QUERY PROCESSING IN TIME-DEPENDENT SPATIAL NETWORKS

by

Ugur Demiryurek

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

December 2012

Copyright 2012 Ugur Demiryurek
Dedication

This thesis is dedicated to my parents and to my wife
Acknowledgments

I would like to thank my advisor, Professor Cyrus Shahabi for his endless support throughout my PhD studies at the University of Southern California. He has greatly motivated my interests in this topic and guided me thoughtfully in my pursuit of knowledge and ideas. I have been always inspired by his vision and his dedication to work. I would also like to thank my proposal and dissertation committee members Professor Gaurav S. Sukhatme, Professor Ulrich Neumann, Professor Aiichiro Nakano, and Professor Genevieve Giuliano for all their valuable advice and guidance from various perspectives. I thank Professor David Kempe for all the stimulating and invaluable conversations. My deep appreciation to Professor Micheal Arbib who has thought me the fundamentals of research and supported me in my first years at USC.

I am also thankful to all my colleagues at Infolab for their friendship and help. I very much enjoyed the time in the past several years with them: Songhua Xing, Kiyoun Yang, Hyunjin Yoon, Colin Gu, Jeff Khosravirafghi, Afsin Akdogan, Mehrdad Jahangiri, Mehdi Sharifzadeh, Leyla Kazemi, Houtan Shirani-Mehr, Ali Khodaei, Bei Pan, Ling Hu, Lian Liu, and Shireesh Asthana. I wish them all personal and professional successes in life.

As always my family have been a constant source of encouragement and support during my studies at USC. My love and gratitude to my parents Hacer and Ihsan Demiryurek who have been my inspiration and motivation for continuing my dreams, my sisters
Emine and Emel Demiryurek, and my brother Ufuk Demiryurek. I thank my dear wife Onur Demiryurek for all of the love and patience she has given through all of these years, and for standing beside me throughout the though times. I also thank my wonderful children: Emre and Arda for always making me laugh and for understanding on those weekends when I was working on this thesis instead of playing with them. I hope that one day they can look into this thesis and smile why I spent so much time in front of my computer.

This landmark in my life’s journey would not have been possible without love, inspiration, and sacrifices of my family. To them, I dedicate this thesis.
Table of Contents

Dedication ii
Acknowledgments iii
List of Tables vii
List of Figures viii
Abstract x

Chapter 1 : Introduction 1
  1.1 Motivation ........................................ 1
  1.2 Challenges ....................................... 4
  1.3 Thesis Statement ................................. 7
  1.4 Overview of Proposed Approaches ................. 7
  1.5 Dissertation Outline ............................. 11

Chapter 2 : Related Work 13
  2.1 Nearest Neighbor Search .......................... 13
    2.1.1 kNN Search in Euclidean Space ............... 13
    2.1.2 kNN Search in Spatial Networks ............... 15
  2.2 Time-dependent Shortest Path .................... 18

Chapter 3 : Problem Definition 22
  3.1 Time-dependent Spatial Network .................. 22
  3.2 Time-dependent Query Processing ................ 24

Chapter 4 : Baseline kNN Search Solutions in Time-dependent Road Networks 26
  4.1 Time-dependent kNN Search using Time-Expanded Networks ........... 27
  4.2 Time-dependent kNN Search using Network Expansion ............. 29
  4.3 Performance Evaluation ............................ 32
  4.4 Summary of Results ............................... 36
Chapter 5 : Efficient kNN Query Processing in Time-Dependent Spatial Networks 38
  5.1 Index Construction ................................................. 39
    5.1.1 Tight Index Construction .................................. 40
    5.1.2 Loose Index Construction ................................. 42
    5.1.3 Tight and Loose R-Tree ................................. 44
  5.2 Time-dependent kNN Query Processing ......................... 47
    5.2.1 Nearest Neighbor Query .................................. 48
    5.2.2 kNN Query .................................................. 49
  5.3 Time-dependent Shortest Path Computation ....................... 52
  5.4 Performance Evaluation ........................................ 55

Chapter 6 : Online Fastest Path Computation in Time-dependent Spatial Networks 61
  6.1 Feasibility of Time-dependent Path Planning .................... 65
  6.2 Time-dependent Fastest Path Computation ....................... 68
    6.2.1 Precomputation Phase .................................. 70
    6.2.2 Online B-TDFP Computation ............................... 74
  6.3 Performance Evaluation ........................................ 80

Chapter 7 : Conclusion and Future Work 87
  7.1 Conclusion ...................................................... 87
  7.2 Discussion of Future Work .................................... 88

Bibliography 92

Appendix A: Indexing Network Voronoi Diagrams 98
  A.1 Preliminaries .................................................. 102
  A.2 Indexing Network Voronoi Cells .............................. 104
    A.2.1 Network Voronoi Diagram Construction .................. 105
    A.2.2 Index Generation on Network Voronoi Diagram ............ 106
  A.3 Performance Evaluation ...................................... 113
List of Tables

4.1 Experimental parameters .................................................. 33
5.1 Experimental parameters for TD-kNN ................................. 56
6.1 Experimental Results ......................................................... 84
6.2 Lower-bound Quality ....................................................... 86
A.1 Experimental parameters .................................................. 114
# List of Figures

1.1 Real-world travel time for a weekday on a segment of I-10 in Los Angeles County ........................................... 1  
1.2 Time-dependent graph .................................................. 3  
1.3 Arrival-time cost functions ............................................. 5  
1.4 Blind network expansion ............................................... 6  
4.1 Time-dependent 1-NN search ........................................... 27  
4.2 A Time-dependent Graph $G_T(V,E)$ ............................... 28  
4.3 A Time-expanded graph ............................................... 28  
4.4 Correctness and impact of k .......................................... 34  
4.5 Response time versus distribution and network size ............... 35  
4.6 Response time versus query/object cardinality and agility ........ 36  
5.1 Tight cell construction of $P_1$ ....................................... 41  
5.2 Tight Cells .......................................................... 41  
5.3 Loose Cells .......................................................... 43  
5.4 LN R-Tree .......................................................... 45  
5.5 Second NN example .................................................. 50  
5.6 TDSP localization .................................................... 53  
5.7 Coverage ratio ....................................................... 57  
5.8 Average number of neighbors ......................................... 57  
5.9 Impact of k .......................................................... 58  
5.10 Network node access ................................................ 58  
5.11 Object cardinality .................................................... 59  
5.12 Query cardinality .................................................... 59  
5.13 Object and Query distribution ....................................... 60  
6.1 Static vs Time-dependent path planning ............................ 63  
6.2 TDFP and FP Comparison ............................................ 67  
6.3 Road network partitioning ............................................ 72  
6.4 Lower-bound distance computation .................................. 73  
6.5 Bidirectional search ................................................ 77  
6.6 TD-ALT Comparison ................................................ 82  
6.7 Speed-up Ratio Analysis ............................................. 86  
A.1 Network Voronoi Diagram ........................................... 100
| A.2 | Voronoi diagram in Euclidean space | 102  |
| A.3 | A Road network and network Voronoi diagram | 105  |
| A.4 | Network Voronoi diagram with \( P = \{p_1, ..., p_{50}\} \) in Los Angeles road network. | 106  |
| A.5 | Network Voronoi cell construction in VR-tree | 107  |
| A.6 | False-negative edges of a NVC in Los Angeles road network | 109  |
| A.7 | Minimum bounding rectangles on network Voronoi cells | 110  |
| A.8 | VQ-tree on Los Angeles road network | 111  |
| A.9 | VQ-tree | 112  |
| A.10 | Impact of object cardinality and distribution | 115  |
| A.11 | Impact of network size | 116  |
| A.12 | Response time vs object cardinality and Index reconstruction | 117  |
Abstract

Recent advances in online map services and their wide deployment in hand-held devices and car-navigation systems have led to extensive use of location-based services. The most popular class of such services are route planning and k-nearest neighbor (kNN) queries where users search for geographical points of interests (e.g., restaurants, gas stations) with corresponding travel-times to these locations. Accordingly, many recent studies focused on developing efficient techniques to answer point-to-point fastest path and k-nearest neighbor search queries in the spatial network space.

However, most of the existing approaches in spatial networks make the simplifying assumption that the cost of traveling each edge of the spatial network is constant (e.g., corresponding to the length of the edge). Whereas in real world, the actual travel cost of a network edge is time-dependent i.e., the cost of a network edge depends on the arrival-time to that edge. Unfortunately, once we consider the time-dependent edge weights in road networks, all proposed kNN and shortest path query solutions assuming constant edge weights fail. With time-dependent edge costs, the network distance between two nodes is not unique and varies based on the departure time from the source. This dynamism of the distance introduces great challenges in developing precomputation techniques to expedite spatial query processing in time-dependent spatial networks.

In this thesis, for the first time we study the problem of k-nearest neighbor search in time-dependent road networks where the weight of each edge is a function of time. We
propose series of index structures that efficiently and accurately answer the k nearest neighbor queries in time-dependent road networks and effectively handle the database updates where points of interests are added or removed. In addition, we study the problem of point-to-point shortest path computation in time-dependent spatial networks and present a technique which speeds-up the path computation using a bidirectional time-dependent A* search based on a novel heuristic function. In this thesis, the efficacy of all proposed techniques for both kNN search and point-to-point shortest path computation have been verified with extensive experiments using real data-sets including a variety of large spatial networks with real traffic data.
Chapter 1

Introduction

1.1 Motivation

With the ever growing popularity of online map services (e.g., Google-Maps) and their wide deployment in hand-held devices (e.g., iPhone) and car-navigation systems, more and more users search for geographical points of interests (e.g., restaurants) and the corresponding directions to these locations. Consequently, many recent research studies (e.g., [3, 28, 29, 36, 39, 49, 57]) focused on developing techniques to accurately and efficiently compute the distance and route between the objects in large road networks.

The majority of existing studies make the simplifying assumption that the cost of traveling each edge of the road network is constant and rely on pre-computation of distances in the network. However, the actual travel-time on road networks heavily depends on the traffic congestion on the edges and hence is a function of the time of

![Figure 1.1: Real-world travel time for a weekday on a segment of I-10 in Los Angeles County](image)

Figure 1.1: Real-world travel time for a weekday on a segment of I-10 in Los Angeles County
the day. For example, Figure 1.1 shows the measured variation of real-world travel-time for a particular segment of I-10 freeway in Los Angeles between 6AM and 8PM on a weekday. Two main observations can be made from this figure. First, the arrival-time to the segment entry determines the travel-time on that segment i.e., travel-time is time-dependent. Second, the change in travel-time is significant and continuous (not abrupt), for example from 8:30AM to 9:00AM, the travel-time of this segment changes from 30 minutes to 18 minutes (40% decrease). These observations have major computation implications: the actual optimal path from a source to a destination may vary significantly depending on the departure-time from the source, and hence, the result of spatial queries (e.g., such KNN or point-to-point fastest path) on such time-dependent network heavily depend on the time at which the query is issued. Unfortunately, once we consider the time-dependent road networks, all the techniques assuming constant edge-weights would fail to address the spatial-queries. To illustrate, we show a simple example in Figure 1.2 where a spatial network is modeled as a graph and edge travel-times are time-dependent. Consider the snapshot of the network (i.e., a static network) with edge weights correspond to travel-time values at $t=0$. With classic fastest path computation approaches that disregard travel-time time-dependency, the fastest path from $s$ to $d$ goes through $v_1,v_2,v_4$ with a cost of 13 time units. However, by the time when $v_2$ is reached (i.e., at $t=5$), the cost of edge $e(v_2,v_4)$ changes from 8 to 12 time units, and hence reaching $d$ through $v_2$ takes 17 time units instead of 13 (as it was anticipated at $t=0$). In contrast, if the time-dependency of edge travel-times are considered and hence path going through $v_1,v_3,v_4$ was taken, the total travel-cost would have been 15 units which is the actual optimal fastest path.

Although time-dependent fastest path computation is the most accurate and realistic path computation method in road networks, we observe (at the time of this thesis is being written) that the existing state-of-the art online map path planning applications (e.g.,
Google Maps, Bing Maps) and car navigation devices do not employ time-dependency in their path computations, and hence, their fastest path recommendation remains the same throughout the day regardless of the departure-time from the source (i.e., query time). While some of these applications provide alternative paths under traffic conditions (which may seem similar to time-dependent planning at first), we note that the recommended alternative paths and their corresponding travel-times still remain unique during the day, and hence no time-dependent planning. To the best of our knowledge, these applications compute $\text{top-}k$ fastest paths (i.e., $k$ alternative paths) and their corresponding travel-times with and without taking into account the traffic conditions. The travel-times which take into account the traffic conditions are simply computed by considering increased edge weights (that corresponds to traffic congestion) for each path.

Meanwhile, an increasing number of navigation companies have started releasing their time-dependent travel-time information for road networks. For example, Navteq [42], the leading provider of navigation services, offer traffic flow services that provide time-dependent travel-time (at the temporal granularity of as low as five minutes) of road network edges up to one year. The time-dependent travel-times are usually extracted
from the historical traffic data and local information like weather, school schedules, and events.

Considering the availability of time-dependent travel-time information for road networks on the one hand and the importance of time-dependency for accurate and realistic route planning on the other hand, this thesis extends existing literature on spatial query processing and planning in road networks to a new family of time-dependent query processing solutions.

1.2 Challenges

The efficient computation of point-to-point fastest path and more complex $k$ nearest neighbor search on time-dependent spatial networks is very challenging due to following reasons:

1) Input Size for Precomputation: Given the nature of online map applications, the fastest path and kNN queries require very fast response time. Towards that end one can consider several precomputation techniques in time-dependent road networks similar to those proposed for static road networks (e.g., [28, 36, 57, 68]). However, precomputation in time-dependent road networks is very challenging due to huge input size (i.e., the number of shortest paths). Specifically, unlike static road networks where the shortest path between any pair of nodes is unique, the shortest path in time-dependent networks depends on the departure time from the source, and hence is not unique. For instance, consider our previous example in Figure 1.2 where a spatial network is modeled as a time-dependent graph and edge-weights are piece-wise linear function of time. There are three possible paths from $v_1$ to $v_4$; $p_1 = \{v_1, v_2, v_4\}$, $p_2 = \{v_1, v_2, v_3, v_4\}$, and $p_3 = \{v_1, v_3, v_4\}$. Since edge weights in the network are function of time, the arrival-time (and the total travel-cost) to destination is also function of time and changes
based on the departure-time from \( v_1 \), i.e., \( f_{p_1} = f_{24}(f_{12}(t)) \), \( f_{p_2} = f_{34}(f_{23}(f_{12}(t))) \), and \( f_{p_3} = f_{34}(f_{13}(t)) \), where \( f_{p_i} \) represent the total travel-cost function of path \( p_i \). Figure 1.3 plots the total travel-time function to destination for different departure-times from \( v_1 \). The blue-line in this figure represents the lower-envelope of the cost functions for all paths. Each piece of the lower-envelope points to the shortest path for the corresponding time-interval in time domain. For instance, one should take \( p_3 \) if (s)he leaves \( v_1 \) between \( t = 3 \) and \( t = 7 \). On the other hand, \( p_1 \) would be the optimal path if the departure-time is between \( t = 7 \) and \( t = 10 \). It has been conjectured that a lower-envelope between any nodes in time-dependent spatial networks contains super-polynomial number of linear pieces (i.e., path selections) [5]. Clearly, an algorithm which precomputes the every possible path for any pair of nodes in large time-dependent networks and stores the corresponding path selections would suffer from exponential time and storage complexity.

![Figure 1.3: Arrival-time cost functions](image)

2) **Object Distribution:** In practice, it is possible to answer kNN queries in time-dependent road networks by implementing an incremental network expansion approach with which starting from \( q \) all network nodes reachable from \( q \) in every direction are visited in order of their shortest time proximity to \( q \) until all \( k \) nearest data objects are located. However, the overhead of executing network expansion is prohibitively high...
particularly in large networks with a sparse (but perhaps large) set of data objects. This is because such a blind search approach has to redundantly visit many network nodes. For example, Figure A.3 depicts a real spatial network (i.e., San Joaquin, CA) and illustrates the set of nodes that network expansion would have to visit (marked by the shaded area) to locate the first nearest data object (1-NN) for the query object \( q \). In this case, 47.2% of the entire set of network nodes must be visited to find the first nearest neighbor.

![Figure A.3](image.png)

**Figure 1.4: Blind network expansion**

3) *Lack of Efficient Index Structures*: Classic spatial index structures (e.g., Quadtree [18], R-tree [25]) have been used to expedite query processing in Euclidean spaces. These indexing schemes have also been used in static road networks (e.g., [57, 68]) to expedite the process of computing shortest paths from that of spatial query processing by decoupling the domain of the participating objects from the domain of the vertices of the spatial network. These index structures are created assuming there exists only one (unique) shortest path between any pair of the nodes in the network and can be used as long as the spatial network is unchanged. It is not feasible, if not impossible, to simply extend these index structures to time-dependent road networks where the shortest path between any pair of nodes is dynamic, i.e., changes based on the departure time from the
source. To the best of our knowledge, there is no index structure for efficient processing of spatial queries in time-dependent spatial networks.

1.3 Thesis Statement

Due to recent sensor instrumentation of road networks in major cities as well as the advances in crowd-sourcing techniques that collect large amount of traffic data from GPS enabled devices such as in-car navigation systems and smart phones, it is now becoming possible to forecast and model time-dependent traffic flows in road networks. Therefore, traffic-sensitive time-dependent spatial queries (e.g., fastest path, k nearest neighbor search, and etc.) on road networks will be a common practice in the near future. Existing spatial query processing techniques in road networks make the simplifying assumption that travel-times of the network edges are constant. However, in real-world road networks, the edge travel-times are time-dependent, where the arrival-time to an edge determines the actual travel-time. This dynamism of the distance posses great challenges in developing efficient algorithms to evaluate spatial queries in time-dependent road networks. Our goal in this thesis is to develop index structures for efficient and exact processing of k nearest neighbor and fastest path queries in time-dependent road networks.

1.4 Overview of Proposed Approaches

In this thesis, we propose solutions for two fundamental spatial queries on time-dependent road networks. First, we study the problem of time-dependent k Nearest Neighbor(TD-kNN) query which finds the k points of interest that are in the shortest network distance (in travel-time) to a given query point. Second, we propose a bidirectional time-dependent A* algorithm based on a novel heuristic function to efficiently
answer point-to-point fastest path queries. The objectives of our proposed algorithms for both kNN and fastest path queries are as follows: 1) find exact (not approximate) solutions in time-dependent road networks, 2) efficiently answer the queries to support location based service applications that require immediate response time, 3) be scalable in order to be applicable to large-scale road networks, 4) enable fast pre-computation time and low space overhead, and 5) efficiently cope with database updates where edge weights and data objects are updated.

**kNN Query Processing in Time-dependent Spatial Networks**

The kNN query searches for the k closest points of interest (e.g., gas station, restaurant) with minimum distance to a query point \(q\), where the distance is defined by a domain-specific metric such as Minkowski distance (e.g., Euclidean distance) or network distance. Unlike the existing studies on kNN queries that assume the cost of traveling each edge of the spatial network is constant (e.g., corresponding to the length of the edge), we consider the cost of a network edge is time-dependent (e.g., travel-time). As a result the network distance between any nodes in the spatial network becomes time-dependent (not unique), hence the term time-dependent spatial network. We introduce two baseline solutions [11] to kNN query processing in time-dependent spatial networks. Firstly, we use an incremental network expansion algorithm with which all network nodes reachable from \(q\) are visited in order of their time-dependent travel-time proximity until all \(k\) nearest objects are located. While this is a practical approach, it cannot scale to large networks with sparse set of data objects due to high overhead of executing network expansion. Secondly, we exploit time-expanded graphs [33] to model the time-dependent networks. With this approach, we discretize the time domain
and at each discrete time instant we use a snapshot of the network to represent the time-dependent network. Although this approach allows exploiting the existing precomputation and index structures developed for static networks, it fails to provide the correct results because the time-expanded model misses the state of the network between any two discrete time instants.

We address the disadvantages of both baseline approaches by developing a novel technique [10] that efficiently and accurately finds k nearest neighbors of a query object in time-dependent road networks. The main idea behind our approach is to employ a filter and refinement strategy by utilizing a distance ranking method, in which we use the time-independent upper-bound and lower-bound distances between data objects to filter-out the obviously wrong answers and filter-in only a small number of objects that are potential candidates when the number of required objects is larger than 1. In particular, we partition the spatial network into neighborhoods around the data objects by creating two subnetworks for each data object called Tight Cell (TC) and Loose Cell (LC). The tight cell $TC(p_i)$ is a subnetwork around data object $p_i$ in which any query object is guaranteed to have $p_i$ as its nearest neighbor in a time-dependent network. On the other hand, the loose cell $LC(p_i)$ is a subnetwork around $p_i$ outside which any point is guaranteed not to have $p_i$ as its nearest neighbor. In other words, data object $p_i$ is guaranteed not to be the nearest neighbor of $q$ if $q$ is outside of the loose cell of $p_i$. The complementary index structures built on the tight and loose cells enable us to localize the search space in time-dependent road networks by immediately finding the first nearest neighbor and then expand the search area (in a much smaller network) to find the remaining k-1 neighbors.
Online Computation of Fastest Path in Time-dependent Spatial Networks

Given a source \( s \) and destination \( d \), and a departure-time \( t_s \) from the source, the fastest path query in time-dependent spatial networks finds the path with the \textit{minimum travel-time} among all paths from \( s \) to \( d \) with the departure-time \( t_s \). The time-dependent fastest path problem was shown first by Dreyfus [16] to be polynomially solvable by a trivial modification to a label-setting (e.g., Dijkstra) algorithm where, analogous to shortest path distances, the arrival-time to the nodes is used as the labels that form the basis of the greedy algorithm. However, Dreyfus’s algorithm is far too slow for online map applications which are usually deployed on very large networks and require almost instant response times.

There are many efficient precomputation approaches that answer fastest path queries in near real-time (e.g., [58], [67]) in static road networks. However, it is infeasible to extend these approaches to time-dependent networks. This is because the input size (i.e., the number of fastest paths) increases drastically in time-dependent networks. Specifically, since the length of a \( s-d \) path changes depending on the departure-time from \( s \), the fastest path is not unique for any pair of nodes in time-dependent networks. It has been shown in [5, 20] that the number of fastest paths between any pair of nodes in time-dependent road networks can be super-polynomial. Hence, an algorithm which considers the every possible path (corresponding to every possible departure-time from the source) for any pair of nodes in large time-dependent networks would suffer from exponential time and prohibitively large storage requirements.

Given these challenges, we propose a bidirectional time-dependent fastest path algorithm [12] based on A* search [27]. There are two main challenges to employ bidirectional A* search in time-dependent networks. First, finding an admissible heuristic function (i.e., lower-bound distance) between an intermediate \( v_i \) node and the destination \( d \) is challenging as the distance between \( v_i \) and \( d \) changes based on the departure-time
from $v_i$. Second, it is not possible to implement a backward search without knowing the arrival-time at the destination. We address the former challenge by partitioning the road network to non-overlapping partitions (an off-line operation) and precompute the intra (node-to-border) and inter (border-to-border) partition distance labels with respect to Lower-bound Graph $G$ which is generated by substituting the edge travel-times in $G$ with minimum possible travel-times. We use the combination of intra and inter distance labels (found by simple table lookups) as a heuristic function in the online computation. To address the latter challenge, we run the backward search on the lower-bound graph ($G$) which enables us to identify the set of the nodes that needs to be explored by the forward search.

1.5 Dissertation Outline

The remainder of this thesis is organized as follows. In Chapter 2, we review the related work on $k$NN queries on both Euclidean and spatial network spaces as well as the time-dependent fastest path algorithms. In Chapter 3, we formally define the problem of $k$NN and fastest path queries in time-dependent spatial networks and introduce the terminology we use throughout this thesis. In Chapter 4, we introduce two baseline approaches for time-dependent $k$NN queries based on a) time-expanded networks, and b) network expansion framework that exploits time-dependent label setting algorithm [16]. We analyze the performance of these solutions on real datasets and discuss their inefficiencies and drawbacks. In Chapter 5, we addresses the disadvantages of both baseline approaches by introducing a novel algorithm that efficiently and accurately answers $k$NN queries in time-dependent road networks. We evaluate the performance of our proposed algorithms with a variety of real-world time-dependent spatial networks with large number of data and query objects. In Chapter 6, we propose an algorithm to
answer fastest path computation in time-dependent spatial networks based on a bidi-
rectional A* search with a novel heuristic function. Finally, we conclude and propose
extensions to our research.
Chapter 2

Related Work

In this section we review the previous studies on $k$NN query processing in both Euclidean space and road networks as well as the time-dependent shortest path computation.

2.1 Nearest Neighbor Search

The $k$NN query searches for the $k$ closest points of interest (e.g., gas station, restaurant) with minimum distance to query points, where the distance is defined by a domain-specific metric such as Minkowski distance (e.g., Euclidean distance) or network distance. The majority of the existing work on kNN queries in the past literature assumes Euclidean space (with Euclidean distance) as the native metric space. These techniques focus on developing spatial access methods (e.g., R-tree, Quad-tree) for efficient processing of kNN queries. However, the access methods that assume Euclidean space are not readily applicable with spatial networks, where the distance (i.e., network distance) between the objects depend on the connectivity of the network. Accordingly, many recent studies focus on developing various pre-computation based techniques to speed-up distance computation and, hence, efficiently answer kNN queries in the spatial network space. Next we review both categories in turn.

2.1.1 $k$NN Search in Euclidean Space

Given a set of $n$ data objects $P = \{p_1, p_2, ..., p_n\}$ in Euclidean space, a query point $q$ and a distance function $d$, the $k$NN query with respect to $q$ finds a subset $P' \subseteq P$
of \( k \) objects with minimum distance to \( q \), i.e., for any object \( p' \in P' \) and \( p \in P - P' \),
\[
d(q, p') \leq d(q, p).
\]
In this case, the distance function is the Minkowski metric (Euclidean metric) and the data object are usually indexed by a spatial index (e.g., R-Tree [25]).

The first nearest neighbor algorithm proposed by Roussopoulos et al. [53] exploits the R-Tree in a depth-first manner, recursively visiting the node with the minimum distance from \( q \). Zheng et al. [76] proposed a \( k \)NN algorithm where the data objects are static and the query objects are mobile. In their work, they pre-compute and store in an R-Tree the Voronoi [46] diagram of data objects. When a NN query submitted, the server computes the nearest neighbor efficiently using the Voronoi diagram as well as the validity time \( T \). While the result remain the same during the \( T \), the mobile query object requests server to re-evaluate the NN after the \( T \). This method is only efficient for 1st NN queries since computing \( k \)NN would require constructing order-\( k \) Voronoi diagrams that is very complex and requires extensive space. Furthermore, estimating the validity time assuming the maximum speed is not very realistic in real-word applications. Zhang et al. [75] improved this method by finding the validity region that the NN set remains the same. In [62], Song and Roussopoulos introduced an algorithm that computes and caches \( m \)-NNs (\( m > k \)) for a moving query object so that if the client moves out of the \( k \)NN range, the new \( k \)NNs can be computed among the cached \( m \) objects. The major drawback of this approach is to define the right value of the \( m \). In [65], Tao et al. presented an algorithm that pre-calculates the \( k \)NN of a query object along the trajectory by utilizing split points. In their work, they divide the trajectory into sub-segments by assuming the query object moves in a steady speed and computed the \( k \)NN for each sub-segment bounded by split points. Later, Tao et al. proposed time-parameterized (TP) queries [63]. With TP when the central server receives a \( k \)NN query it computes the current result set (\( R \)), validity time (\( T \)) and set of data objects that will cause a change (\( C \)) after the \( T \) and returns the client (\( R,T,C \)). Both [65] and [63] assumes that the
query objects move in a steady speed and their movement is constant during the query. Although tree based (e.g., R-tree) data structures are efficient to handle stationary spatial data, they suffer from the node reconstruction caused by location updates with mobile objects. Therefore, some researchers have focused on the simple yet efficient grid-based structures to index and query the moving objects. Kalashnikov et al. and Yu et al. introduced grid-based in-memory algorithms (referred to as Q-Index [31] and YPK-CNN [74] respectively) for efficient and scalable processing of continuous range and $k$NN queries over moving objects. With both studies query indexing technique is used to avoid constant updates to index structures. The shared execution method has been used in SINA [37] for continuous spatio-temporal range queries, and in SEA-CNN [70] for continuous spatio-temporal $k$NN queries. The goal of shared execution algorithm is to abstract numerous spatial queries as spatial joins between the set of moving objects and queries. Finally, Mouratidis et al. [38] proposed an approach termed as CPM that defines a conceptual partitioning of the space by arranging grid cells into rectangles.

All of the above approaches are based on either spatial (e.g., R-Tree) or grid index structures that are applicable to the spaces where the distance between objects is only a function of their spatial attributes (e.g., Euclidean distance). In real-world scenarios, however, the objects move on spatial networks. We review the existing work on $k$NN query processing in spatial networks next.

### 2.1.2 kNN Search in Spatial Networks

Unlike $k$NN query in Euclidean space where the distance can be obtained immediately in constant time, the distance function in road networks is the length of the shortest path between $q$ and the data objects. In this case, a road network is usually modeled as a connected planar graph. Since the distance function in road networks is the shortest
path, the majority of $k$NN query techniques in road networks are based on incremental network expansion \cite{2, 39, 49} which relies on the Dijkstra Algorithm \cite{14}.

In \cite{49}, Papadias et al. studied two different techniques, namely Incremental Network Expansion $INE$ and Euclidean Restriction $ER$ method to answer $k$NN queries in road networks. With $INE$, given a query point $q$ and connected planar graph $G$ (i.e., static road network), starting from $q$ all network nodes reachable from $q$ in every direction are visited in order of their proximity to $q$ (hence, a one-to-many search) until all $k$ nearest data objects are located. Clearly, the overhead of executing network expansion can be very high in large networks with a sparse set of data objects, because such a blind search approach has to redundantly visit many network nodes which are away from the shortest paths to the nearest data objects. On the other hand, $ER$ exploits the fact that Euclidean distance is a lower-bound for the network distance to answer $k$NN queries in static road networks. Specifically, $ER$ uses a filtering mechanism to rapidly identify a set of candidate data objects based on their Euclidean distance from $q$, which is then refined by computing their actual shortest path from $q$ to identify the exact set of nearest neighbors. Since $ER$ relies on lower-bound restriction, it yields better results when network distance and Euclidean distance between $q$ and data objects are correlated (i.e., the network distance is close to the Euclidean distance). In \cite{61}, Shahabi et al. proposed an \textit{road network embedding technique} to transform a spatial network to a constraint-free high dimensional Euclidean space to fast but approximately retrieve nearest objects by applying traditional Euclidean based algorithms. Kolahdouzan and Shahabi utilized the first degree \textit{network Voronoi diagrams} \cite{35, 36} to partition the spatial network to network Voronoi polygons ($NVP$), one for each data object. They indexed the $NVP$s with a spatial access method to reduce the problem to a point location problem in Euclidean space and minimize the on-line network distance computation by precomputing the
NVPs. Cho et al. [3] presented a system UNICONS where the main idea is to integrate the precomputed $k$ nearest neighbors into the Dijkstra algorithm. In [69], Huang et al. addressed $k$NN problem using Island approach [29] where each vertex is associated (and network distance precomputed) to all the data points that are centers of given radius $r$ (so called islands) covering the vertex. With their approach, they utilized a restricted network expansion from the query point while using the precomputed islands. Aside from their specific drawbacks, these algorithms rely on data object dependent precomputations (i.e., the network distance to the data objects are precomputed) and subdivide the spatial network based on the location of the data objects. Therefore, they assume that data objects are static and/or trajectory of query objects is known. This assumption is undesirable in applications where the query and data objects change their positions frequently.

Recently, Huang et al. [28] and Samet et al. [57] proposed two different algorithms that address the drawbacks of data object dependent precomputation. Huang et al. introduced S-GRID where they partition (using grid) the spatial network to disjoint sub-networks and precompute the shortest path for each pair of connected border points. To find the $k$ nearest neighbors, they first perform a network expansion within the sub-networks and then proceed to outer expansion between the border points by utilizing the precomputed information. Samet et al. proposed a method where they associate a label to each edge that represents all nodes to which a shortest path starts with this particular edge. They use these labels to traverse shortest path quadtrees that enables geometric pruning to find the network distance between the objects.

All these studies simplistically assume the network edge weights are constant, and hence they are invalidated with time-dependent edge weights. With our study, we make a fundamentally different assumption that the cost of the network edges are function of
time rather than constant. Our assumption yields a much more realistic scenario and versatile approach for $k$NN query processing in spatial networks.

2.2 Time-dependent Shortest Path

In the last decade, numerous efficient fastest path algorithms with precomputation methods have been proposed (see [58, 59] for an overview). However, there are limited number of studies that focus on efficient computation of time-dependent fastest path (TDFP) problem.

Cooke and Halsey [4] first studied TDFP computation where they solved the problem using Dynamic Programming in discrete time. Another discrete-time solution to TDFP problem is to use time-expanded networks [33]. In general, time-expanded network (TEN) and discrete-time approaches assume that the edge weight functions are defined over a finite discrete window of time $t \in t_0, t_1, ..., t_n$, where $t_n$ is determined by the total duration of time interval under consideration. Therefore, the problem is reduced to the problem of computing minimum-weight paths over a static static network per time window. Hence, one can apply any static fastest path algorithms to compute TDFP. Although these algorithms are easy to design and implement, they have numerous shortcomings. First, TEN models create a separate instance of network for each time instance hence yielding a substantial amount of storage overhead. Second, such approaches can only provide approximate results because the model misses the state of the network between any two discrete-time instants. Moreover, the difference between the shortest path obtained using TEN approach and the optimal shortest path is unbounded. This is because the query time can be always between any two of the intervals which are not captured by the model, and hence the error is is accumulated on each edge along the path. In [21], George and Shekhar proposed a time-aggregated graph approach where
they aggregate the travel-times of each edge over the time instants into a time series. Their model requires less space than that of the TEN and the results are still approximate with no bounds.

In [16], Dreyfus showed that TDFP problem can be solved by a generalization of Dijkstra’s method as efficiently as for static fastest path problems. However, Halpern [26] proved that the generalization of Dijkstra’s algorithm is only true for FIFO networks. If the FIFO property does not hold in a time-dependent network, then the problem is NP-Hard. In [47], Orda and Rom introduced Bellman-Ford based algorithm where they determine the path toward destination by refining the arrival-time functions on each node in the whole time interval $T$. In [32], Kanoulas et al. proposed Time-Interval All Fastest Path (allFP) approach in which they maintain a priority queue of all paths to be expanded instead of sorting the priority queue by scalar values. Therefore, they enumerate all the paths from the source to a destination node which incurs exponential running time in the worst case. In [15], Ding et al. used a variation of Dijkstra’s algorithm to solve the TDFP problem. With their TDFP algorithm, using Dijkstra like expansion, they decouple the path-selection and time-refinement (computing earliest arrival-time functions for nodes) for a given starting time interval $T$. Their algorithm is also shown to run in exponential time for special cases (see [6]). The focus of both [32] and [15] is to find the fastest path in time-dependent road networks for a given start time-interval (e.g., between 7:30AM and 8:30AM).

The ALT algorithm [22] was originally proposed to accelerate fastest path computation in static road networks. With ALT, a set of nodes called landmarks are chosen and then the shortest distances between all the nodes in the network and all the landmarks are computed and stored. ALT employs triangle inequality based on distances to the landmarks to obtain a heuristic function to be used in A* search. The time-dependent variant of this technique is studied in [8] (unidirectional) and [41] (bidirectional A*
search) where heuristic function is computed w.r.t lower-bound graph. However, the landmark selection is very difficult (relies on heuristics) and the size of the search space is severely affected by the choice of landmarks. So far no optimal strategy with respect to landmark selection and random queries has been found. Specifically, landmark selection is NP-hard [52] and ALT does not guarantee to yield the smallest search spaces with respect to fastest path computations where source and destination nodes are chosen at random. Our experiments with real-world time-dependent travel-times show that our approach proposes in Section 2 consumes much less storage as compared to ALT based approaches and yields faster response times. In two different studies, The Contraction Hierarchies (CH) and SHARC methods (also developed for static networks) were augmented to time-dependent road networks in [1] and [7], respectively. The main idea of these techniques is to remove unimportant nodes from the graph without changing the fastest path distances between the remaining (more important) nodes. However, unlike the static networks, the importance of a node can change throughout the time under consideration in time-dependent networks, hence the importance of the nodes are time varying. Considering the super-polynomial input size (as discussed in Section 1.2), and hence the super-polynomial number of important nodes with time-dependent networks, the main shortcomings of these approaches are impractical preprocessing times and extensive space consumption. For example, the precomputation time for SHARC in time-dependent road networks takes more than 11 hours for relatively small road networks (e.g. LA with 304,162 nodes) [7]. Moreover, due to the significant use of arc flags [7], SHARC does not work in a dynamic scenario: whenever an edge cost function changes, arc flags should be recomputed, even though the graph partition need not be updated. While CH also suffers from slow preprocessing times, the space consumption for CH is at least 1000 bytes per node for less varied edge-weights where the storage cost increases with real-world time-dependent edge weights. Therefore, it may not be
feasible to apply SHARC and CH to continental size road networks which can consist of more than 45 million road segments (e.g., North America road network) with possibly large varied edge-weights.
Chapter 3

Problem Definition

In this chapter, we will formally define spatial network, time-dependent travel-time, time-dependent shortest path, time-dependent $k$NN search in spatial networks, and several important concepts that we use throughout this proposal. There are various criteria to define the cost of a path in road networks. Throughout this thesis, the cost of a path is defined as its travel-time and the term shortest path is interchangeably used to denote minimum-travel-time (or fastest) paths.

3.1 Time-dependent Spatial Network

To make the discussion more general, we introduce the concept of a spatial network which is an extension of a network model. The networks are modeled as a weighted graph $G(V, E)$, where $V$ denotes the set of vertices and $E$ denotes the set of edges. A spatial network is an extension of a network such that additional spatial components (e.g., the spatial position of each vertex with respect to a reference coordinate system) are associated with the vertices and/or edges of the graph.

In this thesis, we assume a spatial network (e.g. the Los Angeles road network) containing a set of static data objects (i.e., points of interest such as restaurants, hospitals) as well as static or moving query objects searching for their destination or $k$NN. We model the spatial network as a time-dependent weighted graph (directed) where the non-negative weights are time-dependent travel-times between the nodes. We assume both data and query objects lie on the network edges and all relevant information about the
objects is maintained by a central server. As a query object moves, the central server is updated with the new location of the object. Below, we formally define our terminology.

**Definition 1** Time-dependent Graph A Time-dependent Graph \((G_T)\) is defined as \(G_T(V,E)\) where \(V = \{v_i\}\) is a set of nodes representing the intersections and terminal points, and \(E (E \subseteq V \times V)\) is a set of edges representing the network segments each connecting two nodes. Each edge \(e\) is represented by \(e(v_i,v_j)\) where \(v_i\) and \(v_j\) are starting and ending nodes, respectively, and \(v_i \neq v_j\). For every edge \(e(v_i,v_j) \in E\), there is an edge travel-time function \(c_{i,j}(t)\), where \(t\) is the time variable in time domain \(T\). An edge travel-time function \(c_{i,j}(t)\) specifies how much time it takes to travel from \(v_i\) to \(v_j\) starting at time \(t\). For example, Figure 1.2 depicts a road network modeled as a time-dependent graph \(G_T\) with four nodes and five edges.

**Definition 2** Lower-bound Graph. Given a \(G_T(V,E)\), the corresponding Lower-bound Graph \(\overline{G}(V,E)\) is a graph with the same topology (i.e, nodes and edges) as graph \(G\), where the weight of each edge \(c_{v_i,v_j}\) is fixed (not time-dependent) and is equal to the minimum possible weight \(c_{v_i,v_j}^\text{min}\) where \(\forall e(v_i,v_j) \in E, t \in T c_{v_i,v_j}^\text{min} \leq c_{v_i,v_j}(t)\).

**Definition 3** Upper-bound Graph. Given a \(G_T(V,E)\), the corresponding Upper-bound Graph \(\overline{G}(V,E)\) is a graph with the same topology as graph \(G\), where the weight of each edge \(c_{v_i,v_j}\) is fixed (not time-dependent) and is equal to the maximum possible weight \(c_{v_i,v_j}^\text{max}\) where \(\forall e(v_i,v_j) \in E, t \in T c_{v_i,v_j}^\text{max} \geq c_{v_i,v_j}(t)\).

**Definition 4** Time-dependent Travel Cost. Let \(\{s = v_1,v_2,...,v_k = d\}\) denotes a path which contains a sequence of nodes where \(e(v_i,v_{i+1}) \in E\) and \(i = 1,...,k-1\). Given a \(G_T(V,E)\), a path \((s \leadsto d)\) from source \(s\) to destination \(d\), and a departure-time at the source \(t_s\), the time-dependent travel cost \(TT(s \leadsto d,t_s)\) is the travel-time it takes to travel the path. Since the travel-time of
an edge varies depending on the arrival-time to that edge, the travel-time of a path is computed as follows:

\[ TT(s \rightsquigarrow d, t_s) = \sum_{i=1}^{k-1} c_{v_i, v_{i+1}}(t_i) \text{ where } t_1 = t_s, t_{i+1} = t_i + c(v_i, v_{i+1})(t_i), i = 1, ..., k. \]

**Definition 5 Lower-bound Travel Cost.** The lower-bound travel-time \( LTT(s \rightsquigarrow d) \) of a path is less than the actual travel-time along that path and computed with respect to \( G(V, E) \) by considering the minimum possible travel-times of all edges along the path \( s \rightsquigarrow d \), i.e.,

\[ LTT(s \rightsquigarrow d) = \sum_{i=1}^{k-1} c_{v_i, v_{i+1}}^{\min}, i = 1, ..., k, \]

**Definition 6 Upper-bound Travel Cost.** The upper-bound travel-time \( UTT(s \rightsquigarrow d) \) of a path is greater than the actual travel-time of the path and computed with respect to \( \overline{G}(V, E) \) by considering the maximum possible travel-times of all edges along the path \( s \rightsquigarrow d \), i.e.,

\[ UTT(s \rightsquigarrow d) = \sum_{i=1}^{k-1} c_{v_i, v_{i+1}}^{\max}, i = 1, ..., k. \]

Note that we do not need to consider arrival-dependency when computing \( UTT \) and \( LTT \) hence; \( t \) is not included in their definitions. Given the definitions of \( TT \), \( UTT \) and \( LTT \), the following property holds for any path in \( G_T \): 

\[ LTT(s \rightsquigarrow d) \leq TT(s \rightsquigarrow d, t_s) \leq UTT(s \rightsquigarrow d). \]

We will use this property in subsequent sections to establish some properties of our algorithm.

### 3.2 Time-dependent Query Processing

**Definition 7 Time-dependent Shortest Path** Given a time-dependent graph \( G_T \) with a source \( s \in V \) and a destination \( d \in V \), and a starting time \( t_s \in T \), the time-dependent shortest path (also referred as time-dependent fastest path in this thesis) \( TDSP(s, d, t) \)

\[ TDSP(s, d, t) = \min_{\text{paths } P} \{ TT(P) \} \]
is a path with the minimum travel-time among all paths from $s$ to $d$ with the departure-time $t_s$.

**Definition 8 Time-dependent $k$ Nearest Neighbor Query (TD-$k$NN)** A time-dependent $k$ nearest neighbor query in spatial networks is defined as a query that finds the $k$ nearest neighbors of a query object moving on a time-dependent network $G_T$. Considering a set of $n$ data objects $P = \{p_1, p_2, ..., p_n\}$, the TD-$k$NN query with respect to a query point $q$ finds a subset $P' \subseteq P$ of $k$ objects with minimum time-dependent travel-time to $q$, i.e., for any object $p' \in P'$ and $p \in P - P'$, $TDSP(q, p', t) \leq TDSP(q, p, t)$.

In the rest of this paper, we assume that the edge travel-time functions are given as positive piece-wise linear functions of time and all piece-wise functions have a finite number of pieces. This is consistent with how traffic trends are reported for a given edge in real-world road networks. We also assume that the spatial network $G_T$ satisfies the First-In-First-Out (FIFO) property [5]. This property suggests that moving objects exit from an edge in the same order they enter the edge. Finally, with our algorithm, we do not allow objects to wait at a node, because, in most real-world applications, waiting at a node is not realistic as it requires the moving object to get out of the current road (e.g., the exit freeway) and find a place to park and wait.
Chapter 4

Baseline kNN Search Solutions in Time-dependent Road Networks

In this section, we explain two different baseline algorithms to evaluate $k$ nearest neighbor queries in time-dependent spatial networks. With the first approach, we model the time-dependent road network as a time-expanded graph [33, 48] that approximates the time-dependent network with a snapshot of the network in each time interval. With the second approach, we generalize the incremental network expansion [49] method (proposed for static spatial networks) to time-dependent road networks. Our incremental network expansion method uses the time-dependent arrival times as the labels of the nodes to form the greedy search [16].

Figure 4.1 illustrates an example of time-dependent $k$ nearest neighbor search. With this example, an ambulance is looking for the nearest hospital at 8:30AM and 2:00PM on the same day on a particular road network. Note that the travel times on the edges (the number shown on the edges) change with time in Figures 4.1(a) and 4.1(b). Therefore, the queries launched by the ambulance at 2PM and 5PM return different results.
4.1 Time-dependent kNN Search using Time-Expanded Networks

Given a time-dependent graph $G_T(V, E)$, a time-expanded model discretizes the time domain $T = [t_0; t_n]$ into $n$ points of time, and constructs a static graph $G(V, E)$ by making $n$ copies of each node and each edge, respectively. Specifically, time-expanded network replicates the original network for each discrete time unit $t = 0, 1, ..., t_n$, where $t_n$ is determined by the total duration of the time interval under consideration. This model connects a node and its copy at the next instant in addition to the edges in the original network, replicated for every time instant. The weight of an edge in time-expanded network is the time difference between the time events associated with its endpoints. Therefore, a time-dependent edge costs can be interpreted as a static flow in the corresponding time-expanded network. Figure 4.3 shows four consecutive snapshots (taken every 10 minutes) and the corresponding time-expanded model of the time-dependent network in Figure 4.2. In this figure, for example, the weight (i.e., travel-time) of edge $(v_1, v_2)$ at $t = 0$ is represented by connecting the copy of node $v_1$ at $t = 0$ to the copy of node $v_2$ at $t = 20$ (see Figure 4.3(e)).
The time-expanded network approach enables time-dependent $k$ nearest neighbor problem to be solved by applying any precomputation techniques developed for static networks (e.g., [57]). However, there are two drawbacks with any solution based on
time-expanded networks. First, since the original network is replicated across time instants, the size of the network increases hence, resulting in high storage overhead and slower response time. The storage requirement for a time expanded-network is \( O(|V|T) + O(|V| + |E|T) \), where \( T \) is the total number of snapshots. Second, the difference between the shortest path obtained using time-expanded model and the optimal shortest path is very sensitive to the parameter \( n \), and is unbounded. Because, the query time (or the arrival-time to an edge) can always be between any two time points (e.g., between \( t_0 \) and \( t_1 \)), but the edge weights are only captured in either of the time points. For example, consider a shortest path query executed at \( t = 12 \) in Figure 4.3, and an error \( \epsilon \) between the optimal path and the path found using time-expanded network model. In this case, the network snapshot at \( t = 10 \) is used to compute the shortest path for \( t = 12 \) and \( \epsilon \) is accumulated on each edge along the path. Our experiments show that the error rate is especially high during rush hours (see 4.3), hence causing time-expanded models to generate inaccurate results.

### 4.2 Time-dependent kNN Search using Network Expansion

In this section, we propose an algorithm that generalizes the incremental network expansion method [49] (originally proposed for static road networks) to answer \( k \) nearest neighbor queries in time-dependent road networks. With this algorithm, starting from the query object \( q \) all network nodes reachable from \( q \) in every direction are visited in order of their proximity (i.e., time-dependent travel-time) to \( q \) until all \( k \) nearest data objects are located. We use four main data structures to enable the network expansion solution. The \textit{adjacency component} captures the network connectivity. The \textit{edge component} includes the poly-line representation of each network edge \((u, v)\), length of
the edge, and a pair of pointers to the disk pages containing the adjacency lists of its endpoints $u$, $v$. The travel-time component includes the travel-time functions of each network edge. We use hash table to associate travel-time functions to network edges. The last component is R-tree [25] that indexes the edges’ MBRs. Each leaf entry of R-tree contains a pointer to the disk page storing the corresponding edge.

With this approach, we adopt a variant of Dijkstra algorithm (proposed in [16]) to expand the network based on the time-dependent travel-time to each node around $q$ until $k$ objects are found. We outline our proposed approach in Algorithm 6. The algorithm takes three parameters as the input, i.e., query location $q$, number of desired nearest neighbors $k$, and query time $t_q$. We first execute a findEdge($q$) operation to find the edge that contains $q$ by performing a point location query on R-tree index. Let $t(v)$ denote the time taken to travel from $q$ to node $v$ along the shortest path in a time-dependent network, i.e., time to travel for $TDSP(q, v, t_q)$. Analogous to shortest path distances, for every edge $e(u, v)$, we have $t(v) \leq f_e(t(u))$ and this forms the basis of our greedy algorithm. Note that $f_e(t(u))$ is the time taken to travel from $q$ to $u$ plus the time travel from $u$ to $v$. We use a priority queue $S$ to keep track of the nodes to be examined. With $S$, we maintain a) the set explored nodes (which includes the nodes for which we have calculated the actual $t(v)$), and b) the label $l(v)$ for the nodes not in $S$, where the label $l(v)$ is our current estimate for the least time it takes to reach $v$ from $q$. As shown in Algorithm 6, we update $l(v)$ based on $\min(l(v), f_e(t(u)))$ (Line 8) only considering the edges $(u, v)$ where $u \in S$. Finally, we pick the node $w / S$ with smallest $l(\cdot)$ value and add it to the set $S$ and repeat. If the recently added node is a data object (data objects can easily be modeled as network nodes), we add that data object to nearest neighbor array NN and accordingly compute its travel time (Line 11-13). This process is repeated until the algorithm finds $k$ data objects. It is important to note that Algorithm 6 holds for FIFO networks in which greedy property is maintained.
Algorithm 1: kNN(q,k,t_q)

1: // S: set of nodes, q: query location, dt: departure-time from q
2: // tt: travel-time of the fastest path, v_i: last node added to S
3: // NN: array of current NNs
4: Initialize S = {q}, t(q) = 0, l(v) = ∞ for all v ∈ S
5: v_i = q
6: while S ≠ V do
7:  for each e(v_i, v_j) ∈ E where v_j ∉ S
8:  l(v_j) = min(l(v_j), f_e(t(v_i)))
9:  Let w ∉ S such that l(w) = min_{v_j ∉ S} l(v_j)
10:  S = S ∪ {w}, t(w) = l(w), v_i = w
11:  If v_i is a dataObject Then
12:    add v_i to NN;
13:    tt = t(v_i) - t_q, //compute travel-time to v_i
14:  End If;
15:  If NN.size() = k Then break;
16: end while
17: return NN and travel times

Lemma 1 Algorithm 6 is correct

Proof 1 The proof of correctness for Algorithm 6 follows that of the Dijkstra algorithm. Let us consider any node w, and a set S just before w is added to S. Let P_w and t_w represent the path q ⇝ w and the time it takes to travel along s ⇝ w, respectively. Note that t(w) = l(w) = min_u′∈S: (u′, w) ∈ E f_{u′,w}(t(u′)), so if u′ ∈ S is the node that attains the minimum in l(w), then P_w is obtained by adding the edge (u′, w) to the path P_u′ (which is obtained recursively). Now we show that t(w) is indeed the least time it takes to reach w along any path q ⇝ w. Consider any q ⇝ w path P. Let v be the first node on P that is not in S, and u ∈ S be the node just before v. Let P′ be the portion of the path P from q to v and t_1 and t_2 represent the times at which we reach u and v, respectively, by traveling along the path P. Let e_1 = (v, v_1), e_2, ..., e_k = (v_{k-1}, w) be the portion of the path from v to w. Then, the time it takes to reach w by traveling along P is TT(P) = f_{e_k}(f_{e_{k-1}}(...f_{e_2}(f_{e_1}(t_2))...)) ≥ t_2, because ∀e f_e(t) ≥ t. Also,
\( t_2 = f_{u,v}(t_1) \geq f_{u,v}(t(u)) \) since \( t_1 \geq t(u) \) and the \( f_e \) function is monotonic. Thus, we have \( TT(P) \geq f_{u,v}(t(u)) \geq l(v) \geq l(w) \), where the last two inequalities follow from the definition of \( l(.) \), and since we choose \( w \) to add to the set \( S \). This holds for any path \( P \), and concludes the proof. \( \square \)

### 4.3 Performance Evaluation

We conducted several experiments with different spatial networks and various parameters (see Table 1) to evaluate the performance of both TD-\( k \)NN algorithms. As our road network dataset, we use Los Angeles (LA) and San Joaquin (SJ) road networks with 304,162 and 24,123 road segments, respectively. We evaluate our proposed techniques using both syntectic and actual time-dependent travel-times gathered from real-world traffic sensor data. To generate time-dependent edge costs (travel-time) we use real-world traffic sensor dataset that we have been collecting (past 2 years) and archiving from a collection of approximately 7000 sensors located on the road network of Los Angeles. We collect speed, occupancy, volume information from these sensors and the sampling rate of the data is 1 reading/sensor/min. We spatially and temporally aggregate (average) historical sensor data based on 7 days (Monday to Sunday) of each month by assigning interpolation points for each 5 minutes. The interpolation points represent the travel-times at different times of a particular day. For example, an edge is assigned 180 travel-time attributes to represent how traffic tends to change between 6:00AM and 9:00PM for a particular date in a particular month, e.g., Monday traffic pattern in September. We assume all roads are un-congested between 9:00PM and 6:00AM, and hence consider static edge weights during this interval. However, unfortunately not every edge has a sensor in road networks. In order to generate time-dependent edge weights on SJ and for the edges that does not contain any sensor in LA, we developed a
Table 4.1: Experimental parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of objects</td>
<td>10 (K)</td>
<td>1,5,10,15,20(K)</td>
</tr>
<tr>
<td>Number of queries</td>
<td>3 (K)</td>
<td>1,2,3,4,5 (K)</td>
</tr>
<tr>
<td>Number of k</td>
<td>20</td>
<td>1,10,20,30,40,50</td>
</tr>
<tr>
<td>Object Distribution</td>
<td>Uniform</td>
<td>Uniform, Gaussian</td>
</tr>
<tr>
<td>Query Distribution</td>
<td>Uniform</td>
<td>Uniform, Gaussian</td>
</tr>
</tbody>
</table>

A traffic modeling approach that creates edge travel-time profiles [13]. Our approach uses spatial (e.g., locality, connectivity) and temporal (e.g., rush hour, weekday) characteristics to generate travel-time of network edges that does not have readily available sensor data.

We generated the parameters represented in Table 1 using a simulator prototype developed in Java. We conducted our experiments on a workstation with 2.7 GHz Pentium Core Duo processor and 12GB RAM memory.

We computed the time-expanded network model of both LA and SJ networks by discritizing the networks for each 5 minutes corresponding to our interpolation points. Similar to Algorithm 6 (denoted by TD-NE), we implemented a network expansion method to find $k$ nearest neighbors in time-expanded networks (denoted by TE). We continuously monitored each query for 50 timestamps in both of the implementations. For each set of experiments, we only vary one parameter and fix the remaining to the default values in Table 1.

Since the experimental results with both LA and SJ networks differ insignificantly, we only present the results from LA dataset.
Correctness and Impact of k

With this experiment, we compare the correctness of the two algorithms (i.e., percentage of correctly identified nearest neighbors). Figure 4.4(a) plots the correctness versus time ranging from 6 am to 6 pm, while using default settings in Table 1 for all other parameters. As shown, while TD-NE returns correct results all the time, TE’s correctness is substantially low around rush hours (i.e., 7-9 am, 4-6 pm). This is because time-dependent weights of each network segment change rapidly especially at the boundaries of the traffic peak periods, resulting the error accumulating along the path.

![Correctness versus time and Impact of k on response time](image)

(a) Correctness versus time  
(b) Impact of k on response time

Figure 4.4: Correctness and impact of k

Next, we compare the performance of the two algorithms with regard to $k$. Figure 4.4(b) shows the average query efficiency versus $k$ ranging from 1 to 50. The results indicate that TD-NE outperforms TE with all values of $k$ and the response time of both algorithms increases with the large values of $k$. Note that the slower response time of TE in this and the following experiments is due to increased size of the network because of replication. It is important to note that one can adopt a precomputation technique (proposed for static networks) to accelerate the response time of TE.
Impact of Object/Query Distribution and Network Size

With this experiment, we study the impact of object and query distribution as well as network size. Figure 5.13 shows the response time of both algorithms where the objects and queries follow either uniform or Gaussian distributions. As illustrated, TD-NE yields better performance for queries with Gaussian distribution. This is because as queries are clustered in the spatial network with Gaussian distribution, their nearest neighbor would overlap; hence, allowing TD-NE to save computation.

In addition, we measured the performance of both algorithms with respect to the network size. In order to evaluate the impact of network size, we conducted experiments with the sub-networks of LA dataset ranging from 50K to 250K segments. Figure 4.5(b) illustrates the response time of both algorithms with different network sizes. In general, with the default parameters in the Table 1, the response time increases for both algorithms as the network size increases.

Figure 4.5: Response time versus distribution and network size
Impact of Object and Query Cardinality

With this set of experiments, we compare the performance of the two algorithms by varying the cardinality of the data objects (P) from 1K to 20K while using default settings in Table 1 for all other parameters. Figure 5.11 illustrates the impact of the growing object cardinality on response time. The results indicate that the response time linearly increases with the number of data objects in both methods, where TD-NE outperforms TE for all cases. From P=1K to 5K, the response time is slower. Because, since the objects are sparsely distributed when P is small, network expansion visits more redundant network nodes causing extra processing time. Figure 5.12 shows the impact of the query cardinality (Q), ranging from 1K to 5K, on response time. As shown, TD-NE scales better with large number of Q and the performance gap between the approaches increases as Q grows.

![Diagram](image)

(a) Impact of object cardinality (b) Impact of query cardinality

Figure 4.6: Response time versus query/object cardinality and agility

4.4 Summary of Results

In this chapter, we studied the problem of $k$ nearest neighbor queries in time-dependent spatial networks. We formulated a generalized type of $k$ nearest neighbor query where
we, unlike the existing studies, assume the edge weights of the network are time varying rather than fixed. We proposed two baseline solutions by exploiting time-expanded network and network expansion framework. We evaluated and compared the efficiency of these solutions with real-world data-sets, including a variety of large spatial networks with real traffic-data. Although time-expanded network framework provides a mechanism to use existing $k$NN algorithms (and their precomputation techniques) developed for static networks, the experimental results suggest that the error rate (incorrectly identified nearest neighbors) of this approach is very high especially during traffic peak hours. On the other hand, while network expansion yields correct results at all times, the overhead of executing network expansion is very high in large networks with a sparse set of data objects. Hence, the network expansion approach is not suitable for online $k$ nearest neighbor applications. In the next chapter, we propose an algorithm that address the shortcomings of the both baseline solutions.
Chapter 5

Efficient kNN Query Processing in Time-Dependent Spatial Networks

In the previous chapter, we introduced two baseline solutions for the time-dependent $k$NN problem based on the time-expanded networks and incremental network expansion. With time-expanded graphs the time domain is discretized and at each discrete time instant a snapshot of the network is used to represent the network. Hence, the time-dependent $k$NN problem is reduced to the problem of computing the minimum-weight paths through a series of static networks. Although this approach allows for exploiting the existing algorithms and precomputations for $k$NN computation on static networks, it often fails to provide the correct results because the model misses the state of the network between any two discrete time instants. Secondly, we developed a solution based on the incremental network expansion approach where Dreyfus’s modified Dijkstra algorithm is used for time-dependent distance calculation. With this approach, starting from a query object $q$ all network nodes reachable from $q$ are visited in order of their time-dependent travel-time proximity to $q$ until all $k$ nearest objects are located (i.e., blind network expansion). However, considering the prohibitively high overhead of executing blind network expansion particularly in large networks with a sparse set of data objects, this approach is far too slow to scale for real-time $k$NN query processing.

In this chapter, we address the disadvantages of both baseline approaches by developing a novel technique that finds $k$NN of a query object in time-dependent road networks. Our solution a) efficiently and accurately answers the queries in order to support
moving object \( k \)NN search on time-dependent networks, b) is independent of density and distribution of the data objects, and c) effectively handles the database updates where nodes, links, and data objects are added or removed. Towards that end we develop two types of complementary index structures. The core idea behind these index structures is to localize the search space and minimize the costly time-dependent shortest path computation between the objects hence incurring low computation costs.

Our proposed algorithm involves two phases: an off-line time-dependent network indexing phase and an on-line query processing phase. In the off-line phase, we partition the time-dependent spatial network into \textit{Tight Cells (TC)} and \textit{Loose Cells (LC)} for each data object \( p \) and construct two complementary indexing schemes, namely \textit{Tight Network Index (TNI)} and \textit{Loose Network Index (LNI)}. In the on-line phase, we use \textit{TNI} and \textit{LNI} structures to efficiently answer \( k \) nearest neighbor queries. Specifically, with TNI, we can find the nearest objects without performing any shortest path computation. Our experiments show that in 70% of the cases the nearest neighbor can be found with this index. For those cases that the nearest objects cannot be identified by TNI, LNI allows us to filter in only a small number of objects that are potential candidates (and filter out the rest of the objects). Subsequently, we only need to perform the shortest path computation only for these candidates. In the following sections, we will first mention the construction of our spatial index structures and then describe our query processing algorithm that utilizes these index structures.

## 5.1 Index Construction

In this section, we present the construction of our tight and loose network index structures.
5.1.1 Tight Index Construction

Before we explain the tight index structure, we introduce the concept of tight cell and it’s properties. A Tight Cell $TC(p_i)$ is a sub-network around each data object $p_i$ in which any query object is guaranteed to have $p_i$ as its nearest neighbor in a time-dependent spatial network. We compute the tight cell of a data object by utilizing the parallel Dijkstra algorithm. Specifically, we expand from $p_i$ (i.e., the generator of the tight cell) assuming upper-bound travel-time (i.e., UTT), while in parallel we expand from each and every other data object assuming the lower-bound travel-time (i.e., LTT). We stop the expansions when the shortest path trees meet. We repeat the same process for each data object to compute its tight cell.

The main idea behind tight cells is that if the upper-bound travel time between the query object $q$ and a particular data object $p_i$ is less than any of the lower-bound travel time from $q$ to any other data object $p_j$, then we can conclude that $p_i$ is guaranteed to be the nearest neighbor of $q$. Figure 5.1 depicts the network expansion from the data objects during the tight cell construction for $p_1$. For the sake of clarity, we represent the tight cell of each data object with a polygon as shown in Figure 5.2. We generate the edges of the polygons by connecting the adjacent border nodes (i.e., nodes where the shortest path trees meet) of a generator to each other.

We refer the readers to [17] for the implementation details of parallel Dijkstra algorithm that we used to construct tight cells. Note that the complexity of computing the tight cells with parallel Dijkstra is asymptotically not worse than the complexity of Dijkstra’s algorithm (i.e., $O(|E| + |V| \log |V|)$).

The formal lemma and the proof of tight cells is as follows.

**Lemma 2** Let $P$ be a set of data objects $P = \{p_1, p_2, ..., p_n\}$ in $G_T$ and $TC(p_i)$ be the tight cell of a data object $p_i$. For any query point $q \in TC(p_i)$, the nearest neighbor of $q$ is $p_i$, i.e., $\forall q \in TC(p_i), \forall p_j \in P, p_j \neq p_i, TDSP(q, p_i, t) < TDSP(q, p_j, t)$.
Proof 2 We prove the lemma by contradiction. Assume that $p_i$ is not the nearest neighbor of the query object $q$. Then there exists a data object $p_j$ ($p_i \neq p_j$) which is closer to $q$; i.e., $TDSP(q, p_j, t) < TDSP(q, p_i, t)$. Let us now consider a point $b$ (where the shortest path trees of $p_i$ and $p_j$ meet) on the boundary of the tight cell $TC(p_i)$. We denote shortest upper-bound path from $p_i$ to $b$ (i.e., the shortest path among all $UTT(p_i \rightsquigarrow b)$ paths) as $D_{UTT}(p_i, b)$, and similarly, we denote shortest lower-bound path from $p_j$ to $b$ (i.e., the shortest path among all $LTT(p_j \rightsquigarrow b)$ paths) as $D_{LTT}(p_j, b)$. Then, we have $TDSP(q, p_i, t) < D_{UTT}(p_i, b) = D_{LTT}(p_j, b) < TDSP(q, p_j, t)$. This is a contradiction; hence, $TDFP(q, p_i, t) < TDFP(q, p_j, t)$.

With our proposed algorithm, we utilize TCs in the following way. If a query point $q$ is inside a specific $TC(p_i)$, one can immediately identify the generator of that tight cell (i.e., $p_i$) as the nearest neighbor for $q$. This stage can be expedited by using a spatial index structure generated on the TCs. Although TCs are constructed based on the network distance metric, each TC is actually a polygon in Euclidean space. Therefore, TCs can be indexed using spatial index structures (e.g., R-tree [25]). This way a function (e.g., contain($q$)) invoked on the spatial index structure would efficiently return the TC.
whose generator has the minimum time-dependent network distance to $q$. We formally define Tight Network Index as follows.

**Definition 9 Tight Network Index (TNI).** Let $P$ be the set of data objects $P = \{p_1, p_2, \ldots, p_n\}$, the Tight Network Index is a spatial index structure generated on tight cells of $P$, $\{TC(p_1), TC(p_2), \ldots, TC(p_n)\}$.

As illustrated in Figure 5.2, the set of tight cells often does not cover the entire network. For the cases where $q$ is located in an area which is not covered by any tight cell, we utilize the Loose Network Index (LNI) to identify the candidate nearest data objects. Next, we describe loose network index.

### 5.1.2 Loose Index Construction

The loose cell $LC(p_i)$ is a sub-network around $p_i$ outside which any point is guaranteed not to have $p_i$ as its nearest neighbor in a time-dependent spatial network. In other words, the data object $p_i$ is guaranteed not to be the nearest neighbor of $q$ if $q$ is outside of the loose cell of $p_i$. Similar to the construction process for $TC(p_i)$, we use the parallel shortest path tree expansion to construct $LC(p_i)$. However, this time, we use minimum travel-time between the nodes of the network (i.e., $LTT$) to expand from $p_i$ (i.e., the generator of the loose cell) and maximum travel-time (i.e., $UTT$) to expand from every other data object. Lemma 3 proves the property of $LC$:

**Lemma 3** Let $P$ be a set of data objects $P = \{p_1, p_2, \ldots, p_n\}$ in $G_T$ and $LC(p_i)$ be the loose cell of a data object $p_i$. If $q$ is outside of $LC(p_i)$, $p_i$ is guaranteed not to be the nearest neighbor of $q$, i.e., $\forall q \notin LC(p_i), \exists p_j \in P, p_j \neq p_i, TDSP(q, p_i, t) > TDSP(q, p_j, t)}$.

**Proof 3** We prove the lemma by contradiction. Assume that $p_i$ is the nearest neighbor of a $q$, even though the $q$ is outside of $LC(p_i)$; i.e., $TDSP(q, p_i, t) < TDSP(q, p_j, t)$. 
Suppose there exists a data object $p_j$ whose loose cell $LC(p_j)$ covers $q$ (such a data object must exist, because as we will next prove that set of loose cells cover the entire network). Let $b$ be a point on the boundary of $LC(p_i)$. Then, we have, $TDSP(q,p_j,t) < DU_{LT}(p_j,b) = DL_{LT}(p_i,b) < TDSP(q,p_i,t)$. This is a contradiction; hence, $p_i$ cannot be the nearest neighbor of $q$. □

As illustrated in Figure 5.3, loose cells, unlike TCs, collectively cover the entire network and have some overlapping regions among each other.

**Lemma 4** Loose cells collectively cover the network and they may overlap.

**Proof 4** As we mentioned, during loose cell construction, LTT is used for expansion from the generator of the loose cell. Since the parallel Dijkstra algorithm traverses every node until the priority queue is empty as described in [17], every node in the network is visited; hence, the network is covered. Since the process of expansion with LTT is repeated for each data object, in the overall process some nodes are visited more than once; hence, the overlapping areas. Therefore, loose cells cover the entire network and may have overlapping areas. Note that if the edge weights are constant, the LCs would not overlap, and TCs cells and LCs would be the same. □
Based on the properties of tight and loose cells, we know that loose cells and tight
cells have common edges (i.e., all the tight cell edges are also the edges of loose cells).
We refer to data objects that share common edges as **direct neighbors** and remark that
loose cells of the direct neighbors always overlap. For example, consider Figure 5.3
where the direct neighbors of \( p_2 \) are \( p_1, p_3, \) and \( p_6 \). This property is especially useful
for processing **k-1** neighbors (see Section 5.2.2) after finding the first nearest neighbor.
We determine the direct neighbors during the generation of the loose cells and store the
neighborhood information in a data component. Therefore, finding the neighboring cells
does not require any complex operation.

Similar to **TNI**, we can use spatial index structures to access loose cells efficiently.
We formally define the Loose Network Index (**LNI**) as follows.

**Definition 10** **Loose Network Index (LNI).** Let \( P \) be the set of data objects \( P = \{p_1, p_2, \ldots, p_n\} \), the Loose Network Index is a spatial index structure generated on loose
cells of \( P \), \( \{TC(p_1), TC(p_2), \ldots, TC(p_n)\} \).

Note that **LNI** and **TNI** are complementary index structures. In particular, if a
\( q \) cannot be located with **TNI** (i.e., \( q \) falls outside of any **TC**), then we use **LNI** to
identify the **LCs** that contain \( q \); based on Lemma 3, the generators of such **LCs** are the
only NN candidates for \( q \).

### 5.1.3 Tight and Loose R-Tree

With our approach, we adopt R-Tree [25] data structure to implement TNI and LNI,
termed TN R-tree and LN R-tree, respectively. Figure 5.4 depicts LN R-tree (TN R-tree
is a similar data structure without extra pointers at the leaf nodes, hence not discussed).
As shown, LN R-tree has the basic structure of an R-tree generated on minimum bounding
rectangles of loose cells. The difference is that we modify R-tree by linking its
leaf nodes to the the pointers of additional components that facilitate TD-kNN query processing. These components are the direct neighbors \((N(p_i))\) of \(p_i\) and the list of nodes \((VL_{p_i})\) that are inside \(LC(p_i)\). While \(N(p_i)\) is used to filter the set of candidate nearest neighbors where \(k > 1\), we use \(VL_{p_i}\) to prune the search space during TDSP computation (see Section 5.3).

\[
\begin{array}{c}
R \\
\downarrow \\
\ldots \\
\downarrow \\
\ldots \\
\downarrow \\
\ldots \quad MBR(LC(p_j)) \quad \ldots \\
\downarrow \\
\ldots \quad VL(p_2) = \{\ldots\} \\
\quad N(p_2) = \{p_1, p_3, p_6\} \\
\downarrow \\
\ldots \quad MBR(LC(p_j)) \quad \ldots \\
\downarrow \\
\ldots \quad VL(p_5) = \{\ldots\} \\
\quad N(p_5) = \{p_1, p_6\} \\
\end{array}
\]

Figure 5.4: LN R-Tree

Our proposed index structures need to be updated when the set of data objects and/or the travel-time profiles change. Fortunately, due to local precomputation nature of TD-kNN, the affect of the updates with both cases are *local*, hence requiring minimal change in tight and loose cell index structures. Below, we explain each update type.

*Data Object Updates:* We consider two types of object update; insertion and deletion (object relocation is performed by a deletion following by insertion at the new location). With a location update of a data object \(p_i\), only the tight and loose cells of \(p_i\)’s neighbors are updated locally. In particular, when a new \(p_i\) is inserted, first we find the loose cell(s) \(LC(p_j)\) containing \(p_i\). Clearly, we need to shrink \(LC(p_j)\) and since the loose cells and tight cells share common edges, the region that contains \(LC(p_j)\) and \(LC(p_j)\)’s direct neighbors needs to be adjusted. Towards that end, we find the neighbors of \(LC(p_j)\); the tight and loose cells of these direct neighbors are the only ones affected by the
insertion. Finally, we compute the new TCs and LCs for \( p_i \), \( p_j \) and \( p_j \)’s direct neighbors by updating our index structures. Deletion of a \( p_i \) is similar and hence not discussed.

**Edge Travel-time Updates:** With travel-time updates, we do not need to update our index structures. This is because the tight and loose cells are generated based on the minimum (LTT) and maximum (UTT) travel-times of the edges in the network that are time-independent. The only case we need to update our index structures is when minimum and/or maximum travel-time of an edge changes, which is not that frequent. Moreover, similar to the data object updates, the affect of the travel-time profile update is local. When the maximum and/or minimum travel-time of an edge \( e_i \) changes in the network, we first find the loose cell(s) \( LC(p_j) \) that overlaps with \( e_i \) and thereafter recompute the tight and loose cells of \( LC(p_j) \) and its direct neighbors.

As mentioned, given a query point \( q \), tight and loose cells in a time-dependent road network, the first step in answering a kNN query is to locate the network tight or loose cell that contains \( q \). Considering the large size of the underlying space (e.g., a continental size road network) with numerous data objects as well as the online nature of the queries that requires fast response-time, an index structure is necessary to efficiently access the portion of the network associated with \( q \). Without loss of generality, we used R-tree index structures to index tight and loose cells. We note that although tight and loose cells are created based on the network distance, they are treated as regular polygons (by connecting border points) to be able to use R-tree. This approach in some cases may cause misclassification of the network edges (i.e., false-negative edges) in the cells due to network topology. For example, a network edge not belonging to the tight cell of \( p_i \) may be classified as a member of \( TC(p_i) \) (see an example of such misclassification in Appendix A). One solution to avoid inaccurate results due to false-negative edges is to perform a refinement step. In particular, one can maintain false-negative edges (along with their corresponding tight or loose cell generators) in a separate data
structure. This structure can be checked before each index scan of TNI and LNI. If \( q \) is located in any of the false-negative edges, the corresponding tight (or loose) cell generator is returned as the result. Otherwise, the search is continued based on TNI and LNI as explained above. It is possible to overcome extra refinement step and hence improve the performance of kNN search by using quad-tree index on tight and loose cells. In Appendix A, we discuss a quad-tree index structure to index network Voronoi diagram (NVD) which partitions the road network into Voronoi cells similar to tight and loose cells. This approach can easily be extend to index tight and loose cells.

5.2 Time-dependent kNN Query Processing

So far, we have defined the properties of TNI and LNI. We now explain how we use these index structures to process kNN queries in time-dependent spatial networks. As we discussed, the tight cells do not cover the entire area thus causing unclassified areas in between them. On the other hand, loose cells cover the entire area but they overlap with each other. We proved that if the query object \( q \) is inside one of the tight cells, the generator of that tight cell is the first nearest neighbor of \( q \). If \( q \) falls in to an unclassified area between the tight cells, then we investigate the loose cells that contain \( q \), the generator of these loose cells are the only candidates for the first nearest neighbor. Hence, we will only compute the (point-to-point) time-dependent shortest path distance between \( q \) and those candidates. Clearly, this is much more efficient as compared to the naive way of computing time-dependent network distance from \( q \) to all the sites. After finding the first nearest neighbor we need to find the \( k-1 \) nearest neighbors. In order to find the \( k-1 \) nearest neighbor efficiently, we use the direct neighbor property. Below, we first describe our algorithm to find the nearest neighbor (i.e., \( k=1 \)), and then we extend it to address the kNN case (i.e., \( k \geq 1 \)).
5.2.1 Nearest Neighbor Query

Our proposed approach to determine the nearest neighbor of a query object is based on $TNI$ and $LNI$ discussed above. We present the algorithm to process the nearest neighbor query in Algorithm 2. Given the location of a query object $q$, first we carry out a depth-first search from the $TNI$ root to the node that contains $q$ (Line 5 of Algorithm 2). If a tight cell that contains $q$ is located, we return the generator of that tight cell as the first nearest neighbor. Our experiments show that, in most cases (7 out of 10), we can find $q$ with $TNI$ search (see Section 5.4). If we cannot locate $q$ in $TNI$ (i.e., when $q$ falls outside all tight cells), we proceed to search $LNI$ (Line 7). At this step, we may find one or more loose cells that contain $q$. Based on Lemma 3, the generators of these loose cells are the only possible candidates to be the NN for $q$. Therefore, we compute time-dependent shortest path (TDSP) to find the distance between $q$ and each candidate in order to determine the first NN (Line 8-12). We store the candidates in a minimum heap based on their travel-time to $q$ (Line 10) and retrieve the nearest neighbor from the heap (Line 12).

Algorithm 2: \texttt{NN-Algorithm}(q,t;TNI,LNI)

\begin{algorithmic}
\State // $q$: location of the query object, $t$: query time
\State // $S$: an array containing the candidate set,
\State // $H$: a minimum heap, $p$: the first NN
\State Initialize $S$ and $H$;
\State $p \leftarrow \textit{contain}_{TNI}(q)$;
\If {$p$ is null}
\State $S \leftarrow \textit{contain}_{LNI}(q)$;
\For {each data object $s_i$ in $S$}
\State $\textit{computeTDSP}(q,s_i,t)$;
\State insert $s_i$ to $H$;
\EndFor
\State $p \leftarrow \textit{deHeap} H$;
\EndIf
\State return $p$;
\end{algorithmic}
The time complexity of Algorithm 2 is as follows. The major time consuming steps are traversing TNI or (if necessary) LNI, and point-to-point TDSP computation. Therefore, the complexity of the index search is \( O(2\log(P)) \) in the worst case where \( P \) is the total number of data objects. The complexity of the point-to-point TDFP is same as in Dijkstra (see Section 5.3). Thus, the total time complexity is \( O(\log(P) + S(E + V\log V)) \) where \( S \) is the number of overlapping cells.

5.2.2 \( k \)-NN Query

Our proposed algorithm for finding the remaining \( k-1 \) nearest neighbor is based on the direct neighbor property discussed above. We argue that the second nearest neighbor must be among the direct neighbors of the first NN. Once we identify the second NN, we continue by including the neighbors of the second NN to find the third NN and so on. This search algorithm is based on the following Lemma which is derived from the properties of TNI and LNI.

**Lemma 5** The \( i \)-th nearest neighbor of \( q \) is always among the neighbors of the \( i-1 \) nearest neighbors of \( q \).

**Proof 5** We prove this lemma by induction. We prove the base case (i.e., the second NN is a direct neighbor of the first NN of \( q \)) by contradiction. Consider Figure 5.5 where \( p_2 \) is the first NN of \( q \). Let's assume that \( p_5 \) (which is not a direct neighbor of \( p_2 \)) is the second NN of \( q \). Since \( p_2 \) and \( p_5 \) are not direct neighbors, a point \( w \) on the time-dependent shortest path between \( q \) and \( p_5 \) can be found that is outside both \( LC(p_2) \) and \( LC(p_5) \). However, due to Property 2, \( p_5 \) cannot be a candidate NN for \( w \), because \( w \) is not in \( LC(p_5) \). Thus, there exists another object such as \( p_1 \) for instance which is closer to \( w \) as compared to \( p_5 \). Therefore, \( TDSP(w, p_5, t) > TDSP(w, p_1, t) \). However, as shown in Figure 5.5, we have \( TDSP(q, p_5, t) = TDSP(q, w, t) + TDSP(w, p_5, t) > \)}
\[ TDFP(q, w, t) + TDFP(w, p_1, t) = TDFP(q, p_1, t). \] Thus, \( p_5 \) is farther from \( q \) than both \( p_2 \) and \( p_1 \), which contradicts the assumption that \( p_5 \) is the second NN of \( q \).

Let us now prove the inductive step. The proof of inductive step is straightforward and similar to the above proof by contradiction. Suppose the inductive hypothesis holds for \( k-1 \), we prove that it also holds for \( k \). Let \( S = \{p_1, p_2, \ldots, p_{k-1}\} \) be the \( k-1 \) nearest neighbors of the query object \( q \), we prove that the \( k \)-th neighbor \( p_k \) is among the neighbor cells of \( S \). Consider a \( k \)-th nearest neighbor point \( p_k \) which is not a neighbor of \( S \). Then, there exists a point \( w \) on the time-dependent shortest path from \( q \) to \( p_k \) where \( w \) does not belong to any of the \( LC(p_k) \) or \( LC(p_1), \ldots, LC(p_{k-1}) \). Thus, \( p_k \) is not the nearest neighbor of \( w \). Suppose the nearest neighbor of \( w \) is \( p_i \) where \( p_i \neq p_1, p_2, \ldots, p_k \). Hence \( TDFP(q, p_k) = TDFP(q, w) + TDFP(w, p_k) \geq TDFP(q, w) + TDFP(w, p_i) = TDFP(q, p_i) \). Therefore, since \( p_k \) is farther from \( q \) than \( p_1, p_2, \ldots, p_{k-1} \) and \( p_i \) which contradicts the assumption that \( p_k \) is the \( k \)-th nearest neighbor. \( \square \)

![Figure 5.5: Second NN example](image)

The complete TD-\( k \)NN query process is given in Algorithm 3. Algorithm 3 calls Algorithm 2 to find the first NN and add it to an array \( N \), which maintains the current set of nearest neighbors (Lines 4-5). To find the remaining \( k - 1 \) NNs, we expand
the search area by including the direct neighbors of the first NN. Specifically, we add all direct neighbors of the current NN set to a candidate set to find the next NN. We compute the TDSP for each candidate by inserting them to a minimum heap (Lines 7-9) based on its time-dependent travel-time to \( q \). Thereafter, we select the one with minimum distance as the second NN (Line 11). Once we identify the second NN, we continue by investigating the neighbor loose cells of the second NN to find the third NN and so on. Our experiments show that the average number of neighbors for a data object is a relatively small number less than 9 (see Section 5.4). Note that, with Algorithm 3, we use a min heap data structure to store the candidate nearest neighbors. This allows us to report the results incrementally even without a pre-specified value of \( k \).

**Algorithm 3:** \( k \)NN-Algorithm\( (q, k, t, TNI, LNI) \)

1: // \( q \): location of the query object, \( k \): number of NN
2: // \( t \): query time, \( N \): an array of NN set, \( H \): Min Heap
3: Initialize \( H, N \)
4: \( p \leftarrow \text{NN-Algorithm}(q, t, TNI, LNI) \);
5: add \( p \) to \( N \)
6: while \( N.\text{size} \leq k \) do
7: for each direct neighbor \( p_i \) of \( N \) do
8: \( \text{computeTDSP}(q, p_i, t) \)
9: add \( p_i \) to \( H \)
10: end for
11: \( p \leftarrow \text{deheap} H \); //find next NN
12: add \( p \) to \( N \)
13: end while
14: Return \( N \)

The time complexity of Algorithm 3 is as follows. The major time consuming step of Algorithm 3 is the point-to-point TDSP computation (Line 7-9) to find \( k-1 \) neighbors among the direct neighbors of the current kNN set. Let \( C \) be the average number of neighbors for each site. Then, in the worst case, there are \( k \times C \) candidates. Since the complexity of the point-to-point TDSP is same as in Dijkstra (see Section 5.3), the total complexity is \( O(kCE + kCV\log V) \) plus the complexity of the Algorithm 3. Our
experiments with real-world datasets show that $C$ is a relatively small number less than 9 (see Section 5.4).

5.3 Time-dependent Shortest Path Computation

As we explained, once the nearest neighbor of $q$ is found and the candidate set for the second NN is determined based on the direct neighbors, we need to compute time-dependent shortest path from $q$ to all candidates in order to find the second NN (and so on). Before we explain our TDSP computation, we note a very useful property of loose cells. That is, given $p_i$ is the nearest neighbor of $q$, the time-dependent shortest path from $q$ to $p_i$ is guaranteed to be in $LC(p_i)$ (see Lemma 6). This property indicates that we only need to consider the edges contained in the loose cell of $p_i$ when computing TDSP from $q$ to $p_i$. Obviously, this property allows us to localize the time-dependent shortest path search by extensively pruning the search space. Since the localized area of a loose cell is substantially smaller as compared to the complete graph, the computation cost of TDSP is significantly reduced. Note that the subnetwork bounded by a loose cell is on average $1/n$ of the original network where $n$ is the total number of sites.

**Lemma 6** If $p_i$ is the nearest neighbor of $q$, then the time-dependent shortest path from $q$ to $p_i$ is guaranteed to be inside the loose cell of $p_i$.

**Proof 6** We prove by the lemma contradiction. Assume that $p_i$ is the NN of $q$ but a portion of TDFP from $q$ to $p_i$ passes outside of $LC(p_i)$. Suppose a point $l$ on that outside portion of the path. Since $l$ is outside $LC(p_i)$, then $\exists p_j \in P, p_j \neq p_i$ that...
satisfies $D_{LTT}(p_i, l) > D_{UTT}(p_j, l)$ and hence $TDFP(p_i, l, t) > TDFP(p_j, l, t)$. Then, $\begin{align*}
TDFP(p_i, q, t) &= TDFP(p_i, l, t) + TDFP(l, q, t) > TDFP(p_j, l, t) + TDFP(l, q, t) = TDFP(p_j, q, t),
\end{align*}$
which contradicts the fact that $p_i$ is the NN of $q$. □

We note that for TD-kNN with $k > 1$, the TDFP from $q$ to the $k$th nearest neighbor will lie in the combined area of neighboring loose cells. Figure 5.6 shows an example query with $k > 1$ where $p_2$ is assumed to be the nearest neighbor (and the candidate neighbors of $p_2$ are, $p_1$, $p_6$ and $p_3$). To compute the TDFP from $q$ to data object $p_1$, we only need to consider the edges contained in $LC(p_1) \cup LC(p_2)$. Below, we explain how we compute the TDSP from $q$ to each candidate.

As initially showed by Dreyfus [16], the TDSP problem in FIFO networks can be solved by modifying any label-setting or label-correcting static shortest path algorithm. The asymptotic running times of these modified algorithms are same as those of their static counterparts. With our approach, we implement a time-dependent A* search (a
label-setting algorithm) to compute TDFP between $q$ and the candidate set. The main idea with A* algorithm is to employ a heuristic function $h(v)$ (i.e., lower-bound estimator between the intermediate node $v_i$ and the target $t$) that directs the search towards the target and significantly reduces the number of nodes that have to be traversed. With static road networks where the length of an edge is considered as the cost, the Euclidean distance between $v_i$ and $t$ is the lower-bound estimator. However, with time-dependent road networks, we need to come up with an estimator that never overestimates the travel-time between $v_i$ and $t$ for all possible departure-times (from $v_i$). One simple lower-bound is $d_{euc}(v_i, t)/\max(speed)$, i.e., the Euclidean distance between $v_i$ and $t$ divided by the maximum speed among the edges in the entire network. Although this estimator is guaranteed to be a lower-bound between $v_i$ and $t$, it is a very loose bound, hence yields insignificant pruning. Fortunately, our approach can use Lemma 6 to obtain a much tighter lower-bound. Since the shortest path from $q$ to $p_i$ is guaranteed to be inside $LC(p_i)$, we can use the maximum speed in $LC(p_i)$ to compute the lower-bound.

In Chapter 6, we extend our time-dependent A* algorithm to a more generic framework to answer point-to-point fastest path queries in time-dependent road networks.

We outline our time-dependent A* algorithm in Algorithm 4 where essential modifications (as compared to [16]) are in Lines 3, 10 and 14. As mentioned, to compute TDSP from $q$ to candidate $p_i$, we only consider the nodes in the loose cell that contains $q$ and $LC(p_i)$ (Line 3). To compute the labels for each node, we use arrival time and the estimator to each node, i.e., $cost(v_i) + h_{LC}(v_i)$ where $h_{LC}(v_i)$ is the lower-bound estimator calculated based on the maximum speed in the loose cell (Line 10). In Lines 10 and 14, $TT(v_i, v_j, t_{v_i})$ finds the time-dependent travel-time from $v_i$ to $v_j$ as described in Section 3.
Algorithm 4: TDSP\( (q,d,t) \)

1: // \( q \) : source, \( d \) : target, \( t_v \) : departure-time from node \( v \),
2: // \( \text{cost}(v) \) : cost from \( s \) to \( v \), \( \text{pre}(v) \) : previous node in optimal path
3: \( Q \leftarrow \) set of nodes in \( LC(q) \) and \( LC(d) \)
4: \( \forall v \in Q \text{cost}(v) = \infty, \text{cost}(q) = 0 \)
5: \textbf{while} \( Q \) is not empty \textbf{do}
6: \( v_i \leftarrow \) node in \( Q \) with smallest cost
7: \( \text{remove} \ v_i \ \text{from} \ Q \)
8: \( \text{IF} \ v_i = d \ \text{THEN} \) return path
9: \( \text{for each neighbor} \ v_j \ of \ v_i \)
10: \( l(v_j) = \text{cost}(v_i) + h_{LC}(v_i) + TT(v_i, v_j, t_{v_i}) \)
11: \( \text{IF} \ l(v_j) < \text{cost}(v_j) \ \text{THEN} \)
12: \( \text{cost}(v_j) = l(v_j) \)
13: \( \text{pre}(v_j) = v_i \)
14: \( t_{v_j} = d_t(v_i) + TT(v_i, v_j, t_{v_i}) \)
15: \textbf{end while}

5.4 Performance Evaluation

We conducted several experiments with different spatial networks and various parameters (see Table 1) to evaluate the performance of both TD-\( k \)-NN algorithms. As our road network dataset, we use Los Angeles (LA) and San Joaquin (SJ) road networks with 304,162 and 24,123 road segments, respectively. We evaluate our proposed techniques using both syntectic and actual time-dependent travel-times gathered from real-world traffic sensor data. To generate time-dependent edge costs (travel-time) we use real-world traffic sensor dataset that we have been collecting (past 2 years) and archiving from a collection of approximately 7000 sensors located on the road network of Los Angeles. We collect speed, occupancy, volume information from these sensors and the sampling rate of the data is 1 reading/sensor/min. We spatially and temporally aggregate (average) historical sensor data based on 7 days (Monday to Sunday) of each month by assigning interpolation points for each 5 minutes. The interpolation points represent the travel-times at different times of a particular day. For example, an edge is assigned 180
Table 5.1: Experimental parameters for TD-kNN

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of objects</td>
<td>10 (K)</td>
<td>1,5,10,15,20(K)</td>
</tr>
<tr>
<td>Number of queries</td>
<td>3 (K)</td>
<td>1,2,3,4,5 (K)</td>
</tr>
<tr>
<td>Number of k</td>
<td>20</td>
<td>1,10,20,30,40,50</td>
</tr>
<tr>
<td>Object Distribution</td>
<td>Uniform</td>
<td>Uniform, Gaussian</td>
</tr>
<tr>
<td>Query Distribution</td>
<td>Uniform</td>
<td>Uniform, Gaussian</td>
</tr>
</tbody>
</table>

cost attributes to represent how traffic tends to change between 6:00AM and 9:00PM for a particular date in a particular month, e.g., Monday traffic pattern in September. We assume all roads are un-congested between 9:00PM and 6:00AM, and hence consider static edge weights during this interval. However, unfortunately not every edge has a sensor in road networks. In order to generate time-dependent edge weights on SJ and for the edges that does not contain any sensor in LA, we developed a traffic modeling approach that creates edge travel-time profiles [13]. Our approach uses spatial (e.g., locality, connectivity) and temporal (e.g., rush hour, weekday) characteristics to generate travel-time of network edges that does not have readily available sensor data.

We run our experiments on a workstation with 2.7 GHz Pentium Duo Processor and 12GB RAM memory by monitoring each query for 100 timestamps where we we pick query location and query time uniformly at random. For each set of experiments, we only vary one parameter and fix the remaining to the default values in Table 5.1. With our experiments, we measured the tight cell hit ratio and the impact of $k$, data and query object cardinality as well as the distribution.

**Impact of Tight Cell Hit Ratio and Direct Neighbors**

As we explained, if a $q$ is located in a certain tight cell $TC(p_i)$, our algorithm immediately reports $p_i$ as the first NN. Therefore, it is essential to assess the coverage area of
the tight cells over the entire network. Figure 5.7 illustrates the coverage ratio of the tight cells with varying data object cardinality (ranging from 1K to 20K) on two data sets. As shown, the average tight cell coverage is about 68% of the entire network for both LA and SJ. This implies that the first NN of a query can be answered immediately with a ratio of 7/10 with no further computation. Another important parameter affecting the TD-$k$NN algorithm is the average number of direct neighbors for each data object. Figure 5.8 depicts the average number of neighbor cells with varying data object cardinality. As shown, the average number of neighbors is less than 9 for both LA and SJ.

As mentioned in Chapter 4, we developed an incremental network expansion algorithm (based on [16]) to evaluate $k$NN queries in time-dependent networks. Below we compare our results with this baseline approach. For the rest of the experiments, since the experimental results with both LA and SJ networks differ insignificantly, we only present the results from LA dataset.
Impact of $k$

In this experiment, we compare the performance of both algorithms by varying the value of $k$. Figure 5.9 plots the average response time versus $k$ ranging from 1 to 50 while using default settings in Table 5.1 for other parameters. The results show that TD-$k$NN outperforms naive approach for all values of $k$ and scales better with the large values of $k$. As illustrated, when $k=1$, TD-$k$NN generates the result set almost instantly. This is because a simple `contain()` function is enough to find the first NN. As the value of $k$ increases, the response time of TD-$k$NN increases at linear rate. Because, TD-$k$NN, rather than expanding the search blindly, benefits from localized computation. In addition, we compared the average number of network node access with both algorithms. As shown in Figure 5.10, the number of nodes accessed by TD-$k$NN is less than the naive approach for all values of $k$.

![Figure 5.9: Impact of $k$](image1)

![Figure 5.10: Network node access](image2)

Impact of Object and Query Cardinality

Next, we compare the algorithms with respect to cardinality of the data objects ($P$). Figure 5.11 shows the impact of $P$ on response time. The response time linearly increases with the number of data objects in both methods where TD-$k$NN outperforms the naive
approach for all cases. From P=1K to 5K, the performance gap is more significant. This is because, for lower densities where data objects are possibly distributed sparsely, naive approach requires larger portion of the network to be retrieved. Figure 5.12 shows the impact of the query cardinality (Q) ranging from 1K to 5K on response time. As shown, TD-kNN scales better with larger Q and the performance gap between the approaches increases as Q grows.

![Figure 5.11: Object cardinality](image1)

![Figure 5.12: Query cardinality](image2)

**Impact of Object/Query Distribution**

Finally, we study the impact of object, query distribution. Figure 5.13 shows the response time of both algorithms where the objects and queries follow either uniform or Gaussian distributions. TD-kNN outperforms the naive approach significantly in all cases. TD-kNN yields better performance for queries with Gaussian distribution. This is because as queries with Gaussian distribution are clustered in the network, their nearest neighbors would overlap hence allowing TD-kNN to reuse the path computations.
Figure 5.13: Object and Query distribution
Chapter 6

Online Fastest Path Computation in Time-dependent Spatial Networks

The fastest path problem in time-dependent road networks was first shown by Dreyfus [16] to be polynomially solvable in FIFO networks by a trivial modification to Dijkstra algorithm where, analogous to shortest path distances, the arrival-time to the nodes is used as the labels that form the basis of the greedy algorithm. The FIFO property, which typically holds for many networks including road networks, suggests that moving objects exit from an edge in the same order they entered the edge \(^1\). However, the modified Dijkstra algorithm [16] is far too slow for online map applications which are usually deployed on very large networks and require almost instant response times. On the other hand, there are many efficient precomputation approaches that answer fastest path queries in near real-time (e.g., [58]) in static road networks. However, it is infeasible to extend these approaches to time-dependent networks. This is because the input size (i.e., the number of fastest paths) increases drastically in time-dependent networks. Specifically, since the length of a \(s-d\) path changes depending on the departure-time from \(s\), the fastest path is not unique for any pair of nodes in time-dependent networks. It has been conjectured in [5] and settled in [20] that the number of fastest paths between any pair of nodes in time-dependent road networks can be super-polynomial. Hence, an algorithm which considers the every possible path (corresponding to every possible departure-time

\(^1\)The fastest path computation is shown to be NP-hard in non-FIFO networks where waiting at nodes is not allowed [47]. Violation of the FIFO property rarely happens in real-world and hence is not the focus of this study.
from the source) for any pair of nodes in large time-dependent networks would suffer from exponential time and prohibitively large storage requirements. For example, the time-dependent extension of Contraction Hierarchies (CH) [1] and SHARC [7] speed-up techniques (which are proved to be very efficient for static networks) suffer from the impractical precomputation times and intolerable storage complexity.

Although time-dependent fastest path computation is the most accurate and realistic path computation method in road networks, we observe (at the time of this thesis is being written) that most of the existing state of the art path planning applications (e.g., Google Maps, Bing Maps) do not employ time-dependency in their path computations, and hence, their fastest path recommendation remains the same throughout the day regardless of the departure-time from the source (i.e., query time). While some of these applications provide alternative paths under traffic conditions (which may seem similar to time-dependent planning at first), we note that the recommended alternative paths and their corresponding travel-times still remain unique during the day, and hence no time-dependent planning. To the best of our knowledge, these applications compute $\text{top-k}$ fastest paths (i.e., $k$ alternative paths) and their corresponding travel-times with and without taking into account the traffic conditions. The travel-times which take into account the traffic conditions are simply computed by considering increased edge weights (that corresponds to traffic congestion) for each path. However, our time-dependent path planning results in different optimum paths for different departure-times from the source. For example, consider Figure 6.1(a) where Google Maps offer two alternative paths (and their travel-times under no-traffic and traffic conditions) for an origin and destination pair in Los Angeles road network. Note that the path recommendation and the travel-times remain the same regardless of when the user submits the query. On the other hand, Figure 6.1(b) depicts the time-dependent path recommendations (in different colors for different departure times) for the same origin and
destination pair where we computed the time-dependent fastest paths for 38 consecutive departure-times between 8AM and 5:30PM, spaced 15 minutes apart. As shown, the optimal paths change frequently during the course of the day.

![Static and Time-dependent path planning](image)

(a) Static path planning  (b) Time-dependent path planning

Figure 6.1: Static vs Time-dependent path planning

One may argue against the feasibility of time-dependent path planning algorithms due to a) unavailability of the time-dependent edge travel-times, or b) negligible gain of time-dependent path planning (i.e., how much time-dependent planning can improve the travel-time) over static path planning. To address the first argument, note that recent advances in sensor networks enabled instrumentation of road networks in major cities for collecting real-time traffic data, and hence it is now feasible to accurately model the time-dependent travel-times based on the vast amounts of historical data. For instance, at our research center, we maintain a very large traffic sensor dataset of Los Angeles County that we have been collecting and archiving the data for past two years (see Section 4.3 for the details of this dataset). As another example, PeMS [50] project developed by UC Berkeley generates time-varying edge travel-times using historical traffic data.

---

The paths are computed using the algorithm presented in Section 6.2 where time-dependent edge travel-times are generated based on the two-years of historical traffic sensor data collected from Los Angeles road network.
sensor data throughout California. Meanwhile, we also witness that the leading navigation service providers (such as Navteq [42] and TeleAtlas [66]) started releasing their time-dependent travel-time data for road networks at high temporal resolution. With regards to the second argument, several recent studies showed the importance of time-dependent path planning in road networks where real-world traffic datasets have been used for the assessment. For example, in we show in the following section that the fastest path computation that considers time-dependent edge travel-times in Los Angeles road network decreases the travel-time by on average 36% over the fastest path computation that assumes constant edge travel-times. A similar observation has been done in another study [24] under IBM’s Smart Traffic Project where the time-dependent fastest path computation in Stockholm road network can improve the travel-time accuracy significantly. Considering the availability of high-resolution time-dependent travel-time data for road networks, and the importance of time-dependency for accurate and useful path planning, the need for efficient algorithms to enable next-generation time-dependent path planning applications becomes apparent and immediate.

In this section, we propose a bidirectional time-dependent fastest path algorithm (B-TDFP) based on A* search [27]. There are two main challenges to employ bidirectional A* search in time-dependent networks. First, finding an admissible heuristic function (i.e., lower-bound distance) between an intermediate \( v_i \) node and the destination \( d \) is challenging as the distance between \( v_i \) and \( d \) changes based on the departure-time from \( v_i \). Second, it is not possible to implement a backward search without knowing the arrival-time at the destination. We address the former challenge by partitioning the road network to non-overlapping partitions (an off-line operation) and precompute the intra (node-to-border) and inter (border-to-border) partition distance labels with respect to \( \text{Lower-bound Graph} \ G \) which is generated by substituting the edge travel-times in \( G \) with minimum possible travel-times. We use the combination of intra and inter distance
labels as a heuristic function in the online computation. To address the latter challenge, we run the backward search on the lower-bound graph ($G$) which enables us to filter-in the set of the nodes that needs to be explored by the forward search.

The remainder of this section is organized as follows. In Section 3, we formally define the time-dependent fastest path problem in spatial networks. In Section 6.2, we establish the theoretical foundation of our proposed bidirectional algorithm and explain our approach. In Section 6.3, we present the results of our experiments for both approaches with a variety of spatial networks with real-world time-dependent edge weights.

6.1 Feasibility of Time-dependent Path Planning

As we discussed there are handful of studies that focus on efficient computation of time-dependent fastest path. However, none of these studies investigate the practicality of time-dependent planning in real-world road networks. In this section, for the first time we assess the importance and the practical usefulness of time-dependent planning by comparing the results of time-independent fastest computation on a real-world spatial network with real time-varying edge travel-times. We focus on answering two specific questions: i) how much does time-dependent path planning reduce the travel-time as compared to static path planning, and ii) how different are the time-dependent fastest path and the static fastest path for a given source and destination. We answer these question in the following section based on our experimental evaluation with real-world datasets.

Towards this end we conducted extensive experiments to evaluate the practical usefulness of TDFP. As of our dataset, we used California (CA) and Los Angeles (LA) road network data [42] with approximately 1,965,300 and 304,162, respectively. In our
lab, we maintain large-scale and high resolution (both spatially and temporally) traffic sensor (i.e., loop detector) dataset collected from the entire Los Angeles County highways and arterial streets. This dataset includes both inventory and real-time data (with update rate as high as every 1 minute) for 6300 traffic sensors covering approximately 3200 miles. The sampling rate of the streaming data is 1 reading/sensor/min. We have been continuously collecting and archiving the traffic sensor data past two years. We use this real-world dataset create time varying edge weights, we spatially and temporally aggregate sensor data by assigning interpolation points (for each 5 minutes) that depict the travel-times on the network segments. Based on our observation, all roads are uncongested between 9PM and 6AM, and hence we assume static edge weights between those times. In order to create time-dependent edge weights for the local streets in LA as well as the entire CA road network, we developed a traffic modeling approach (based on our observations from LA dataset) that synthetically generates edge travel-time profiles [13]. Our approach uses spatial (e.g., locality, connectivity) and temporal (e.g., rush hour, weekday) characteristics to generate travel-time of network edges that does not have readily available sensor data.

In this section we report our experimental results from our fastest path queries in which we determine the source $s$ and destination $d$ nodes uniformly at random. We also pick our departure time randomly and uniformly distributed in time domain $T$. The average results are derived from 1000 random $s$-$d$ queries.

With our experiment we investigate how much TDFP improves the total travel-time as compared to static fastest path (FP). We use Dreyfus’s algorithm [16] to compute time-dependent fastest path for a given $s$ and $d$. To compute static fastest path (with Dijkstra’s algorithm), we use the maximum attainable speed (hence minimum travel-time) on the network edges. We conduct our experiments with the following settings.
Figure 6.2: TDFP and FP Comparison

Given $s-d$ path and for each 5 minutes from 6AM to 9PM, we first determine the time-dependent $P^* = \{n_1, ..., n_t\}, t$ and time-independent $P = \{n'_1, ..., n'_k\}$ optimum paths as well as their corresponding travel-times to $d$, i.e., $A^t_{P^*}(t)$ and $A_P(t)$, respectively. Next, we compute the actual time-dependent travel-time $A^t_P(t)$ of time-independent path $P$. Specifically, we take $P = \{n'_1, ..., n'_k\}$ and determine actual time-dependent cost of travel along $P$ departing from $n'_1$ for a given $t$. Figure 6.2(a) plots the improvement gained by the TDFP over its static counter part for which we measure the relative percentage increase of FP’s travel-time over that of TDFP computed as $A^t_P(t)/A^t_{P^*}(t) - 1$.

As shown, the cost of the path found by the TDFP is on average 36% better than that of FP and the difference is more significant (i.e., 68%, 43%) during rush hours (i.e., 7-9:30AM, 4-6PM). The reason for significant difference during rush hours is that the edge weights change rapidly especially at the boundaries of the traffic peak periods and hence causing an overall increase in the cost of FP. However, TDFP avoids the congestion by selecting alternative segments and hence yields better travel-times. As expected, the paths found by TDFP and FP is often same before 6AM and after 9PM. Figure 6.2(b) depicts the standard deviation (in minutes) of $A^t_P(t) - A^t_{P^*}(t)$. As illustrated, the standard deviation is also more significant during rush hours.
We also compared the path similarity (number of identical edges in $P^*$ and $P$) of TDFP and FP. Our results showed that the path found by the TDFP deviates on average 28% with maximum recorded deviation of 87% (where TDFP finds almost completely different path than that of FP) from FP. One interesting observation from this experiment is that although different departure times return different optimum paths, there exists only a limited number of different paths during a day for a given $s$ and $d$. In particular, we used 180 different departure times, and on average the number of distinct optimum-time path computed by time-dependent fastest path algorithm was on average 7, and at most 12.

In sum, we observe that the use of time-dependent information can significantly reduce the travel-times especially during peak hours when the faster travel-time routes needed the most. In addition, one interesting observation is that although time-dependent fastest path computation returns different optimal-paths for different departure times, there are only a limited number of distinct paths (i.e., 8 on average) for a given source and destination.

### 6.2 Time-dependent Fastest Path Computation

In this section, we explain our bidirectional time-dependent fastest path approach that we generalize bidirectional A* algorithm proposed for static spatial networks [51] to time-dependent road networks. Our proposed solution involves two phases. At the pre-computation phase, we partition the road network into non-overlapping partitions and precompute lower-bound distance labels within and across the partitions with respect to $G(V,E)$. Successively, at the online phase, we use the precomputed distance labels as a
heuristic function in our bidirectional time-dependent A* search that performs simultaneous searches from source and destination. Below we first remind the definitions that we will use in this section and then elaborate on both phases.

**Definition 11 Time-dependent Graph.** A Time-dependent Graph is defined as $G(V, E, T)$ where $V = \{v_i\}$ is a set of nodes and $E \subseteq V \times V$ is a set of edges representing the network segments each connecting two nodes. For every edge $e(v_i, v_j) \in E$, and $v_i \neq v_j$, there is a cost function $c_{v_i,v_j}(t)$, where $t$ is the time variable in time domain $T$. An edge cost function $c_{v_i,v_j}(t)$ specifies the travel-time from $v_i$ to $v_j$ starting at time $t$.

**Definition 12 Time-dependent Travel Cost.** Let $\{s = v_1, v_2, ..., v_k = d\}$ denotes a path which contains a sequence of nodes where $e(v_i, v_{i+1}) \in E$ and $i = 1, ..., k - 1$. Given a $G(V, E, T)$, a path $s \leadsto d$ from source $s$ to destination $d$, and a departure-time at the source $t_s$, the time-dependent travel cost $TT(s \leadsto d, t_s)$ is the time it takes to travel the path. Since the travel-time of an edge varies depending on the arrival-time to that edge, the travel-time of a path is computed as follows:

$$TT(s \leadsto d, t_s) = \sum_{i=1}^{k-1} c_{v_i,v_{i+1}}(t_i) \text{ where } t_1 = t_s, t_{i+1} = t_i + c_{v_i,v_{i+1}}(t_i), i = 1, ..., k.$$  

**Definition 13 Lower-bound Graph.** Given a $G(V, E, T)$, the corresponding Lower-bound Graph $\underline{G}(V, E)$ is a graph with the same topology (i.e, nodes and edges) as graph $G$, where the weight of each edge $c_{v_i,v_j}$ is fixed (not time-dependent) and is equal to the minimum possible weight $c_{v_i,v_j}^{\min}$ where $\forall e(v_i, v_j) \in E, t \in T c_{v_i,v_j}^{\min} \leq c_{v_i,v_j}(t)$.

**Definition 14 Lower-bound Travel Cost.** The lower-bound travel-time $LTT(s \leadsto d)$ of a path is less than the actual travel-time along that path and computed w.r.t $\underline{G}(V, E)$ as

$$LTT(s \leadsto d) = \sum_{i=1}^{k-1} c_{v_i,v_{i+1}}^{\min}, i = 1, ..., k.$$
It is important to note that for each source and destination pair \((s, d)\), \(LTT(s \rightsquigarrow d)\) is time-independent constant value and hence \(t\) is not included in its definition. Given the definitions of \(TT\) and \(LTT\), the following property always holds for any path in \(G(V, E, T)\): \(LTT(s \rightsquigarrow d) \leq TT(s \rightsquigarrow d, t_s)\) where \(t_s\) is an arbitrary departure-time from \(s\). We will use this property in subsequent sections to establish some properties of our proposed solution.

**Definition 15 Time-dependent Fastest Path (TDFP).** Given a \(G(V, E, T)\), \(s\), \(d\), and \(t_s\), the time-dependent fastest path \(TDFP(s, d, t_s)\) is a path with the minimum travel-time among all paths from \(s\) to \(d\) for starting time \(t_s\).

### 6.2.1 Precomputation Phase

The precomputation phase of our proposed algorithm includes two main steps in which we partition the road network into non-overlapping partitions and precompute lower-bound border-to-border, node-to-border, and border-to-node distance labels.

**Road Network Partitioning**

Real-world road networks are built on a well-defined hierarchy. For example, in United States, highways connect large regions such as states, interstate roads connect cities within a state, and multi-lane roads connect locations within a city. Almost all of the road network data providers (e.g., Navteq [42]) include road hierarchy information in their datasets. In this paper, we partition the graph to non-overlapping partitions by exploiting the predefined edge class information in road networks. Specifically, we first use higher level roads (e.g., interstate) to divide the road network into large regions. Then, we subdivide each large region using the next level roads and so on. We adopt this technique from [23] and note that our proposed algorithm is independent of the
partitioning method, i.e., it yields correct results with all non-overlapping partitioning methods.

With our approach, we assume that the class of each edge \(\text{class}(e)\) is predefined and we denote the class of a node \(\text{class}(v)\) by the lowest class number of any incoming or outgoing edge to/from \(v\). For instance, a node at the intersection of two freeway segments and an arterial road (i.e., the entry node to the freeway) is labeled with class of the freeway rather than the class of the arterial road. The input to our hierarchical partitioning method is the road network and the level of partitioning \(l\). For example, if we like to partition a particular road network based on the interstates, freeways, and arterial roads in sequence, we set \(l = 2\) where interstate edges represent the class 0. The road network partitions can be conceptually visualized as the areas after removal the nodes with \(\text{class}(v) \leq l\) from \(G(E, V)\).

**Definition 16** Given a graph \(G(V, E)\), the partition of \(G(V, E)\) is a set of subgraphs \(\{S_1, S_2, ..., S_k\}\) where \(S_i = (V_i, E_i)\) includes node set \(V_i\) where \(V_i \cap V_j = \emptyset\) and \(\bigcup_{i=1}^{k} V_i = V, i \neq j\).

Given a \(G(E, V)\) and level of partitioning \(l\), we first assign to each node an empty set of partitions. Then, we choose a node \(v_i\) that is connected to edges other than the ones used for partitioning (i.e., a node with \(\text{class}(v_i) > l\)) and add partition number (e.g., \(S_1\)) to \(v_i\)’s partition set. For instance, continuing with our example above, a node \(v_i\) with \(\text{class}(v_i) > 2\) represent a particular node that belongs a less important road segment than an arterial road. Subsequently, we expand a shortest path tree from \(v_i\) to all it’s neighbor nodes reachable through the edges of the classes greater than \(l\), and add \(S_1\) to their partition sets. Intuitively, we expand from \(v_i\) until we reach the roads that are used for partitioning. At this point we determine all the nodes that belong to \(S_1\). Then, we select another node \(v_j\) with an empty partition set by adding the next partition number (e.g., \(S_2\)) to \(v_j\)’s partition set and repeat the process. We terminate the process when
all nodes are assigned to at least one partition. With this method we can easily find the border nodes for each partition, i.e., those nodes which include multiple partitions in their partition sets. Specifically, a node \( v \), with \( \text{class}(v) \leq l \) belongs to all partitions such that there is an edge \( e \) (with \( \text{class}(e) > l \)) connecting \( v \) to \( v' \) where \( v' \in S_i \) and \( i = 1, ..., k \), is the border node of the partitions that it connects to. Note that \( l \) is a tuning parameter in our partitioning method. Hence, one can arrange the size of the partitions by increasing or decreasing \( l \).

Figure 6.3 shows the partitioning of San Joaquin (California) network based on the road classes. As shown, higher level edges are depicted with different (thicker) colors. Each partition is numbered starting from the north-west corner of the road network. The border nodes between partitions \( S_1 \) and \( S_4 \) are shown in the circled area. We remark that the number of border nodes (which can be potentially large depending on the density of the network) in the actual partitions have a negligible influence on the storage
complexity. We explain the effect of the border nodes on the storage cost in the next section.

**Distance Label Computation**

In this step, for each pair of partitions \((S_i, S_j)\) we compute the lower-bound fastest path cost w.r.t. \(G\) between each border in \(S_i\) to each border node in \(S_j\). However, we only store the minimum of all border-to-border fastest path distances. As an example, consider Figure 6.4 where the lower-bound fastest path cost between \(b_1\) and \(b_3\) (shown with straight line) is the minimum among all border-to-border distances (i.e., \(b_1-b_4\), \(b_2-b_4\), \(b_2-b_3\)) between \(S_1\) and \(S_2\). In addition, for each node \(v_i\) in a partition \(S_i\), we compute the lower-bound fastest path cost from \(v_i\) to all border nodes in \(S_i\) w.r.t. \(G\) and store the minimum among them. We repeat the same process from border nodes in \(S_i\) to \(v_i\). For example, border nodes \(b_1\) and \(b_4\) in Figure 6.4 are the nearest border nodes to \(s\) and \(d\), respectively. We will use the precomputed node-to-border, border-to-border, and border-to-node lower-bound travel-times (referred to as distance labels) to construct our heuristic function for online time-dependent A* search. We used a similar distance label precomputation technique to expedite shortest path computation between network Voronoi polygons in static road networks [34].

![Figure 6.4: Lower-bound distance computation.](image)
We maintain the distance labels by attaching three attributes to each node representing a) the partition $S_i$ that contains the node, b) minimum of the lower-bound distances from the node to border nodes, and c) minimum of the lower-bound distances from border nodes to the node (this is necessary for directed graphs). We keep border-to-border distance information in a hash table. Since we only store one distance value for each partition pair, the storage cost of the border-to-border distance labels is negligible. Another benefit of our proposed lower-bound computation is that the lower-bounds need to be updated when it is necessary. Specifically, we update the intra and inter distance labels only when the minimum travel-time of an edge changes, otherwise, the travel-time updates are discarded. Note that intra distance label computation is local, i.e., we only update the intra distance labels for the partitions in which the minimum travel-time of an edge changes.

### 6.2.2 Online B-TDFP Computation

As showed in [16], the time-dependent fastest path problem (in FIFO networks) can be solved by modifying Dijkstra algorithm. We refer to modified Dijkstra algorithm as time-dependent Dijkstra (TD-Dijkstra). TD-Dijkstra visits all network nodes reachable from $s$ in every direction until destination node $d$ is reached. On the other hand, a time-dependent A* algorithm can significantly reduce the number of nodes that have to be traversed in TD-Dijkstra algorithm by employing a heuristic function $h(v)$ that directs the search towards destination. To guarantee optimal results, $h(v)$ must be admissible and consistent (a.k.a, monotonic). The admissibility implies that $h(v)$ must be less than or equal to the actual distance between $v$ and $d$. With static road networks where the length of an edge is constant, Euclidian distance between $v$ and $d$ is used as $h(v)$. However, this simple heuristic function cannot be directly applied to time-dependent road networks, because, the optimal travel-time between $v$ and $d$ changes based on the
departure-time $t_v$ from $v$. Therefore, in time-dependent road networks, we need to use an estimator that never overestimates the travel-time between $v$ and $d$ for any possible $t_v$. One simple lower-bound estimator is $d_{\text{euc}}(v,d)/\max(speed)$, i.e., the Euclidean distance between $v$ and $d$ divided by the maximum speed among the edges in the entire network. Although this estimator is guaranteed to be a lower-bound, it is a very loose bound, and hence yields insignificant pruning.

With our approach, we obtain a much tighter bound by utilizing the precomputed distance labels. Assuming that an on-line time-dependent fastest path query requests a path from source $s$ in partition $S_i$ to destination $d$ in partition $S_j$, the fastest path must pass through from one border node $b_i$ in $S_i$ and another border node $b_j$ in $S_j$. We know that the time-dependent fastest path distance passing from $b_i$ and $b_j$ is greater than or equal to the precomputed lower-bound border-to-border (e.g., $LTT(b_i,b_l)$) distance for $S_i$ and $S_j$ pair. We also know that a time-dependent fastest path distance from $s$ to $b_i$ is always greater than or equal to the precomputed lower-bound fastest path distance of $s$ to its nearest border node $b_s$. Analogously, same is true from the border node $b_d$ (i.e., nearest border node) to $d$ in $S_j$. Thus, we can compute a lower-bound estimator of $s$ by $h(s) = LTT(s,b_s) + LTT(b_l,b_t) + LTT(b_d,d)$.

**Lemma 7** Given an intermediate node $v_i$ in $S_i$ and destination node $d$ in $S_j$, the estimator $h(v_i)$ is admissible, i.e., a lower-bound of time-dependent fastest path distance from $v_i$ to $d$ passing from border nodes $b_i$ and $b_j$ in $S_i$ and $S_j$, respectively.

**Proof 7** Assume $LTT(b_i,b_l)$ is the minimum border-to-border distance between $S_i$ and $S_j$, and $b_i', b_j'$ are the nearest border nodes to $v_i$ and $d$ in $G$, respectively. By definition of $G(V,E)$, $LTT(v_i,b_i') \leq TDFP(v_i,b_i,t_v)$, $LTT(b_l,b_t) \leq TDFP(b_i,b_j,t_{b_i})$, and $LTT(b_d,d) \leq TDFP(b_d,d,t_{b_d})$ Then, we have $h(v_i) = LTT(v_i,b_i') + LTT(b_l,b_t) + LTT(b_d,d) \leq TDFP(v_i,b_i,t_{v_i}) + TDFP(b_i,b_j,t_{b_i}) + TDFP(b_d,d,t_{b_d})$  □
We can use our $h(v)$ heuristic with unidirectional time-dependent A* search in road networks. The time-dependent A* algorithm is a best-first search algorithm which scans nodes based on their time-dependent cost label (maintained in a priority queue) to source similar to [16]. The only difference to [16] is that the label within the priority queue is not determined only by the time-dependent distance to source but also by a lower-bound of the distance to $d$, i.e., $h(v)$ introduced above.

To further speed-up the computation, we propose a bidirectional search that simultaneously searches forward from the source and backwards from the destination until the search frontiers meet. However, bidirectional search is challenging in time-dependent road networks for two following reasons. First, it is essential to start the backward search from the arrival-time at the destination $t_d$ and exact $t_d$ cannot be evaluated in advance at the query time (recall that arrival-time to destination depends on the departure-time from the source in time-dependent road networks). We address this problem by running a backward A* search that is based on the reverse lower-bound graph $\overset{\leftarrow}{G}$ (the lower-bound graph with every edge reversed). The main idea with running backward search in $\overset{\leftarrow}{G}$ is to determine the set of nodes that will be explored by the forward A* search. Second, it is not straightforward to satisfy the consistency (the second optimality condition of A* search) of $h(v)$ as the forward and reverse searches use different distance functions. Next, we explain bidirectional time-dependent A* search algorithm (Algorithm 1) and how we satisfy the consistency.

Given $G = (V, E, T)$, $s$ and $d$, and departure-time $t_s$ from $s$, let $Q_f$ and $Q_b$ represent the two priority queues that maintain the labels of nodes to be processed with forward and backward A* search, respectively. Let $F$ represent the set of nodes scanned by the forward search and $N_f$ is the corresponding set of labeled vertices (those in its priority queue). We denote the label of a node in $N_f$ by $d_{fv}$. Analogously, we define $B$, $N_b$, and $d_{bv}$ for the backward search. Note that during the bidirectional search $F$
and $B$ are disjoint but $N_f$ and $N_b$ may intersect. We simultaneously run the forward and backward A* searches on $G(V, E, T)$ and $\overrightarrow{G}$, respectively (Line 4 in Algorithm 1). We keep all the nodes visited by backward search in a set $H$ (Line 5). When the search frontiers meet, i.e., as soon as $N_f$ and $N_b$ have a node $u$ in common (Line 6), the cost of the time-dependent fastest path ($TDFP(s, u, t_s)$) from $s$ to $u$ is determined. At this point, we know that $TDFP(u, d, t_u) > LTT(u, d)$ for the path found by the backward search. Hence, the time-dependent cost of the paths (found so far) passing from $u$ is the upper-bound of the time-dependent fastest path from $s$ to $d$, i.e., $TDFP(s, u, t_s) + TDFP(u, d, t_u) \geq TDFP(s, d, t_s)$.

Figure 6.5: Bidirectional search

If we stop the searches as soon as a node $u$ is scanned by both forward and backward searches, we cannot guarantee finding the time-dependent fastest path from $u$ to $d$ within the set of nodes in $H$. This is due to inconsistent potential function used in bidirectional search that relies on two independent potential functions for two inner A* algorithms. Specifically, let $h_f(v)$ (estimated distance from node $v$ to target) and $h_b(v)$ (estimated distance from node $v$ to source) be the potential functions used in the forward and backward searches, respectively. With the backward search, each original edge $e(i, j)$ considered as $e(j, i)$ in the reverse graph where $h_b$ used as the potential
function, and hence the reduced cost $^3$ of $e(j, i)$ w.r.t. $h_b$ is computed by $c_{h_b}(j, i) = c(i, j) - h_b(j) + h_b(i)$ where $c(i, j)$ is the cost in the original graph. Note that $h_f$ and $h_b$ are consistent if, for all edges $(i, j)$, $c_{h_f}(i, j)$ in the original graph is equal to $c_{h_b}(j, i)$ in the reverse graph. If $h_f$ and $h_b$ are not consistent, there is no guarantee that the shortest path can be found when the search frontiers meet. For instance, consider Figure 6.5 where the forward and backward searches meet at node $u$. As shown, if $v$ is scanned before $u$ by the forward search, then $TDFP(s, u, t_s) > TDFP(s, v, t_s)$. Similarly if $w$ is scanned before $u$ by the backward search, the $LTT(u, d) > LTT(w, d)$ and hence $TDFP(u, d, t_u) > TDFP(w, d, t_w)$. Consequently, it is possible that $TDFP(s, u, t_s) + TDFP(u, d, t_u) \geq TDFP(s, v, t_s) + TDFP(w, d, t_w)$. To address this challenge, one needs to find a) a consistent heuristic function and stop the search when the forward and backward searches meet or b) a new termination condition. In this study, we develop a new termination condition (the proof of correctness is given below) in which we continue both searches until the $Q_b$ only contains nodes whose labels exceed $TDFP(s, u, t_s) + TDFP(u, d, t_u)$ by adding all visited nodes to $H$ (Line 9-11). Recall that the label (denoted by $d_{bv}$) of node $v$ in the backward search priority queue $Q_b$ is computed by the time-dependent distance from the destination to $v$ plus the lower-bound distance from $v$ to $s$, i.e., $d_{bv} = TDFP(v, d, t_u) + h(v)$. Hence, we stop the search when $d_{bv} > TDFP(s, u, t_s) + TDFP(u, d, t_u)$. As we explained, $TDFP(s, u, t_s) + TDFP(u, d, t_u)$ is the length of the fastest path seen so far (not necessarily the actual fastest path) and is updated during the search when a new common node $u'$ found with $TDFP(s, u', t_s) + TDFP(u', d, t_{u'}) < TDFP(s, u, t_s) + TDFP(u, d, t_u)$. Once both searches stop, $H$ will include all the candidate nodes that can possibly be part of the time-dependent fastest path to $d$. Finally,

$^3$A* search is equivalent to Dijkstra’s algorithm on a transformed network in which the cost of each edge $c(i, j)$ is equal to $c(i, j) - h(i) + h(j)$. 

78
we continue the forward search considering only the nodes in $H$ until we reach $d$ (Line 12).

**Algorithm 5: B-TDFP Algorithm**

1: //Input: $G_T, \overrightarrow{G}, s$:source, $d$:destination,$t_s$:departure time
2: //Output: a $(s, d, t_s)$ fastest path
3: //FS():forward search, BS():backward search, $N_f/N_b$: nodes scanned by 
   FS()/BS(), $d_{bv}$: label of the minimum element in BS queue
4: FS($G_T$) and BS($\overrightarrow{G}$) //start searches simultaneously
5: $N_f \leftarrow$ FS($G_T$) and $N_b \leftarrow$ BS($\overrightarrow{G}$)
6: If $N_f \cap N_b \neq \emptyset$ then $u \leftarrow N_f \cap N_b$
7: $M = TDFP(s, u, t_s) + TDFP(u, d, t_u)$
8: end If
9: While $d_{bv} \geq M$
10: $N_b \leftarrow$ BS($\overrightarrow{G}$)
11: End While
12: FS($N_b$)
13: return $(s, d, t_s)$

**Lemma 8** Algorithm 1 finds the correct time-dependent fastest path from source to destination for a given departure-time $t_s$.

**Proof 8** We prove Lemma 2 by contradiction. The forward search in Algorithm 1 is the same as the unidirectional A* algorithm and our heuristic function $h(v)$ is a lower-bound of time-dependent distance from $u$ to $v$. Therefore, the forward search is correct. Now, let $P(s, (u), d, t_s)$ represent the path from $s$ to $d$ passing from $u$ where forward and backward searches meet and $\omega$ denotes the cost of this path. As we showed $\omega$ is the upper-bound of actual time-dependent fastest path from $s$ to $d$. Let $\phi$ be the smallest label of the backward search in priority queue $Q_b$ when both forward and backward searches stopped. Recall that we stop searches when $\phi > \omega$. Suppose that Algorithm 1 is not correct and yields a suboptimal path, i.e., the fastest path passes from a node outside of the corridor generated by the forward and backward searches. Let $P*$ be the
fastest path from $s$ to $d$ for departure-time $t_s$ and cost of this path is $\alpha$. Let $v$ be the first node on $P^*$ which is going to be explored by the forward search and not explored by the backward search and $h_b(v)$ is the heuristic function for the backward search. Hence, we have $\phi \leq h_b(v) + LTT(v, d)$, $\alpha \leq \omega < \phi$ and $h_b(v) + LTT(v, d) \leq LTT(s, v) + LTT(v, d) \leq TDFP(s, v, t_s) + TDFP(t, t_v) = \alpha$, which is a contradiction. Hence, the fastest path will be found in the corridor of the nodes labeled by the backward search.

6.3 Performance Evaluation

We conducted extensive experiments with different spatial networks to evaluate the performance of our proposed bidirectional time-dependent fastest path (B-TDFP) approach. As of our dataset, we used California ($CA$), Los Angeles ($LA$) and San Joaquin County ($SJ$) road network data (obtained from Navteq [42]) with approximately 1,965,300, 304,162 and 24,123 nodes, respectively. We evaluate our proposed techniques using both syntectic and actual time-dependent travel-times gathered from real-world traffic sensor data. To generate time-dependent edge costs (travel-time) we use real-world traffic sensor dataset that we have been collecting (past 2 years) and archiving from a collection of approximately 7000 sensors located on the road network of Los Angeles. We collect speed, occupancy, volume information from these sensors and the sampling rate of the data is 1 reading/sensor/min. We spatially and temporally aggregate (average) historical sensor data based on 7 days (Monday to Sunday) of each month by assigning interpolation points for each 5 minutes. The interpolation points represent the travel-times at different times of a particular day. For example, an edge is assigned 180 cost attributes to represent how traffic tends to change between 6:00AM and 9:00PM for a particular date in a particular month, e.g., Monday traffic pattern in September. We
assume all roads are un-congested between 9:00PM and 6:00AM, and hence consider static edge weights during this interval. However, unfortunately not every edge has a sensor in road networks. In order to generate time-dependent edge weights on SJ and CA and for the edges that does not contain any sensor in LA, we developed a traffic modeling approach that creates edge travel-time profiles [13]. Our approach uses spatial (e.g., locality, connectivity) and temporal (e.g., rush hour, weekday) characteristics to generate travel-time of network edges that does not have readily available sensor data.

In this section, we report the experimental results from our fastest path queries in which we determine the s and d nodes uniformly at random. We also pick our departure-time randomly and uniformly distributed in time domain T. The average results are derived from 1000 random s-d queries. We only present the results for LA and CA, the experimental results for both SJ and LA are very similar. We conducted our experiments on a server with 2.7 GHz Pentium Core Duo processor with 12GB RAM memory.

**Comparison with ALT**

In this set of experiments we compare our algorithm with time-dependent ALT (TD-ALT) approaches [8, 41] with respect to storage and response time. We run our proposed algorithm both unidirectionally and bidirectionally (in CA network) and compare with [8] and [41], respectively. As we mentioned, selecting good landmarks that lead to good performance is very difficult and hence several heuristics have been proposed for landmark selection. Among these heuristics, we use the best known technique; max-Cover (see [8]) with 64 landmarks. We computed travel-times between each node and the landmarks with respect to G. Under this setting, to store the precomputed distances, TD-ALT attaches to each node an array of 64 elements corresponding to the number of landmarks. Assuming that each array element takes 2 bytes of space, the additional storage requirement of TD-ALT is 63 Megabytes. On the other hand, with our algorithm, we
divide CA network to 60 partitions and store the intra and inter distance labels. The total storage requirement of our proposed solution is 8.5 Megabytes where we consume, for each node, an array of 2 elements (corresponding to from and to distances to the closest border node) plus the border-to-border distance labels. Since the experimental results for both unidirectional and bidirectional searches differ insignificantly and due to space limitations, we only present the results from unidirectional search below. As shown in Figure 6.6 the response time of our unidirectional time-dependent A* search (U-TDFP) is approximately three times better than that of TD-ALT for all times. This is because the search space of TD-ALT is severely affected by the quality of the landmarks which are selected based on a heuristic. Specifically, TD-ALT may yield very loose bounds based on the randomly selected $s$ and $d$, and hence the large search space. In addition, with each iteration, TD-ALT needs to find the best landmark (among 64 landmarks) which yields largest triangular inequality distance for better pruning; it seems that the overhead of this operation is not negligible. On the other hand, U-TDFP yields a more directional search with the help of intra and inter distance labels with no additional computation.

Figure 6.6: TD-ALT Comparison
Performance of B-TDFP

In this set of experiments, we compare the performance of our proposed approach to other existing TDFP methods w.r.t to a) preprocessing time, b) storage (byte per node), c) the average number of relaxed edges, and d) average query time. Table 1 shows the preprocessing time (Pre Processing), storage (Storage), number of scanned nodes (#Nodes), and response time (Res. Time) of time-dependent Dijkstra (TD-Dijkstra) implemented based on [16], unidirectional (U-TDFP) and bidirectional (B-TDFP) time-dependent A* search implemented using our proposed heuristic function, time-dependent Contraction Hierarchies (TD-CH) [1], and time-dependent SHARC (TD-SHARC) [7]. To implement U-TDFP and B-TDFP, we divide CA and LA network to 60 (which roughly correspond to counties in CA) and 25 partitions, respectively. Comparing TD-Dijkstra with our approach, we observe a very high trade-off between the query results and precomputation in both LA and CA networks. Our proposed B-TDFP performs 23 times better than TD-Dijkstra depending on the network while preprocessing and storage overhead is relatively small. As shown, the preprocessing time and storage complexity is directly proportional to network size.

Comparing the time-dependent variant of SHARC (TD-SHARC) and CH (TD-CH) with our approach, we observe B-TDFP outperforms TD-SHARC and TD-CH in preprocessing and response time. We also observe that as the graph gets bigger or more edges are time-dependent, the preprocessing time of TD-SHARC increases drastically. The preprocessing of TD-SHARC takes very long for both road networks, i.e., up to 20 times more than B-TDFP. The reason for the performance gap is that TD-SHARC’s contraction routine cannot bypass the majority of the nodes in time-dependent road networks as in the static road networks. Recall that the importance of a node can change throughout the time under consideration in time-dependent road networks. In addition,
Table 6.1: Experimental Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>PreProcessing [h:m]</th>
<th>Storage [B/node]</th>
<th>#Nodes</th>
<th>Res. Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD-Dijkstra</td>
<td>0:00</td>
<td>0</td>
<td>1162323</td>
<td>4104.11</td>
</tr>
<tr>
<td>U-TDFP</td>
<td>1:13</td>
<td>6.82</td>
<td>90575</td>
<td>310.17</td>
</tr>
<tr>
<td>B-TDFP</td>
<td>1:13</td>
<td>6.82</td>
<td>67172</td>
<td>182.06</td>
</tr>
<tr>
<td>TD-SHARC</td>
<td>19:41</td>
<td>154.10</td>
<td>75104</td>
<td>227.26</td>
</tr>
<tr>
<td>TD-CH</td>
<td>3:55</td>
<td>1018.33</td>
<td>70011</td>
<td>209.12</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD-Dijkstra</td>
<td>0:00</td>
<td>0</td>
<td>210384</td>
<td>2590.07</td>
</tr>
<tr>
<td>U-TDFP</td>
<td>0:27</td>
<td>3.51</td>
<td>11115</td>
<td>197.23</td>
</tr>
<tr>
<td>B-TDFP</td>
<td>0:27</td>
<td>3.51</td>
<td>6681</td>
<td>101.22</td>
</tr>
<tr>
<td>TD-SHARC</td>
<td>11:12</td>
<td>68.47</td>
<td>9566</td>
<td>168.11</td>
</tr>
<tr>
<td>TD-CH</td>
<td>1:58</td>
<td>740.88</td>
<td>7922</td>
<td>140.25</td>
</tr>
</tbody>
</table>

TD-SHARC is very sensitive to edge cost function changes, i.e. whenever cost function of an edge changes, the preprocessing phase needs to be repeated to determine the by-pass nodes. While TD-CH tend to have better response times than TD-SHARC, the space consumption of TD-CH is significantly high (approximately 1000 bytes per node in CA network). For this reason, TD-CH is not feasible for very large road networks such as North America and Europe. We note that, to improve the response and preprocessing time, several variations of TD-SHARC and TD-CH algorithms are implemented in the literature. These variations trade-off between the optimality of the solution and the response time. For example, the response time of Heuristic TD-SHARC [7] is shown much better than that of original TD-SHARC algorithm. However, the path found by the Heuristic TD-SHARC is not optimal and the error rate is not bounded. As another example, the performance of TD-SHARC can be improved by combining with another technique called Arc-Flags [7]. Similar performance improvements can be applied to
our proposed approach. For instance, we can terminate the search when the search frontiers meet and report the combination of path found by the forward and backward search as the result. However, as mentioned in Section 6.2.2, we cannot guarantee the optimal solution in this setting. Moreover, based on our initial observation and implementation, we can also integrate our algorithm with Arc-Flags. However, the focus of our study is to develop a technique that yields exact solutions. Hence, for the sake of simplicity and fair comparison, we only compare the original algorithms that yields exact results and do not consider integrating different methods.

Quality of Lower-bounds

As discussed, the performance of time-dependent A* search depends on the lower-bound distance. In this set of experiments, we analyze the quality of our proposed lower-bound computed based on the Distance Labels explained in Section 6.2.1. We define the lower-bound quality by

\[ \lg = \frac{\delta(u,v)}{d(u,v)} \]

where \( \delta(u,v) \) and \( d(u,v) \) represent the estimated and actual travel-times between nodes \( u \) and \( v \), respectively. Table 6.2 reports \( \lg \) based on three different heuristic function, namely Naive, ALT, and DL (i.e., our heuristic function computed based on Distance Labels). Similar to other experiments, the values in Table 6.2 are obtained by selecting \( s, d \) and \( t_s \) uniformly at random between 6AM and 9PM. We compute the naive lower-bound estimator by

\[ d_{euc}(u,v) \text{max(speed)} \]

i.e., the Euclidean distance between \( u \) and \( v \) is divided by the maximum speed among the edges in the entire network. We obtain the ALT lower-bounds based on \( G \) and the maxCover ([8]) technique with 64 landmarks. As shown, DL provides better heuristic function in both LA and CA. The reason is that the ALT’s \( \lg \) relies on the distribution of the landmarks, and hence depending on the location of \( s \) and \( d \) it is possible to get very loose bounds. On the other hand, the lower-bounds computed based on Distance Labels are more directional. Specifically, with our approach the \( s \) and \( d \) nodes must reside in one
of the partitions and the (border-to-border) distance between these partitions is always considered for the lower-bound computation.

Table 6.2: Lower-bound Quality

<table>
<thead>
<tr>
<th>Network</th>
<th>Naive (%)</th>
<th>ALT (%)</th>
<th>DL (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>21</td>
<td>42</td>
<td>63</td>
</tr>
<tr>
<td>LA</td>
<td>33</td>
<td>46</td>
<td>66</td>
</tr>
</tbody>
</table>

Bidirectional vs Unidirectional Search

In another set of experiments, we study the impact of path length (i.e., distance from $s$ to $d$) on the speed-up of bidirectional search. Hence, we measure the performance of B-TDFP and U-TDFP with respect to distance by varying the path distance (1 to 300 miles) between $s$ and $d$. Figure 6.7 shows the speed-up with respect to distance. We observe that the speed-up is significantly more especially for long distance queries. The reason is that for short distances the computational overhead incurred by B-TDFP is not worthwhile as U-TDFP visits less number of nodes anyway.

![Figure 6.7: Speed-up Ratio Analysis](image)

Figure 6.7: Speed-up Ratio Analysis
Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we studied the problem of time-dependent k nearest neighbor and fastest path queries in spatial networks where the weight of each edge is a function of time. We formulated a generalized type of k nearest neighbor query where we, unlike the existing studies, assume the edge weights of the network are time varying rather than fixed. We introduced indexing schemes that partition the network space based on object locations using time-independent network distance metrics between the objects. This facilitates the localization of the search space, hence reduces the invocation of the expensive fastest-path computation between the query point and data objects in time-dependent spatial networks. Our proposed index structures are independent of density and distribution of the data objects, and effectively handle the database updates where nodes, links, and data objects are added or removed.

To process the fastest path queries in time-dependent road networks, we study a bidirectional A* search based on a novel heuristic function. Since the number of fastest paths between any pair of nodes in time-dependent road networks can be theoretically super-polynomial, it is infeasible to extend existing shortest path precomputation techniques proposed for static road networks to time-dependent road networks. With our approach, we partition the road network to non-overlapping partitions and precompute the intra (node-to-border) and inter (border-to-border) partition distance labels with respect to time-independent lower-bound graph. We use the combination of intra and
inter distance labels as a heuristic function to efficiently prune the search space during the online computation.

7.2 Discussion of Future Work

We plan to extend our work in three main directions. First, although our proposed approach is efficient for predefined static data objects, it will not scale in case of mobile data objects due to frequent node reconstruction of index structures. Therefore, we will address this fundamental challenge in kNN queries with non-predefined data objects. Second, the correctness and the performance evaluation of our proposed algorithm and it’s extensions rely on the accurate and realistic modeling of time-dependent spatial networks. Thus, we will focus on developing a framework that generates realistic and well-defined data for the time-dependent spatial networks. Third, our time-dependent fastest path computation is based on a bidirectional A* search. However, we observe that response time of this approach may degrade with relatively long paths queries (e.g., Los Angeles to Seattle). To improve the performance of our proposed algorithm we plan to study hierarchical fastest path computation in time-dependent road networks. Below we elaborate each of the three future tasks in turn.

Construction of Dynamic Index Structures for Time-dependent Road Networks

We plan to extend our time-dependent kNN algorithm to support arbitrarily moving data and/or query objects. This extension will enable us to support continuous monitoring of kNN queries on time-dependent road networks. As an example of continuous kNN problem, consider that the queries correspond to pedestrians, and the data objects are transportation vehicles (such as taxis, buses, and trains). As transportation vehicles and pedestrians move, each pedestrian wishes to know his/her k closest transportation
vehicles in terms of time-dependent travel-time. This problem and its variations have been extensively studied in Euclidean space (e.g., [37, 38]) and there exists a few studies assuming static road networks [9, 40]. To the best of our knowledge, continuous $k$NN queries for moving objects and queries on time-dependent road networks is an open problem. The main challenge with continuous $k$NN monitoring on time-dependent road networks is that the server(s) need to continuously compute and update the results of each query in real-time (or close to real-time). One approach to address this challenge is to design dynamic index structures that are updatable with minimal incremental cost as the data objects move. The basic idea is to localize the update cost of the index structure to the local tight or loose sub-networks in order to avoid global reconstruction of the indexes with each update. In addition, with lazy update techniques we can control the accuracy of the index versus its efficiency.

**Accurate Modeling of Time-dependent Road Networks**

The accurate modeling of time-dependent road networks is critical for the following reasons. First, the design, development and correctness of the time-dependent road network query algorithms depend on the correctness of the model. Second, due to the availability of traffic data, we envision that many researchers in industry and academia will consider developing new time-dependent query algorithms. Clearly, the performance evaluation and comparison of these proposed algorithms under various conditions are critical to the success of this research area. Therefore, to enable systematic and comprehensible evaluation and comparison of the proposed algorithms, there is a need for a model that produces realistic and well-defined test data for the time-dependent road networks. Last but not least, full-scale and realistic modeling of the time-varying edge weights on road networks will enable more accurate Intelligent Transportation Systems (ITS) and
Advanced Traveler Information Systems (ATIS) development, and facilitate transportation systems analysis to improve mobility and throughput. In [13], we conducted some preliminary work in creating traffic flow models for the Los Angeles County freeways. However, through our experiments in [DKS09a], we observed that the freeway network model is not sufficient to evaluate time-dependent query algorithms. This is because 1) number of freeways in a city, even a megacity such as Los Angeles, is too small to evaluate the scalability of our algorithms, and 2) only a small number of traffic patterns exist on freeways. Moreover, in [13], we only used spatial characteristics of road segments (e.g., if a segment is in residential or business area) to cluster and model their corresponding traffic data. Clearly, we need to bring temporal clustering to the mix, for example, cluster segments based on the time of the day (e.g., rush hour). Hence, we propose to create a framework for realistic and accurate modeling of time-dependent edge weights for the entire road network of Los Angeles County considering both spatial and temporal characteristics of the data. Our initial study suggests that there are a finite number of profiles for a given city (e.g., about 59 for Los Angeles). The idea is whether we can parameterize these profiles for different cities; for example, include the start and end of the morning and afternoon rush-hours as parameters to a single weight-profile to generate traffic patterns for different cities.

**Hierarchical Time-dependent Fastest Path Computation**

We plan to develop a hierarchical path planning approach (H-TDSP) that exploits the road hierarchies (e.g., freeways, arterials, alleys) inherit in real-world road networks. This solution offers to make a trade-off between the computation time and the optimality of the shortest path. The intuition here is that in real-world, given that source and destination are sufficiently far away, humans tend to select paths hierarchically, i.e., drive on the nearest main street which connects to freeway, drive on the freeway to a location
close to the destination, exit freeway and reach the destination via a main street. The hierarchical path may not necessarily be optimal but largely considered as the best path due to the fact that higher level road segments offer uninterrupted (e.g., smaller number of turns, no traffic lights) and safer travel. Because of their simplicity and popularity in real-world traveling, hierarchical route planning approaches have been widely deployed by industry for static road networks. To the best of our knowledge, hierarchical route planning solution in time-dependent road networks has not yet been studied. We plan to develop an approach where we start the path search at the first level and then carry out exploring the hierarchy in the network in ascending order. Since the number of nodes at each level shrinks rapidly, the total number of explored nodes is considerably smaller than that of the plain shortest path algorithm. The main challenge however with hierarchical path planning in time-dependent networks is to identify the transit node through which we start searching the adjacent higher levels. This is due to the fact that the travel-time from the search node to the potential transit nodes can change significantly throughout the day, and hence unlike their static counterpart solutions, it is not possible to have predetermined transit nodes. We will address this challenge by extending our tight and loose cell methods that pre-compute two sub-networks around each transit node. Given the source and destination nodes, tight and loose cells around transit nodes will enable us to localize the search space by identifying the best transit node by executing either none or very limited number of shortest path computations.
Bibliography


[23] Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, and John Paul Sondag. Adaptive fastest path computation on a road network: A traffic mining approach. In *VLDB*, 2007.


[27] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic
determination of minimum cost paths. *IEEE Transactions on Systems Science and
Cybernetics*, 1968.

[28] Xuegang Huang, Christian S. Jensen, Hua Lu, and Simonas Saltenis. S-grid: A
versatile approach to efficient query processing in spatial networks. In *SSTD*, pages

[29] Xuegang Huang, Christian S. Jensen, and Simonas Saltenis. The islands approach


[31] Dmitri V. Kalashnikov, Sunil Prabhakar, and Susanne E. Hambrusch. Main mem-


[33] Ekkehard Kohler, Katharina Langkau, and Martin Skutella. Time expanded graphs

[34] Mohammad Kolahdouzan and Cyrus Shahabi. Voronoi-based k nearest neighbor


[36] Mohammad R. Kolahdouzan and Cyrus Shahabi. Voronoi-based k nearest neigh-

[37] Mohamed F. Mokbel, Xiaopeing Xiong, and Walid G. Aref. Sina: scalable incre-
mental processing of continuous queries in spatio-temporal databases. In *SIG-

[38] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual
partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIG-
MOD*, 2005.

54, 2006.


Appendix A

Indexing Network Voronoi Diagrams

The latest developments in wireless technologies as well as the widespread use of GPS-enabled mobile devices have led to the recent prevalence of location-based services. An important class of location based queries consists of proximity queries such as k Nearest Neighbor (kNN) query \[30, 34, 54, 62, 76\] and its variations, e.g., Reverse k Nearest Neighbor (RkNN) \[64, 73\], k Aggregate Nearest Neighbor (kANN) [72]. The proximity queries in general search for data objects that minimize a distance-based function with reference to one or more query objects.

With proximity queries, potentially the distance between the query point and every object in the database (e.g., all the points-of-interest) must be computed in order to find the closest (or the \(k\) closest) object(s) to the query point. Hence, the main research focus has been on indexing the objects to avoid the exhaustive search. Earlier studies assumed Euclidean distance as the distance function and hence indexed the objects in Euclidean space (e.g., \[62, 65, 75, 76\]) using R-tree like index structures. With the advent of online mapping systems such as Google Maps and Mapquest and the availability of accurate nation-wide road network data, the proximity queries have been extended from Euclidean space to the road network space as natural artifact. The challenge in processing proximity queries on road networks is that the computation of the distance function is complex and hence the indexing techniques incorporated some sort of pre-computation of distances (in network) into their structures. One such approach is based on using network Voronoi diagrams [44].
A network Voronoi diagram is a specialization of a Voronoi diagram in which the locations of objects are restricted to the network edges and the distance between objects is defined as the length of the shortest network distance (e.g., shortest path or shortest time), instead of the Euclidean distance. Any network node located in a Voronoi cell has a shortest path to its corresponding Voronoi generator that is always shorter than that to any other Voronoi generator. A large number of studies adopted network Voronoi diagrams [44] to evaluate variety of proximity queries on road networks (e.g., [34, 43, 45, 55, 71]). For example, in [45] Okabe et al. introduced six different types of network Voronoi diagrams (each corresponds to very important real-world applications) whose generators are based on points, sets of points, lines and polygons, and whose distances are given by inward/outward distances, and additively/multiplicatively weighted shortest path distances.

Given a query point \( q \) and network Voronoi diagram (\( NVD \)), the first step in answering any proximity query is to locate the network Voronoi cell \( NVC(p_i) \) that contains \( q \) (the generator \( p_i \) of \( NVC(p_i) \) is the nearest neighbor of \( q \)). We refer to this operation as \( contain(q) \) in the rest of the paper. Considering the large size of the underlying space (e.g., a continental size road network) with numerous data objects as well as the online nature of the queries that requires fast response-time, an index structure is necessary to efficiently access the portion of \( NVD \) associated with \( q \). Although the existing approaches successfully used network Voronoi diagrams as a pre-computation approach for partitioning the network space, they overlooked the indexing techniques that enable efficient evaluation of \( contain(q) \). Currently, indexing network Voronoi diagram with R-tree (referred as Voronoi R-tree or VR-tree for short) is the only known method for locating the network Voronoi cell that contains a particular point or edge of the network. VR-tree is first proposed in [34] and later used in many other approaches based on NVD (e.g., [43, 55, 71]).
In this paper, we show that VR-tree has two main problems. First, VR-tree may yield inaccurate results due to the way the Voronoi cells are formed in network space, i.e., although a NVD is generated based on the network distance metric, its Voronoi cells are created and indexed as regular polygons in Euclidean space. This inconsistency may result in a network edge belonging to a cell \( NVC(p_i) \), to be classified as a member of the cell \( NVC(p_j) \) because due to the network topology, the edge falls inside the polygon of \( NVC(p_j) \) even though its network distance is closer to the generator of \( NVC(p_i) \). For example, Figure A.1 depicts the network Voronoi diagram of a hypothetical road network where each line style corresponds to network Voronoi cells of the generators \( p_1, p_2 \) and \( p_3 \). With VR-tree the network Voronoi cells are formed by connecting the border points (i.e., \( \{b_1, b_2, \ldots, b_7\} \)) \(^1\) and bounded by straight line segments (i.e., bold lines in the Figure). As shown, the edges marked by false-negative are included in the Voronoi cell of \( p_1, NVC(p_1) \), however the network distance from any point on the false-negative edges to \( p_3 \) is shorter than that to \( p_1 \).

Second, VR-tree is inefficient because of the non-disjoint partitioning of the space. Specifically, VR-tree splits the network space with hierarchically nested and largely overlapping minimum bounding rectangles (MBR) created around network Voronoi

\(^1\)We discuss the network Voronoi diagram generation in Section A.2.1
The overhead of executing \textit{contain}(q) query is prohibitively high particularly in large networks with a dense (but perhaps large) set of data objects. This is because VR-tree has to redundantly visit the parent node(s) of the overlapping MBRs (aka, backtracking problem) in the index structure.

To address both of the aforementioned drawbacks, we propose a new indexing approach for network Voronoi diagrams based on region Quad-tree [56], termed \textit{Voronoi-Quad-tree} or VQ-tree for short. VQ-tree, unlike VR-tree that approximates network Voronoi cells using regular polygons in the Euclidean space, enables exact representation of the network Voronoi cells based on quad-tree blocks in the network space, and hence always yields correct results. VQ-tree does not suffer from the backtracking problem of VR-tree. This is because VQ-tree enables disjoint decomposition of the network space and encodes each of the quad-tree blocks to indicate the identity of the network Voronoi cell of which it is a member. Thus, once the quad-tree block containing \( q \) is located, VQ-tree immediately identifies the nearest Voronoi generator based on the encoded value of that block. Our experiments with real-world datasets show that the ratio of false-negative edges is \( \%16 \) on average with respect to the total number of edges in the network and VQ-tree outperforms VR-tree with 12 times improved response time (see Section 6.3).

The remainder of this section is organized as follows. In Section A.1, we overview Network Voronoi diagrams and it’s properties. In Section A.2, we establish the theoretical foundation of the proposed solution for indexing Network Voronoi diagrams for efficient and accurate processing of proximity queries in spatial networks. In Section 6.3, we present the results of our experiments with a variety of spatial networks with large number of query and data objects.
A.1 Preliminaries

In this section, we review the principles of Euclidean and Network Voronoi diagrams. We first introduce 2-dimensional Euclidean space Voronoi diagrams and describe the properties of Voronoi diagrams. We then explain the network Voronoi diagram. We refer readers to [44] for a comprehensive discussion of Euclidean and network Voronoi diagrams.

Voronoi Diagrams

Let $P = \{p_1, p_2, ..., p_n\}$ be a set of $n$ distinct sites (i.e., generator points) distributed in the Euclidean space. These generator points can be considered any spatial type of objects (e.g., gas station, restaurant). We define the Voronoi diagram of $P$ as the subdivision of the space into $n$ cells, one for each site in $P$, with the property that a point $q$ lies in the cell corresponding to a site $p_i$ if and only if $\text{distance}(q, p_i) < \text{distance}(q, p_j)$ for each $p_j \in P$ with $j \neq i$. Figure A.2 shows the ordinary Voronoi diagram of eight points where the distance metric is Euclidean.

![Voronoi diagram in Euclidean space](image)

Figure A.2: Voronoi diagram in Euclidean space

We refer to the region containing the point $p_i$ as its Voronoi cell $VC(p_i)$ or Voronoi polygon (see $VC(p_4)$ in the Figure). In Euclidean space, $VC(p_i)$ is a convex polygon.
Each edge of $VC(p_i)$ is a segment of the perpendicular bisector of the line segment connecting $p$ to another point of the set $P$. We call each of these edges a Voronoi edge. The Voronoi cells that have common edges are called adjacent cells and their generators are called adjacent generators. The Voronoi cells are collectively exhaustive and mutually exclusive except their boundaries (i.e., Voronoi edges). We define the Voronoi cell and Voronoi diagram as follows.

**Definition 17** Consider $P : \{p_1, p_2, \ldots, p_n\}$ where $2 \leq n$ and $p_i \neq p_j$ for $i \neq j$, $i, j \in I_n = 1, \ldots, n$. The region given by $VC(p_i) = p|d(p, p_i) \leq d(p, p_j)$ where $d(p, p_i)$ is the minimum Euclidean distance between $p$ and $p_i$ is called the Voronoi Cell (VC) associated with $p_i$.

**Definition 18** The set of Voronoi cells given by $VD(P) = \{VC(p_1), ..., VC(p_n)\}$ is called the Voronoi Diagram (VD) generated by $P$.

**Network Voronoi Diagrams**

With network Voronoi diagrams ($NVD$), the $VD$ described above is generalized by replacing the Euclidean space with a spatial network (e.g., road network), hence the distance with the network distance (e.g., shortest-path) between the objects.

**Definition 19** A road network is represented as a directional weighted graph $G(N, E)$, where $N$ is a set of nodes representing intersections and terminal points, and $E$ ($E \subseteq N \times N$) is a set of edges representing the network edges each connecting two nodes. Each edge $e$ is denoted as $e(n_i, n_j)$ where $n_i$ and $n_j$ are starting and ending nodes, respectively.

In this study, we consider planar graph where edges intersect only at their endpoints. We assume that Voronoi generators are located on the network segments as the graph
nodes. Each edge connecting nodes $p_i$, $p_j$ stores the network distance $d_N(p_i, p_j)$. For nodes that are not directly connected, $d_N(p_i, p_j)$ is the length of the shortest path from $p_i$ to $p_j$.

Given a weighted graph $G(N, E)$ consisting of a set of nodes $N = \{p_1, \ldots, p_n, p_{n+1}, \ldots, p_o\}$ where the first $n$ nodes represent the Voronoi generators and a set of edges $E = \{e_1, \ldots, e_k\}$ that connects the nodes, we define the set dominance region and border points as follows,

**Definition 20** The dominance region of $p_i$ over $p_j$

$\text{Dom}(p_i, p_j) = \{p|p \in \bigcup_{o=1}^{k} e_o, d_N(p, p_i) \leq d_N(p, p_j)\}$ represents all points in all edges in $E$ that are closer (or equal distance) to $p_i$ than $p_j$.

**Definition 21** The border points between $p_i$ and $p_j$

$b(p_i, p_j) = \{p|p \in \bigcup_{o=1}^{k} e_o, d_N(p, p_i) = d_N(p, p_j)\}$ represent all points in all edges that are equally distanced from $p_i$ and $p_j$.

**Definition 22** Based on the above definitions, the Voronoi edge set $V_{\text{edge}}$ of $p_i$ as

$V_{\text{edge}}(p_i) = \bigcup_{j \in I_n \setminus \{i\}} \text{Dom}(p_i, p_j)$ represents all the points in all edges in $E$ that are closer to $p_i$ than any other generator point in $N$. Consequently, we define network Voronoi diagram $NVD(P)$ w.r.t set of points $P$ as $NVD(P) = \{V_{\text{edge}}(p_1), \ldots, V_{\text{edge}}(p_n)\}$.

Similar to $VD$ described in Section A.1, the elements of $NVD$ are mutually exclusive and collectively exhaustive.

### A.2 Indexing Network Voronoi Cells

In this section, we will first explain how to construct a network Voronoi diagram in road networks and then discuss two different index structures, namely the Voronoi R-tree and
Voronoi Quad-tree that efficiently identifies the subdivision of the network space that contains a particular query point or network edge.

### A.2.1 Network Voronoi Diagram Construction

The network Voronoi diagrams can be constructed using parallel Dijkstra algorithm [17] with the Voronoi generators as multiple sources. Specifically, one can expand shortest path trees from each Voronoi generator simultaneously and stop the expansions when the shortest path trees meet.

![Road Network](image1)

![Network Voronoi Diagram](image2)

Figure A.3: A Road network and network Voronoi diagram

Figure A.3 shows an example of road network and the corresponding network Voronoi diagram. Figure A.3a depicts the original weighted graph $G(N, E)$ which consists of $N = \{p_1, p_2, p_3, p_4, ... p_{16}\}$ nodes where $p_1$, $p_2$, and $p_3$ are the Voronoi generators (i.e., data objects such as restaurants, hotels) and $p_4$ to $p_{16}$ are the intersections on a road network that are interconnected by a set of edges. Figure A.3b shows the NVD of the road network where each line style corresponds to the shortest path tree based on the generators \{\(p_1, p_2, p_3\}\}. Each shortest path tree composes a network Voronoi cell and some edges (e.g., $e(p_4, p_5)$) can be partially contained in different network Voronoi...
cells. The border points $b_1$ to $b_7$ are the nodes where the shortest path trees meet as a result of the parallel Dijkstra algorithm. The border points between any two generator $p_i$ and $p_j$ are equally distanced from $p_i$ and $p_j$. Figure A.4 shows a real network Voronoi diagram with respect to 50 data objects in Los Angeles road network. Each network node marked with a different color corresponds to a network Voronoi cell.

Figure A.4: Network Voronoi diagram with $P = \{p_1, ..., p_{50}\}$ in Los Angeles road network.

### A.2.2 Index Generation on Network Voronoi Diagram

As we discussed, to answer any proximity query with respect to a query point $q$, one first needs to find the Voronoi cell that contains $q$. There remains a basic question concerning how to efficiently access the portion of the NVD associated with a particular query point $q$. This can be achieved by utilizing a spatial index structure that is generated on Voronoi cells. Below, we discuss two types of spatial index structures that can be used to index NVCs, namely, the Voronoi R-tree (VR-tree) and Voronoi Quad-tree (VQ-tree).
The Voronoi R-tree (VR-tree)

VR-tree is first introduced in [34] where NVD is used to evaluate kNN queries in road networks. VR-tree is based on the R-tree that splits the network space with hierarchically nested Minimum Bound Rectangels (MBR) generated around network Voronoi cells. Given the location of a query point $q$, a $\text{contain}(q)$ query invoked on VR-tree starts from the root node and iteratively checks the MBRs (of NVCs) with respect to a $q$ to decide whether or not to further search the child nodes.

![Figure A.5: Network Voronoi cell construction in VR-tree](image)

VR-tree has two main shortcomings. First, VR-tree may yield inaccurate results for a $\text{contain}(q)$ query. This is because VR-tree makes the simplifying assumption that although the NVD is computed based on the network distance metric, its NVCs are treated as regular polygons (by connecting border points of NVCs) and indexed using R-tree that is designed for the Euclidean distance metric. However, such approach may cause misclassification of the network edges (i.e., false-negative edges) in the network Voronoi cells, and hence inaccurate results. Specifically, a network edge belonging to a network Voronoi cell of $p_i, NVC(p_i)$ may be classified as a member of another network Voronoi cell $NVC(p_j)$. For instance, continuing with our running example
in Figure A.3, Figure A.5(a) shows how adjacent border points are connected to each other: if two adjacent border points are between two similar generators (e.g., $b_5$ and $b_7$ are between $p_1$ and $p_3$), they can be connected with an arbitrary line. Three or more adjacent border points (e.g., $b_2$, $b_3$ and $b_5$) can be connected to each other through an arbitrary auxiliary point (e.g., $v$ in the figure). As a result, similar to its Euclidean counterpart, the NVCs are represented with polygons in the network space. However, to illustrate why VR-tree may fail to yield correct results, consider Figure A.5(b) where we introduce two new edges (as an extension of $p_{12}$) to the road network. As shown, although the new edges (marked by false-negative edges in the Figure) are included inside the Voronoi cell of $p_1$, the network distance from any point on the false-negative edges to $p_3$ is shorter than that to $p_1$. Thus, with VR-tree, when $q$ is located on false-negative edges, a contain$(q)$ will return incorrect Voronoi generator as the NN. With our example we only show one particular case that can happen in real-world road networks. Arguably, it is possible to increase the number of such examples under different road network topologies. Figure A.6 depicts the NVC of a particular data object in Los Angeles road network where border nodes and false-negative edges are marked by light blue and red color, respectively.

One naive solution to the inaccuracy problem of VR-tree is to perform an additional refinement step. Specifically, one can maintain false-negative edges (along with their corresponding Voronoi generators) in a separate index structure and, for each contain$(q)$ query, check $q$ against this index structure. If $q$ is located in any of the false-negative edges, the corresponding Voronoi generator is returned as the nearest neighbor. Otherwise, VR-tree continues the search based on MBRs of the Voronoi cells as explained above.
Second, VR-tree is inefficient due to non-disjoint partitioning of the space. Specifically, with VR-tree the hierarchy of NVCs is enforced by minimum bounding rectangles created around network Voronoi cells. Depending on the different topologies of the road network and the distribution of the objects on the network segments, the overlapping areas of MBRs of network Voronoi cells may be quite large, and hence significant computation overhead in traversing R-tree for \textit{contain}(q) query. For example, Figure A.7 illustrates the MBRs of network Voronoi cells in Figure A.4. For the sake of clarity, we do not include the Voronoi cells in the picture. As shown, the MBRs around network Voronoi cells result in a \textit{non-disjoint decomposition} of the underlying space which means that the location occupied by a Voronoi cell may be contained in several bounding boxes. This degrades the search performance in VR-tree because of the backtracking [25] problem, i.e., the parent node(s) of the overlapping MBRs have to be accessed repeatedly in order to search the child nodes that contain \(q\). Thus, with VR-tree the amount of work often depends on the overlapping areas of MBRs. We also implemented VR-tree with R+ tree [60] to reduce the impact of overlapping areas. However,
we observe that the performance of VR+ tree is still less as compared to VQ-tree (see Section A.3).

Figure A.7: Minimum bounding rectangles on network Voronoi cells

The Voronoi Quad-tree (VQ-tree)

The alternative to VR-tree is to index network Voronoi cells using Quad-tree [19, 56], termed Voronoi Quad-tree (VQ-tree), that enables disjoint decomposition of the underlying space. The main observation behind VQ-tree is that each color coded area in Figure A.4 is a spatially contiguous region in the network space. The regions are mutually exclusive as they do not have any overlapping areas and collectively exhaustive as every location in the network space is associated with at least one generator. Therefore, an exact approximation of the network Voronoi diagram can be obtained by using a region quad-tree [56] where the leaf nodes of the quad-tree correspond to a region in a Voronoi cell in NVD. In particular, with VQ-tree the root node represents the rectangular region enclosing the entire span of the road network (and hence NVD) under consideration. We subdivide this rectangular region into four equal quadrants where each quadrant is one of the four child nodes of the root. Subsequently, we recursively subdivide the quadrants until each quadrant contains only one network Voronoi cell information. That is,
for each quadrant, we search for two (or more) different color-coded nodes\(^2\). If we find such a quadrant (meaning that the quadrant includes more than one network Voronoi cell), we subdivide that quadrant into four subquadrants. This subdivision process continues recursively until all nodes in a quadrant have the same color code.

Figure A.8 illustrates the quad-blocks generated on the road network in Figure A.4. We note that the leaf nodes of VQ-tree does not store any information about the network nodes. As shown in Figure A.9, the leaf nodes only store the region information (i.e., coordinates) of the quad-blocks as well as a single value (e.g., a color code or an integer number) which indicates the identity of the network Voronoi cell of which the quad-tree block is a member. We note that a leaf node in the quad-tree corresponds to a particular subdivision of a network Voronoi cell.

As shown in A.8, each network Voronoi cell \( NVC \) consists of disjoint quad-tree blocks. The disjoint decomposition of the network Voronoi diagram with VQ-tree

\(^2\)During NVD construction parallel Dijkstra algorithm can encode each node with a Voronoi cell identifier, e.g., a color
Figure A.9: VQ-tree

addresses the two drawbacks of VR-tree. Specifically, unlike VR-tree that roughly estimates the network Voronoi cells with polygons in the Euclidean space, VQ-tree enables the exact representation of the network Voronoi cells using quad-tree blocks and hence always yield correct results. VQ-tree does not suffer from the backtracking problem of VR-tree, and hence fast response time for \( \text{contain}(q) \). This is due to non-overlapping partitioning of the network Voronoi cells: once the quad-tree block containing \( q \) is located in the leaf nodes, VQ-tree immediately identifies the nearest Voronoi generator based on the value (e.g., a color code) of that block.

Algorithm 6 presents the outline for VQ-tree. Given a set of \( N \) nodes with their color codes and bounding box \([x_1; x_2] \times [y_1; y_2]\) that contains \( N \) as an input, Algorithm 6 creates VQ-tree by recursively splitting the quadrants until all the nodes in a quadrant have the same color code.
Algorithm 6: VQ-Tree Algorithm

\[
VQuadTree(N, x_1, x_2, y_1, y_2) \{
/* Scan distinct color codes in the region
\text{cellColor}[] \leftarrow \text{checkRegion}(N, x_1, x_2, y_1, y_2);
/* If there exist more than one color-code then split
\text{if} \\text{cellColor.length} > 1 \text{ then}
/*Initialize intermediate node
\text{node} \leftarrow \text{QuadTreeNode}();
/*Set Quadrants
\text{node.SE} \leftarrow VQuadTree(N, x_1, (x_2+x_1)/2, y_1, (y_1+y_2)/2);
\text{node.SW} \leftarrow VQuadTree(N, (x_2+x_1)/2, x_2, y_1, (y_1+y_2)/2);
\text{node.NE} \leftarrow VQuadTree(N, x_1, (x_2+x_1)/2, (y_1+y_2)/2, y_2);
\text{node.NW} \leftarrow VQuadTree(N, (x_2+x_1)/2, x_2, (y_1+y_2)/2, y_2);
\text{else}
/*Create leaf node
\text{QuadTreeLeafNode(cellColor[0]);}
\text{end if}
\}

A.3 Performance Evaluation

We conducted experiments with different spatial networks and various parameters to evaluate the performance of VQ-tree and VR-tree. We measured the ratio of false-negative edges with varying object cardinality (i.e., number of Voronoi generators) and object distribution in the road network. In addition, we compared the precomputation, index rebuilding (for dynamic environments) and response time of VQ-tree and VR-tree with respect to different network sizes and object cardinality. As of our dataset, we used California (CA), Los Angeles (LA) and San Joaquin County (SJ) road network data (obtained from Navteq [42]) with approximately 1,965,300, 304,162 and 24,123 nodes, respectively. Since the experimental results with LA and SJ networks differ insignificantly, we only present the results from the CA and LA datasets. We conducted our experiments on a workstation with 2.7 GHz Pentium Core Duo processor and 12GB
RAM memory. For each set of experiments, we only vary one parameter and fix the remaining to the default values in Table 1.

Table A.1: Experimental parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Cardinality</td>
<td>100</td>
<td>10,50,100,500,1000</td>
</tr>
<tr>
<td>Road Network</td>
<td>LA</td>
<td>SJ, LA, CA</td>
</tr>
<tr>
<td>Object Distribution</td>
<td>Uniform</td>
<td>Uniform, Gaussian</td>
</tr>
</tbody>
</table>

**Ratio of False-negative Edges**

First, we study the ratio of false-negative edges with respect to object cardinality (i.e., number of Voronoi generators) and object distribution. To identify false-negative edges, we compare the encoded values (i.e., color code) of each node based on VR-tree and VQ-tree. Specifically, we first encode each edge to its corresponding Voronoi generator by using VR-tree polygons and then compare the encoded values to that we obtained from VQ-tree. We repeat each experiment 100 times and report the average number of incorrectly encoded (i.e., false-negative) edges with respect to total number of edges in the network. Figure A.10(a) shows the ratio of false-negative edges of both networks where the object cardinality ranging from 10 to 1000. As illustrated, the ratio of incorrectly identified edges is %16 on average in both networks. The maximum recorded false-negative edge ratio for LA and CA road networks is %24 and %29, respectively.

Figure A.10(b) illustrates the ratio of false-negative edges with different object distribution for both CA and LA road networks. We observe that the number of false-negative edges is less in Gaussian distribution. This is because as objects are clustered in the spatial network with Gaussian distribution, the corresponding shortest path trees would be less disperse and hence spatially close border points. As mentioned, with VR-tree we encode the edges based on the Euclidean polygon generated by connecting the border
points. The more spatially close border points provides the more accurate presentation of the NBCs and hence less false-negative edges.

![Graph](image1.png)

(a) Impact of object cardinality

![Bar graph](image2.png)

(b) Impact of object distribution

Figure A.10: Impact of object cardinality and distribution

**Precomputation Time**

With another set of experiments, we compare the precomputation (i.e., index construction) time of VR-tree and VQ-tree with varying network sizes and number of objects. In order to evaluate the impact of network size, we conducted experiments with the sub-networks of CA dataset ranging from 50K to 250K segments. We set the the node size of VR-tree to 4K bytes in all cases. Figure A.11(a) shows the precomputation time of VQ-tree and VR-tree in CA road network with varying network size. The results indicate that the precomputation time increases with the network size in both methods where VQ-tree outperforms VR-tree with all numbers of edges. This is because as the network size increases the perimeters of the polygons (and hence the number of connected line segments that form a polygon) grow in VR-tree. Arguably, the overhead of generating MBRs (to be used in VR-tree) around the polygons composed of numerous connected line strings is time-consuming as the coordinates (that form the lines) needs to be scanned to find the ultimate corners of the MBR. On the other hand, VQ-tree is
constructed based the underlying space (rather than objects in VR-tree) by recursively dividing the road network to quad-blocks each corresponding to one NVC.

Figure A.11(b) illustrates the impact of object cardinality over precomputation time in LA road network (the results are similar in CA network and hence not presented). We observe that as the number of objects in the road network increases, the preprocessing time for both approaches increases. As shown, the precomputation time for VQ-tree outperforms VR-tree. The reason is that the time for hierarchically clustering polygons in VR-tree for a large datasets is relatively expensive. We also observe that the depth of VQ-tree increases with the increasing number of data objects. This is because large number of data objects yields smaller VCs and hence more splits.

![Figure A.11: Impact of network size](image)

(a) Impact of network size  
(b) Impact of object cardinality

**Index Reconstruction**

Next, we compare the index reconstruction overhead of VR-tree and VQ-tree with respect to object updates. In this set of experiments, we update the location of the randomly selected data objects and measure the index reconstruction overhead in both VR-tree and VQ-tree. Figure A.12(a) shows the index reconstruction time of both index structures with varying object update ratio (i.e., the percentage of data objects whose locations changed). We observe that VQ-tree outperforms VR-tree with respect to index
reconstruction. This is because the insert operations in VR-tree are expensive. When new data objects are inserted into VR-tree, besides updating leaf nodes, it is likely that updates are also required to non-leaf nodes (i.e., more than one branch of the tree may be expanded), which leads to a large overhead during insertion. On the other hand, with VQ-tree we observe that most of the index updates take place in the leaf nodes.

![Graphs showing response time vs object cardinality and index reconstruction](image)

(a) Index reconstruction  (b) Impact of object cardinality

Figure A.12: Response time vs object cardinality and Index reconstruction

**Response Time**

In this experiment, we compare the performance (i.e., the response time for containing($q$) query) of VQ-tree and VR-tree with varying object cardinality. We determine the location of the query object $q$ uniformly at random and report average of 100 queries. As we mentioned the original VR-tree proposed in [34] may yield inaccurate results. In order to provide correct results with VR-tree, we modify VR-tree by adding an additional index structure that maintains false-negative edges. Specifically, we construct an R-tree on the false-negative network edges along with their Voronoi generators. With each containing($q$) query, we check $q$ against this index structure. If we locate $q$ on any of the false-negative edges, the corresponding data object is returned as the first NN. Otherwise, VR-tree continues the search based on the polygons explained in A.2.2. Figure A.12(b) plots the average response time for containing($q$) query. The results indicate that
VQ-tree outperforms VR-tree with all data objects and scales better with large number of data objects. The response time of VQ-tree is approximately 12 times better than that of VR-tree with more than 200 data objects. This is because of the overlapping MBRs of network Voronoi cells. With VR-tree the amount of work often depends on the size of the overlapping areas. In particular, the overlapping areas may belong to more than one NVC and hence during the search the parent node(s) of the overlapping MBRs have to be accessed repeatedly. Moreover, with each \textit{containing}(q) query VR-tree performs an additional step to check if \( q \) is located on false-negative edges; it seems that the overhead of this operation is not negligible.