the current traffic of \( e_1 \) does not affect the remaining route of the user along \( \pi \). However, since the system has no information about the position of this user, whenever it knows that there is a traffic update on \( e_1 \) and one of the two conditions is satisfied based on this traffic, the system has to notify the user to change to another candidate path in the output set. Third, \([47]\) restricts the greatest distance between a source and destination vertex to 50 miles, due to scalability issues.

### 6.4 Exact Algorithm

Let \( u \) be a source vertex and \( v \) be a destination vertex. In this section, we present our exact algorithm finding \( P_{dh}(u, v) \) in Section 6.4.1 and our exact algorithm finding \( P_{dr}(u, v) \) in Section 6.4.2.

#### 6.4.1 Algorithm Finding \( P_{dh}(u, v) \)

In this section, we present an exact algorithm finding \( P_{dh}(u, v) \) and an exact algorithm finding \( P_{dr}(u, v) \) where \( u \) is a source vertex and \( v \) is a destination vertex. We first focus on the algorithm finding \( P_{dh}(u, v) \). Since the summary of the historical information is stored at the client side, there is no need for communication between the client and the server. Thus, each client just executes the algorithm at the client side.

The algorithm finding \( P_{dh}(u, v) \) is straightforward.

- **Step 1: Traversing the First Edge:** Initially, a vehicle/client is at vertex \( u \) at timestamp \( T_u \). It issues an existing algorithm \([17]\), which returns the fastest static historical path \( P \) from \( u \) to \( v \). It then obtains the first edge \( e \) along this path and traverses this edge. Let \( e = (u, u') \). It leaves \( u \) at \( T_u \) and reaches \( u' \) at \( T_{u'} \). Note that \( T_{u'} \) may be different from \( T_u \) used in the existing algorithm which returns the fastest static historical path \( P \) from \( u \) to \( v \).

- **Step 2: Traversing the Second Edge:** At timestamp \( T_{u'} \), it issues the existing algorithm which returns the fastest static historical path \( P \) from \( u' \) to \( v \). It then obtains the first edge \( e' \) along this path and traverses via this edge. Similarly, let \( e' = (u', u'') \). It leaves \( u' \) at \( T_{u'} \) and reaches \( u'' \) at \( T_{u''} \).

- **Step 3: Traversing the \( i \)-th Edge:** Let \( v_i \) be the \( i \)-th vertex traversed by a vehicle. At timestamp \( T_{v_i} \), it issues the existing algorithm which returns the
fastest static historical path $P$ from $v_i$ to $v$. It then obtains the first edge $e_i$ along this path and traverses via $e_i$.

- **Step 4: Traversing the Last Edge:** It repeats the above step until the edge just traversed contains the destination vertex $v$.

It is easy to verify the correctness of the above algorithm.

This algorithm can be executed at the client side without any communication with the server since the summary of the historical information is stored at the client side.

**Time Complexity Analysis:** Given the source vertex $u$, a destination vertex $v$, and graph $G = (V, E)$, the time complexity of computing the fastest static historical path is given by $O(|E| + |V|\log|V|) \cdot \gamma(t))$. Here, $\gamma(t)$ is the time required to look up the corresponding edge weight at timestamp $t$\cite{17}. The time complexity of the operation $\gamma(t)$ is $O(1)$. In the worst case, the first-pass path can traverse via $|V|$ vertices, the second-pass path can traverse $|V| - 1$ vertices and so on till we reach the destination vertex. Let $A$ denote the time complexity of computing a path. Then, the total number of such operations is $O(|V| + (|V| - 1) + \ldots + 1)$. Thus, the final time complexity of the algorithm for finding $P_{dh}(u, v)$ is $O(|V|^2 \cdot A) = O(|V|^2 \cdot (|E| + |V|\log|V|) \cdot \gamma(t))$. In practice, we bound the total number of possible path switches and set it to 5 in our experiments.

**Space Complexity Analysis:** Each client has to store the travel time histogram of each edge $e$ on the network denoting $f(t, e)$ where $t$ is a timestamp. The travel time histogram of an edge $e$ is a histogram denoting the travel time of traversing $e$ at each pre-defined time (e.g., 9am every Monday). Let $H$ be the size of the travel time histogram. The space complexity of this algorithm at the client side is $O(|E| \cdot H)$.

### 6.4.2 Algorithm Finding $P_{dr}(u, v)$

We propose an algorithm finding $P_{dr}(u, v)$ which is exactly the same as the algorithm finding $P_{dh}(u, v)$ except the following differences.

- Since we focus on the current traffic information instead of the historical data, instead of the fastest static historical path from the current vertex to the destination, the algorithm computes the path from the current vertex to the destination based on the current traffic information by running the fastest-path algorithm \cite{17} on the road network where the weight of each edge denotes the travel time of a vehicle passing this edge recorded at the current timestamp.

- The client obtains current traffic information from the server and computes the fastest path at the client side. Since the vehicle/client needs to obtain the current traffic information, there are communications between the client and the server. Whenever there is an update on the traffic information of an edge, the server sends the message about this update to the client.

It is also easy to verify the correctness of the above algorithm.
As we can notice, some messages about the current traffic information are sent from the server to the client. If all information on the whole network $G$ is sent from the server to the client, then the algorithm returns $P_{dr}(u,v)$ exactly. However, sending all information on the whole network $G$ is too costly, and thus in the next section, we present an approximate algorithm with the travel time error guarantee if the current traffic information is “consistent” with the historical information.

**Time Complexity Analysis:** The time complexity analysis is similar to the previous section.

**Space Complexity Analysis:** Each client has to keep track of the current travel time of each edge and thus each edge is associated with an additional variable denoting the current travel time. Similarly, the space complexity of the algorithm at the client side is $O(|E| \cdot H)$.

### 6.5 Approximate Algorithm

Given an error parameter $\epsilon \in [0, 1]$, we propose a $(1 + \epsilon)$-approximate algorithm that takes as input the whole graph $G$ and outputs a reduced sub-network $G'_\epsilon$. Fastest paths computed in $G'_\epsilon$ have their greatest actual travel time to be at most $(1 + \epsilon)$ times the greatest actual travel time computed in the original graph $G$. If $\epsilon$ is smaller, then the actual travel time of the path returned by the algorithm is nearer to the actual travel time of the fastest dynamic real-time path.

Given a travel time $f$, a time range $\Delta t$ and an edge $e$, we say that $f$ is within the expected time range of $e$ over the time range $\Delta t$ if $f_{\min}(t', e) \leq t \leq f_{\max}(t', e)$ for each $t' \in \Delta t$. Given a time range $\Delta t$, we say that the current traffic information over the time point $\Delta t$ is consistent with the historical information if the actual travel time of a vehicle via each edge recorded for each time point in $\Delta t$ is within the expected travel time range of this edge over the time range $\Delta t$.

In the following, we assume that the current traffic information is consistent with the historical information so that we can have an approximate solution with the traffic time error guarantee. This assumption makes sense since drivers of vehicles expect that the current traffic should be in their expectation. To the best of our knowledge, this is the first work to study the traffic time error guarantee in the context of real-time traffic. This is the first initial step for this fastest dynamic real-time problem. How to handle the case when the assumption is violated will be discussed in Section 6.6.

We notice that sending the current traffic information on the whole network $G$ from the server to the client is costly. A better technique is to send this information on a smaller sub-network $G'_\epsilon$ of $G$, which is sufficient to return a $(1 + \epsilon)$-approximate solution. In Section 6.5.1, we describe a method to find a sub-network $G'_\epsilon$ of $G$ so that the algorithm returns a $(1 + \epsilon)$-approximate solution.

#### 6.5.1 Finding a Portion of the Network

Before we introduce how to generate a sub-network within which the algorithm returns a $(1 + \epsilon)$-approximate solution, we introduce three important concepts, namely
6.5. APPROXIMATE ALGORITHM

$G_{\text{max}}$, $G'_{0}$ and $G'_{\epsilon}$.

The first concept is “$G_{\text{max}}$”. We define a sub-network, denoted by $G_{\text{max}}$, to be a graph containing the same set of vertices and the same set of edges as the original graph $G$ but the weight of each edge $e$ in $G_{\text{max}}$ is set to $\max_{t \in T} f_{\text{max}}(t, e)$ where $T$ is a set of time points recorded in the historical information. We denote the weight of each edge $e$ in $G_{\text{max}}$ by $w_{G_{\text{max}}}(e)$.

Given a vertex $u$ and a vertex $v$, the shortest weight distance between $u$ and $v$ is defined to be the shortest distance between $u$ and $v$ on $G_{\text{max}}$ where the weight of each edge is interpreted as a distance. Given a vertex $u$ and a non-negative real number $d$, we denote $VS(u, d)$ to be the set of vertices in $G_{\text{max}}$ each of which has its shortest weight distance to $u$ at most $d$.

The second concept is “$G'_{0}$”. Consider a source vertex $u$ and a destination vertex $v$. Let $d_{\text{max}} = d_{G_{\text{max}}}(u, v)$. Formally, $G'_{0} = G_{\text{max}}[VS(u, d_{\text{max}}) \cap VS(v, d_{\text{max}})]$.

It is easy to verify that all possible fastest dynamic real-time paths from $u$ to $v$ can be found in $G'_{0}$ because $G'_{0}$ was generated based on $G_{\text{max}}$ in which the weight of each edge denotes the greatest travel time of traversing this edge. To compute $G'_{0}$, we perform a bi-directional iterative deepening depth-first search from $u$ and from $v$, limited by depth $d_{\text{max}}$.

In practice, $G'_{0}$ is much smaller than the whole network $G$. In our experiments, in the default dataset $NJ$, $G$ has 2,049,540 vertices and $G'_{0}$ has 456,450 vertices, which is about 22% of the number of vertices in $G$. It also achieves similar compression on $G$ for the largest dataset $JU$.

The third concept is “$G'_{\epsilon}$”.

**Definition 7 (Errorless-Subnetwork $G'_{0}$ for $(u, v)$)** $G'_{0}$ is defined to be the subgraph (or sub-network) of the original graph/network $G$ induced by the intersection of vertex sets $VS(u, d_{\text{max}})$ and $VS(v, d_{\text{max}})$. Formally, $G'_{0} = G_{\text{max}}[VS(u, d_{\text{max}}) \cap VS(v, d_{\text{max}})]$.

Next, we propose a greedy algorithm to generate $G'_{\epsilon}$ as shown in Algorithm 6.

The major steps in Algorithm 6 are as follows. First, sort all the edges in ascending order of their weights, and initialize a graph $G'$ with the same set of vertices as graph $G'_{0}$ (i.e., a subset of $V$) and an empty edge set $E'$. We use the definition of setting distances between vertices to infinite when there is no path between them. Each edge is processed one by one, and an edge $(a, b)$ is added to $G'$ only under the condition that $d_{G'}(a, b) > (1 + \epsilon) w_{G'_{0}}((a, b))$. Here, $d_{G'}(a, b)$ means the length of
### Algorithm 6: Algorithm for Generating $G'_\epsilon$

1: $E^* \leftarrow$ a list of all edges in $E$ sorted in ascending order of their weights
2: $E' \leftarrow \emptyset$
3: $G'_\epsilon \leftarrow (V, E')$
4: for each $(a, b) \in E^*$ processed with the ordering in $E^*$ do
5: \hspace{1em} if $d_{G'_\epsilon}(a, b) > (1 + \epsilon)w_{G'_0}((a, b))$ then
6: \hspace{2em} $E' \leftarrow E' \cup \{(a, b)\}$
7: \hspace{2em} $G'_\epsilon \leftarrow (V, E')$
8: end if
9: end for
10: return $G'_\epsilon$

the shortest path from a vertex $a$ to another vertex $b$ on $G'_\epsilon$ where the weight of each edge is regarded as a distance.

Since the algorithm performs $O(|E|)$ times of computing $d_{G'_\epsilon}(a, b)$ each of which takes $O(|E| + |V| \log |V|)$ time, the overall time complexity of Algorithm 6 is $O(|E|^2 + |E| \cdot |V| \log |V|)$.

**Lemma 8** Given a source vertex $u$ and a destination vertex $v$, the greatest actual travel time of the path from $u$ to $v$ found based on $G'_\epsilon$ is at most $(1 + \epsilon)$ times the greatest actual travel time of the fastest dynamic real-time path from $u$ to $v$ based on $G'_0$.

**Proof:** Let the fastest dynamic real-time path from $u$ to $v$ computed based on $G'_0$ be $\pi : \langle u = v_1, v_2, \ldots, v_k = v \rangle$. Let $\delta(a, b)$ denote the actual travel time of this path.

We construct a path $\pi'$ as follows. Initially, $\pi'$ is set to $\pi$. For each edge $(v_j, v_{j+1}) \notin G'_\epsilon$ where $j = 1, \ldots, k - 1$, we replace $(v_j, v_{j+1})$ with the shortest path from $v_j$ to $v_{j+1}$ on $G'_\epsilon$ where the weight of each edge is regarded a distance. Note that $d_{G'_\epsilon}(v_j, v_{j+1}) \leq (1 + \epsilon)w_{G'_0}((v_j, v_{j+1}))$ (since edge $(v_j, v_{j+1})$ cannot be found in the edge set of $G'_\epsilon$). Besides, it is easy to verify that $\pi'$ is a path on $G'_\epsilon$. \qed

### 6.5.2 Finding $\epsilon$-Approximate Path

The algorithm for finding an $\epsilon$-approximate solution of $P_{dr}(u, v)$ is exactly the algorithm described in Section 6.4.2 where $u$ is a source vertex, $v$ is a destination vertex and $\epsilon$ is an error parameter. The only difference is that this algorithm is based on a smaller subgraph/sub-network $G'_\epsilon$ instead of the whole network $G$.

### 6.6 Discussion

This section deals with the fundamental problem of dealing with edge traversal time updates that exceed the historical maximum travel time of the edge $e$, i.e., they exceed $\max_{t \in T} \hat{f}_{\max}(t, e)$. This case usually arises in case of unforeseen events like traffic accidents and adverse weather conditions.
Intuitively, in the case where an edge travel time update exceeds its historical maximum travel time, we recompute a new errorless-subgraph $G'_{0''}$ in such a way that we re-use the edges from the previous errorless-subgraph $G'_{0}$ and only introduce new edges or modify edge weights to reflect the changes in the subgraph.

![Figure 6.4: Re-computation of subgraph $G'_{0}$ to give new errorless-subgraph $G'_{0''}$.

In Figure 6.4, $u$ is the source vertex and $v$ is the destination vertex. Our initial $G_{max}$ has all its edges set to their respective upper bound values. Then, $d_{max}$ represents the total actual travel time of the fastest dynamic real-time path in $G_{max}$. $VS(u, d_{max})$ (from the source vertex) and $VS(v, d_{max})$ (from the destination vertex) are computed, shown with bold outlines in Figure 6.4 and the vertices that lie in their intersection form the vertex set from which the induced subgraph $G'_{0}$ (errorless-subgraph) is formed. Besides, we mark the boundary vertices (E.g., $b_1 – b_7$ in Figure 6.4) for both subgraphs $VS(u, d_{max})$ and $VS(v, d_{max})$. The nodes adjacent to each boundary vertex are classified as being inside or outside the boundaries of $VS(u, d_{max})$ and $VS(v, d_{max})$. For example, boundary node $b_2$ has neighbors $i_1$ and $i_2$ inside the boundary, while neighbors $o_1$ and $o_2$ are outside the boundary.

When, an edge in $G'_{0}$ has an edge update where its travel time is higher than its current upper bound (based on our historical data), this edge weight is updated and accordingly a new $G'_{0''}$ must be computed and sent to the client. Consider that the edge $e$ whose travel time exceeds $\max_{T} \tilde{f}(t, e)$ . Two cases occur.

**Case (i): The edge $e$ belongs to $d_{G'_{0}}(u, v)$. In this case, we recompute the fastest path to get a new one, i.e., $d'_{max}$ (shown in Figure 6.4). Then, we expand with depth $d'_{max} - d_{max}$ from the boundary nodes outwards, to compute both $VS(v, d'_{max})$ and $VS(u, d'_{max})$, to finally compute $G'_{0''}$. Note that only the outer neighbors of the boundary nodes are considered when expanding outwards. Only the new edges with edge weights and old edges with modified edge weights are then relayed to the client, thus saving on communication costs as well.**

**Case (ii): The edge $e \notin d_{G'_{0}}(u, v)$. In this case, there is no need for a re-computation, but we send the new edge weight update to the client so that the client
can avoid using this expensive edge for future path changes.

6.7 Experiments

In this section, we empirically study the performance of our approach and compare it with other baselines [11, 47]. We identify $k$-DJPath [47] as the state-of-the-art because it is the only related work that takes into account real-time traffic updates to continuously provide alternate fastest paths to the clients. In $k$-DJPath, we implemented the algorithm outlined in [47] and set a user parameter $K$-candidate-paths of $k$-DJPath as 5, as outlined to be the best setting in their experiments [47].

In Corridor [11], we generated a sub-network also called Corridor (same as the algorithm name) as described in [11] using the naive method, with no contraction-heirarchy (CH) based speedup techniques because real-time travel time updates would cause more CH re-computations that would be very computation-intensive. Following the experimental settings outlined in [11], we set a user parameter $k$-wrong-turns of Corridor to 6. This setting ensured that the sub-network was large enough for drivers to reach their destination.

Furthermore, we implemented another baseline called A-Corridor, which stands for the adapted-Corridor method, where we generated the initial sub-network and then used real-time traffic information, similar to our methods, to recompute fastest paths within the sub-network that is shipped to a client. In contrast, Corridor method computed the sub-network and only used a probabilistic driving model that would make a wrong-turn on every 10th vertex crossed along the fastest path. Thus, the Corridor method ignored all real-time information after the creation of the sub-network.

The different path computation techniques are fastest static historical path ($sh$), fastest dynamic historical path ($dh$), and fastest dynamic real-time path ($dr$). Specifically, we denote the different path computation methods generated by our algorithms as OurAlg-Pa, where Pa is a placeholder for the various path types, with possible values \{sh, dh, dr\}. All results are shown on the reduced $(1 + \epsilon)$-approximation subgraph ($G_\epsilon$), but we also conducted the same experiments on the original graph $G$ to compare the effects.

6.7.1 Experimental Setup

**Road Networks:** We obtained three real road networks from OpenStreet Map (OSM) of: Aalborg (AA) with 149,430 vertices and 378,420 edges; North Jutland (NJ) with 2,049,540 vertices and 4,906,380 edges (our default dataset); and Jutland (JU) with 13,357,520 vertices and 32,459,480 edges.

OSM provides the road segments as 2-D polylines (called Ways in OSM) with vertices (called Nodes in OSM) placed at points of inflection, i.e., where the polyline’s curvature changes. We computed all intersections of the 2-D polylines to generate the underlying graph of the road network.
Geographically, AA is a major city located in NJ, which in turn is the northern region of JU. In comparison to [47], the state-of-the-art approach, i.e., k-DJPath, used a single road network of Boston BO with 30,000 vertices and 40,000 edges, which is much smaller in size than the datasets used in our experiments.

For our scalability experiments, we generated datasets of different sizes by introducing more vertices (Nodes) at the points of inflection in the 2D polylines from OSM. Later, the graphs were extracted in a similar fashion as was outlined for the original datasets.

**GPS Data:** A large real-world GPS dataset consisting of nearly 180 million GPS records, measured at a frequency of 1 Hz, spanning years 2007 – 2008, was used in our experiments to validate our approaches. Historical traffic patterns were deduced from this data and it was also used to replay actual real-time traffic updates that occurred in that time period on road segments. The GPS data covers the largest region, i.e. JU.

**Real Case Study:** For our GPS dataset, we found an unplanned closure event where a major accident caused major traffic delays and a planned closure event where some major roads had to be blocked due to a planned running event.

**Unplanned Case Study:** We incorporated in our traffic updates, an accident that took place on a 6-lane, motorway tunnel (0.5 kms long), which connects the central portions of dataset AA to each other. The accident took place at 15:30 on August 6, 2008, causing a traffic delay of 31 minutes in the surrounding regions, affecting all incoming/outgoing traffic to/from AA.

**Planned Case Study:** The annual “Aalborg Brutal Marathon” took place at 13:00 on March 21, 2008 in Aalborg (AA), on busy city roads and forest paths on a very hilly course near the railway station. Many major roads were blocked for nearly 12 hours during this event.

**Query Settings:** We generated user queries randomly. Specifically, for each user query, we randomly generate a source vertex and a destination vertex. Since different source vertices and different destination vertices give different shortest network distances and thus different fastest paths, we classify two types of user queries, namely short and long running queries. Short running queries had short network distances between source and destination vertices, with corresponding short travel times, while Long running queries had larger network distances between source and destination vertices. For brevity, only experimental results with long-running queries are shown. The results with short-running queries were not shown here. This was done in order to test the capacity of various architectures for handling traffic updates, while serving clients with accurate up-to-date fastest path information. In addition to the source vertex and the destination vertex, we generate the starting time $t$ that the user will start the trip in the network. Here, in dataset AA, we define short-running and long-running queries that ranged between 1-2 and 5-6 km, respectively. In dataset NJ, we define short-running and long-running queries that ranged between 2-5 and 20-50 kms, respectively. In dataset JU, we define short-running and long-running queries that ranged between 1-50 and 150-200 kms, respectively. Note that k-DJPath imposed a maximum closeness-constraint that the source vertex is at most 80 km (or
50 miles) away from the destination vertex on the network, which is quite restrictive. **Parameter Settings:** The experiments were conducted by varying several parameters to study the effect of the trade-offs among accuracy, efficiency, and memory usage. We vary the following parameters: dataset sizes, the approximation error $\epsilon$, the number of concurrent clients, the number of training months and the GPS sampling rate. The dataset size corresponds to the total number of vertices in the graph. The number of concurrent clients corresponds to the total number of user queries issued concurrently. The number of training months will be described in detail in Section 6.7.4. The GPS sampling rate will be described in detail in Section 6.7.5.

Table 6.2 shows the parameters with their default values in bold. The dataset sizes shown in Table 6.2 reflect the datasets that were constructed for our scalability experiments. $NJ$ was the default road network in all our experiments. Experiments were conducted with default parameter values unless explicitly stated. **Implementation:** Preprocessing tasks were written in C++. To closely mimic the dynamic nature of vehicles in traffic, we chose to implement our designs in Erlang (http://www.erlang.org/). Erlang, developed by Ericsson Computer Science Labs (http://www.ericsson.com/), allows to build massively scalable soft real-time systems, using a message-passing framework with very little shared memory. Recently, Facebook Chat was rewritten in Erlang to handle the enormous number of users chatting simultaneously (http://tinyurl.com/o9z452e). All experiments were conducted on a Debian 7.1 Linux machine, with an Intel Xeon CPU E5-2650 2.00GHz (20MB cache, hyper-threading, 8 cores) and 32 GB internal memory. **Measurement Units:** Here, we explain the meaning of our measurements.

*Estimated Travel Time:* is computed by summing up the estimated edge traversal time for each edge that constitutes the path, generated by an algorithm, for a client to reach its destination. This measurement is different for various path kinds, e.g., static historical, dynamic historical and realtime dynamic paths. *Actual Travel Time:* is computed by summing up the actual edge traversal time for each edge that constitutes the path taken by the client. *Preprocessing Time:* is the time it takes to compute $G'_0$ (or $G'_\epsilon$) in our algorithms, the sub-network in the Corridor-based algorithms and the top-$k$ paths in $k$-Dijkstra algorithm. *Message Size Received at a Client:* is the total message size a client receives. This includes real-time travel time updates on edges and header messages used in the communication protocol. *Path Recomputation Time:* is the time an algorithm takes to recompute a fastest path, when reacting to a change in travel time (new travel time update was received). This computation is done in the reduced subgraph at the client. *Client Max Memory:* is the maximum possible

<table>
<thead>
<tr>
<th>Dataset Sizes $D$ (million points)</th>
<th>4M, 8M, 12M, 16M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approx. Error $\epsilon$</td>
<td>0, 0.1, 0.2, 0.3, 0.4</td>
</tr>
<tr>
<td>Number of Concurrent Clients $N$</td>
<td>100, 500, 1K, 5K, 10K</td>
</tr>
<tr>
<td>Number of Training Months $X$</td>
<td>4, 8, 12, 16, 20</td>
</tr>
<tr>
<td>GPS Sampling Rate</td>
<td>0.25, 0.5, 0.75, 1</td>
</tr>
</tbody>
</table>

Table 6.2: Parameter Settings
6.7. EXPERIMENTS

In order to highlight the significance of our contributions, we study various effects individually to present a complete and comprehensive experimental study. In Section 6.7.2, we compare our proposed methods with existing methods. In Section 6.7.3, we study the impact of our approximation subgraphs that were theoretically outlined in Section 6.5. In Section 6.7.4, we study an abnormal event sensitive study. In Section 6.7.5, we study the effect of lowering the GPS sampling rates on our algorithms. Section 6.7.6 depicts the scalability of our methods on different dataset sizes. In Section 6.7.7, we give case studies. Finally, Section 6.7.8 summarizes the insight gained from the experiments.

6.7.2 Comparison with Existing Methods

In this section, we compare our proposed methods with existing methods by varying the number of concurrent users issuing long-running queries. In Figure 6.5(a), the total estimated travel time calculated by OurAlg-sh is the lowest (fastest) because it does not take into account any real-time traffic delays and hence severely underestimates the travel time to the destination. k-DJPath estimates a much longer travel time because on every real-time travel time delay, it forces the clients to backtrack to the original starting vertex and then follow the next-ranked path in the top-k edge-
disjoint alternate paths. Comparing the time differences between the estimated travel time and the actual travel time (Figure 6.5(b)), we find our newly proposed OurAlg-dr to be the most accurate as it has the least time difference. A-Corridor comes a close second, because it has a larger subgraph (nearly 5–6 times larger than subgraph in OurAlg-dr) but this larger subgraph is generated heuristically and has no theoretical bound on the quality of the path returned. OurAlg-dr’s estimates are calculated much more dynamically and immediately reflect any real-time traffic delays, whereas most other algorithms still rely on historical data to compute their paths. Figure 6.5(b) also conclusively shows that OurAlg-dr computes the fastest path with the least travel time. Figure 6.5(c) shows that the Corridor based methods incur a preprocessing time that is nearly 2 orders of magnitude larger than our methods. In particular, the preprocessing time of Corridor, k-DJPath and OurAlg (OurAlg-dh, OurAlg-dr and OurAlg-sh) are 260 s (4.3 mins), 12000 s (200 mins) and 3.2 s, respectively. k-DJPath has the largest preprocessing time because it has to compute the top k-fastest paths between the source and the destination, which has a higher time complexity than both the Corridor method and our method. Figure 6.5(d) shows that k-DJPath requires the least client memory as it just needs to store k alternate paths, while our methods need to store a larger subgraph containing historical information (within 100 MB) and the Corridor methods need to store an even larger subgraph, which explains why the Corridor methods consume nearly 6 times the memory needed by our methods.

6.7.3 Study on our Approximate Graphs

Here, we study the impact of our (approximate) $(1 + \epsilon)$-subgraphs on fastest path computation. Figure 6.6(a) shows a steady decline in estimated travel times when error $\epsilon$ is reduced. The error introduced in the graphs results in a loss of edges (i.e., information loss), which in turn results in longer path estimates for larger errors. We also notice a larger gap in terms of the actual travel time between the estimated path and the actual path (shown in Figure 6.6(b)) for the historical paths $(dH)$ due to their reliance on historical data, while on the other hand, the dynamic paths $(dR)$ have tighter estimates. Hence, $dR$ approach seems to be more reactive to changes in the network and recomputes faster paths accordingly. Figure 6.6(c) shows an increase in the total message size in bytes received at a client when error decreases because lower errors result in larger subgraphs with more edge, which capture more targeted real-time traffic updates. Figure 6.6(d) shows that the original graph $G$ contains 2,049, 450 vertices, which after computing $G'_0$ is reduced to 456, 450 vertices, which is approximately 22% of $G$. After running our approximation algorithm, we notice that there is a steady decline in the number of vertices. Graph $G'_{0.4}$, with the highest error, contains just 48, 000 vertices, which is 2.3% of $G$.

All experiments outlined in Section 6.7.3 were repeated on the largest dataset, i.e., JU. On both datasets, we found that for $\epsilon = 0.1$, we obtained a reduced subgraph which had travel time estimates nearly as good as on the error-free graph $G'_0$. We also noticed a substantial reduction in the number of updates and total messages in
general between a client and a server from $G'_0$ to $G'_0$. Hence, $G'_{0,1}$ was chosen as the default graph for all the remaining experiments.

### 6.7.4 Abnormal Event Sensitivity Study

In this section, we study how our proposed methods are sensitive to abnormal events. We first studied how abnormal events affects our proposed methods in real datasets. Then, we studies how abnormal events affect our methods with a simulation for traffic accidents.

#### Study with our Real-world Traffic Data

For this set of experiments, we divided the historical data collected over a period of 24 months into training data ($X$ months) and testing data ($24 - X$ months). The min/max edge traversal times were computed from the $X$ months of training data. Additionally, we defined % Edge Violations as the % of edges in the network for $24 - X$ remaining months, which exceeded their max edge traversal times recorded in training months $X$. We then varied the size of the training data to study its effect on our algorithms. Note that larger training data (i.e., $X$ is larger) captures more information about accidents and hence there are fewer accidents in the testing data (i.e., the data from the $24 - X$ months) and thus fewer recomputations of our errorless
Figure 6.7: Impact of varying training months for accidents, Dataset: NJ
subgraph \( G_0 \). When edge travel times exceed their maximum possible edge travel times, i.e. \( \tilde{f}_{max}(t,e) \), we term this as an edge violation. When we vary \( X \), we keep the same set of user queries in order to see the effect of \( X \).

In Figures 6.7(a) and (b), we notice a downward trend in estimated and actual travel time. With fewer training months, we come across more accidents that have not yet been seen in the historical data (testing data) and hence more edge violations. When edge violations are encountered, \( G_0 \) is recomputed and covers a larger spatial region. This larger spatial region allows drivers to take a larger detour from their original path and this larger spatial region also captures more edge delays. Thus, faster paths are able to be computed when there is more historical data available and we also further notice that 12 months worth of training data is sufficient to make good estimations because for more training data, i.e., \( X > 12 \), we notice very little change in actual travel times and also most other measurements, such as maximum client memory and the message size received at a client.

Figure 6.7(c) shows that the preprocessing time remains fixed with respect to change in the number of training months. The preprocessing time here is the time it takes to compute \( G_0 \), which is independent of the number of training months used. We notice exactly the same behavior as in Figure 6.5(c).

In Figure 6.7(d), for fewer training months, there is a larger use of client memory because it has more maximum edge violations, which result in the storage of a larger spatial region for \( G_0 \).

Figure 6.7(e) shows the percentage of edges in \( G_0 \) that have travel times that exceed their historical maximums. As expected, with more training data made available, there is also a drop in the number of edge violations. In 12 months training data, we notice that only 7% of the edges have violations.

Figure 6.7(e) shows the number of edge violations spread across various training months and Figure 6.7(f) shows the extent by which edges exceed their maximums. For example, the leftmost (first) bar in Figure 6.7(f) shows that 25% of the edge violations lie within the range of 0% to 15% of their original edge maximum values. We notice that a majority of the edge violations lie within the 0–100% range (first four bars).

Figure 6.7(g) shows more messages received by clients for fewer training months, due to a larger spatial region for which edge updates must be received by the clients.

Study with Traffic Accident Simulation

We wanted to gain deeper insight into the robustness of our algorithms when faced with unseen traffic accidents, that cause much longer delays than exhibited by our real-world data. We randomly selected a set of queries, for which we computed \( G_0 \) and simulated accidents by exceeding the edge maximum travel time by up to 1600%, for 5% of that total edges in \( G_0 \). We also propagated the edge delays to the adjacent edges and used an exponential decay function with a decay factor of 0.05, to simulate edge travel times returning back to normal when accidents are cleared up.
An increase in the extent by which the edge maximum travel times where exceeded, caused an increase in the actual travel times, due to the re-computation of $G_0$, which would cover a much larger spatial region, as shown in Figure 6.8(a). Figure 6.8(b) shows that there is no significant change on the preprocessing time when the percentage increase of the maximum travel time increases.

### 6.7.5 Effect of GPS Sampling Rates

We also varied the GPS sampling rates to study the effect of having lower frequency GPS updates coming in. This was simulated by dropping some GPS samples from our real-time GPS updates and estimating the fastest travel time between samples based on historical information. Hence, for lower sampling, there was greater reliance on historical GPS data instead of the real-time travel times.

The GPS sampling rate affects the travel time update computations and the rate at which updates are sent from the server to the clients. Thus, for low frequency GPS data, the clients have more stale information and are not up to date with the latest traffic updates. Thus, the paths computed by the clients with less information are longer, as shown in Figure 6.9(a). The sampling rates do not affect the preprocessing...
times, shown in Figure 6.9(b), and we see a similar behavior exhibited as was seen in Figure 6.5(c).

6.7.6 Scalability Studies

Here, we study the scalability of our algorithms and the Corridor based methods, by varying the dataset size (defined as the total number of vertices in the graph). \(k\)-\(DJPath\) was omitted from this study because it could not be completed on such large datasets.

Since, the number of default users is \(1K\) on the \(NJ\) dataset, we accordingly scale the number of users for each dataset. Thus, for each of the datasets, in increasing order of size, we have 1950, 3900, 5850 and 7800 clients/users, respectively. The distance separation between the start and end points for the queries are also increased accordingly.

In Figure 6.10(a), as expected, the paths get longer as we increase the size of the datasets. More importantly, the gap between estimated and actual travel times remain nearly the same for the \(OurAlg-sh\) and \(OurAlg-dh\) historical paths, while improves slightly with increasing dataset sizes, for \(OurAlg-dr\) real-time paths. This suggests that for longer paths, the real-time approach can estimate better because it has more opportunities to correct its fastest path, when reacting to edge weight updates, well in advance.

Figure 6.10(b) shows that the preprocessing time for computing the Corridor increases quite drastically with increasing dataset size, while our methods have a much slighter increase. This is due to the fact that Corridor based method computes the entire shortest path tree from the initial source on a very large graph, while in comparison, our method stops the bi-directional deepening method when \(d_{max}\) is reached, which is much smaller compared to the diameter of the entire graph.

In addition, we also notice that the path re-computation times on \(G_0^r\) versus \(G\) improves significantly as the dataset size increases. We note an increase of 4 orders of magnitude, specifically 11,054 times, on the largest dataset, which has 16 million vertices.
6.7.7 Case Studies

Here, we study some cases studies (i.e., the impact of Planned and Unplanned closure events of road segments). In the case of unplanned closure events, we notice that all algorithms gave nearly similar estimated travel times in absence of an event, and in the presence of an event gave very different estimates. The actual travel times also exhibit the same behavior. Figure 6.11(a) shows that when an event occurs, both OurAlg-dr and A-Corridor give actual travel times that have smaller delays as compared to the other algorithms. This is because both these algorithms react well to real-time events and manage to disperse the traffic along other routes that become available and are faster.

The preprocessing times, shown in Figure 6.11(b), show a slight increase for our algorithms because the shape and the size of $G_0'$ has a dependence on $d_{\text{max}}$, whose magnitude is in turn affected by delays caused due to closure events. The corridor based algorithms show no variance in size because the corridor remains unaffected if the path $\pi$ is the same but only the edges have more traversal delays on them, but is only affected when there are more edges in path $\pi$. Similarly, $k$-DIPPath uses the $k$-shortest path algorithm to compute the alternate paths and is also mostly unaffected.
6.8. CONCLUSION AND OUTLOOK

by delays on edges. Thus, the Corridor based algorithms and $k$-DJPath have the same preprocessing times.

In our experiments, we found a similar behavior for planned closure events, shown in Figure 6.12 but with smaller differences in actual travel times, when compared to the unplanned closure events.

6.7.8 Summary

The actual travel time of OurAlg-dr is the smallest among all algorithms. Besides, the difference between the actual travel time and the estimated travel time of OurAlg-dr is the smallest among all algorithms. In the largest dataset, the memory consumption of OurAlg-dr at the client side is at most 650 MB which is affordable to many electronic devices (e.g., the minimum memory specification of iPhone 5 (i.e., 1GB)). A baseline method, $k$-DJPath, has the greatest actual travel time in all experiments. Although the other baseline methods, Corridor and A-Corridor, have a similar actual travel time as OurAlg-dr, the memory consumption of these algorithms at the client side is very large (e.g., at least 3.85 GB in the largest dataset) and the preprocessing times of these algorithms are very large (e.g., at least 2 hours in the largest dataset). Since the preprocessing time can be regarded as the waiting time to start the system and issue the shortest query, a large preprocessing time is not affordable which means that Corridor and A-Corridor are not desirable. Besides, the path returned by $k$-DJPath and the path returned by Corridor are nearly 475% and 157% slower than the path returned by OurAlg-dr, respectively.

6.8 Conclusion and Outlook

In this paper, we are the first to propose the problem of finding fastest dynamic real-time/historical paths based on real-time traffic information. We also proposed exact algorithms and approximate algorithms for this problem. We also proposed a method of generating a smaller graph to reduce the communication cost between the client and the server. Finally, we conducted experiments to show that the dynamic paths proposed by us are better than the traditional static paths, and our proposed algorithms are effective and scalable.

There are a lot of promising future directions. The first direction is to propose a model estimating the travel time of traversing an edge by considering both the historical information and the real-time traffic. The second possible direction is to study whether we can use a MapReduce like approach for this problem when the current traffic information changes frequently over time.
Bibliography


