In Search of Points of Interest: A Story of Decoupled Heuristics on Road Networks

By

Tenindra Abeywickrama

Thesis

Submitted in Fulfillment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisor: Prof. Muhammad Aamir Cheema

Co-Supervisor: Prof. David Taniar

Faculty of Information Technology, Clayton

Monash University

February 2019

I dedicate this thesis to my loving parents who value knowledge above all...

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Tenindra Abeywickrama

June 24, 2019

Abstract

Today is the era of ubiquitous mobile computing, heralded by the massive proliferation of GPS-enabled smartphones. The smartphone combined with cheap network bandwidth has seen *map-based services* become an essential part of daily life for many people. The key to many of these map-based services is to efficiently locate the nearest points of interest (POIs) through the road network. For example, a ride-hailing application locates the nearest drivers to a user's location by their road network travel time.

Consequently, POI search in road networks has recently become a popular area of scientific endeavor. However, we find that heuristics used to guide POI search have not been carefully considered. We first revisit a simple Euclidean heuristic in a thorough experimental investigation into the state-of-the-art for the road network k Nearest Neighbor (kNN) query. To our surprise, with a simple improvement, we find this long-forgotten heuristic outperforms the state-of-the-art. Moreover, we find that the Euclidean heuristic can even replace and improve some of the more complex dedicated heuristics used by state-of-the-art techniques (namely the G-tree and Distance Browsing techniques).

This surprising observation forms the thread we weave through our study. We identify the simple Euclidean heuristic as an example of a broader family of *decoupled heuristics* that use a separate technique to compute network distances. This paradigm has several underrated benefits, for example being able to easily leverage 60-years of research into network distance computation. We confirm that Euclidean distance is a less effective heuristic in certain scenarios (like travel time), making its superior performance on kNN queries in that setting even more remarkable. Motivated by this, we propose alternative decoupled heuristics based on landmarks and techniques to overcome the challenges in using them efficiently, to further improve kNN querying.

We show that the decoupled heuristic paradigm has broad applications in POI search by creating a framework to answer spatial keyword queries. In doing so, we can overcome

the disadvantages experienced by all other techniques for road network spatial keyword queries from the use of *keyword aggregation*. We also show that, while the paradigm itself is simple, it can incorporate more complex heuristics. This is demonstrated through our more sophisticated heuristic for efficient Aggregate kNN querying, which exhibits different properties to regular kNN queries due to the presence of multiple query locations. Across the board, our techniques are orders of magnitude faster than the state-of-the-art on a vast majority of experimental settings and real-world datasets. Ultimately, we provide strong evidence for a better direction in heuristic development for POI search.

Acknowledgments

It has been a long, enlightening, perilous, rewarding, grinding roller-coaster of a journey. I would like to thank all the people that have helped make this Ph.D. the life-changing journey that it has been. First and foremost, my mum and dad, who have been unwavering in their support and have always been there to catch me when I fall. Having you both behind me has made me stronger and saying thank you does not do justice to how I feel.

Having a mentor that both cares about you and understands you is a precious and rare thing. For that, I am very thankful to my supervisor, Prof. Aamir Cheema. I cannot imagine a kinder and more emotionally supportive Ph.D. supervisor. After each time I discussed something with you, be it work or personal, it felt like all was right in the world again. Thank you for giving me this opportunity and for all the effort that you have put into getting me to where I am. I hope I did you proud.

I am thankful for my co-supervisor Prof. David Taniar for always being kind and encouraging. I am indebted to Prof. Arijit Khan for giving me the opportunity to work in Singapore and greatly appreciate the continued support you have shown even after I left. I credit Dr. John Shepherd at UNSW for inspiring me to pursue a Ph.D. in databases through his teaching. I would not be here without his recommendation to Prof. Cheema.

But before I was inspired to do a Ph.D., I was inspired to change myself. I am grateful to my old boss Lester Munro for his mentorship and wise advice. My time in your team was probably the most pivotal in my professional life. Your pearls of wisdom like "Tenindra, if you kill time, time will kill you", spurred me to find a new direction, which led me to a Ph.D. and the most enriching experience of my life.

I am grateful to my colleagues from Monash who shared this journey with me. Special mention to Nader Chmait, Srinibas Swain and Shenjun Zhong for the fun hangouts, decompression sessions, and lunchtime chats. Thank you to Agnes Haryanto, Arif Hidayat and everyone else from the Monash Spatial Group for being such a pleasant group to work

alongside. I'm glad to have met everyone (past and present) from Room 140. I really appreciate the empathetic and caring support of the graduate student coordinators, Helen Cridland, Aidan Solla, and Danette Derianne, without whom I would not have had such a smooth ride through all the processes. Helen in particular moved mountains to help me get my Hong Kong fellowship. I also want to recognize Dr. John Betts and Dr. Daniel Harabor from my academic panel for always appreciating my hard work and giving thoughtful feedback during my milestone seminars. My research outputs (and paying my bills) were made possible by an Australian Government RTP Scholarship.

Finally, I would like to thank my awesome friends outside academia for their moral, emotional and often material support. I am very grateful to Yasanthi and Mark Drover and family for always making me feel like part of the family. You guys will always be my Melbourne family and I'll miss red curry salmon risotto! I am especially thankful to Vas Withanage and Iresha Perera, for making me feel so welcome in Canberra and visiting me to share some big moments in Melbourne and Hong Kong. I am glad Jonathan and Rajita Jayanthakumar were in Singapore during my internship (my first time living overseas). I don't think I would have stuck it out if not for our hangouts! I am also thankful to Lisa Wu and Jonathan Li for all the fun times while on my Hong Kong fellowship. I can't wait to check out Ocean Park again! And to my other friends who checked up on me or visited Melbourne, it always brightened up my week.

This journey would not have been possible without you all, and most definitely would not have been nearly as enjoyable. I will end by quoting another pearl from Lester Munro. "Everyone brings me joy, Tenindra, some as they arrive, others as they leave". So even if I didn't mention you, rest easy knowing that you have surely brought me joy! Jokes aside... So long, and thanks for the Ph.D.!

Publications

Below is the list of publications and manuscripts arising from this thesis.

Published Works

- Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. *k-Nearest Neighbors on Road Networks: A Journey in Experimentation and Implementation*, appeared in the International Conference on Very Large Databases (**VLDB**) 2016.

- Tenindra Abeywickrama and Muhammad Aamir Cheema. *Efficient Landmark-Based Candidate Generation for kNN Queries on Road Networks*, appeared in the International Conference on Database Systems for Advanced Applications (**DASFAA**) 2017.

- Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. *k-Nearest Neighbors on Road Networks: Euclidean Heuristic Revisited (Extended Abstract)*, appeared in the Symposium on Combinatorial Search (**SoCS**) 2016.

- Tenindra Abeywickrama, Muhammad Aamir Cheema, Arijit Khan. *K-SPIN: Efficiently Processing Spatial Keyword Queries on Road Networks*, to appear in IEEE Transactions on Knowledge and Data Engineering (**TKDE**) 2019.

Other Manuscripts

- Tenindra Abeywickrama, Muhammad Aamir Cheema. *Hierarchical Graph Traversal for Aggregate kNN Search in Road Networks*, To Be Submitted

- Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. *k-Nearest Neighbors on Road Networks: A Journey in Experimentation and Implementation (Technical Report)*, arXiv preprint arXiv:1601.01549 2016.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

> Research is to see what everybody else has seen and to think what nobody else has thought.
>
> Albert Szent-Gyorgyi

We are in the age of ubiquitous mobile computing, heralded by the massive proliferation of GPS-enabled devices. According to market research, over 3 billion smartphones were in use by the end of 2018 [New18]. This omnipresence of smartphones combined with cheap network data has driven a surge in the adoption of *map-based services* in day-to-day life. For example, according to Pew Research, 90% of smartphone users in the US utilize map applications such as Google Maps [And16]. These map-based services perform a wide range of roles in our daily activities, such as getting us from point A to B, helping us find the nearest Thai restaurant, or hailing a ride-sharing vehicle. Consequently, the efficiency, quality, and capabilities of the algorithms and data structures that enable, and support map-based services have been an important area of scientific research.

The key to many map-based services lies in issuing queries related to the road network. For example, an on-demand transport app, like the one shown in Figure 1.1, needs to locate the nearest drivers to a user's location by the estimated time of arrival. This, in turn, depends on the travel time of the fastest road network route from each available driver. To solve these problems algorithmically, the road network is most commonly represented as a graph like the example depicted in Figure 1.2. We define the road network graph more formally shortly, but using the graph representation of a road network allows the fastest route and its travel time to be computed through graph algorithms such as Dijkstra's famous shortest path algorithm [Dij59].

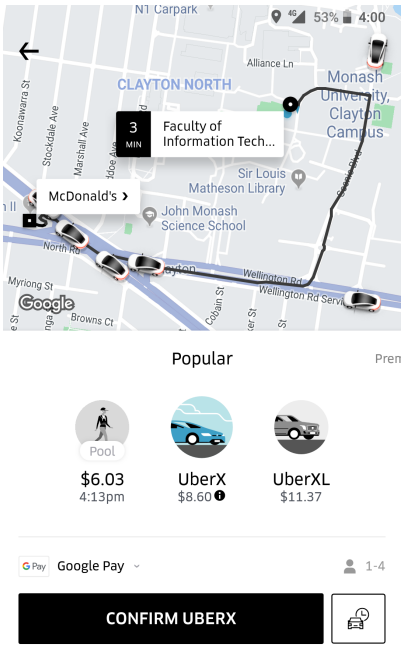


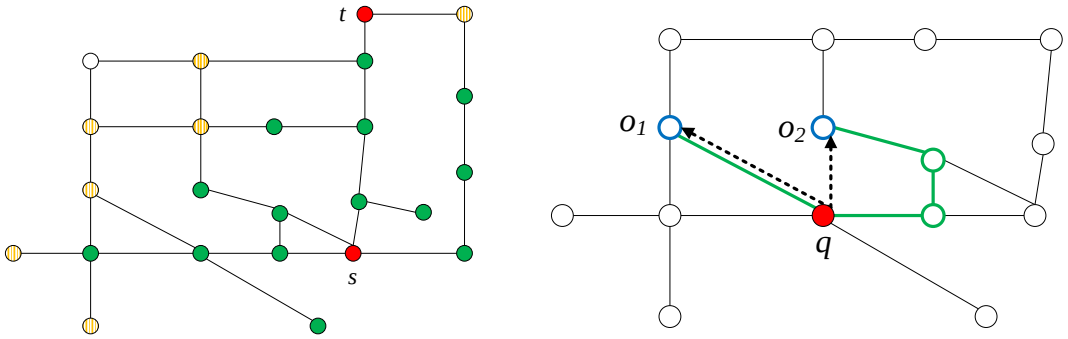
Figure 1.1: Uber: An On-Demand Ride-Hailing Application

These shortest path *queries* have received a significant amount of attention, with 60-years of research resulting in an array of increasingly advanced techniques. More recently, the problem of finding points of interest (POIs) in the road network has spawned a wide variety of new queries, which have not received nearly as much attention. These queries have arisen specifically to meet the needs of map-based services, such as on-demand transport, which cannot be met by traditional Euclidean POI search methods.

Given the incredible growth in demand for map-based services and the huge throughput required to meet it, any new technique must be as efficient as possible. We model the road network as a graph and provide an overarching formal definition in Section 1.1. In Section 1.2, we describe the challenges faced in pursuing this goal and briefly introduce some popular POI search queries. In Section 1.3, we introduce the key insights that form the backbone of our hypotheses. Section 1.4 presents the main contributions made by this thesis to POI search. Lastly, Section 1.5 outlines the thesis organization.

1.1 Road Network: A Formal Definition

We use the definition of a road network as a connected undirected graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. Vertices in V generally represent real-world intersections and the edges in E represent the road segments connecting vertices. Each



(a) Vertices Visited by Dijkstra (b) POIs by Euclidean vs. Network Distance

Figure 1.2: Example Road Network Graph

edge $(u, v) \in E$, where $u, v \in V$ are adjacent vertices, is also associated with a weight $w(u, v) \in \mathbb{R}_{>0}$ representing the length of the road segment between its endpoints. In fact, this "length" may not represent physical distance at all but could be one of a multitude of highly useful metrics, e.g., the time taken to traverse the edge or the toll cost. This model allows the application of graph algorithms to solve real-world user queries, such as using Dijkstra's algorithm [Dij59] to answer shortest path queries. The shortest path $P(u, v)$ between vertices $u, v \in V$ represents the sequences of edges connecting u and v with the minimum sum of weights. The shortest path distance, or *network distance*, $d(u, v)$ represents the sum of the weights of the edges in the shortest path $P(u, v)$. Figure 1.2 depicts two sample road networks. In the case of Figure 1.2(a), there are 24 vertices indicated by the circles and 28 bi-directional edges indicated by straight lines connecting the circles. Note that a single line represents a bi-directional edge, i.e., implying both (u, v) and (v, u). We will use this definition of road networks throughout our study.

1.2 Motivations

Let us consider, for the moment, the shortest path distance query. This query computes the *network distance* between a source vertex s and destination vertex t in a graph G with $s, t \in V$. That is, it computes the minimum sum of weights over all sequences of edges in G connecting s and t. This sequence is also known as the shortest path. One approach is to simply use Dijkstra's classic algorithm, which will compute a result in $O(n \log n)$ time (where $n = |V|$). However, this involves visiting every vertex that is closer to the source than the destination, resulting in unacceptable response times of up to tens of seconds

[Wu+12]. Let us consider the example road network in Figure 1.2(a). Assuming unit edge-weights, using Dijkstra's search to find the shortest path from vertex s to vertex t involves visiting *at least* the green (solid colored) vertices, and potentially all the orange (vertical striped) vertices as well, before the search can terminate. In contrast, if we pre-compute the distance between every pair of vertices in the road network in an offline pre-processing phase, the online query can be answered in $O(1)$ time. But this entails the creation of an index typically requiring $O(n^2 \log n)$ pre-processing time and $O(n^2)$ space. To put this into perspective, for the continental US road network where the number of vertices n is 24 million[1], this is estimated to consume 1.2 petabytes in storage [Bas+15] and take 45 days to compute using a parallelized algorithm on a server with 48 cores [Del+11]. The challenge faced by researchers following Dijkstra has been to develop an indexing method with the ideal compromise between these two extremes, minimizing both pre-processing cost and query time.

Naturally, this indexing dilemma is similarly faced by POI search techniques in road networks as they also involve computing network distances to POIs like ride-sharing vehicles or bottle shops. In fact, searching for POIs involves additional challenges. Not only must we compute the network distance to target POIs efficiently, but we must also only do so to the POIs that are actual results, potentially among thousands of non-result POIs. For example, it is easy to imagine multiple target vertices in Figure 1.2(a) resulting in even greater numbers of vertices being visited. To avoid this problem, applications may consider using POI search techniques in Euclidean space such as R-trees [Gut84]. However, Euclidean distance (i.e., "as the crow flies") cannot support other metrics like travel time, which is a more accurate measure of a POI's proximity to a user. Let us demonstrate using Figure 1.2(b). Assume the edge-weights for this example road network are physical distance and are drawn to scale. Imagine vertices o_1 and o_2 are gas stations near a user at vertex q. o_2 is closest by Euclidean distance (dotted lines with arrows). However, by network distance (thick edges) o_1 is clearly closer. While Euclidean distance may be sufficient for some applications, in an age where users are demanding greater accuracy from a crowded marketplace and in extremely time-sensitive applications like on-demand transportation where time costs money, this is simply not possible. Moreover, POI search in road networks is a relatively new area of research and has not received the same amount

[1] According to datasets provided by the 9th DIMACS Challenge found at http://www.dis.uniroma1.it/%7Echallenge9/, which are based on data collected by the U.S. Census Bureau

of attention as shortest path computation. Hence these increasingly important queries are the subject of our research.

1.2.1 Types of POI Search Queries on Road Networks

In this section, we describe three prominent types of POI search queries that we study in this thesis and highlight their importance in the context of map-based services.

*k*NN Queries on Road Networks

A k Nearest Neighbor (kNN) query returns the k nearest objects (POIs) to a query location by network distance. For example, a user may wish to find the nearest Thai restaurants or a ride-hailing app may want to locate the nearest driver to the user's location. Formally, given a query vertex q and a set of object vertices $O \subseteq V$, a kNN query retrieves the set of k objects in O closest to q by their network distances. Figure 1.3 shows our running example road network with four object vertices o_1 to o_4 with thick borders. Assuming unit edge-weights, a 2-NN query issued from query vertex q would retrieve o_4 and o_2 with network distances 2 and 3, respectively.

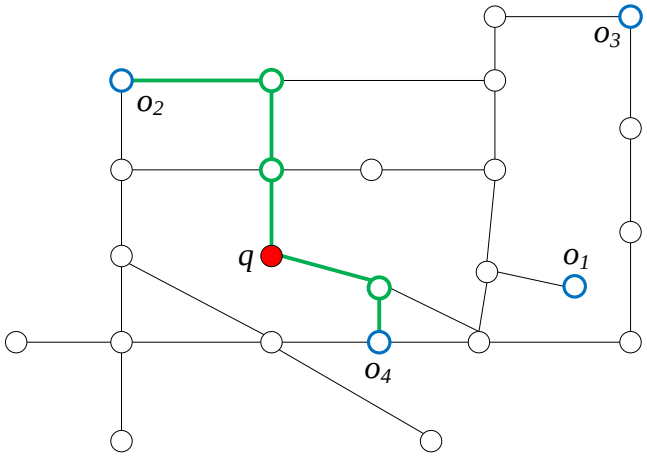


Figure 1.3: k-Nearest Neighbors on a Road Network

Spatial Keyword Queries on Road Networks

POIs are often associated with rich textual descriptions. In this setting, users want to locate POIs by their textual similarity to their desired keywords as well as the spatial proximity to their location. To meet this need, a Boolean kNN (BkNN) query retrieves

the nearest POIs by network distance whose keywords satisfy some Boolean logic, e.g., POIs that contain *all* query keywords in conjunctive BkNN queries. On the other hand, users may be willing trade-off proximity for similarity (and vice versa). This scenario is handled by top-k spatial keyword queries, which retrieve POIs ranked by a score that combines their network distance and textual similarity. In this way, a user can choose a less relevant POI to their desired keywords if it is close enough to their location. For example, a user may search for a POI that matches keywords "Thai, restaurant, delivery", but may be willing to compromise on keyword "delivery" if the restaurant is close enough to travel to.

Aggregate kNN Queries on Road Networks

While the queries described so far involved one user at a single location, POI search is not necessarily limited to this setting. Aggregate k Nearest Neighbor (AkNN) queries involve multiple users at different locations and retrieve POIs by their *aggregate distance*. This distance is computed by aggregating the network distances to the POI from each user according to some aggregate function. For example, a group of friends may wish to locate the pub that minimizes the maximum travel time for the group. Specifically, given a set of query vertices $Q \in V$, a set of object vertices $O \in V$, and an aggregate function agg, an AkNN query retrieves the k objects in O which minimize the aggregate network distance from each query vertex $q \in Q$ to the object according to function agg.

1.3 Key Insights

Let us frame finding POIs as a heuristic search problem. Then, our goal is to search the road network graph for POIs matching the user's query, e.g., the nearest by network distance to the user's location. There are two factors that affect the performance of this search. First, it is affected by the efficiency of computing network distances to POIs that we find. Second, it depends on the efficiency of the heuristic, i.e., the ability of the heuristic to find POIs that are actual results and avoid those that are not. These two factors combine to give the overall performance. Clearly, an inaccurate heuristic has a compound effect on overall performance, as it results in wasted network distance computations. We observe that past work on POI search has not considered the interplay

between these factors in-depth. In particular, we find that heuristic efficiency has not been thoroughly considered or investigated. For example, G-tree [Zho+15; Zho+13], one of the state-of-the-art techniques to answer kNN queries, proposed a novel and interesting heuristic using the minimum distance to subgraphs in its road network index without comparing the effectiveness against simpler heuristics. Motivated by this observation, the focus of our study is to first understand heuristic efficiency in POI search and then to develop better heuristics for popular POI search queries.

A better understanding of heuristic performance leads us to a second key insight. One strategy to solve POI search problems is to decouple the heuristic from network distance computation. As it turns out, even using this strategy with simple heuristics like Euclidean distance often outperforms more complex state-of-the-art heuristics (as detailed in Chapter 3). Decoupling network distance computation also enables us to leverage the 60-years of shortest path research that has produced exceedingly fast techniques. Moreover, the effectiveness of these *decoupled heuristics* has a greater likelihood of passing the test of time as any future improvements to network distance computation can be easily incorporated. We show that this simple paradigm forms a viable and effective platform to develop new indexing and query processing techniques to efficiently answer a wide variety of POI search queries. Moreover, this is achieved in a manner where heuristic efficiency can be quantitatively studied, as we show in our experimental comparisons.

1.4 Contributions

We now outline the contributions made by this thesis to several types of important POI search queries including the associated research outputs in terms of publications.

1.4.1 *k* Nearest Neighbor Queries

In our review of past work on k Nearest Neighbor (kNN) queries on road networks, we found that the efficacy of a long-forgotten technique [Pap+03] using a simple decoupled POI search heuristic using Euclidean distance had never been properly investigated. Motivated further by other discrepancies and gaps in past experimental comparisons, we present an in-depth experimental comparison of state-of-the-art kNN techniques. While making numerous improvements, we implemented all methods from scratch to obtain as

fair a comparison as possible. To our surprise, the neglected technique based on a simple Euclidean heuristic outperformed the state-of-the-art on most experimental settings. The overarching implication of this result, and the driving force behind this thesis, is that existing heuristics are not as effective as they could be. We also observed the drastic effect implementation in main memory has on query performance. We demonstrate how a bad implementation can make it impossible to determine which heuristics work better. In response, we provide guidelines for future implementers to avoid common pitfalls and produce more reliable experimental results. In fact, these observations apply to all query processing techniques, not just POI search. This work [ACT16b] has been peer-reviewed and published in the *Proceedings of the VLDB Endowment* (**PVLDB**) 2016. An extended version [ACT16a] with additional experiments and full documentation of our improvements made to each method has been uploaded to arXiv. An extended abstract based on the PVLDB paper, written from the perspective of the AI heuristic search community, was published in the Annual Symposium on Combinatorial Search (**SoCS**) 2018 [ACT18].

We identify the simple Euclidean heuristic mentioned above as part of a broader class of *decoupled heuristics*, which use a heuristic to retrieve candidate POIs and another technique to compute their network distances. In the case of the neglected technique, Euclidean distance is used as a lower-bound network distance to retrieve candidate objects until all kNNs are found. One of the reasons its performance was so surprising is because Euclidean distance is not an accurate lower-bound for network distance, e.g., when travel time is concerned [GH05]. To improve on this, we propose a different heuristic using tighter lower-bounds based on landmarks and POI candidate generation using a partitioning of the road network. In our extensive experiments we show that the proposed heuristic outperforms the Euclidean method both in terms of traditional query performance and heuristic efficiency. This research was peer-reviewed and appeared in the *International Conference on Database Systems for Advanced Applications* (**DASFAA**) 2017 [AC17].

1.4.2 Spatial Keyword Queries

As discussed earlier, POIs are often associated with rich textual descriptions. Spatial keyword queries search for POIs by considering both spatial proximity and textual relevance to a set of keywords specified by the user. We observe that all existing spatial keyword

query techniques on road networks use a keyword aggregation strategy that is disadvantageous on road networks. Inspired by the decoupled heuristic paradigm, we present a flexible framework using a *keyword separation* approach combined with a novel heap data structure that delays and avoids expensive operations. We also propose new techniques including a new data structure to ease the impractical pre-processing costs normally associated with keyword separation, resulting in an index that is viable and even lightweight. Our query algorithms are up to two orders of magnitude more efficient than the state-of-the-art. Our in-depth experimental investigation shows that this is true for various spatial keyword queries, parameter settings, and real-world road network and keyword datasets. This work has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering (**TKDE**) 2019 [ACK19].

1.4.3 Aggregate k Nearest Neighbor Queries

Aggregate k Nearest Neighbor (AkNN) queries retrieve POIs by aggregating the network distances from a group of users according to some aggregate function. Naturally, kNN query results will be in the vicinity of the single user's location and expansion-based heuristics like the one we propose in Chapter 4 are highly effective based on this intuition. However, this assumption is not necessarily true for AkNN query results due to multiple users being present. The results are unlikely to be in the vicinity of any single user, and a hierarchical approach is more intuitive. We propose a decoupled heuristic based on hierarchically traversing subgraphs of the road network by accurate landmark-based lower-bounds to efficiently answer AkNN queries. Moreover, we observe an interesting property of our COLT data structure for convexity-preserving aggregate functions that allows us to significantly improve heuristic efficiency and consider fewer candidates. The result is an orders of magnitude improvement in both query performance and heuristic efficiency, as demonstrated in our detailed experimental investigation. In the wider context of the decoupled heuristic paradigm, we show that it is possible to further improve POI search by incorporating more sophisticated heuristics in the simple paradigm, like the one we propose.

1.5 Thesis Organization

The organization of the remainder of this thesis is outlined below.

- **Chapter 2** reviews the literature related to road network algorithms, including shortest path computation and POI search.
- **Chapter 3** covers our experimental investigation into state-of-the-art k Nearest Neighbor query techniques on road networks [ACT16b; ACT16a].
- **Chapter 4** presents an improved heuristic to find kNN results based on landmark lower-bounds [AC17], directly influenced by our surprising findings regarding decoupled heuristics in Chapter 3.
- **Chapter 5** describes our flexible K-SPIN framework [ACK19] which leverages the decoupled heuristic paradigm and an improved indexing strategy for keywords to efficiently answer various spatial-keyword queries on road networks.
- **Chapter 6** presents a more sophisticated decoupled heuristic to answer aggregate kNN queries using a novel data structure for efficient hierarchical traversal of the road network graph [AC].
- **Chapter 7** summarizes our findings, provides the overarching conclusions of our work, and some potential directions for future research.

Chapter 2

Literature Review

> The quest for certainty blocks the search for meaning. Uncertainty is the very condition to impel man to unfold his powers.
>
> Erich Fromm

In this chapter, we review the existing work in the literature that is most relevant to our study. Specifically, we first provide a brief overview of the developments in shortest path computation in Section 2.1, where road network indexing techniques originated. Then in Sections 2.2 to 2.4, we describe techniques related to each query we study in this work, as introduced in Section 1.2.1. Finally, we briefly survey some of the other types of POI search queries in Section 2.5.

2.1 Shortest Path Computation on Road Networks

Computing shortest paths in graphs is a fundamental problem closely related to point of interest (POI) search in road networks. It all began with the development of Dijkstra's algorithm [Dij59] to answer *single source shortest path* (SSSP) queries. An SSSP query finds shortest paths to *all* vertices in the graph G from a source vertex q. The algorithm works by using a priority queue to store vertices "seen" so far, with the queue element key being a tentative network distance from the source q. This queue initially contains the query vertex q with a key of 0. In each iteration the vertex u with the smallest key is dequeued first, with the key value necessarily being the network distance to u. Now this distance from q to u is used to "relax" the neighboring vertices of u. That is, if a neighbor vertex v is in the queue then its key is updated if the distance to v through u is smaller than the current key. If v is not in queue, then a new element for v is inserted

with the distance from q to v through u. Queue elements are also associated with the predecessor vertex used to compute the tentative network distance, in order to retrieve the shortest path. But predecessor vertices can be omitted if we are only interested in computing network distances.

In the context of map-based services and road networks, we are more interested in the point-to-point (P2P) shortest path query. A P2P query involves computing the shortest path from a source vertex to a single destination vertex, e.g., to find the shortest route from our home to our favorite restaurant. Dijkstra's algorithm is adapted for this problem by simply terminating when the target vertex is dequeued. However, as discussed in Section 1.2, this involves visiting every vertex that is closer to the source than the target. This still leads to a very large search space (as described earlier using Figure 1.2(a)). Moreover, this search space increases rapidly as the source and target become further apart. Consequently, the worst-case query time is $O(|V| \log |V|)$, depending on the type of priority queue used and since typically $|E| = O(|V|)$ in road networks. Modifications have been proposed to reduce the search space, for example, bi-directional Dijkstra's search [Poh71] that simultaneously searches forward from the source and backward from the target (but on the reversed version of the graph) or A* Search [HNR68] which uses Euclidean distance to prune vertices that cannot be on the shortest path.

2.1.1 Road Network Properties and the Indexing Trade-Off

Even with improvements to the search space, Dijkstra's algorithm is not efficient enough for real-world P2P shortest path queries for use in map-based services. For example, a user is unlikely to tolerate tens of seconds to get a route as reported by Wu *et al.* [Wu+12]. Moreover, it is a generalized graph algorithm, not taking advantage of the unique properties of road networks compared to other graphs. For example, there is a natural embedding of vertices in Euclidean space, as each vertex corresponds to a real-world location. Other examples include near-planarity as tunnels and overpasses are not common and a small average degree as most road intersections tend do have 4 or fewer outward edges. As a result, *road network indexes* that exploit such properties have been created to answer shortest path and other queries more efficiently. The essential idea is to use some property to create a "summarized" data structure (an index) that can

be interrogated more efficiently to answer queries. Thus, an index trades *offline* pre-processing space and time for faster *online* query processing. However, this idea is not without its drawbacks and introduces new challenges in finding an ideal trade-off.

A particularly interesting example is the use of the "path coherence" property in the Spatially Induced Linkage Cognizance (SILC) index by Sankaranarayanan *et al.* [SAS05], which is among the fastest shortest path query techniques [Wu+12]. This technique exploits the idea that traveling from a source location to any location among a group of destination locations near each other, will involve traversing much of the same paths. Unfortunately, SILC exemplified the potential disadvantages of the indexing trade-off, with massive pre-processing costs in terms of both time and space. Due to this cost, the SILC index has never been built for road networks with more than 1 million vertices [SSA08], which eliminates the possibility of using it on continental sized road networks. Path Coherent Pairs Decomposition (PCPD) [SSA09] was an evolution of the SILC index with $O(n)$ space cost (versus $O(n^{1.5})$ for SILC). However, in reality, PCPD possessed an even larger index than SILC in practice due to the presence of hidden constants based on inconsistent assumptions on road networks [Wu+12]. We will discuss SILC in greater depth in Section 2.2 as it was later extended to answer kNN queries. In any case, SILC and PCPD serve to highlight the key challenges involved in creating road network indexes for map-based services.

2.1.2 Road Network Indexing Techniques

Numerous other road network indexes have been proposed over the last three decades. However, no method has been able to achieve a trade-off that minimizes the pre-processing cost while maximizing the query performance [Bas+15]. Among current state-of-the-art techniques, Contraction Hierarchies (CH) [Gei+08] remains extremely popular. CH works by first imposing a total order on vertices in the road network graph G based on some heuristic measuring "importance". Each vertex is "contracted" one-by-one in order of increasing importance. Contracting a vertex first involves temporarily removing the vertex from the graph. Then *virtual edges* or shortcuts are inserted between the neighbors of the removed vertex if a shortest path between them passed through the removed vertex. For example, in Figure 2.1, let us assume the vertices are numbered in order of increasing importance. v_1, v_2, v_3, and v_4 are iteratively contracted without adding shortcuts as no

shortest paths go through them. Next, v_5 is contracted but the shortest path from v_6 to v_9 passes through v_5, resulting in the addition of a shortcut between v_6 and v_9 (indicated by the blue dashed line). Note this shortcut will have weight equal to the shortest path distance through v_5. Geisberger *et al.* propose several heuristics for ordering vertices to reduce the number of shortcuts added. One example is that vertices with high degree are likely to be involved in a larger number of shortest paths, and therefore are more important. Afterwards, the P2P query algorithm on the CH index proceeds in a similar manner to bi-directional Dijkstra's search, except the search only expands to vertices of higher importance vertices. The presence of shortcuts preserves the correctness, while the search space is significantly reduced. CH performs extremely well on real-world datasets in experiments, however, does not provide any worst-case complexity guarantees. Arterial Hierarchies (AH) [Zhu+13] was proposed as an improvement on CH while providing worst-case complexity guarantees on the query time, although does not significantly improve on CH in practice.

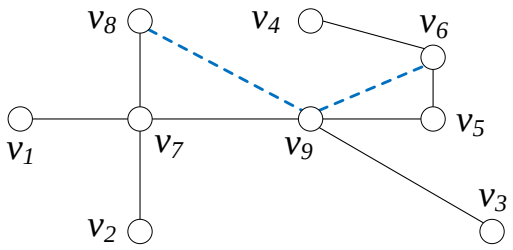


Figure 2.1: Contraction Hierarchies: Road Network Augments by Shortcuts

Transit Node Routing (TNR) [Bas+07] is an alternative index that significantly improves on CH query time at the expense of higher pre-processing cost. TNR works by creating a distance table based on a special property of road networks. When traveling from one location to another location that is further away, we tend to leave and enter via the same vertices. For example, the shortest path when traveling to the city may always involve taking a certain motorway entrance when departing from a suburb, irrespective of the starting point within that suburb. TNR exploits this property by imposing a grid on the road network. For each grid cell, TNR identifies the set of all "access node" vertices, which are the vertices through which any shortest path leaving the grid cell passes through. Interestingly, the number of access nodes tends to remain constant, approximately 10 according to Wu *et al.* [Wu+12], regardless of the granularity of the grid. Since the number

of access nodes is a small constant, it is feasible to pre-compute the distances between the access nodes from one cell to each access node of every other cell. Now, network distance queries can be answered by finding the pair of access nodes in the cells containing the source and destination vertex which give the smallest distance overall. When the source and destination are not in the same grid cell, TNR essentially provides constant time querying. However, when the source and destination are in the same grid cell, the query reverts to Dijkstra's algorithm. The justification for this is that since the cell is small, Dijkstra's search would be limited to a small search space. However, a finer granularity with a smaller search space results in higher space consumption and pre-processing time due to the increased number of cells, even if the number of access nodes per cell remains relatively constant. Thus, while TNR provided exceptional query times for longer shortest paths, it is not the ideal technique.

Some of the most recent methods have been from the 2-Hop Labeling (also known as Hub Labeling) family of techniques [Abr+11; Aki+14; Jia+14; DGW13; Ouy+18]. In a 2-hop labeling index, each vertex u is associated with a *label*. The label consists of a set of *hub* vertices to which network distances from u are known. Given any two vertices, there exists a common hub, such that the hub lies on the shortest path between the two vertices. The minimum number of hubs has been shown to be relatively small in practice, resulting in extremely fast network distance queries [Abr+11]. However, identifying fewer hub vertices typically requires an all-pairs shortest path computation. The pre-processing time is so high that most approaches [Abr+11; DGW13] rely on parallel processing to be practicable for application on continental-sized road networks. In addition, although the number of hubs is relatively small, multiplying by the total number of vertices results in significantly larger index sizes [Bas+15]. Akiba *et al.* [Aki+14] eliminated the need for parallelization by employing *pruned labeling* [AIY13]. This approach significantly reduced the pre-processing time for even continental road networks, at the expense of higher numbers of hubs in labels and, as a result, larger indexes overall. Hub Label Compression [DGW13] on the other hand reduces the index size by avoiding redundancy between labels of different vertices. However, this comes at the expense of pre-processing time and query performance. Ouyang *et al.* [Ouy+18] propose a labeling approach that incorporates ideas from hierarchical indexes like CH to improve query time with comparable pre-processing costs. The myriad

of relative advantages and disadvantages in all these types of techniques further underline the challenges of road network indexing.

The type of road network indexing techniques discussed so far, while originating in shortest path computation, are still highly relevant for answering other types of queries efficiently. Efficiency is particularly important for POI search as they, unlike P2P queries, do not specify a target. Any mistake in identifying a target results in additional work to compute unnecessary shortest paths. This inefficiency is naturally compounded by any inefficiency in computing the paths themselves. Furthermore, it is not always easy to adapt or transfer road network indexing techniques to POI search queries. For example, Contraction Hierarchies (CH) described earlier is efficient due to its bi-directional search. But bi-directional search assumes the target is known, which is not true for POI search. Consequently, using road network indexes like CH for POI search requires careful thought, especially when it comes to answering them efficiently, as exemplified by Liao *et al.* [Lia+15] for CH.

2.2 *k*NN Queries on Road Networks

First formally defined by Papadias *et al.* [Pap+03], the *k*NN query is among the first POI search queries on road networks. In this section we give an overview of various *k*NN techniques. Techniques are sorted into one of three categories, which we present in chronological order. Initially techniques did not involve any indexing of the road network. Subsequent techniques created a single index incorporating road network and object set (POI) information. Finally, state-of-the-art techniques use decoupled indexing that separates the road network from the object set. Several techniques will be described in greater detail in Chapter 3 as part of our experimental investigation into POI search.

2.2.1 Index-Free *k*NN Techniques

Papadias *et al.* [Pap+03] introduced the first road network *k*NN techniques, namely Incremental Network Expansion (INE) and Incremental Euclidean Restriction (IER). Neither of these techniques in their original form involves indexing the road network. INE is an extension of Dijkstra's algorithm with a different termination condition. When an object

is dequeued by INE it is added to the result set and when k objects are dequeued the algorithm terminates. IER, on the other hand, retrieves Euclidean Nearest Neighbors (NNs) as potential *candidates* for the network distance kNNs. Papadias *et al.* use Dijkstra's algorithm to compute the network distance to each candidate, thus not requiring any indexing of the road network. The candidate set is iteratively refined until it cannot be improved further. While these two techniques did not require any pre-processing, as discussed in Section 2.1, indexing allows significant gains to be made in query performance and this direction was taken by subsequent work. Moreover, using Dijkstra's algorithm with IER when other faster network distance techniques were available is a curiosity that turns out to have significant implications for POI search, as we investigate in Chapter 3.

2.2.2 Single Index *k*NN Techniques

The first techniques that followed INE and IER attempted to use the foreknowledge of O, the set of objects (POIs), and the road network graph G in a pre-processing step to create a single index that would aid subsequent online query processing. The first such example, Voronoi-based Network Nearest Neighbor (VN^3) [KS04], computes the network equivalent of a Voronoi diagram [OBS00] for a given object set to partition all road network vertices based on its first nearest neighbor (NN). By pre-computing distances between borders of adjacent Voronoi partitions, VN^3 answers queries using the fact the next NN must be in a Voronoi partition adjacent to the Voronoi partitions of currently found NNs. However, this technique suffered from significant pre-processing overhead, especially when objects were sparse. As partitions are not based on minimizing the number of borders, this leads to the computation of many border-to-border distances, increasing indexing time and space cost.

Several other techniques were proposed that pre-computed NNs for some or all vertices in the road network. Distance Indexing [HLL06] pre-computed the NNs for all vertices and used a compression scheme to reduce the space cost by a factor of 10. Even with such a reduction, the quadratic space complexity still represented too huge an index cost to be practicable on larger datasets. UNICONS [CC05] and Islands [HJŠ05] used a parameter m to limit the number of NNs that were pre-computed for each vertex, but this results in severely degraded performance when $k > m$. Nearest Descendant [HLX06] on the other hand only pre-computes the first NN for each vertex but this is still a significant space

overhead as there may be many object sets. For example, a different object set may exist for hospitals, each type of restaurant, hotels, and so on.

The common thread through the techniques discussed in this section is that kNN queries are answered by first creating a single index by processing both the road network and an object set together. This is highly disadvantageous as it will not scale well with greater numbers of object sets. The index space and time cost for each object set will be a function of $|V|$ even though the cardinality of the object set is generally much smaller than the number of road network vertices ($|V| \gg |O|$). Moreover, we must also reprocess the entire road network for any change to an object set, which may occur frequently, e.g., when a new restaurant is opened, or a shop goes out of business. As a result, these methods are unlikely to be useful in practice, and the current state-of-the-art has moved towards decoupled indexes, as we discuss next.

2.2.3 Decoupled Index *k*NN Techniques

To rectify the disadvantages of creating a single index, the latest techniques decouple indexing the road network from indexing the object set. Thus, the typically far more expensive pre-processing of the road network need only occur once. One of the first methods to attempt this was Distance Browsing (DisBrw) [SSA08] utilizing the SILC shortest path index [SAS05] introduced earlier in Section 2.1. SILC first computes the *shortest path tree* for each road network vertex s. Each neighbor of s is assigned a color. Then every vertex v in the graph is assigned the color corresponding to the neighbor of s on the shortest path from s to v, i.e., the "first move". Figure 3.1 illustrates this for vertex v_6. As identified by Sankaranarayanan *et al.*, due to the principle of *spatial coherence*, this coloring scheme will create large contiguous regions that can be indexed by a region quadtree to significantly reduce space cost. Answering shortest path queries is a simple matter of iteratively accessing quadtrees to determine the next vertex in the shortest path to a target. DisBrw stores additional information in each quadtree block to compute a minimum and maximum distance for the whole block. By building an additional quadtree for the object set (using the Euclidean coordinates of the objects), it can be intersected with the colored quadtree to visit the most promising regions first and prune regions that cannot contain objects. However, DisBrw still suffers from the same

huge pre-processing costs of SILC, in terms of both size and space, making it impractical on large road networks.

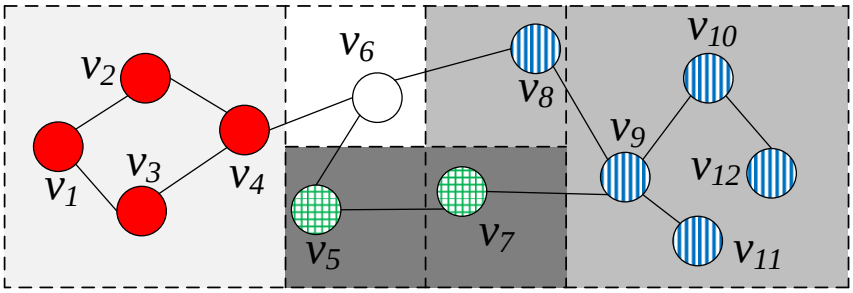


Figure 3.1: SILC: Coloring Scheme and Quadtree for v_6 (repeated from page 35)

Lee *et al.* propose Route Overlay and Association Directory (ROAD) [LLZ09; Lee+12] to address the weaknesses of INE by creating an index that allows "skipping" of regions that do not contain objects. This involves partitioning the road network into a hierarchy of subgraphs or *Rnets* to a parameterized depth l, as shown with the solid and broken-line containers in Figure 3.2 for depth $l = 3$. For each subgraph, shortcut edges are computed from every border vertex to every other border vertex. Lastly, given an object set, an Association Directory is built indicating whether an Rnet contains objects or not. Then kNN queries can be answered by using an algorithm similar to INE. Whenever the search encounters an Rnet border, ROAD will bypass the largest possible Rnet not associated with any object, using the shortcuts to preserve shortest path distances. Of course, if no Rnet at any level of the hierarchy can be bypassed, then the original edges of the vertex are used to visit adjacent vertices in exactly the same way as INE. In that case, the Rnet checking is an unnecessary overhead. Hence, ROAD's superiority over INE greatly depends on the time saved bypassing Rnets versus the overhead added by traversing shortcuts, which can be quite small for dense objects sets.

G-tree [Zho+13; Zho+15] is a road network index that partitions the road network G in a similar way to ROAD. However, unlike ROAD, which terminates partitioning at a certain depth, G-tree stops partitioning when a subgraph contains a certain number of vertices. Figure 3.3 shows the hierarchical partitioning of the road network G_0 into G-tree's subgraphs. The key difference between G-tree and ROAD is G-tree's use of a *distance matrix*. In non-leaf nodes the matrix stores the distance from every subgraph border to every other border (like ROAD's shortcuts), while in leaf nodes the matrix

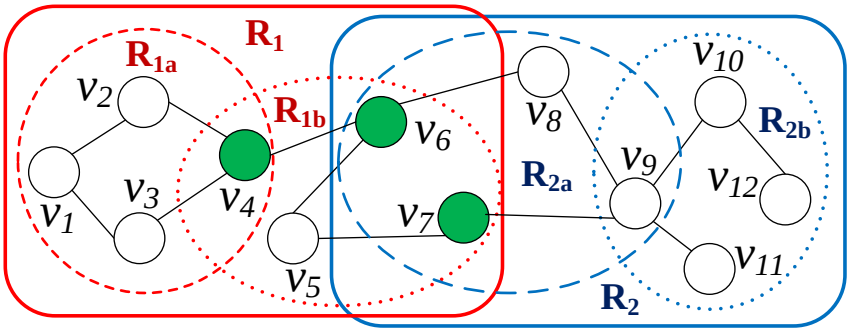


Figure 3.2: ROAD (repeated from page 36)

stores the distance from every subgraph vertex to every border. The subgraph hierarchy and distance matrices provide a mechanism for network distances to be *assembled* using a dynamic programming algorithm. Moreover, it is also possible to compute a lower-bound network distance to a subgraph at any level of the hierarchy by computing the distance to the closest border. Finding kNNs involves traversing the G-tree hierarchy by computing minimum network distances to the subgraphs which contain objects until reaching a leaf node where the distance to the object can be computed. This is aided by a secondary index called an Occurrence List which indicates the subgraphs that contain objects, similar to ROAD's Association Directory. Heuristically speaking, G-tree uses the closest border of a subgraph in the hierarchy to guide the search towards kNNs. It is worth noting that, unlike decoupled heuristics, this heuristic is dedicated and can only be used in the G-tree index. But objects may occur quite far from this border, thus it may not always be an accurate heuristic. Nonetheless, G-tree is shown to outperform all other kNN techniques that preceded it and incur reasonable pre-processing costs [Zho+13]. Recently, Li *et al.* [LCW19] proposed an optimized G*-tree index that improves on the original G-tree by better handling queries for vertices that are close in the road network but distant in the G-tree hierarchy.

Decoupled indexing has become the benchmark for kNN query processing. However, while the pre-processing advantages are obvious, this has naturally come with a cost. This is particularly true in the case of search heuristics, which is highly relevant to our study. For example, in the case of G-tree, the minimum network distance to a border of a subgraph is used to determine whether that subgraph looks promising or not. But since the partitioning only depends on the road network, it is blind to the presence or location

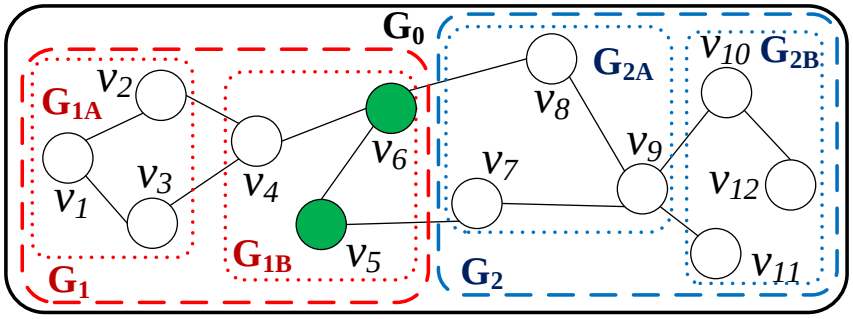


Figure 3.3: G-tree (repeated from page 38)

of objects within the subgraph. While Occurrence Lists help determine the presence or absence of objects, objects may still be quite far from the nearest border of the subgraph used to guide the search. This is the trade-off that has resulted from decoupled indexing in that it is more difficult to incorporate object information to help guide kNN search. Liao *et al.* [Lia+15] attempted to remedy this by re-processing the Contraction Hierarchy road network index based on the object set. But this has the same drawbacks as single index techniques and is not a scalable solution when there are many object sets. Therefore, this dilemma warrants further study.

2.3 Spatial Keyword Queries on Road Networks

Spatial keyword queries, also known as spatio-textual queries, involve combining textual similarity and spatial proximity to find objects (POIs). Compared to kNN queries, they involve finding the nearest and *relevant* objects, where relevance is measured by the similarity of keywords associated with the object to some input keywords specified by the user. This is advantageous in cases where it is laborious to sanitize and categorize objects into object sets as required by kNN querying. For example, we do not need to create and maintain different object sets for restaurants and vegetarian restaurants. Instead, we simply associate objects with keywords "restaurant" and "vegetarian" where applicable. Moreover, relevance is a relative measure, allowing ranking based on partially matching keywords. Several types of spatial keyword queries have been widely studied using Euclidean distance as the metric for spatial proximity [CJW09; WCJ12; Che+13; ZCT14], which as we have discussed is a less accurate and less flexible metric. In road

networks, where spatial keyword query techniques are few and far between, indexing keyword datasets is more challenging. We now describe the relevant techniques for each type of spatial keyword query.

Top-k Queries. Spatial keyword top-k queries in road networks retrieve objects by ranking them based on a score that combines an object's network distance from the query location and the textual similarity of its keywords to the query keywords. Textual similarity is usually computed by adopting methods for textual relevance from text mining, such as *cosine similarity* [ZM06]. Rocha-Junior and Nørvåg [RN12] first adapted Dijkstra's algorithm and the previously described ROAD index to answer top-k queries. In this instance, ROAD Rnets are used to compute textual relevance for all objects contained within the Rnet. During top-k search, a maximum score is computed for any object within the Rnet and Rnets that cannot improve the result set are bypassed in a similar manner to kNN querying. The disadvantage is that, since the maximum score is based on aggregated keyword occurrences, the score for an Rnet may not be very accurate and result in fewer bypasses, especially at higher levels.

Zhong *et al.* [Zho+13] also adapted the G-tree index to answer spatial keyword top-k queries. Both ROAD and G-tree involve hierarchically partitioning the road network. For example, Figure 5.1 shows a simplified single level partitioning of an example road network into 4 subgraphs indicated by the dotted containers. The road network includes several objects and their associated keywords. The approach by both ROAD and G-tree for incorporating textual information into their road network indexes involves combining the occurrences of keywords for entire subgraphs. Then these aggregated keyword occurrences are used to determine the maximum textual relevance for the whole subgraph. Like with ROAD's Rnets, top-k queries using G-tree can be answered by only visiting the subgraphs in the hierarchy that are promising and may indeed improve the results. The difference between ROAD and G-tree is largely the way the subgraph hierarchy is stored and accessed. G-tree offers better cache performance and, as a result, better query times compared to ROAD, as we verify in Chapter 3. However, in addition to the heuristic problems discussed in the previous section involving unreliable minimum network distances to subgraphs, both techniques will also suffer heuristic inefficiency from combining keyword occurrences due to the resulting inflation of top-k scores. In fact, the inefficiency is costlier on road networks

due to the high cost of network distance computation compared to similar approaches in Euclidean space, as we detail in Chapter 5.

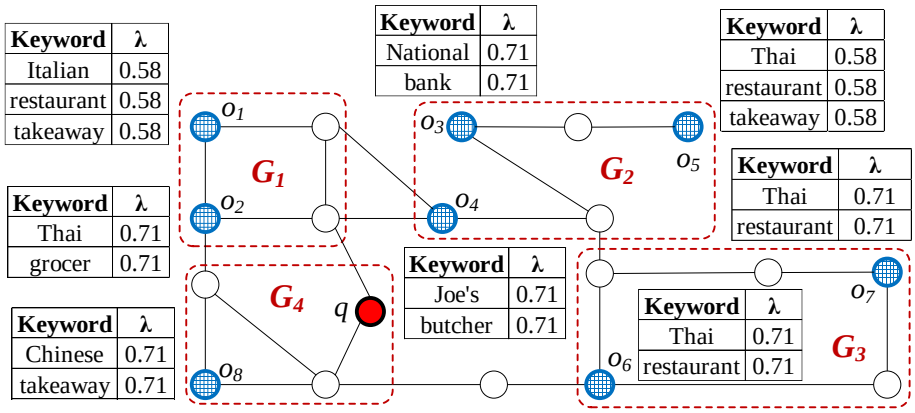


Figure 5.1: Road Network and Objects with Textual Information (repeated from page 91)

Boolean kNN Queries. FS-FBS [JFW15] is a Boolean kNN query technique improving on earlier approximate techniques [Qia+13] by providing exact results. FS-FBS uses a 2-hop labeling index described in Section 2.1.2 and its inverse, a backward label index. Recall that, in 2-hop labeling, each vertex has a *label* with a set of *hub* vertices and their distances to the vertex. Given any two vertices, it is guaranteed that they have a common hub that appears on the shortest path between them, hence allowing fast computation of the network distance between the two vertices by finding the common hub. A backward label for hub vertex h is the list of vertices for which h is in their label. For frequent keywords, FS-FBS employs a hierarchy of bit-array hashes to indicate the presence of keywords in the reverse label. During querying FS-FBS uses the hierarchy to narrow down on the nearest vertices containing relevant keywords. For infrequent keywords, FS-FBS simply computes network distances to *all* vertices containing the infrequent keyword. This is problematic because it is difficult to decide how to differentiate between frequent and infrequent keywords. While a metric is suggested, it is still necessary to verify the best performing frequency experimentally. Moreover, inverted lists for infrequent keywords, which are more common according to Zipf's law [ZM06], cannot be accessed by FS-FBS in order. Thus, it is not possible to terminate without evaluating the entire list. As the hierarchical hashing scheme is a keyword aggregation approach, the occurrence of collisions results in FS-FBS similarly suffering from the same drawbacks as G-tree and ROAD.

2.4 Aggregate kNN Queries on Road Networks

Aggregate kNN (AkNN) queries were first described in the context of Euclidean space by Papadias *et al.* [Pap+04]. On road networks, these queries have not received a significant amount of attention and by extension neither have the heuristics used to search for AkNNs. The few techniques that have been proposed have been heavily influenced by early approaches to answer kNN queries, which have long been superseded by new kNN techniques and, as we elaborate, may not be suitable for AkNN search anyway. Here we briefly describe the current state-of-the-art for AkNN queries on road networks.

Yiu *et al.* adapted INE and IER, previously described in Section 2.2, to solve AkNN queries on road networks [YMP05]. Their IER-based technique involved traversing the R-tree index, containing the objects, in a top-down manner. This traversal was guided by computing a lower-bound aggregate distance for all objects contained within an R-tree node. The lower-bound is computed using the minimum Euclidean distance from each query location to the R-tree node's Minimum Bounding Rectangle (MBR). Now the branches with the most promising R-tree nodes can be searched first, while those that cannot contain results are pruned. Similar to IER on kNN queries, the lower-bound becomes inaccurate when non-physical distance edge-weights are considered, and this error is only compounded by the aggregate function.

Ioup *et al.* [Iou+07] proposed to improve query efficiency by only providing approximate AkNN results. Their M-tree indexes transform the road network into higher dimensional space where AkNNs can be retrieved approximately in a similar manner to the RNE index proposed by Shahabi *et al.* [SKS03]. VN^3 has also been adapted for the AkNN problem [Zhu+10]. However, the use of network Voronoi diagrams is less suitable for the AkNN problem than the kNN problem, because result objects are unlikely to be close to any single query vertex when there are multiple query vertices potentially separated by large distances. Moreover, like its kNN counterpart, it suffers from high pre-processing costs.

A recent study by Yao *et al.* generalized the AkNN problem by proposing Flexible Aggregate Nearest Neighbor (FANN) queries [Yao+18]. FANN queries additionally specify a parameter ϕ. This parameter is used to define the proportion of query vertices for which the aggregate distance must be minimized. For example, perhaps it is sufficient to return

objects that minimize the aggregate distance for 10% ($\phi = 0.1$) of the query locations. To solve this generalized problem, techniques based on INE and IER are similarly proposed, experiencing the same benefits and drawbacks as we have discussed in-depth in the context of kNN querying.

2.5 Other POI Queries

The types of POI search queries we have described so far have covered the most studied settings, i.e., queries based purely on network distance (kNNs), then queries incorporating keywords and lastly queries incorporating multiple user locations. Beyond these, other queries have been proposed which vary the setting in some way. For example, some variations involving making the setting more dynamic. While in this study we focus on the major settings discussed so far, we give an overview of the types of variations that exist for the interested reader.

Dynamic POI search has many different interpretations. One example is to consider the objects as moving as described by Shahabi *et al.* [SKS03], who also identified the inadequacy of Euclidean k Nearest Neighbors for POI search. Their technique, Road Network Embedding (RNE) [SKS03], was the first work to propose a solution to the kNN problem specifically for road networks and moving objects. It proposes a better approximation of kNNs in road networks than Euclidean kNNs by pre-computing certain shortest path distances and then retrieving kNNs in higher dimensional space. Li *et al.* [Li+18] go further by using a lazy update approach combined with GPU-based processing to accelerate both updates and queries. Cao *et al.* [Cao+18] incorporate moving objects by answering *snapshot* kNN queries and utilize hierarchical grids to limit the search space.

Another variation is when we consider the query vertex to be moving. Given a path P, this might involve computing the kNNs for the entire path, i.e., all possible kNNs for all intervals on path P. Cho and Chung [CC05] use pre-computed lists of objects and an INE-like algorithm to answer such queries. On the other hand, Chen *et al.* [Che+09] consider a single set of kNNs for the whole path P, rather than multiple sets of kNNs for each possible interval on the path.

When it comes to dynamic queries, the possible variations are endless. LARC [Zhe+16] is an adaptation of FS-FBS for continuous Boolean kNN spatial keyword queries. Mouratidis *et al.* [Mou+06] consider both the object and the query location as moving. Moreover, variations can also involve more complex indexing techniques such as the Contraction Hierarchy-based technique proposed by Luo *et al.* [Luo+18] to answer dynamic kNN queries while providing a parameter that can trade query time for update time depending on which is more frequent. In their follow-up work, Luo *et al.* [Luo+19] considered efficiently and concurrently executing kNN queries and updates to indexes in a multi-core setting. On the other hand, He *et al.* [He+19] consider how to maintain the "correctness" of kNN results as the positions of moving objects are updated.

The variations we have discussed so far have involved some change to the assumptions about the objects or query vertex. However, this may also apply to the road network. One example is to consider the road network as time-dependent [DBS10], which involves representing the edge-weight as a time-varying function that, for example, has different travel times for peak versus off-peak times of the day. This scenario requires the static indexing techniques discussed in 2.1.2 to be adapted, which is not always trivial to accomplish, as Li *et al.* [LWZ19] found when adapting hub labeling for the time-dependent case. Dynamic road networks, on the other hand, incorporate real-time updates to the edge weights based on events like accidents or roadwork. This setting was considered by Delling and Werneck [DW15] in a technique that bypasses subgraphs in a similar manner to ROAD while offering fast updates to the road network index for edge-weight changes.

In Section 2.3, we discussed combining textual information with spatial proximity to retrieve POIs. [Zha+17] extend this concept by also considering social information such as check-ins on social media. This representation can then be used, for example, to answer reverse top-k queries that identify the nearest and most influential customers of a particular restaurant. Zhao *et al.* [Zha+18] then consider *why not* queries in the same setting to identify the reasons and generate the queries that would retrieve missing objects from top-k results. Guo *et al.* [Guo+19] similarly extend the group or aggregate nearest neighbor query discussed earlier in Section 2.4 by adding a social dimension, to find relevant POIs to groups of users that are more socially connected.

Queries related POIs are not necessarily limited to search. Yawalkar and Ran [YR19] attempt to find routes that visit as many of a given set of POIs as possible. Ridesharing

vehicles are an important example of moving POIs often represented on road networks, and Cheng *et al.* [CXC17] consider the utility maximization problem in this setting. Here the set of vehicles and users is known, and the goal is to maximize the utilization of available capacity for ridesharing. Both problems are shown to be NP-hard [YR19; CXC17].

Chapter 3

An Experimental Journey Into *k*NN Queries on Road Networks

> Truth is stranger than fiction, but it is because Fiction is obliged to stick to possibilities; Truth isn't.
>
> Oscar Wilde

In this chapter we present our experimental investigation into the state-of-the-art for kNN query techniques on road networks. Notably, we investigate a simple Euclidean heuristic that has never been properly compared, finding that through a simple improvement it is often the best performing technique, even when compared to more complex heuristics. In addition, we thoroughly document our entire journey and the other many insights we gleaned in the process. The work in this chapter was published in [ACT16b; ACT16a].

3.1 Overview

Market research company Newzoo reports that there were 3 billion smartphones in use at the end of 2018, with that number expected to increase to 3.8 billion by 2021. Due to this surge in the adoption of smartphones and other GPS-enabled devices, and cheap wireless network bandwidth, map-based services have become ubiquitous. Accordingly, 90% of these smartphone users have reported using a map application such as Google Maps [And16]. Finding nearby facilities (e.g., restaurants, ATMs) are among the most popular queries issued on maps. Due to their popularity and importance, k nearest neighbor (kNN)

queries, which find the k closest points of interest (objects) to a given query location, have been extensively studied in the recent past.

While related to the shortest path problem in many ways, the kNN problem on road networks introduces new challenges. Since the total number of objects is usually much larger than k it is not efficient to compute the shortest paths (or network distances) to all objects to determine which are kNNs. The challenge is to not only ignore the objects that cannot be kNNs but also the road network vertices that are not associated with objects. Recently, there has been a large body of work to answer kNN queries on road networks. Some of the most notable algorithms include *Incremental Network Expansion* (INE) [Pap+03], *Incremental Euclidean Restriction* (IER) [Pap+03], *Distance Browsing* [SSA08], *Route Overlay and Association Directory* (ROAD) [LLZ09; Lee+12], and *G-tree* [Zho+15; Zho+13]. In this chapter, we conduct a thorough experimental evaluation of these algorithms.

3.1.1 Motivation

We identify several important observations of the state-of-the-art for kNN queries that motivated us to conduct this study, as follows.

1. Neglected Competitor. IER [Pap+03] was among the first kNN algorithms on road networks. It has often been the worst performing method and as a result, is no longer included in comparisons. The basic idea of IER is to compute shortest path distances using Dijkstra's algorithm to the closest objects in terms of Euclidean distance. Although many significantly faster shortest path algorithms have been proposed in recent years, surprisingly, IER has never been compared against other kNN methods using any algorithm other than Dijkstra. To ascertain the true performance of IER it must be integrated with state-of-the-art shortest path algorithms.

2. Discrepancies in Existing Results. We note several discrepancies in the experimental results reported in some of the most notable papers on this topic. ROAD is seen to perform significantly worse than Distance Browsing and INE in [Zho+15]. But according to Lee *et al.* [Lee+12], ROAD is experimentally superior to both Distance Browsing and INE. The results in both [Lee+12] and [Zho+15] show Distance Browsing has worse performance than INE. In contrast, Distance Browsing is shown to be more efficient than INE in [SSA08]. These contradictions identify the need for reproducibility.

3. Implementation Does Matter. Similar to a recent study [ŠJ14], we observe that simple implementation choices can significantly affect algorithm performance. For example, G-tree utilizes *distance matrices* that can be implemented using either hash-tables or arrays and, on the surface, both seem reasonable choices. However, the array implementation, in fact, performs more than an order of magnitude faster than the hash-table implementation. We show that this is due to data locality in G-tree's index and its impact on cache performance. In short, seemingly innocuous choices can drastically change experimental outcomes. We also believe discrepancies reported above may well be due to different choices made by the implementers. Thus, it is critical to provide a fair comparison of existing kNN algorithms using careful in-memory implementations.

4. Overlooked Evaluation Measures/Settings. All methods we study here decouple the road network index from that of the set of objects, i.e., one index is created for the road network and another to store the set of objects. Although existing studies evaluate the road network indexes, no study evaluates the behavior of each individual object index. The construction time and storage cost for these *object indexes* may be critical information for developers when choosing methods, especially for object sets that change regularly. Additionally, kNN queries have not been investigated for travel time graphs (only travel distance), which is also a common scenario in practice. Finally, the more recent techniques (G-tree and ROAD) did not include comparisons for real-world POIs.

3.1.2 Contributions

Below we summarize our contributions.

1. Revived IER: We investigate IER with several efficient shortest path techniques for the first time (see Section 3.5). We show that the performance of IER is significantly improved when better shortest path algorithms are used. This occurs to the point that IER is the best performing method in most settings, including travel time road networks where Euclidean distance is a less effective lower bound.

2. Highly Optimized Algorithms Open-Sourced: We present efficient implementations of five of the most notable methods (IER, INE, Distance Browsing, ROAD, and G-tree). First, we have carefully implemented each method for efficient performance in main memory as described in Appendix A. Second, we thoroughly checked each algorithm and made various improvements that are applicable in any setting, as documented

in Appendix B. The source code and scripts to run experiments have been released as open-source [Abe16], making our best effort to ensure it is modular and re-usable.

3. Reproducibility Study: With efficient implementations of each algorithm, we repeat many experiments from past studies on many of the same datasets in Section 3.6. Our results provide a deeper understanding of the state-of-the-art with new insights into the weaknesses and strengths of each technique. We also show that there is room to improve kNN search heuristics by demonstrating that G-tree can be made more efficient by using a simple Euclidean distance heuristic compare to its own more complex heuristic. This gives us thought to reconsider POI search heuristics in general, and a pathway to develop better heuristics.

4. Extended Experiments and Analysis: Our comprehensive experimental study in Section 3.6 extends beyond past studies by 1) comparing object indexes for the first time; 2) revealing new trends by comparing G-tree with another advanced method (ROAD) on larger datasets for the first time; 3) evaluating all methods (including ROAD and G-tree) on real-world POIs; and 4) evaluating applicable methods on travel time road networks.

5. Guidance on Main-Memory Implementations: In Appendix A, we also demonstrate how simple choices can severely impact algorithm performance. We share an in-depth case study to give insights into the relationship between algorithms and in-memory performance with respect to data locality and cache efficiency. Additionally, we highlight the main choices involved and illustrate them through examples and experimental results, to provide hints to future implementers. Significantly, these insights are potentially applicable to any problem, not just those we study here.

Additionally, Section 3.2 defines the problem and scope of our study. Section 3.3 and 3.4 describes the algorithms evaluated and datasets used, respectively. Finally, Section 3.9 summarizes our findings.

3.2 Background

3.2.1 Problem Definition

We use the definition of a road network as an undirected graph $G = (V, E)$ and network distance as introduced in Section 1.1. For conceptual simplicity, similar to the existing studies [Zho+15; SSA08], we assume that objects (i.e., POIs) and the query locations are

located on vertices in V. Given a query vertex q and a set of object vertices O, a kNN query retrieves the k closest objects in O based on their network distances from q.

3.2.2 Scope

We separate existing kNN techniques into two broad categories based on the indexing they use: 1) blended indexing; and 2) decoupled indexing. Techniques that use blended indexing [KS04; HLL06; CC05] create a single index to store the objects as well the road network. For example, VN^3 [KS04] is a notable technique that uses a network Voronoi diagram based on the set of objects to partition the network. In contrast, decoupled indexing techniques [Pap+03; Lee+12; Zho+15; SSA08] use two separate indexes for the object set and road network, which is more practical and has several advantages as explained below.

First, a real-world kNN query may be applied to one of many object sets, e.g., return the k closest restaurants or locate the nearest parking space. Blended indexing must repeatedly index the road network for each type of object, entailing huge space and pre-processing time overheads. But decoupled indexing requires only one road network index regardless of the number of object sets, resulting in lower storage and pre-processing cost. Second, if there is any change in an object set, blended indexing must update the whole index and reprocess the entire road network, whereas decoupled techniques need only update the object index. For example, the network-based Voronoi diagram must be updated resulting in expensive re-computations [KS04]. Conversely, in decoupled indexing, the object indexes (e.g., R-tree) are typically much cheaper to update. The problem is more serious for object sets that change often, e.g., if the objects are the nearest *available* parking spaces.

Due to these advantages, all recent kNN techniques use decoupled indexing. In this study, we focus on the most notable kNN algorithms that employ decoupled indexing. These algorithms either employ an expansion-based method or a heuristic best-first search (BFS). The expansion-based methods encounter kNNs in network distance order. Heuristic BFS methods instead employ heuristics to evaluate the most promising kNN candidates, not necessarily in network distance order, potentially terminating sooner. We study the five most notable methods which include two expansion-based methods, INE [Pap+03] and ROAD [Lee+12], and three heuristic BFS methods, IER [Pap+03], Distance Browsing (DisBrw) [SSA08] and G-tree [Zho+15].

Given the rapid growth in smartphones and the corresponding widespread use of map-based services, applications must employ fast in-memory query processing to meet the high query workload. In-memory processing has become viable due to the increases in main-memory capacities and its affordability. Thus, we limit our study to in-memory query processing but reflect on this choice in Appendix A.

3.3 Methods

We now describe the main ideas behind each method evaluated by our study. These techniques were briefly introduced in Section 2.2, but we provide detailed descriptions here to help readers better understand how they work and interpret the insights from our experimental investigation. Note that some methods propose a road network index and a kNN query algorithm to use it. In some cases, such as G-tree, we refer to both the index and kNN algorithm by the same name.

3.3.1 Incremental Network Expansion

Incremental Network Expansion (INE) [Pap+03] is a method derived from Dijkstra's algorithm. As with Dijkstra, INE maintains a priority queue of the vertices seen so far, initialized with the query vertex q with a key of zero. The search is expanded to the nearest of these vertices v (i.e., the vertex in the queue with minimum key). If $v \in O$ then it is added to the result set as one of the kNNs and if v is the kth object then the search is terminated. Otherwise, the edges of v are used to relax the distances to its neighbors and the expansion continues. As in Dijkstra's algorithm, relaxation involves updating the minimum network distances to the neighbors of v using the network distance through v. Like Dijkstra, INE has the same disadvantage, in that it must visit every vertex that is closer than the kth object. Naturally, this inefficiency will worsen when objects are sparse and the kth object is found further away, due to the increased search space.

3.3.2 Incremental Euclidean Restriction

Incremental Euclidean Restriction (IER) [Pap+03] uses Euclidean distance as a heuristic to retrieve candidates from O, as it is a lower bound on network distance for road networks with travel distance edge weights. First, IER retrieves the Euclidean kNNs, e.g., using an

R-tree [Gut84]. It then computes the network distance to each of these k objects using another technique (such as Dijkstra's algorithm) and sorts them in network distance order. This set becomes the candidate kNNs and the network distance to the furthest candidate, denoted as D_k, is an upper bound on the distance to the true kth nearest neighbor. Now, IER retrieves the next Euclidean nearest neighbor (NN) p. If the Euclidean distance to p is $d_E(q,p)$ and $d_E(q,p) \geq D_k$, then p cannot be a better candidate by network distance than any current candidate. This is because $d_E(q,p)$ represents a lower bound on network distance, which is true when edge weights are physical distance (it can also be adapted for other edge weights as well). Moreover, since it is the *nearest* Euclidean NN, the search can also be terminated. However, if $d_E(q,p) < D_k$ then p may be a better candidate. In this case, IER computes the network distance $d(q,p)$. If $d(q,p) < D_K$, p is inserted into the candidate set (removing the furthest candidate and updating D_k). This continues until the search is terminated or there are no Euclidean NNs left. Papadias *et al.* used Dijkstra's algorithm to compute network distances. As a result, IER was significantly outperformed by INE in [Pap+03] because the repeated invocation of Dijkstra's algorithm must revisit the same vertices each time.

3.3.3 Distance Browsing

Distance Browsing (DisBrw) [SSA08] uses the Spatially Induced Linkage Cognizance (SILC) index proposed in [SAS05] to answer kNN queries. [SAS05] proposed an incremental kNN algorithm, which DisBrw improves upon by making fewer priority queue insertions.

SILC Index. We first introduce the SILC index used by DisBrw. For a vertex $s \in V$, SILC pre-computes the shortest paths from s to all other vertices. SILC assigns each adjacent vertex of s a unique color. Then, each vertex $u \in V$ is assigned the same color as the adjacent vertex v that is passed through in the shortest path from s to u. Figure 3.1 shows the coloring of the vertices for the vertex $s = v_6$ where each adjacent vertex of v_6 is assigned a unique color and the other vertices are colored accordingly. For example, the vertices v_9 to v_{12} have the same color as v_8 (blue vertical stripes) because the shortest path from v_6 to each of these vertices passes through v_8 (for this example assume unit edge weights).

Observe that the vertices close to each other have the same color resulting in several contiguous regions of the same color. These regions are indexed by a region quadtree

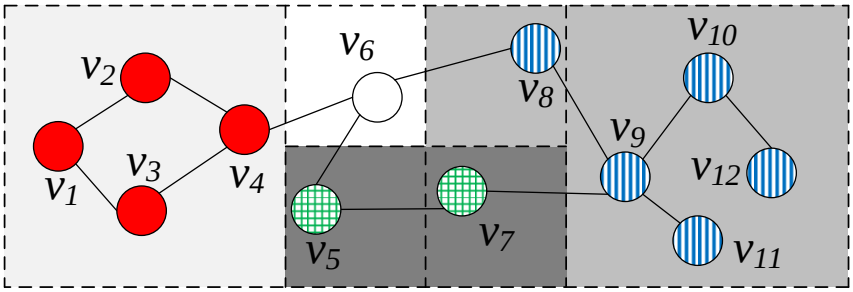


Figure 3.1: SILC Index: Coloring Scheme and Quadtree for v_6

[Sam05] to reduce the storage space. The color of a vertex can be determined by locating the region in the quadtree that contains it. SILC applies the coloring scheme and creates a quadtree for each vertex of the road network. This requires $O(|V|^{1.5})$ space in total and, due to the all-pairs shortest path computation, $O(|V|^2 \log |V|)$ pre-processing time.

To compute the shortest path from s to t, SILC uses the quadtree of s to identify the color of t. The color of t determines the first vertex v on the shortest path from s to t. To determine the next vertex on the shortest path, this procedure is repeated on the quadtree of v. For example, in Figure 3.1, the first vertex on the shortest path from v_6 to v_{12} is v_8 because v_{12} has the same color as v_8. The color of v_{12} is found by locating the quadtree block containing v_{12}. The shortest path can be computed in $O(m \log |V|)$ where m is the number of edges on the shortest path [SSA08].

kNN Algorithm. To enable kNN search, DisBrw stores additional information in each quadtree. For each vertex v contained in a quadtree block b, it computes the ratio of the Euclidean and network distances between the quadtree owner s and v. It then stores the minimum and maximum ratios, λ^- and λ^+ respectively, with b. Now, given any vertex t, DisBrw computes a *distance interval* $[\delta^-, \delta^+]$ by multiplying the Euclidean distance from s to t by the λ^- and λ^+ values of the block containing t. This interval defines a lower and upper bound on the network distance from s to t and can be used to prune objects that cannot be kNNs. The interval is *refined* by obtaining the next vertex u in the shortest path from s to t (as described earlier), computing an interval for u to t, and then adding the known distance from s to u to the new interval. By refining the interval, it eventually converges to the network distance.

DisBrw used an *Object Hierarchy* in [SSA08] to avoid computing distance intervals for all objects. The basic idea was to compute distance intervals for regions containing

objects, then visit the most promising regions (and recursively sub-regions) first. We found this method did not use the SILC index to its full potential. Instead we retrieve Euclidean NNs as candidate objects for which intervals are then computed. Otherwise, the DisBrw kNN algorithm proceeds exactly as in [SSA08]. We refer the readers to Section 3.5.2 for full details and experimental comparisons with the original method.

3.3.4 Route Overlay & Association Directory

The search space of INE can be considerably large depending on the distance to the kth object. Route Overlay and Association Directory (ROAD) [LLZ09; Lee+12] attempts to remedy this by bypassing regions that do not contain objects by using *search space pruning*.

An *Rnet* is a partition of the road network $G{=}(V, E)$, with every edge in E belonging to at least one Rnet. Thus, an Rnet R represents a set of edges $E_R \subseteq E$. V_R is the set of vertices that are associated with edges in E_R. To create Rnets, ROAD partitions the road network G into $f \geq 2$ Rnets, recursively partitioning resulting Rnets until a hierarchy of $l > 1$ levels is formed (with G being the root at level 0). Figure 3.2 shows Rnets (for $l{=}2$) for the graph in our running example. The enclosing boxes and ovals represent the set V_R of each Rnet. Specifically, $R_1{=}\{v_1, \cdots, v_7\}$ and $R_2{=}\{v_6, \cdots, v_{12}\}$ are the child Rnets of the root G. Each of R_1 and R_2 are further divided into Rnets, e.g., R_1 is divided into $R_{1a}{=}\{v_1, v_2, v_3, v_4\}$ and $R_{1b}{=}\{v_4, v_5, v_6, v_7\}$.

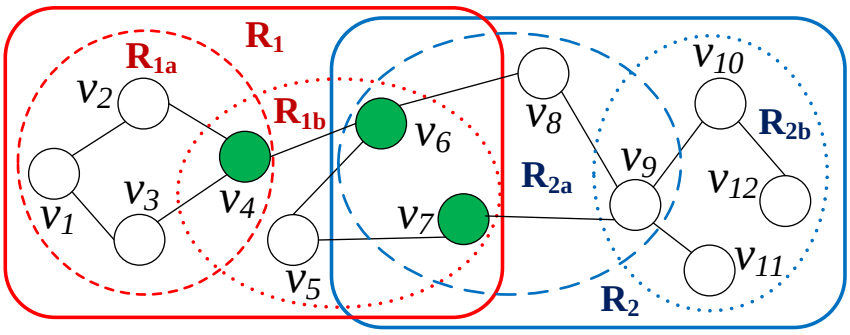


Figure 3.2: Route Overlay and Association Directory Index

For an Rnet R, a vertex $b \in V_R$ with an adjacent edge $(b, v) \notin E_R$ is defined as a *border* of R. For instance, v_4 is a border of R_{1b} but v_5 is not. These borders form the set $B_R \subseteq V_R$, e.g., the border set of R_{1b} consists of v_4, v_6 and v_7. ROAD computes the

network distance between every pair of borders $b_i, b_j \in B_R$ in each Rnet and stores each as the *shortcut* $S(b_i, b_j)$. Now any shortest path between two vertices $s, t \notin V_R$ involving a vertex $u \in V_R$ must enter R through a border $b \in B_R$ and leave through a border $b' \in B_R$. So if a search reaches a border $b \in B_R$ the shortcuts associated with b, $S(b, b')$ $\forall$ $b' \in B_R$, can be traversed to bypass the Rnet R while preserving network distances. For example, in Figure 3.2, the borders of R_{1b} are v_4, v_6 and v_7 (the colored vertices) and ROAD precomputes the shortcuts between all these borders. Suppose the query vertex is v_1 and the search has reached the vertex v_4. If it is known that R_{1b} does not contain any object, the algorithm can bypass R_{1b} by quickly expanding the search to other borders of R_{1b} without the need to access any non-border vertex of R_{1b}. For example, using the shortcut between v_4 and v_7, the algorithm can compute the distance between v_1 to v_7 without exploring any vertex in R_{1b}.

Since child Rnets are contained by their parent Rnet, a border b of an Rnet must be a border of some child Rnet at each lower level. For example, v_6 in Figure 3.2 is a border for R_{1b} and its parent R_1. This allows the shortcuts to be computed in a bottom-up manner, where shortcuts at level i are computed using those of level i+1, greatly reducing pre-computation cost. Only leaf Rnets require Dijkstra searches on the original graph G.

ROAD uses a *Route Overlay* index and an *Association Directory* to efficiently compute kNNs. Recall that a vertex v may be a border of more than one Rnet. The Route Overlay index stores, for each vertex v, the Rnets for which it is a border along with the *shortcut trees* of v. The Association Directory provides a means to check whether a given Rnet or vertex contains an object or not. The kNN algorithm proceeds incrementally from the query vertex q in a similar fashion to INE. However, when ROAD expands to a new vertex v, instead of inspecting its neighbors, it consults the Route Overlay and Association Directory to find the highest level Rnet associated with it that does not contain any object. ROAD then relaxes all the shortcuts in this Rnet in a similar way to edges in INE, to bypass it. Of course, when v is not a border of any Rnet or if all Rnets associated with v contain an object, it relaxes the edges of v exactly as in INE. The search terminates when k objects have been found or there are no further vertices to expand.

3.3.5 G-tree

G-tree [Zho+13; Zho+15] also employs graph partitioning to create a tree index that can be used to efficiently compute network distances through a hierarchy of subgraphs. The partitioning occurs in a similar way to that of ROAD where the input graph G is partitioned into $f \geq 2$ subgraphs. Each subgraph is recursively partitioned until it contains no more than $\tau \geq 1$ vertices. For any subgraph G_i, $V_i \subseteq V$ is defined as the set of road network vertices contained within it. Any vertex $b \in V_i$ with an edge (b, v) where $v \notin V_i$ is defined as a border of G_i and all such vertices form the set of borders B_i. Figure 3.3 shows an example where the colored vertices v_5 and v_6 are borders for the subgraph G_1=$\{v_1, \cdots, v_6\}$.

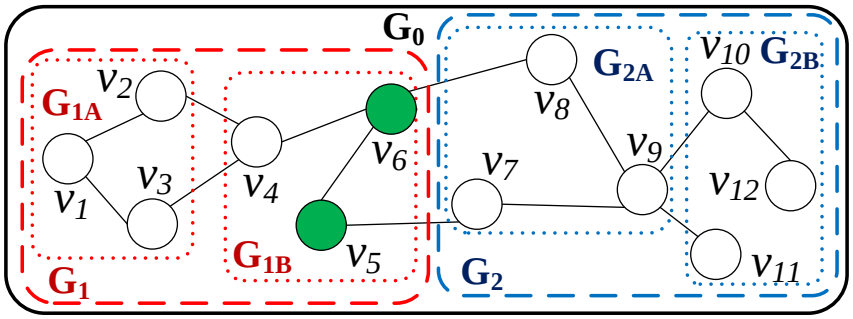


Figure 3.3: G-tree

The partitioned subgraphs naturally form a tree hierarchy with each node in the G-tree associated with one subgraph. Note that we use *node* to refer to the G-tree node while *vertex* refers to road network vertices. Notably, a non-leaf node G_i does not need to store subgraph vertices, but only the set of borders B_i and a *distance matrix*. For non-leaf nodes, the distance matrix stores the network distance from each child node border to all other child node borders. For leaf nodes, it stores the network distance between each of its borders and the vertices contained in it.

Similar to the bottom-up computation of shortcuts in ROAD, the distance matrix of nodes at tree level i can be efficiently computed by reducing the graph to only consist of borders at level i+1 using the distance matrices of that level. Only leaf nodes require a Dijkstra's search on the original graph. Given a planar graph and optimal partitioning method, G-tree is a height-balanced tree with a space complexity of $O(|V| \log |V|)$. The

similarities with ROAD are clear. One major difference is that G-tree uses its border-to-border distance matrices to "assemble" shortest path distances by the path through the G-tree hierarchy. We refer the reader to the original paper [Zho+15] for the details of the assembly method.

Another key difference is the kNN algorithm. To support efficient kNN queries, G-tree introduces the *Occurrence List*. Given an object set O, the Occurrence List of a G-tree node G_i lists its children that contain objects, allowing empty nodes to be pruned. The kNN algorithm begins from the leaf node that contains q, using a Dijkstra-like search to retrieve leaf objects. However, we found this leaf search could be further optimized and detail our improved leaf search algorithm in Appendix B.2.1. The algorithm then incrementally traverses the G-tree hierarchy from the source leaf. Elements (nodes or objects) are inserted into a priority queue using their network distances from q. The network distance to a G-tree node is computed using the assembly method by finding its nearest border to q. Queue elements are dequeued in a loop. If the dequeued element is a node, its Occurrence List is used to insert its children (nodes or object vertices) back into the priority queue. If the dequeued element is a vertex, it is guaranteed to be the next nearest object. The search terminates when k objects are dequeued.

A useful property of assembling distances is that, given a path through the G-tree hierarchy, distances can be *materialized* for already visited G-tree nodes. For example, given a query vertex q and two kNN objects in the same leaf node, after locating one of them, the distances to the borders of this leaf need not be recomputed.

3.4 Datasets

Here we describe the datasets used to supply the road network $G{=}(V, E)$ and set of object vertices $O \subseteq V$ for kNN querying.

3.4.1 Real Road Networks

We study kNN queries on 10 real-world road network graphs as listed in Table 3.1. These were created for the 9th DIMACS Challenge [Pat06] from data publicly released by the US Census Bureau. Each network covers all types of roads, including local roads, and contains real edge weights for travel distances and travel times (both are used in our experiments).

We also conduct in-depth studies for the United States (US) and North-West US (NW) road networks. The US dataset, covering the entire continental United States, is the largest with 24 million vertices. The NW road network (with 1 million vertices), covering Oregon and Washington, represents queries limited to a region or smaller country. Notably, this is the first time DisBrw has been evaluated on a network with more than $500,000$ vertices, previously not possible due to its high pre-processing cost (in both space and time).

Name	Region	# Vertices	# Edges
DE	Delaware	48,812	119,004
VT	Vermont	95,672	209,288
ME	Maine	187,315	412,352
CO	Colorado	435,666	1,042,400
NW	North-West US	1,089,933	2,545,844
CA	California & Nevada	1,890,815	4,630,444
E	Eastern US	3,598,623	8,708,058
W	Western US	6,262,104	15,119,284
C	Central US	14,081,816	33,866,826
US	United States	23,947,347	57,708,624

Table 3.1: Real-World Road Network Datasets

3.4.2 Real and Synthetic Object Sets

We create object sets based on both real-world points of interest (POIs) and synthetic methods as described below.

Real-World POI Sets. We created 8 real-world object sets (listed in Table 3.2) using data extracted from OpenStreetMap (OSM) [Ope] for locations of real-world POIs in the United States. Each object set is associated with one type of POI, e.g., all fast food outlets. POIs were mapped to road network vertices on both the US and NW road networks using their coordinates. While real POIs can be obtained freely from OSM, it is not a propriety system. As a result, the data quality can vary, e.g., the largest object sets in OSM may not be representative of the true largest object sets and the completeness of POI data may vary between regions. So, in addition to real-world object sets, we generate synthetic sets to make generalizable and repeatable observations for all road networks.

Uniform Object Sets. A uniform object set is generated by selecting uniformly random vertices from the road network. As these objects are randomly selected road network vertices, they are likely to simulate real POIs, e.g., areas with more vertices have more POIs (e.g., cities) while those with fewer roads have less (e.g., rural areas). The

Object Set	**United States**		**North-West US**	
	Size	Density	Size	Density
Schools	160,525	0.007	4,441	0.004
Parks	69,338	0.003	5,098	0.005
Fast Food	25,069	0.001	1,328	0.001
Post Offices	21,319	0.0009	1,403	0.001
Hospitals	11,417	0.0005	258	0.0002
Hotels	8,742	0.0004	460	0.0004
Universities	3,954	0.0002	95	0.00009
Courthouses	2,161	0.00009	49	0.00005

Table 3.2: Real-World Object Sets

density of objects sets d is varied from 0.0001 to 1, where d is the ratio of the number of objects $|O|$ to the number of vertices $|V|$ in the road network. High densities can simulate larger object sets which are common occurrences, e.g., ATM machines, parking spaces. Low densities correspond to the sparsely located POIs, e.g., post offices or restaurants in a specific chain. By decreasing the density, we can simulate more difficult queries, as fewer objects imply longer distances and therefore larger search spaces. Uniform objects were used to evaluate G-tree in [Zho+13; Zho+15].

Clustered Object Sets. While some POIs may be uniformly distributed other types, such as fast food outlets, occur in clusters. To create such clustered object sets, given a number of clusters $|C|$, we select $|C|$ central vertices uniformly at random (as above). For each central vertex, we select several vertices (up to a maximum cluster size C_{max}) in its vicinity, by expanding outwards from it. This distribution was used to evaluate ROAD in [Lee+12].

Minimum Object Distance Sets. The worst-case kNN query occurs when the query location is remote. To simulate this, we create minimum distance object sets as follows. We choose an approximate center vertex v_c by using the nearest vertex to the Euclidean center of the road network. We find the furthest vertex v_f from v_c and set D_{max} as the network distance from v_c to v_f. For an object set R_i, $i \in [1, m]$, we choose $|O|$ objects such that the network distance from v_c to each object in R_i is at least $\frac{D_{max}}{2^{m-i+1}}$. For example, for m=5, the set R_1 contains objects within the range $(\frac{D_{max}}{32}, D_{max}]$. Thus, we investigate the effect of increasing minimum object distance by comparing query time on R_i with increasing i.

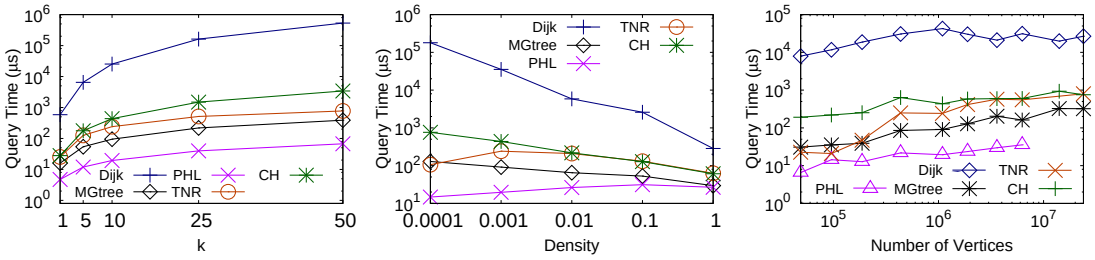
3.5 IER Revisited

Network distance computation is a critical part of Incremental Euclidean Restriction (IER). However, to the best of our knowledge, all existing studies [Pap+03; SSA08; LLZ09; Lee+12] employ Dijkstra's algorithm to compute network distances. Dijkstra's algorithm is not only slow, but it must also revisit the same vertices for subsequent network distance computations. Even if Dijkstra's algorithm is suspended and resumed for subsequent Euclidean NN candidates, this is necessarily no better than INE, which uses Dijkstra-like expansion until all kNNs are found.

To understand the true potential of IER, we combined it with several fast techniques. *Pruned Highway Labeling* [Aki+14] is amongst the fastest techniques. It boasts fast construction times despite being a labeling method but has similarly large index sizes. The G-tree assembly-based method mentioned earlier can also compute network distances. Notably, in a similar manner to G-tree's kNN search, the "materialization" property can be used to optimize repeated network distance queries from the same source (as in IER). The Dijkstra-like leaf-search can also be suspended and resumed. This is doubly advantageous for IER, as it becomes more robust to "false hits" (Euclidean NNs that are not real kNNs), especially if they are in the vicinity of a real kNN. We refer to this version of G-tree as MGtree. Finally, we combined IER with *Contraction Hierarchies* (CH) [Gei+08] and *Transit Node Routing* (TNR) [Bas+07] using implementations made available by a recent experimental paper [Wu+12]. We use a grid size of 128 for TNR as in [Wu+12].

We compare the performance of IER using each method in Figure 3.4 for travel distance edge-weights. These queries are run on a road network dataset representing the North West United States (NW) using the same environment and default parameters as in other experiments in this chapter (as detailed in Section 3.6.1). PHL is the consistent winner, being 4 orders of magnitude faster than Dijkstra and an order of magnitude better than the next fastest method at its peak. G-tree, assisted by materialization, is the next best method. All methods converge with increasing density, as the search space becomes smaller. Note that CH is the technique used to answer local queries in TNR, which explains why TNR and CH are so similar for high densities as the distances are too small to use TNR's access node vertices (described briefly in Section 2.1.2). At lower densities in Figure 3.4(b), access nodes are used more often, leading to a larger speed up. The

superiority of G-tree and PHL is also seen to be true irrespective of the dataset size in Figure 3.4(c). Note that the PHL index was too large to fit into memory for the largest dataset, the continental US, due to the lack of hierarchies in travel distance graphs, but we do include PHL for the travel time version of the US dataset as we explain below. Nonetheless, given these results we include the two fastest versions of IER, i.e., PHL and MGtree, in our main experiments.



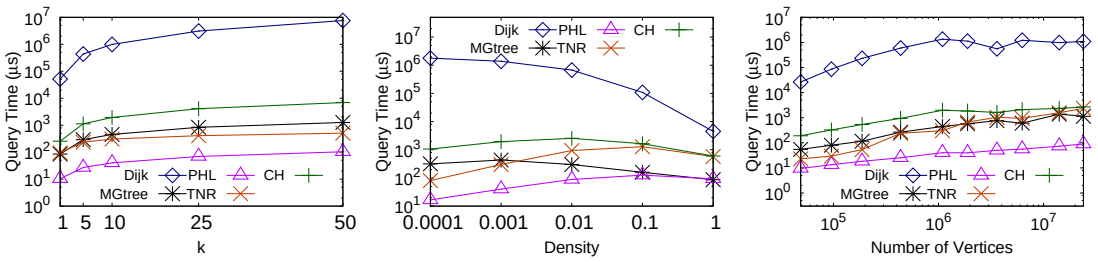
(a) Varying k (b) Varying Density (c) Varying Road Network

Figure 3.4: IER Variants on Travel Distance (NW, d=0.001, k=10, uniform objects)

3.5.1 IER on Travel Time Road Networks

IER can also be adapted for use on road networks with travel-time edge-weights (and by extension any type of edge-weight). IER uses Euclidean distance as a lower bound on the network distance between two points for travel distance edge weights. This can easily be extended for other edge weights. Let w_i (respectively, d_i) represent the edge weight (respectively, Euclidean length) of an edge e_i. We compute $S = max_{\forall e_i \in E}(d_i/w_i)$. For example, if w_i represents travel time, S corresponds to the maximum speed on any edge in the network. Let $d_E(p, q)$ be the Euclidean distance between two points p and q. It is easy to see that $d_E(p, q)/S$ is a lower bound on the network distance between p and q, e.g., the time it takes to travel the Euclidean distance at the maximum possible speed. Thus, we compute S for the network and use the new lower bound in IER. Naturally, we can expect this lower-bound to be looser compared to the actual travel time due to using the fastest speed on the whole network. Landmarks are known to provide better lower bounds on travel time graphs [GH05]. However, using them to incrementally retrieve candidates, like how Euclidean candidates can be retrieved from an R-tree, is a difficult problem that we address in Chapter 4.

In Figure 3.5 we again compare IER with different network distance techniques on the NW road network dataset but with travel-time edge-weights. Note that the performance of shortest path techniques are known to vary between travel distance and travel time graphs. For example, CH and TNR have been seen to perform 5−20× worse on travel distances [Bau+10]. This is because travel distances do not display hierarchies as prominently as travel times. For example, highways may not always provide the shortest travel distance, but generally, provide faster travel time. Methods that rely on these properties, such as CH, PHL, and TNR, are more effective when they are present (as in travel time graphs). For example, when hierarchies are present it is easier to distinguish between more important and less important vertices in CH or PHL and the number of access nodes is lower in TNR. This is particularly useful in the case of PHL, as now the index can be constructed for the largest datasets, and query performance for these datasets are reported in Figure 3.5(c).



(a) Varying k (b) Varying Density (c) Varying Road Network

Figure 3.5: IER Variants on Travel Time (NW, d=0.001, k=10, uniform objects)

All methods perform worse at high densities as IER encounters more false hits as the Euclidean-based lower bound is looser. Interestingly, CH and TNR query times do not change significantly from the travel distance case for lower densities, unlike MGtree. As mentioned before, despite the greater number of false hits, both these methods are faster on travel time graphs. As a result, TNR performs better than MGtree on travel times for low densities. In fact, all methods perform better on density 0.0001 than density 0.001 because there are fewer objects and therefore a lower likelihood of having similar lower-bounds, leading to fewer false hits. MGtree's performance degrades by the smallest amount on high densities, as its optimized repeated computations make it more robust to the increase in false hits. If TNR were to be combined with MGtree to answer local queries (rather than CH), it may be a better option on NW than just MGtree. Regardless of this, PHL

performs significantly better than TNR across the board. Additionally, we also compare IER methods for increasing network size $|V|$ in Figure 3.5(c). The most noteworthy observation is that TNR deteriorates more rapidly than other methods because, for the same grid size, TNR answers fewer queries using access nodes with increasing $|V|$. With increasing $|V|$, grid cells contain more vertices, and as a result distances to more kNNs must be computed using the slower local method.

3.5.2 Distance Browsing via Euclidean NN

We also see an opportunity to apply the Euclidean distance heuristic to the competing method, DisBrw. Rather than using SILC shortest paths to improve IER, this is akin to incorporating IER's Euclidean distance heuristic to improve the original DisBrw kNN search algorithm. The Object Hierarchy is a key component of DisBrw. It is most easily represented by a quadtree containing all objects from the object set. DisBrw visits the most promising branches of this quadtree first, by computing distance intervals to child blocks (i.e., Object Hierarchy nodes). As described in [SSA08], DisBrw retrieves all leaf blocks from the SILC quadtree of the query vertex intersecting with that Object Hierarchy node. It uses these blocks to compute lower and upper bounds on the distance from the query vertex to any object in that node. But computing intersections is not a trivial expense. In the worst-case, all SILC quadtree leaf blocks must be retrieved. Furthermore, many of the same intersections must be recomputed whilst traversing down the hierarchy. This implies a trade-off between the ability to prune regions using the hierarchy and the height of the hierarchy. A larger height improves performance on very high densities but penalizes lower densities. We observed that very shallow Object Hierarchies (with leaf capacities of 500 objects) provided the best overall performance.

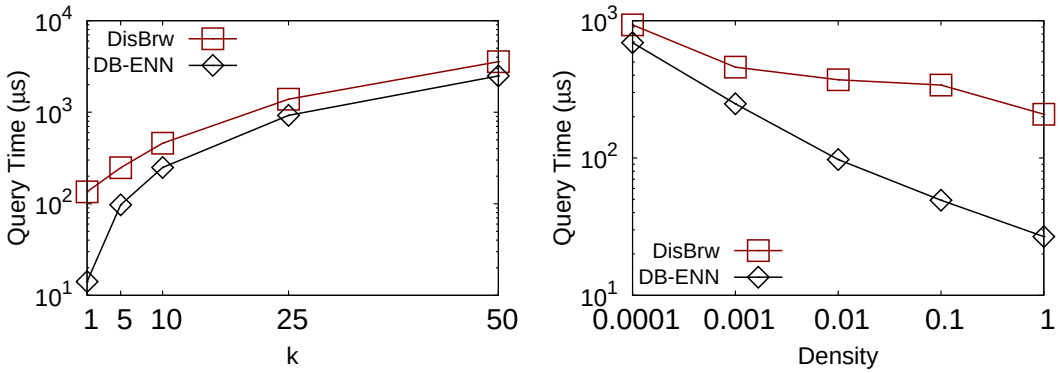
To overcome this, we propose a variant of DisBrw that eliminates computing intersections called DB-ENN presented in Algorithm 1. Essentially, we replace the Object Hierarchy with Euclidean NNs to generate candidates (recall that Euclidean distances are already used to compute distance ratios [SSA08]). We first retrieve Euclidean kNNs using an R-tree as the initial candidates and then suspend the search (i.e., we keep the priority queue E used by the Euclidean kNN search). Now we compute distance intervals for each candidate and insert these candidates into queues Q and L, setting D_k as the largest upper bound. DisBrw proceeds as before, except before dequeuing an element from Q, we check

if $Front(E) < Front(Q)$. If true, there may be a closer Euclidean NN, so we retrieve the next Euclidean NN from E. This object is handled in the same way as an object in a leaf node of the Object Hierarchy (i.e., potentially inserted into Q and L).

Algorithm 1 Alternative SILC-based kNN query algorithm using Euclidean NNs

```
1: function GetKNNByENNSILC(v_q, k, SILC, Rt)
2:   Input: SILC is SILC quadtree for v_q and Rt is R-tree for object set
3:   Initialize min priority queue Q and max priority queue L
4:   Initialize min priority queue E for Rtree NN search and set D_k ← ∞
5:   K ← GetEuclideanKNNs(E, Rt, k)
6:   for each object v_o ∈ K do
7:     ProcessCandidate(Q, L, v_o, 0, D_k))          ▷ D_k will be set if k ≤ |O|
8:   while Q ≠ φ or E ≠ φ do
9:     if Front(E) < Front(Q) then
10:      (e, LB_e) ← GetNextEuclideanNN(E)
11:      ProcessCandidate(Q, L, e, LB_e, D_k))
12:    else
13:      ([e, UB_e, v_n, d], LB_e) ← Dequeue(Q)
14:      if UB_e ≥ D_k then
15:        break
16:      else
17:        if UB_e > Front(Q) or (UB_e = Front(Q) and UB_e ≠ LB_e) then
18:          if UB_e ≤ D_k and Contains(L, e) then
19:            Delete(L, e)
20:          (v_n, d, LB_e, UB_e) ← Refine(v_n, d, LB_e, UB_e)
21:          if UB_e ≤ D_k then
22:            UpdateL(L, e, UB_e, D_k)
23:          if LB_e ≤ D_k then
24:            Enqueue(Q, ([e, UB_e, v_n, d], LB_e))
25:        else                  ▷ e dropped implicitly, no further refinement needed
26:  Populate(R, L)   ▷ Dequeue from L to populate R so results are in distance order
27:  return R
28: function ProcessCandidate(Q, L, o, LB_o, D_k)
29:  (v_n, d, LB_o, UB_o) ← Refine(v_q, 0, LB_o, inf)
30:  if LB_o < D_k then
31:    Enqueue(Q, ([o, UB_o, v_n, d], LB_o))
32:    if UB_o < D_k then
33:      UpdateL(L, o, UB_o, D_k)
```

We compare DisBrw (improved as in Appendix B.1) to DB-ENN in Figure 3.6. DB-ENN's improvement increases with higher density and smaller k as this is when the overhead from the Object Hierarchy is highest. The improvement peaks at 1 order of magnitude. Since this suggests that Object Hierarchies do not use the SILC index to its full potential, and the Euclidean heuristic is more effective, we use DB-ENN in our experiments instead of the original DisBrw algorithm.



(a) Varying k (b) Varying Density d

Figure 3.6: DisBrw vs. DB-ENN (NW, d=0.001, k=10)

3.6 Experiments

3.6.1 Experimental Setting

Environment. We conducted experiments on a 3.2GHz Intel Core i5-4570 CPU and 32GB RAM running 64-bit Linux (kernel 4.2). Our program was compiled with g++ 5.2 using the O3 flag, and all query algorithms use a single thread. To ensure fairness, we used the same subroutines for common tasks between the algorithms whenever possible. We implemented INE, IER, G-tree, and ROAD from scratch. We obtained the authors' code for G-tree, which we used to further improve our implementation, e.g., by selecting the better option when our choices disagreed with the authors' choice of data structures. For Distance Browsing, we partly based our SILC index on open-source code from [Wu+12], but being a shortest path study this implementation did not support kNN queries. As a result, we implemented the kNN algorithms ourselves from scratch, modifying the index to support them, taking the opportunity to make significant improvements (as discussed in Section 3.5, Appendix B and Appendix A). We used a highly efficient open-source implementation of PHL made available by its authors [Aki+14]. All source code and scripts to generate datasets, run experiments, and draw figures have been released as open-source [Abe16] for readers to reproduce our results or re-use in future studies.

Index Parameters. The performance of the G-tree and ROAD indexes are highly dependent on the choice of leaf capacity τ (G-tree), hierarchy levels l (ROAD) and fanout f (both) [Zho+15; Lee+12; LLZ09]. We experimentally confirmed trends observed in those studies and computed parameters for new datasets. As such, we use fanout f=4 for

both methods. For G-tree we set τ to 64 (DE), 128 (VT, ME, CO), 256 (NW, CA, E), and 512 (W, C, US). For ROAD, we set l to 7 (DE), 8 (VT, ME), 9 (CO, NW), 10 (CA, E) and 11 (W, C, US). We chose values of l for ROAD in accordance with the results reported in [LLZ09] that show query performance of ROAD improves for larger l. Specifically, for each dataset, we increased l until either the query performance did not improve or further partitioning was not possible due to too few vertices in the leaf levels.

Query Variables. Table 3.3 shows the range of each variable used in our experiments (defaults in bold). Similar to past studies [Zho+15], we vary k from 1 to 50 with a default of 10. We used 8 real-world object sets as discussed Section 3.4. We vary uniform object set density d from 0.0001 to 1 where d=$|O|/|V|$ with a default value of 0.001. We choose this default density as it closely matches the typical density for real-world object sets as shown in Table 3.2. Furthermore, this density creates a large enough search space to reveal interesting performance trends for methods. We vary over 10 real road networks (listed in Table 3.1) with median-sized NW and largest US road networks as defaults. We use distance edge weights in Section 3.7 for comparison with past studies, especially because IER and DisBrw were developed for such graphs. But we repeat experiments on travel times later in Section 3.8 for completeness.

Parameter	**Values**
Road Networks	DE, VT, ME, CO, **NW**, CA, E, W, C, **US**
k	1, 5, **10**, 25, 50
Density (d)	1, 0.1, 0.01, **0.001**, 0.0001
Synthetic POIs	**uniform**, clustered, min. obj. distance
Real POIs	Refer to Table 3.2

Table 3.3: kNN Experimental Parameters (Defaults in Bold)

Query and Object Sets (Section 3.4.2). All query times are averaged over 10,000 queries. For real-world object sets, we tested each set with 10,000 random query vertices. For uniform and clustered object sets, we generate 50 different sets for each density and number of clusters, respectively, combined with 200 random query vertices. For minimum distance object sets, we generated 50 sets for each distance set R_i with $i \in [1, m]$. We also chose 200 random query vertices with distances from the center vertex in range $[0, \frac{D_{max}}{2^m})$ (i.e., vertices closer than R_1) for use with all sets. We use m=6 for NW and m=8 for US to ensure there were enough objects in each set to satisfy the default density 0.001.

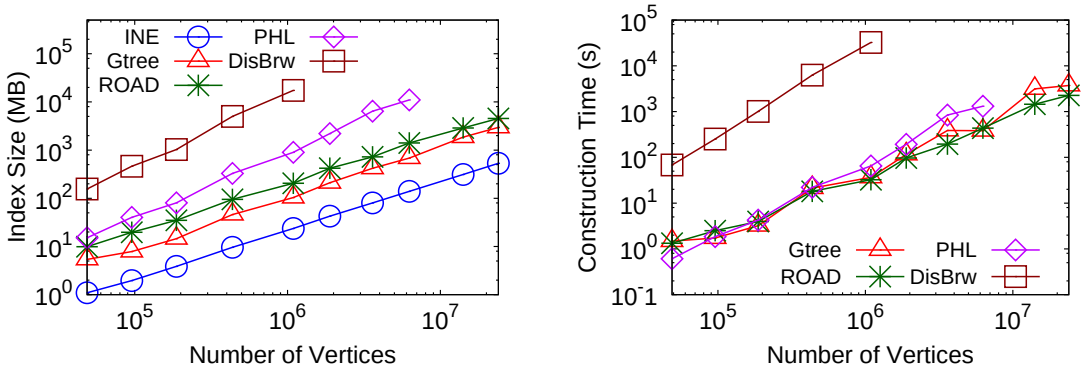
3.7 Travel Distance Experiments

We first conduct our experiments on travel distance road networks, as in past studies, for the state-of-the-art techniques. However, we repeat these experiments for the travel time case in section 3.8.

3.7.1 Road Network Index Pre-Processing Cost

Here we measure the construction time and size of the index used by each technique for all road networks in Table 3.1.

Index Size. Figure 3.7(a) shows the index size for each algorithm. INE only uses the original graph data structure, so its size can be seen as the lower bound on space. DisBrw could only be built for the first 5 road networks before exceeding our memory capacity. This is not surprising given the $O(|V|^{1.5})$ storage complexity. However, in our implementation, we were able to build DisBrw for an index with 1 million vertices (NW) consuming 17GB. PHL also exhibits large indexes, however, it can still be built for all but the 2 largest datasets. We note that PHL experiences larger indexes on travel distance graphs because they do not exhibit prominent hierarchies needed for effective pruning (on travel time graphs we were able to build PHL for all indexes). G-tree consumed less space than ROAD. For example, for the US dataset G-tree used 2.9GB compared to ROAD's 4.4GB. As explained in past studies [Zho+15], ROAD's Route Overlay contains significant redundancy as multiple shortcut trees repeatedly store a subset of the Rnet hierarchy.



(a) Index Size

(b) Construction Time

Figure 3.7: Pre-Processing Cost vs. Road Network Size $|V|$

Construction Time. Figure 3.7(b) compares the construction time of each index for increasing network sizes. DisBrw again stands out as its index (SILC [SAS05]) requires

an all-pairs shortest path computation. However, the computation of each SILC quadtree is independent and can be easily parallelized. We observed a speed-up factor of very close to 4× with our quad-core CPU using OpenMP. Note that other methods cannot be so easily parallelized. Despite this DisBrw still required 9 hours on NW, while parallelization is useful it does not change the asymptotic behavior. PHL takes longer than G-tree and ROAD but surprisingly not significantly so, thanks to pruned labeling [Aki+14]. IER's index performance depends on the network distance method it employs (i.e., G-tree or PHL).

Recall that both ROAD and G-tree must partition the road network. Since the network partitioning problem is known to be NP-complete, ROAD and G-tree both employ heuristic algorithms. As both methods require the same type of partitioning, we use the same multilevel graph partitioning algorithm [KK98] used in G-tree. This method uses a much faster variant of the Kernighan-Lin algorithm recommended in ROAD [Lee+12]. Consequently, we are able to evaluate ROAD for much larger datasets for the first time, with ROAD being constructed in less than one hour for even the largest dataset (US) containing 24 million vertices. The construction time of ROAD is comparable to G-tree, because both use the same partitioning method, and employ bottom-up methods to compute shortcuts and distance matrices, respectively.

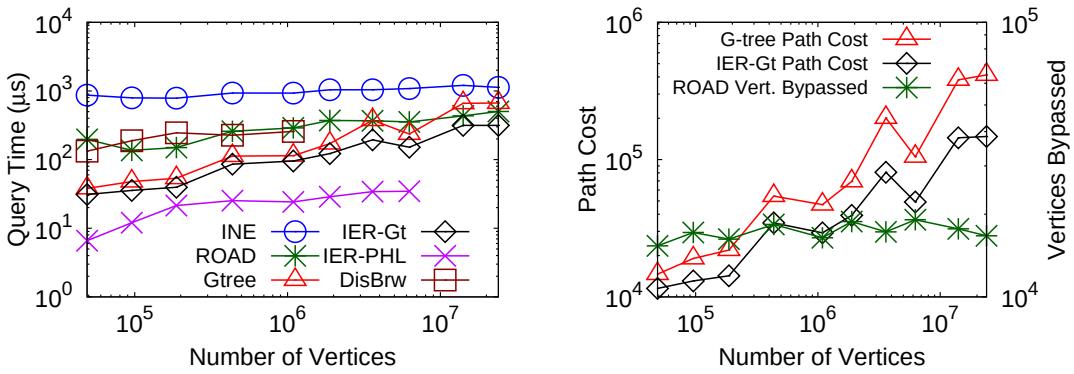
We remark that, while most existing studies have focused on improving query processing time, there is a need to develop algorithms and indexes providing comparable efficiency with a focus on reducing memory usage and construction time.

3.7.2 Query Performance

We investigated kNN query performance over several variables: road network size, k, density, object distance, clusters, and real-world POIs. Implementations have been optimized according to Appendix A. We have applied numerous improvements to each algorithm, as detailed in Section 3.5 and Appendix B. IER network distances are computed using both PHL [Aki+14] (when its index fits in memory) and G-tree with materialization (shown as IER-PHL and IER-Gt, respectively).

Varying Network Size

Figure 3.8(a) shows query times with increasing numbers of road network vertices $|V|$ for all 10 road networks in Table 3.1 on uniform objects. We observe the consistent superiority of IER-based methods. Figure 3.8(a) clearly shows the reduced applicability of DisBrw. Even though its performance is close to ROAD, its large index size makes it applicable to only the first 5 datasets.



(a) Query Time (b) G-tree & ROAD Stats

Figure 3.8: Effect of Road Network Size $|V|$ (d=0.001,k=10)

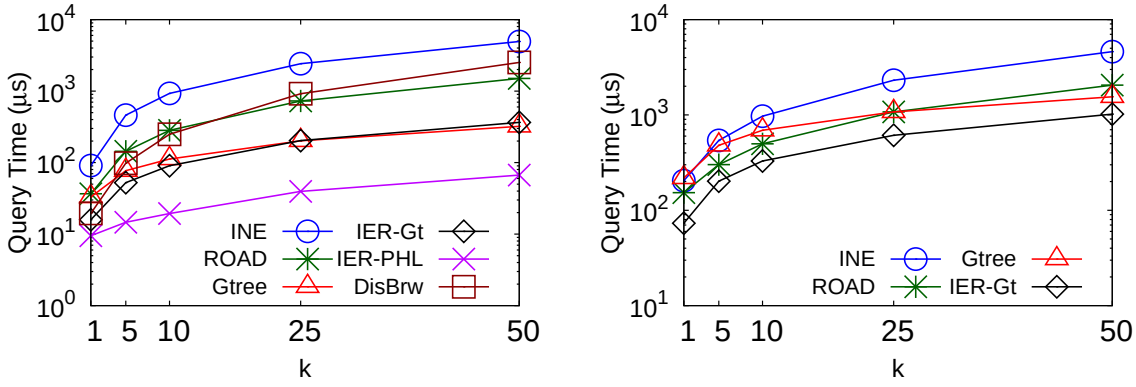
Surprisingly G-tree's advantage over ROAD decreases with increasing network size $|V|$. Recall that ROAD can be seen as an optimization on INE, where the expansion can bypass object-less regions (i.e., Rnets). Thus, ROAD's relative improvement over INE depends on the time saved bypassing Rnets versus additional time spent descending shortcut trees. In general, given the same density, we can expect a region of similar size to contain the same number of objects irrespective of the network size $|V|$. This explains why INE remains relatively unaffected by $|V|$. It also means that regions without objects are similarly sized. Although Rnets may grow, the size of the Rnets we do bypass also grows, so ROAD bypasses similar numbers of vertices. So, the time saved bypassing regions does not increase greatly. Hence, ROAD's query time with increasing $|V|$ mainly depends on the depth of shortcut trees. But the depth is bounded by l, which we know does not increase greatly, and as a result, ROAD scales extremely well with increasing $|V|$.

G-tree's non-materialized distance computation cost is a function of the number of borders of G-tree nodes (i.e., subgraphs) involved in the tree path to another node or object. With increasing network size, a G-tree node at the same depth has more borders and the path cost is consequently higher. Thus, we see G-tree "catch-up" to ROAD on

the US dataset. These trends are demonstrated in 3.8(b). G-tree's path cost (in border-to-border computations) increases while the number of vertices ROAD bypasses remains stable with increasing $|V|$ (note these are not directly comparable).

Varying k

Figures 3.9(a) and 3.9(b) show the results for varying k for the NW and US datasets, respectively, on uniform objects. Significantly, IER-PHL is 5× faster than any other method on NW. While PHL could not be constructed for the US dataset for travel distances, IER-Gt takes its place as the fastest method, being twice as fast as G-tree. Interestingly, this is despite both using the same index, also materializing intermediate results, and IER-Gt having the additional overhead of retrieving Euclidean NNs. So, this is really an examination of heuristics used by G-tree. Essentially G-tree visits the closest subgraph (i.e., by one of its borders) while IER-Gt visits the subgraph with the next Euclidean NN. IER-Gt can perform better because its heuristic incorporates an estimate on distances to objects within subgraphs while G-tree does not. Each time G-tree visits a subgraph not containing a kNN it pays a penalty in the cost of non-materialized distance computations. We have seen this cost increases with network size, which explains why the improvement of IER-Gt is greater on the US than on NW. This is verified in Figure 3.8(b), which shows IER-Gt involves fewer computations than G-tree and the gap increases with network size.



(a) NW Dataset

(b) US Dataset

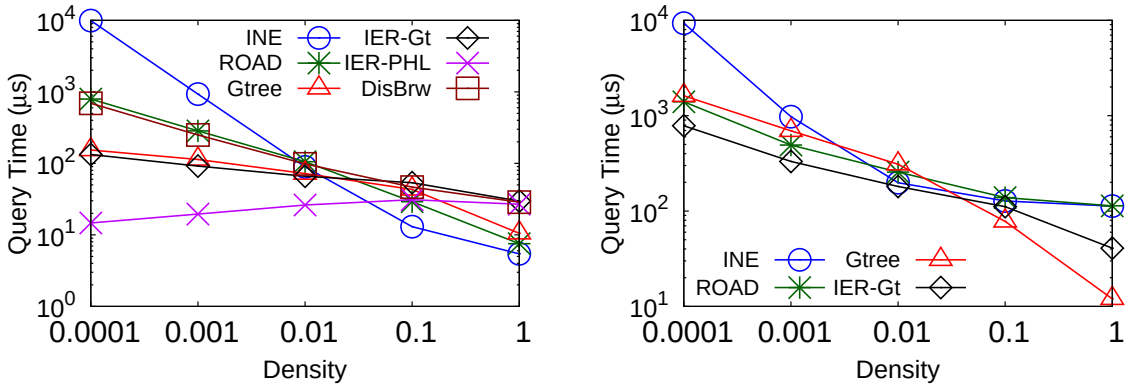
Figure 3.9: Effect of k (d=0.001,uniform objects)

We observe that G-tree outperforms ROAD, DisBrw, and INE on NW, with a trend similar to previous studies [Zho+15]. INE is the slowest as it visits many vertices. For $k = 1$ the ROAD, DisBrw and G-tree methods are indistinguishable as a small area is

likely to contain the 1NN. ROAD and DisBrw scale very similarly with k. G-tree scales better than both, at its peak nearly an order of magnitude better than ROAD and DisBrw. As more objects are located, more paths in the G-tree hierarchy are traversed, allowing greater numbers of subsequent traversals to be materialized. As explained in Section 3.7.2, we again see G-tree's relative improvement over ROAD decrease in Figure 3.9(b) for the larger US dataset.

Varying Density

We evaluate performance for varying uniform object densities in Figure 3.10. With increasing density, the average distance between objects decreases and in general query times are lower. The rate of improvement for heuristic-based methods (DisBrw, G-tree, IER) is slower because they are less able to distinguish better candidates. For IER this means more false hits, explaining why IER-PHL's query times increase (slightly) as it has no means to re-use previous computations like IER-Gt does. The rate of improvement is higher for expansion-based methods as their search spaces become smaller. ROAD falls behind INE beyond density 0.01 indicating the tipping point at which the time spent traversing shortcut trees exceeds the time saved bypassing Rnets (if any). The query times plateau at high densities on the US dataset for ROAD and INE because it is dominated by the bit-array initialization cost (refer to Appendix A.3). G-tree performs well at high densities as more kNNs are found in the source leaf node. In this case, it reverts to a Dijkstra-like search (which we improved as in Appendix B.2.1) providing comparable performance to INE and ROAD on NW. G-tree exceeds them on the US as a bit-array is not required due to G-tree's leaf search being limited to at most τ vertices.



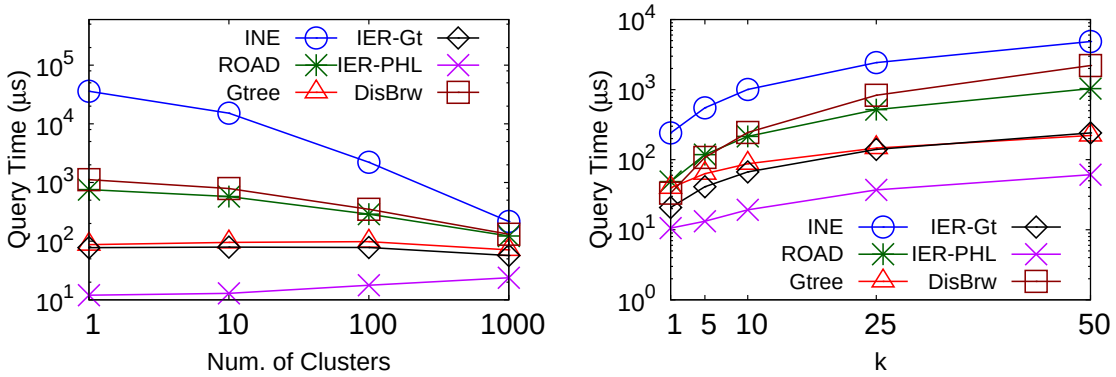
(a) NW Dataset

(b) US Dataset

Figure 3.10: Effect of Density (k=10,uniform objects)

Varying Clusters

In this section, we evaluate performance on clustered object sets proposed in Section 3.4.2. Figure 3.11 shows the query time with increasing numbers of clusters and varying k. Cluster size is at most 5 in both cases. Figure 3.11(b) uses an object density of 0.001. As the number of clusters increases the average distance between objects decreases leading to faster queries. This is analogous to increasing density, thus showing the same trend as for uniform objects. IER-PHL's superiority is again apparent. One difference to uniform objects is IER-based methods find it more difficult to differentiate between candidates as the number of clusters increases, and query times increase (but not significantly). Similarly, in Figure 3.11(b), as k increases, IER-PHL visits more clusters, causing its performance lead to be slightly smaller than for uniform objects. IER-Gt on the other hand is more robust to this, as it is able to materialize most results. G-tree again performs better than DisBrw and ROAD. Due to clustering, objects in the same cluster will likely be in the same G-tree leaf node. After finding the first object, G-tree can quickly retrieve other objects without recomputing distances to the leaf node, thus remaining relatively constant.



(a) Varying No. of Clusters

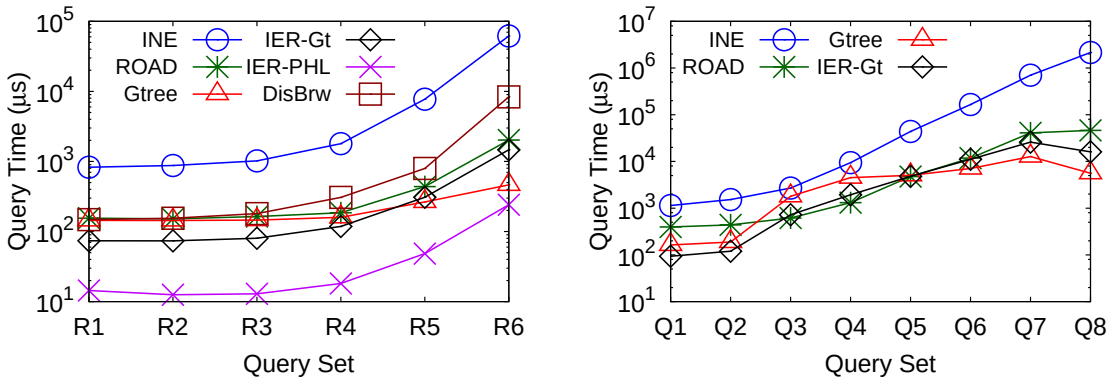
(b) Varying k

Figure 3.11: Effect of Clustered Objects (NW,$|C|$=0.001,k=10,clustered objects)

Varying Minimum Object Distance

Each set R_i in Figure 3.12 represents an exponentially increasing network distance to the closest object with increasing i, as described in Section 3.4.2. For the smallest sets, objects still tend to be found further away, as there are fewer closer vertices. However, as distance increases further, we see the effect of "remoteness". INE scales badly due to the increasing search space. IER-based methods scale poorly as the Euclidean lower bounds

become less accurate with increasing network distance. This is particularly noticeable in Figure 3.12(b) as G-tree eventually overtakes IER-Gt on the US. But IER-PHL still outperforms all methods on NW. DisBrw performs poorly for a similar reason, making many interval refinements. G-tree scales extremely well in both cases, as more paths are visited through the G-tree hierarchy and more computations can be materialized for subsequent traversals.



(a) NW Dataset (b) US Dataset

Figure 3.12: Effect of Minimum Object Distance (d=0.001,k=10,distance-based objects)

Real-World Object Sets

Varying Object Sets. In Figures 3.13 and 3.13, we show query times of each technique on typical real-world object sets from Table 3.2. These are ordered by decreasing size, which is analogous to decreasing density, showing the same trend as in Figure 3.10. Schools represent the largest object set and all methods are extremely fast as seen for high density. A more typically searched POI, like hospitals, are less numerous and show the differences between methods more clearly. Regardless, IER-PHL on NW and IER-Gt on the US consistently and significantly outperform other methods on most real-world object sets. Also note query times for G-tree are higher on the US than NW for the same sets, confirming our observations in Section 3.7.1.

Varying k. Figure 3.15 shows the behavior of two typically searched POIs, fast food outlets and hospitals, on the NW dataset. Hospitals display a trend similar to that of uniform objects for increasing k, as they tend to be sparse. IER-PHL is again significantly faster than G-tree. Although still fastest, IER-PHL has marginally lower performance for fast food outlets as these tend to appear in clusters where Euclidean distance is less able to

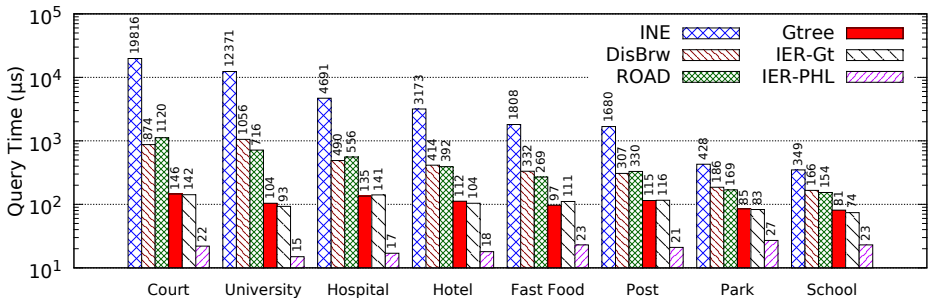


Figure 3.13: Varying Real-World Object Sets on NW Road Network (NW,k=10)

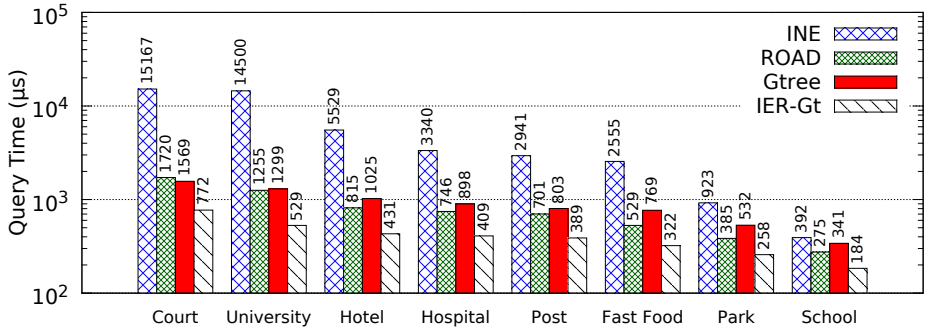
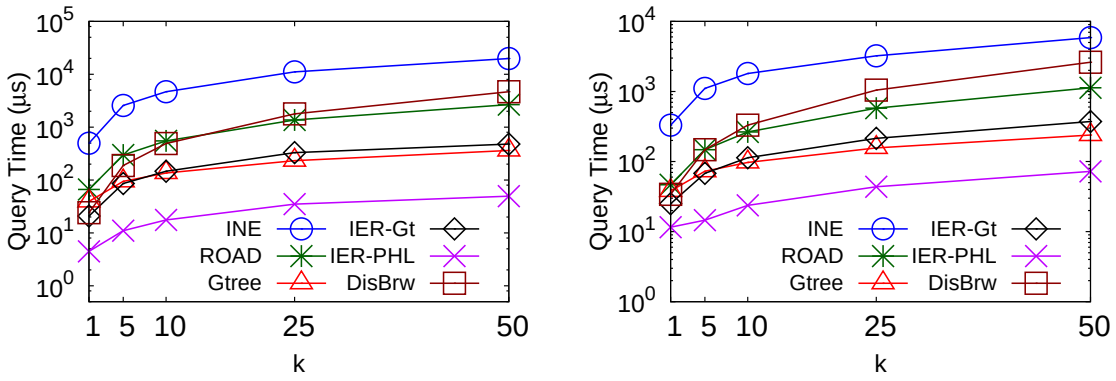


Figure 3.14: Varying Real-World Object Sets on US Road Network (US,k=10)

distinguish better candidates, similar to synthetic clusters in Figure 3.11(b). Thus, trends observed for equivalent synthetic object sets in previous experiments are also observed for real-world POIs.



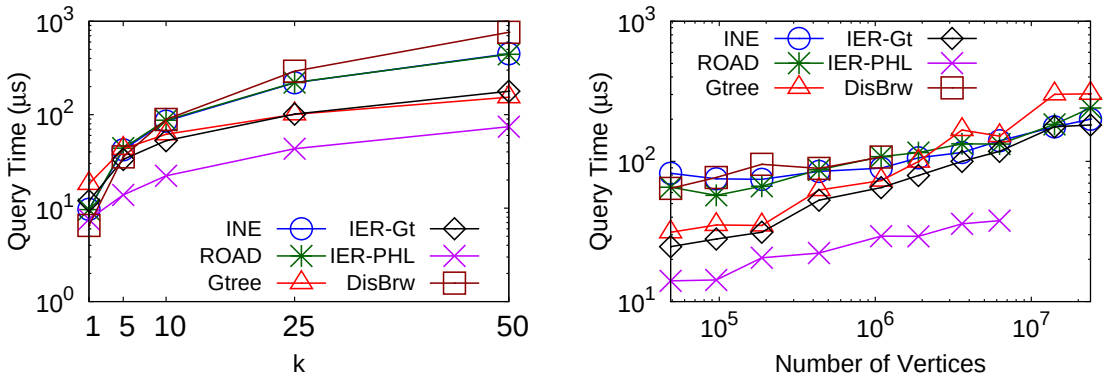
(a) Hospitals (b) Fast Food

Figure 3.15: Varying k for Real-World Objects (NW)

Original Settings

A recent experimental comparison [Zho+15] used a higher default density of d=0.01. While we choose a more typical default density, we reproduce results using d=0.01 in Figure 3.16

for varying k and network size. Note that we use the smaller Colorado dataset in Figure 3.16(a) for direct comparison with [Zho+15]. First, all methods compared in [Zho+15] now answer queries in less than 1ms. While our CPU is faster, it cannot account for such a large difference. This suggests our implementations are indeed efficient. Second, most methods are difficult to differentiate, as such a high density implies a very small search space (i.e., queries are "easy" for all methods).



(a) Varying k (b) Varying $|V|$

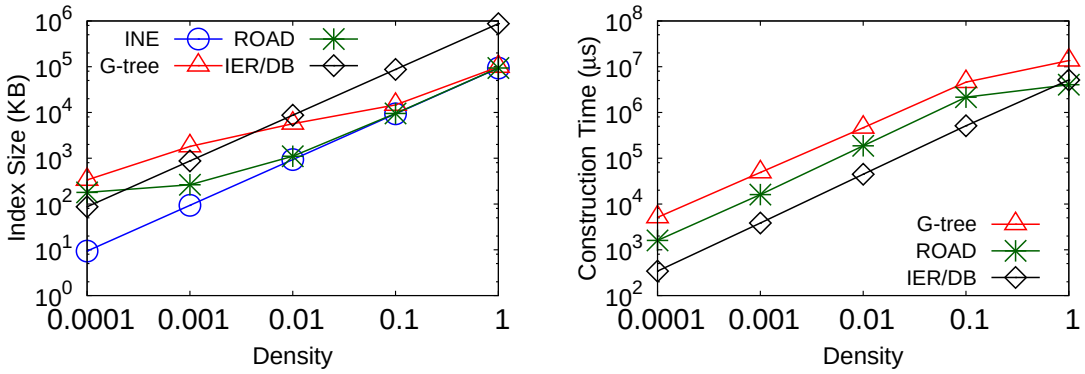
Figure 3.16: Default Settings from [Zho+13] (CO,d=0.01,k=10,uniform objects)

3.7.3 Object Set Index Pre-Processing Cost

The original ROAD paper [Lee+12] included the pre-processing cost for a single fixed object set in its road network index statistics. But there may be many object sets (e.g., one for each type of restaurant) or objects may need frequent updating (e.g., hotels with vacancies). Thus, we are interested in the performance of individual object indexes over varying size (i.e., density). We evaluate 3 object indexes on the US dataset, namely: *R-trees* used by IER, *Association Directories* used by ROAD and *Occurrence Lists* used by G-tree. Note that in our study DisBrw also uses R-trees (see Section 3.5.2).

Index Size. In practice, object indexes for all object sets would be constructed offline, loaded into memory and the appropriate one injected at query time. We investigate the index sizes (in KB) in Figure 3.17(a) to gauge what effect each density has on the total size. The size of the input object set used by INE is the lower bound storage cost. ROAD's object index is smaller than G-tree's because it need only store whether an Rnet contains an object or not, which is easily done in a low memory bit-array. G-tree's object index must additionally store the child nodes containing objects. Both indexes must however

store the actual objects, which gradually dominates the index size with increasing density. Note that we chose R-tree parameters (e.g., node capacity) for best performance. As a result, R-trees fall behind after density 0.01, but this can be remedied by increasing the node capacity at the expense of Euclidean kNN performance. We note that object indexes are much smaller than road network indexes, as they are simpler data structures, and real-world object sets with high densities are less frequent.



(a) Index Size vs. d

(b) Build Time vs. Density d

Figure 3.17: Object Indexes for US (uniform objects)

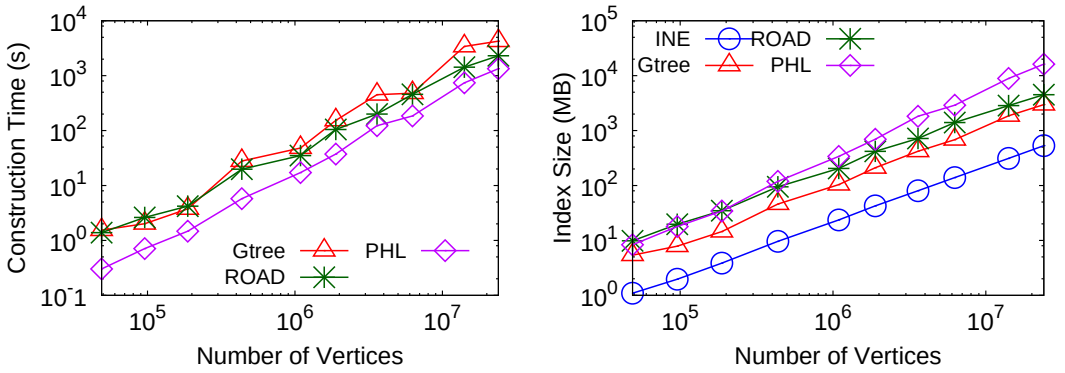
Construction Time. Figure 3.17(b) similarly shows the object index construction times. Again, they are all constructed much faster than road network indexes, due to being simpler data structures. The ROAD and G-tree object indexes incur the largest build time due to bottom-up propagation of the presence of objects through their respective hierarchies. However, the R-trees used by IER are significantly faster to build. As R-trees support updates, this suggests the possibility of use in real-time settings.

3.8 Travel Time Experiments

kNNs may just as commonly be required in terms of travel time. In this section, we reproduce query results on road networks with travel time edge weights. Notably, the state-of-the-art techniques have never previously been compared in this setting. Note that we do not test DisBrw on travel times, as the additional information (i.e., distance ratios) stored in the SILC index relies heavily on Euclidean distance, making it more complex to adapt than IER and likely to perform significantly slower than on travel distances.

3.8.1 Road Network Pre-Processing and Space

Figure 3.18 shows the index construction time and index size for the travel time edge weight versions of the road networks in Table 3.1. The key difference to travel distances is that PHL is constructed faster (in fact faster than the other methods) and uses significantly less memory allowing it to be constructed for all datasets up to and including the US dataset with 24 million vertices. Travel time graphs display better hierarchies, allowing for more effective pruning, leading to smaller label sizes on average. Note we do not need to repeat object index comparisons for travel times as they will be the same as for travel distances. For example, the same partitioning of the road network is used to construct a G-tree (respectively, ROAD) index in either case, which means Occurrence Lists (respectively, Association Directories) will be identical to the travel distance versions.



(a) Construction Time (b) Index Size

Figure 3.18: Pre-Processing Cost vs. Road Network Size $|V|$

3.8.2 Query Performance

We now present results for query performance on travel time road networks. We note that overall, sometimes surprisingly, many of the trends observed for travel distances are similarly observed for travel times.

Varying Network Size

Similar to the travel distance case, G-tree is seen to degrade with increasing $|V|$ in Figure 3.19 as before. However, we were also able to construct the PHL index for *all* datasets, due to the previously described presence of hierarchies in travel time road networks, allowing

us to compare IER-PHL for even the continental US dataset. More surprisingly, IER-PHL outperforms all other techniques on all datasets in Figure 3.19. This is despite the Euclidean lower bound being much looser on travel times (as described in Section 3.5.1).

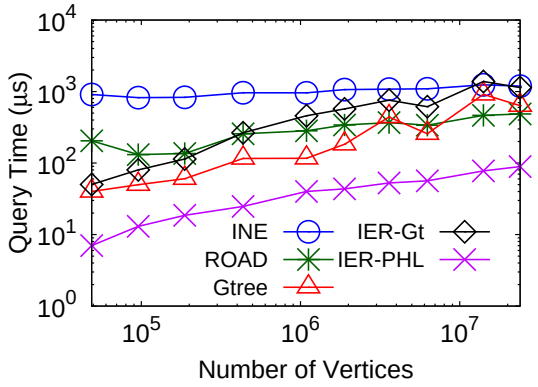
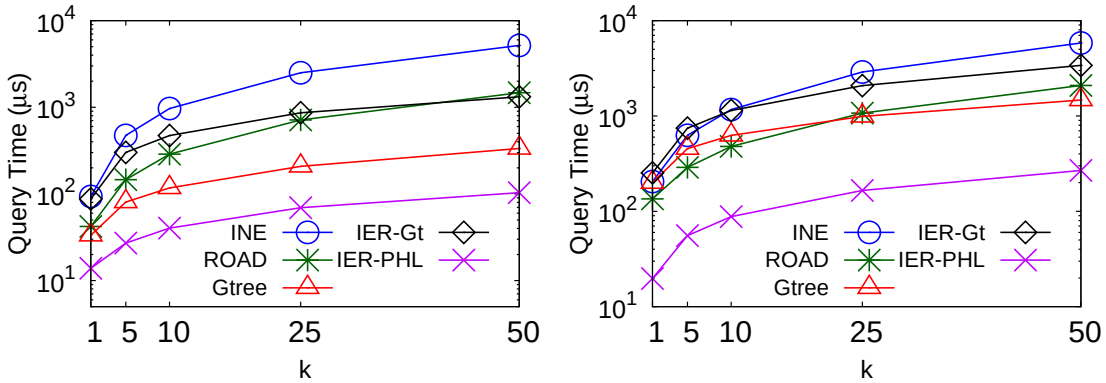


Figure 3.19: Effect of Road Network Size $|V|$ (d=0.001,k=10,uniform objects)

Varying k

In general, we can expect IER-based methods to experience more false hits due to the looser lower bound on travel times. This explains why IER-Gt is now outperformed by G-tree in Figure 3.20. But the penalty paid by IER in false hits is not enough to stop IER-PHL still significantly outperforming all other methods, remaining the fastest in most situations. In fact, the margin of improvement achieved by IER-PHL over the other techniques remains similar to travel distances. This is due to the increased cost of false hits being partly offset by the reduced label sizes for PHL, which makes network distance queries faster.

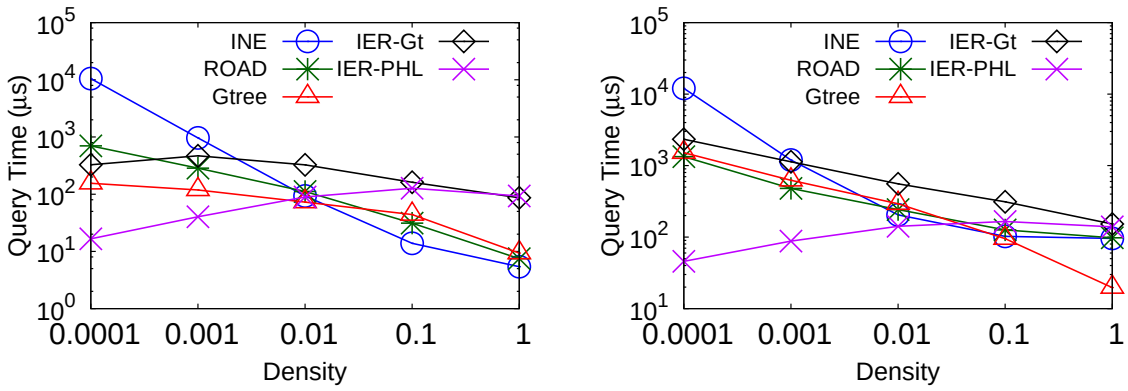


(a) NW Dataset (b) US Dataset

Figure 3.20: Effect of k (d=0.001,uniform objects)

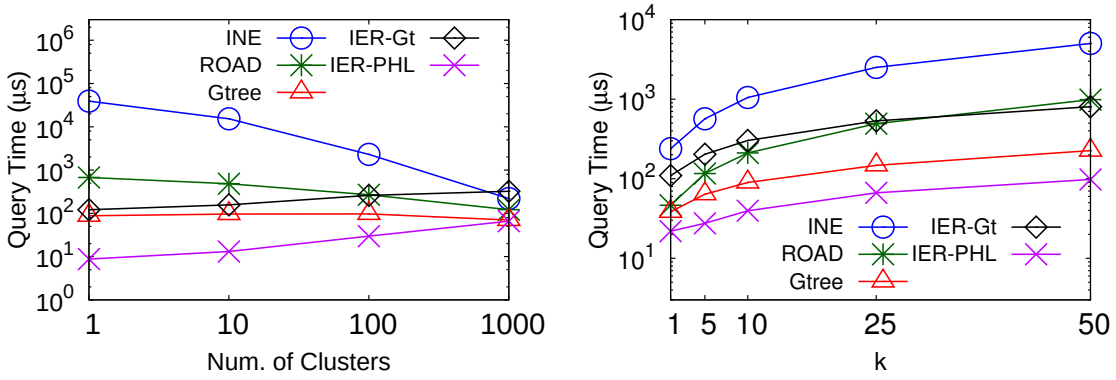
Varying Density, Clusters and Minimum Object Distance

There are a few notable situations where looser lower bounds aggravate cases where Euclidean distance was already less effective on travel distances. For example, IER was already less able to distinguish better candidates with increasing density, and as a result, IER-PHL degrades faster on travel times in Figure 3.21. However, we note that a density of 1, where every vertex has an object is not a realistic real-world scenario, and in such cases, INE is the optimal method to use anyway. This is similarly observed for increasing numbers of clusters in Figure 3.22 and increasing network distance in Figure 3.23 as the lower-bounding error introduced by Euclidean distance worsens.



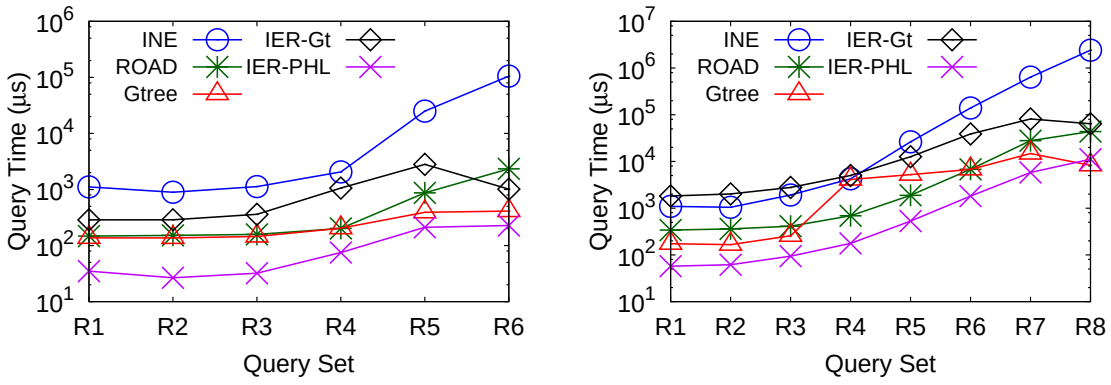
(a) NW Dataset (b) US Dataset

Figure 3.21: Effect of Density (k=10,uniform objects)



(a) Varying No. of Clusters (b) Varying k

Figure 3.22: Effect of Clustered Objects (NW,$|C|$=0.001,k=10,clustered objects)



(a) NW Dataset (b) US Dataset

Figure 3.23: Effect of Minimum Object Distance (d=0.001,k=10,distance-based objects)

Real-World Object Sets

We repeat the experiments for real-world object sets in Figures 3.24, 3.25 and 3.26. All observations we have made so far are similarly observed here. For example, we observe the same trends for increasing real-world object set size in Figure 3.24 as for increasing object density. We also observe that G-tree again performs worse on the US dataset than on NW. One major difference is that IER-PHL is included in comparisons for the US dataset in Figure 3.25, since its index can be constructed for all datasets. This offsets the degraded performance of IER-Gt. The sparse object set (Hospitals) and the clustered object set (Fast Food) again show similar trends, with IER having degraded performance on the clustered object set as it is less able to distinguish candidates. Again, this effect is aggravated for travel times.

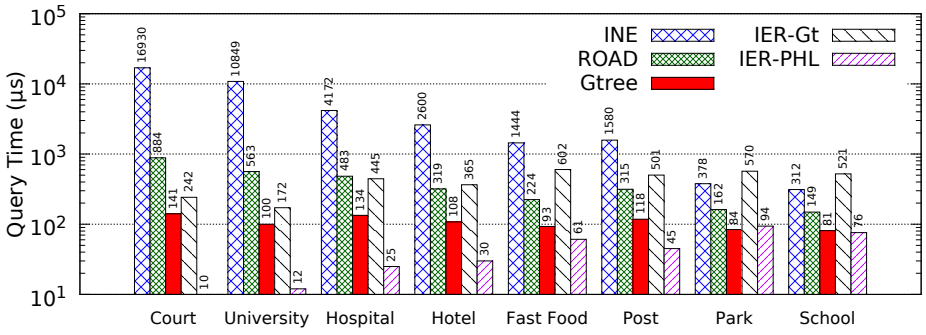


Figure 3.24: Varying Real-World Object Sets on NW Road Network (NW,k=10)

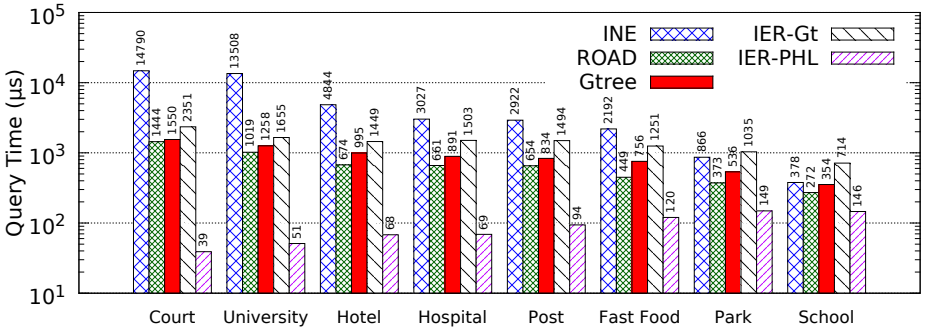
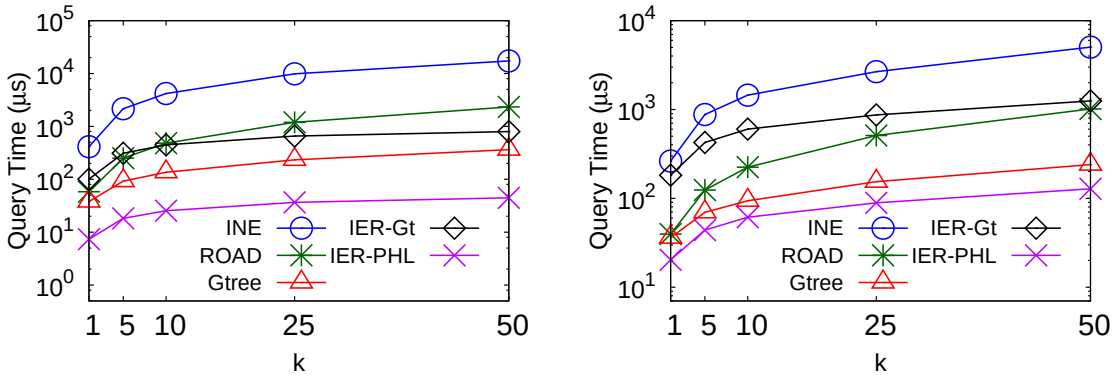


Figure 3.25: Varying Real-World Object Sets on US Road Network (US,k=10)



(a) Hospitals (b) Fast Food

Figure 3.26: Varying k for Real-World Objects (NW)

3.9 Summary

We have presented an extensive experimental study for the kNN problem on road networks, settling unanswered questions by evaluating object indexes, travel time graphs and real-world POIs. We verify that G-tree generally outperforms INE, DisBrw and ROAD, but the relative improvement is much smaller and at times reversed, demonstrating the impact of implementation efficiency. Table 3.4 provides the ranking of the algorithms under different criteria.

Criteria	INE	G-tree	ROAD	IER	DisBrw
Query Performance					
Default Settings	5th	2nd	=3rd	1st	=3rd
Small k	5th	=3rd	=3rd	1st	2nd
Large k	5th	2nd	3rd	1st	4th
Low Density	5th	2nd	=3rd	1st	=3rd
High Density	1st	3rd	2nd	4th	5th
Small Networks	5th	2nd	=3rd	1st	=3rd
Large Networks	4th	=3rd	2nd	1st	N/A
Network and Object Index Pre-Processing					
Time (Network)	1st	3rd	2nd	4th	5th
Time (Objects)	1st	5th	4th	=2nd	=2nd
Space (Network)	1st	2nd	3rd	4th	5th
Space (Objects)	1st	5th	2nd	=3rd	=3rd

Table 3.4: Ranking of kNN Algorithms Under Different Criteria

Our most significant conclusions are regarding IER, which we investigated with fast network distance techniques for the first time. IER-PHL significantly outperformed every competitor in all but a few cases, even on travel time graphs where Euclidean distance is less effective. IER provides a flexible framework that can be combined with the fastest shortest path technique allowed by the users' memory capacity and must be included in future comparisons. Additionally, on travel distances, we saw that IER-Gt often outperformed the original G-tree kNN algorithm despite using the same index. We even see that this observation can be generalized further as we also improved DisBrw using Euclidean distance.

3.9.1 To Blend or Not to Blend

Our results for IER-Gt and DisBrw show that Euclidean distance is a better heuristic than those proposed in the original works. In the case of G-tree, the original G-tree kNN algorithm searches its hierarchy by visiting G-tree nodes in the hierarchy closest to the query vertex. The distance to a G-tree node is determined by computing the distance to

the nearest border vertex of the associated subgraph. Thus, there is no object specific information involved in this part of the kNN search. In fact, the only information used to guide the search unique to the object set is the indication of whether the subgraph contains an object or not (via the Occurrence List object index). On the other hand, using G-tree in the IER algorithm can be thought of as guiding the search through the G-tree hierarchy by using the minimum Euclidean distance to an object. Our results suggest that even though Euclidean distance underestimates network distance, this object specific information is more accurate than using the network distance to the nearest border, which may be very far from the actual object contained in the subgraph.

Now, generally, this identifies significant room for improvement in kNN search heuristics, if such a simple heuristic like Euclidean distance not known to be particularly accurate outperforms state-of-the-art heuristics. More specifically, given the comparison between searching the G-tree hierarchy using the original heuristic and Euclidean distance in the previous paragraph, it suggests that the object index component of decoupled indexes need to incorporate more information specific to the object set. In Section 3.2.2, we compared the relative advantages of decoupled indexing (like the use of the G-tree road network index with the Occurrence List object index) and often prohibitive disadvantages of blended indexing. The significant pre-processing benefits of decoupled indexes are hard to ignore, especially given the increasingly large road network datasets used today. But it appears the process of decoupling and the resulting performance increases from advances in computing network distance has led to a devaluing of incorporating object set specific information into the object indexes. In blended indexing, this happened naturally, as each index was constructed for a specific object set. We use this insight as a motivating factor for the direction of our subsequent work, starting with improving heuristics for kNN search in Chapter 4. That is, to incorporate more object set specific information into object indexes while preserving the pre-processing benefits of decoupled indexing.

Chapter 4

Landmark-Based Strategies for kNN Search Heuristics

> The whole art of war consists of guessing at what is on the other side of the hill.
>
> Duke of Wellington

We identified the unexpected impressive query performance of a simple Euclidean heuristic compared to state-of-the-art techniques in Chapter 3. Remarkably, this was true even on travel-time road networks. However, from studies on the shortest path problem, Euclidean distance is known to be a less effective heuristic on road networks with travel-time edge-weights. In this chapter, we use our earlier insight into the effectiveness of the Euclidean distance decoupled heuristic to develop new heuristics that further improve kNN querying. The research in this chapter appeared in [AC17].

4.1 Overview

As described earlier, a k Nearest Neighbor (kNN) query finds the k closest POIs (objects) to a query location by their network distance. But computing network distances to all of the potentially thousands of objects, only to report a few, is infeasible in the context of real-time map-based services. Recall from Chapter 3 that Incremental Euclidean Restriction (IER) [Pap+03] is a kNN technique that uses a simple Euclidean distance heuristic to avoid having to compute network distances to all objects. IER retrieves Euclidean kNNs as *candidates* and computes network distances to each one using a shortest path algorithm

(e.g., Dijkstra). The kth furthest candidate implies an upper bound network distance to the real kth NN. IER then iteratively retrieves further Euclidean NNs as candidates, computes network distances to each candidate and updates the k best candidates if closer POIs are found. Since Euclidean distance is a lower bound on network distance, IER terminates when the distance to the next Euclidean NN is larger than the network distance to the kth candidate, as the candidate set cannot be improved.

Our improvement of IER using faster network distance techniques and subsequent experimental comparisons to the state-of-the-art in Chapter 3, showed that this approach is an unexpectedly effective heuristic compared to the state-of-the-art. This was especially true in the case of road networks with travel distance edge-weights. However, while it may have outperformed existing techniques, Euclidean distance is only a "good" heuristic when it is a tight lower-bound on network distance. For example, the Euclidean heuristic only provides a loose lower-bound in travel-time road networks. As we discuss next, using alternative lower-bounding heuristics with tighter lower-bounds is a challenge in itself.

4.1.1 Motivations & Contributions

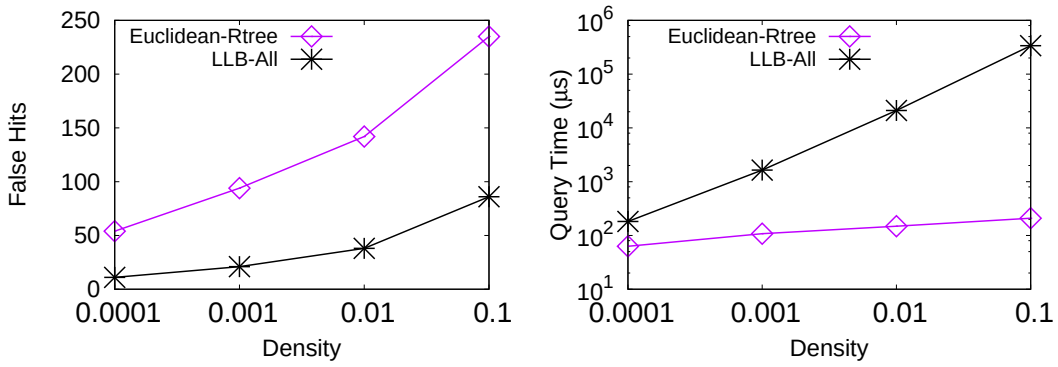
Euclidean distance is a lower-bound on the road network distance when edge weights in the road network are physical distances. It can also be made into a lower-bound for other metrics. For example, for travel time edge weights, we can divide the Euclidean distance by the maximum speed over all edges to obtain a minimum possible travel time as shown in Section 3.5.1. This however has a negative impact on kNN querying by (1) making Euclidean NN candidates less likely to be true kNN results and (2) taking longer to terminate candidate generation as the lower-bounds are looser compared to network distance. This results in wasteful network distance computations to "false hit" candidates that are not real kNNs. This was evident as IER's advantage was smaller in several travel time experiments in Section 3.8.

Landmark Lower Bounds (LLBs) are an alternative lower-bounding method based on pre-computed network distances to *landmark* vertices and the triangle inequality. LLBs were used by Goldberg and Harrelson [GH05] to significantly improve A* search due to the tighter lower-bounds provided than Euclidean distance. Naturally, as the shortest path problem is closely related to the kNN problem, this raises the question of whether these landmark lower bounds can be used to similarly improve IER's kNN query performance.

Until our study, the answer to this question has been "no". As we elaborate next, this is because using LLBs efficiently is not a trivial task.

Incrementally retrieving the candidate with the smallest Euclidean distance is efficiently achieved using a Euclidean NN search algorithm on an R-tree. By "incrementally" we mean by computing only a small number of lower-bounds to determine the next Euclidean candidate. However, an analogous and efficient method to incrementally retrieve candidates by their LLBs has never been proposed. Consequently, all past attempts to use LLBs for kNN querying [Kri+07; Kri+08] have resorted to computing LLBs for *all* POIs. By sorting all LLBs, the object with the smallest LLB can be found. But this is neither efficient as it is required for every new query, nor is it practicable as most POI sets number in the thousands, e.g., the 25,000 fast food outlets in the US (Table 3.2).

Figure 4.1 demonstrates the magnitude of this problem for kNN queries on the US travel time road network. Figure 4.1(a) shows that fewer false hits are encountered when using landmark lower-bounds. However, Figure 4.1(b) suggests that this does not translate into improved query time when all LLBs are being computed. In fact, increasing object density (i.e., the ratio of POIs to vertices) requires more LLB computations resulting in increasingly poorer performance. This clearly demonstrates our argument, in that (1) LLB-based methods provide better lower bounds as evident from the fewer false hits and (2) LLB-based methods perform very poorly without the ability to retrieve candidates incrementally. We propose methods that remedy (2) such that we can, finally, take advantage of (1).



(a) False Hits

(b) Query Time

Figure 4.1: Euclidean kNN vs. Landmark kNN Querying (US,k=10,uniform objects)

We identify IER, more generally, as an example of a *decoupled heuristic*. The heuristic is decoupled in the sense that retrieving candidate objects is independent of computing network distances. In this chapter, we propose new decoupled heuristics that effectively utilize landmark lower bounds to efficiently answer kNN queries. Specifically:

- We propose the *object-first* strategy to incrementally retrieve candidates. We revisit a neglected data structure, the Network Voronoi Diagram (NVD), and use it to retrieve objects which may be the next candidate to which we compute LLBs. We then utilize a useful property of NVDs to avoid computing LLBs to other objects. This approach is extremely efficient for kNN querying, significantly outperforming competing methods on running time and heuristic performance.

- We also propose the *lower-bound first* strategy. This approach searches a list of sorted distances to find the LLB belonging to the next candidate. In the process, we propose a new index, Object Lists (OL), with very low pre-processing time and index size in theory and practice. This approach demonstrates the difficulties in using LLBs but is efficient in some scenarios.

- We empirically verify that LLBs produce more accurate kNN candidates than Euclidean distance on a variety of experimental settings, to the best of our knowledge, for the first time. Moreover, our study of "false hits" (i.e., candidates that are not real kNNs) in addition to running time verifies the improvements made by our techniques in a machine-independent manner. Conclusions based on this metric are applicable irrespective of the experimental environment or network distance technique used.

4.2 Preliminaries

We utilize the common definition of the road network graph $G = (V, E)$, shortest path and network distance as described in Section 1.1. As in Chapter 3 and other studies we consider POIs (objects) and query points located on vertices in V. Similarly, given a query vertex q and a set of object vertices O, a kNN query retrieves the k closest objects in O based on their network distances from q in G.

4.2.1 Landmark Lower Bounds

To compute a Landmark Lower Bound (LLB) between two vertices, firstly a pre-processing phase is conducted offline where a set of m *landmark* vertices $L = \{l_1, \ldots, l_m\} \subseteq V$ is selected. Then from each landmark $l_i \in L$ the distances to all vertices in V are computed. Now given any source vertex q and destination vertex o, we can compute a lower bound $LB_{l_i}(q, o)$ on the network distance $d(q, o)$ using the distances to landmark l_i and the triangle inequality as defined in (4.1). We obtain the "tightest" lower bound (i.e., closest to $d(q, o)$) by choosing the maximum lower bound $LB_{max}(q, o)$ over all m landmarks as defined in (4.2).

$$LB_{l_i}(q, o) = |d(l_i, q) - d(l_i, o)| \leq d(q, o) \tag{4.1}$$

$$LB_{max}(q, o) = \max_{l_i \in L}(|d(l_i, q) - d(l_i, o)|) \tag{4.2}$$

4.2.2 ALT Index

LLBs were first applied to road networks by Goldberg and Harrelson [GH05] in their *ALT* technique. The *ALT* index simply stores, for each l_i in the set of m landmarks, the distance $d(l_i, v_j)$ from l_i to every vertex $v_j \in V$. This can be visualized as an array of distances for each vertex as shown in Figure 4.2. ALT takes $O(m|V|)$ space and given the set of m landmarks vertices can be constructed in $O(m|V| \log |V|)$ time using Dijkstra's algorithm.

Figure 4.2: Vertex-Landmark Distances in ALT Index

Two considerations arising from LLBs are (a) the vertices to select as landmarks and (b) the number of landmarks. For example, increasing the number of landmarks tends to increase LB_{max}, as the probability of finding a "good" landmark increases. But this entails higher space cost (as there will be additional columns in Figure 4.2) and finding the tightest lower-bound over all landmarks also becomes more expensive. Choosing landmarks suitable for kNN querying is a challenging problem, as we elaborate in Section 4.3.1.

In practice, it is more beneficial to store the distances in arrays as shown in Figure 4.2 rather than the transpose. That is, have an array for each *vertex* with distances to landmarks rather than the other way around. Then $LB_{max}(q, o)$ can be efficiently computed by iterating over the lists for q and o. Since accessed values will be sequential memory, this leads to better CPU cache utilization and faster computation. Also note that (4.1) and (4.2) apply to undirected graphs, but the idea can easily be extended to directed graphs by computing and storing distances to and from landmarks.

4.2.3 Multi-Step *k*NN Algorithm

Incremental Euclidean Restriction (IER) [Pap+03] is an instance of the multi-step kNN algorithm first described by Seidl and Kriegel [SK98] as in Algorithm 2. This decoupled heuristic algorithm generalizes IER to consider *any* lower-bounding heuristic, such as LLBs. We refer to kNN algorithms using this paradigm with other heuristics as Incremental Lower Bound Restriction (ILBR).

Algorithm 2 Multi-Step kNN algorithm by Seidl and Kriegel [SK98]

```
1: function GetKNNsByILBR(k, q, V)
2:   PQ ← φ                          ▷ Queue of objects in ascending lower-bound distance from q
3:   R ← φ                           ▷ Max priority queue containing k best candidates
4:   D_k ← ∞                         ▷ Network distance to kth candidate in R
5:   while MinKey(PQ) ¡ D_k do
6:     c ← Extract-Min(PQ)
7:     Compute network distance d(q, c)
8:     if d(q, c) < D_k then
9:       Insert(R, c, d(q, c))      ▷ Update R and D_k if needed
10:  return R
```

Algorithm 2 iteratively retrieves the candidate object with the smallest lower-bound distance from minimum priority queue $\mathcal{PQ}$. The true network distance from query vertex q to the extracted candidate c is computed using another technique. Then c is inserted into the result set R if it improves it. D_k records the network distance to the current k-th

furthest candidate in R (initially infinity). When the next smallest lower-bound (e.g., at the front of $\mathcal{PQ}$) is larger than D_k, R can no longer be improved, and the algorithm terminates.

For IER, $\mathcal{PQ}$ is the priority queue maintained by the Euclidean NN search on an R-tree indexing all objects in O. This algorithm avoids computing Euclidean distances to all objects by using minimum bounding rectangles to avoid R-tree nodes. However, there is no efficient analogous method for LLBs. In past studies [Kri+07; Kri+08], LB_{max} in (4.2) is computed for *all* objects. Each object is inserted into $\mathcal{PQ}$ by its LB_{max} value in order to find the object with minimum LB_{max}. This is neither efficient nor practicable, as it is required every time a query is issued and there may also be many objects.

4.3 Techniques

In this section we propose two approaches to solve the problem of incrementally retrieving candidate objects for use by Algorithm 2 and avoid the previously mentioned problems of using LLBs. We first attempt a "lower-bound first" by introducing the Object Lists index based on ALT that searches for the next best lower-bound (and associated object) in Section 4.3.1. Then in Section 4.3.2, we propose an "object first" approach which searches for the best object to which we compute a lower-bound.

4.3.1 Object Lists

For an object set O, we create an index consisting of a set of *Object Lists* (OL), one for each landmark $l_i \in L$. The list OL_i contains an element for every object $o \in O$, with each element containing o and its distance from the landmark $d(l_i, o)$. The list is sorted on distances $d(l_i, o)$. Figure 4.3 illustrates a set of *unsorted* Object Lists. An OL index is pre-computed offline since object sets are known before query time (e.g., the set of all restaurants). It can be efficiently constructed using an ALT index shared between all object sets, in $O(m|O|)$ time and space, which is linear to the input.

OL-Based Candidate Generation for kNN Queries

To generate candidates for kNN querying using OL, we must find the object $p \in O$ with the smallest $LB_{max}(q, o)$ as defined by (4.3). Then, p can be returned as a candidate to

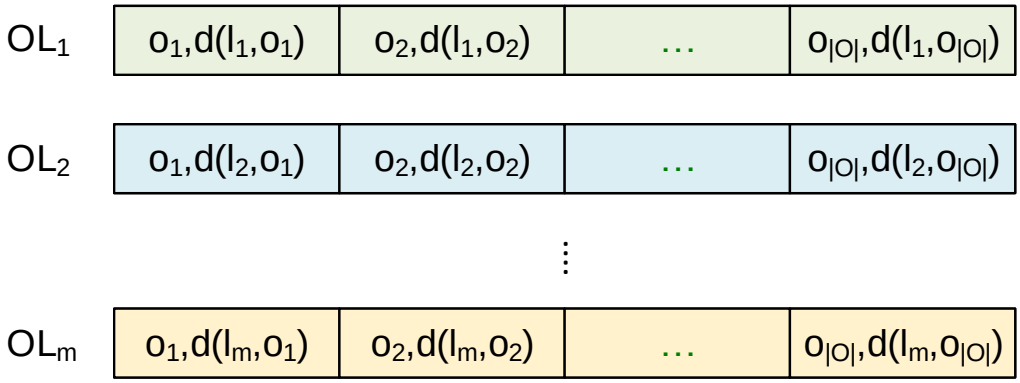


Figure 4.3: Unsorted Object Lists for m Landmarks

ILBR (Algorithm 2). Rather than computing LLBs for *all* objects to find p, we attempt to do this more optimistically and compute fewer LLBs in the process.

$$p = \min_{p \in O}(\max_{l_i \in L}(|d(l_i, q) - d(l_i, p)|)) \tag{4.3}$$

In Algorithm 3, given a query vertex q and a landmark vertex l_q, we use the object list OL_q of l_q to populate and refine a set of *potential candidates* until we find object p that minimizes (4.3). We will elaborate on choosing l_q shortly but consider l_q to be the nearest landmark to q for now.

Algorithm 3 Retrieve object with minimum lower-bound using Object Lists

1: **function** EXTRACT-MIN-OL($q, l_q, d(l_q, q), \mathcal{PQ}, RP_q, LP_q$)
2: **if** OL_q positions RP_q and LP_q are not set **then**
3: $pos \leftarrow OL_q.FindClosest(d(l_q, q))$
4: $p \leftarrow OL_q[pos].obj, RP_q \leftarrow pos + 1, LP_q \leftarrow pos - 1$
5: $\mathcal{PQ}.Enqueue(p, ALT.ComputeLBMax(q, p))$
6: **while** $LB_{l_q}(q, p) < \mathcal{PQ}.Top()$ for p at RP_q or LP_q **do**
7: $p \leftarrow$ object at RP_q or LP_q with smaller LB_{l_q}
8: $\mathcal{PQ}.Enqueue(p, ALT.ComputeLBMax(q, p))$
9: Increment RP_q or decrement LP_q based on choice at Line 7
10: **return** $o_c \leftarrow \mathcal{PQ}.Dequeue()$

The first potential candidate is the object c with the minimum lower-bound for the given landmark l_q. The lower-bound $LB_{l_q}(q, o)$ for every object $o \in O$ is defined by (4.1), i.e., $|d(l_q, q) - d(l_q, o)|$. But for each query, $d(l_q, q)$ is constant as the query vertex stays the same. Thus, this is a single-variable absolute value function of the form $f(x) = |C - x|$

where $x \in \{d(l_q, o) | o \in O\}$. This means the domain of x is stored in the object list OL_q of landmark l_q in increasing order.

Single-variable absolute value functions are convex and minimized at the vertex at $x = C$. Thus, the minimum value over the domain of x is given by x closest to constant C. This is also the minimum lower-bound for landmark l_q and can be found by searching OL_q for $d(l_q, c)$ closest to $d(l_q, q)$ for some object $c \in O$. Since OL_q is sorted this can be achieved efficiently using a modified binary search in $\log |O|$ time (line 3 in Algorithm 3). The object c corresponding to the distance $d(l_q, c)$ closest to $d(l_q, q)$ is the object which minimizes $LB_{lq}(q, o)$. While c may not be unique, any object with the minimum lower-bound will suffice. However, c may not minimize (4.3) and we must search further to ensure p is found, as we show next.

Example 1. *Figure 4.4 depicts OL_q for a set of 7 objects $o_1, ..., o_7$ with distances from a landmark l_q. Let us say $d(l_q, q) = 4$, then the binary search will find the element at index 3 (shaded) as closest to 4. Therefore $c = o_5$ minimizes (4.1) with $LB_{l_q}(q, o_5) = |4 - 4| = 0$.*

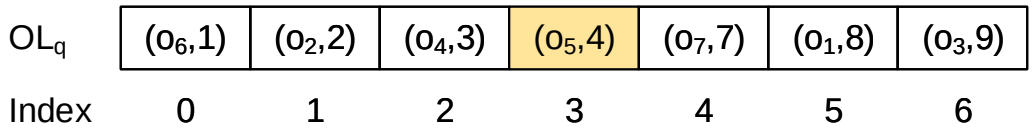


Figure 4.4: Example Object List OL_q for Landmark l_q

Now object c is inserted into a minimum priority queue $\mathcal{PQ}$ keyed by $LB_{max}(q, c)$ computed using the ALT index [GH05] as described in Section 4.2.2. We use the ALT-based lower-bound for c as it may provide a tighter lower-bound than $LB_{l_q}(q, c)$. More importantly, it helps us determine how far we must search the object list OL_q to find the object p which minimizes (4.3). To formalize this search, we propose Lemma 1:

Lemma 1. *Given an object c and a landmark l_q, any other object $p \in O$ with $LB_{l_q}(q, p) < LB_{max}(q, c)$ may also have $LB_{max}(q, p) < LB_{max}(q, c)$.*

Proof. Lemma 1 is trivially true when $LB_{max}(q, p) = LB_{l_q}(q, p)$, i.e., when l_q gives the tightest lower-bound for p. □

Now object c_{next} with the next smallest lower bound by l_q is immediately to the left or right of object c found earlier by binary search on OL_q (line 4 in Algorithm 3). If

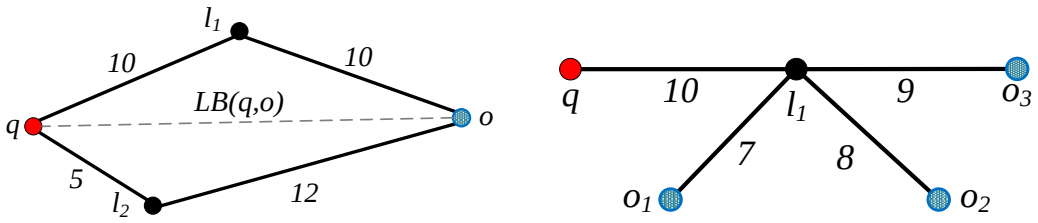
$LB_{l_q}(q, c_{next}) < Top(\mathcal{PQ})$ then by Lemma 1, c_{next} may have smaller LB_{max} than any object in $\mathcal{PQ}$. While $LB_{l_q}(q, c_{next}) < Top(\mathcal{PQ})$, we search left or right from p. When an object satisfies the condition, we compute LB_{max} and insert it into $\mathcal{PQ}$. When neither the next left nor right object satisfies the condition, the algorithm terminates, and the top element in $\mathcal{PQ}$ is returned as the object that minimizes (4.3). This is correct as any object further left or right must have $LB_{l_q}(q, c_{next}) \geq Top(\mathcal{PQ})$ and cannot satisfy Lemma 1. Furthermore, by saving $\mathcal{PQ}$ and the indices in OL_q of the last left and right elements evaluated so far, we can incrementally retrieve the object with the next smallest LB_{max}.

Example 2. *Continuing Example 1, let us say* $LB_{max}(q, o_5) = 2$ *and hence* $Top(\mathcal{PQ}) = 2$ *after inserting* o_5*. In Figure 4.4, the objects to the left and right of* o_5 *are* o_4 *and* o_7*, with lower bounds* $LB_{l_q}(q, o_4) = |4 - 3| = 1$ *and* $LB_{l_q}(q, o_7) = |4 - 7| = 3$ *respectively. By Lemma 1,* o_4 *may have a lower* LB_{max} *so we compute* $LB_{max}(q, o_4)$ *and insert it into* $\mathcal{PQ}$*. Let us say we still have* $Top(\mathcal{PQ}) = 2$ *after inserting* o_4*. For the next element to the left,* o_2*, we have* $LB_{l_q}(q, o_2) = |4 - 2| = 2$*. In that case, neither lower bounds for the object to the left or right is less than* $Top(\mathcal{PQ})$ *and therefore cannot improve on the objects in* $\mathcal{PQ}$ *and the search terminates.*

Difficulty in Choosing an Object List

In the above, we used the object list of the landmark l_q closest to the query vertex q to increase the probability of objects being further from l_q than q. Then we have $d(l_q, q) < d(l_q, o)$ for a greater number of objects and hence produce higher lower-bounds by (4.1). This scenario is illustrated in Figure 4.5(a), which shows the landmark distances for query vertex q, the 1NN object o and two landmarks l_1 and l_2. The lower-bound distances using each landmark are $LB_{l_1}(q, o) = |10 - 10| = 0$ and $LB_{l_2}(q, o) = |12 - 5| = 7$. Landmark l_2 gives a better lower-bound for o than l_1 as it is closer to q. In this way, we expect to benefit from having to only search a single object list OL_q with lower-bounds closer to their LB_{max} on average and hence find p in (4.3) sooner.

This principle holds for smaller datasets where the ratio of landmarks to objects is lower, as confirmed in practice by our experimental investigation (Section 4.4). However, when dataset size increases in either (a) the number of road network vertices $|V|$ or (b) the number of objects $|O|$, it becomes less likely to be true. Recall that the number of landmarks m is fixed so in both cases the likelihood of finding a landmark that is closer



(a) Close Landmark (b) Far Landmark

Figure 4.5: Effect of Landmark Location on Lower-Bounds

to the query location than to other objects decreases. This disadvantageous scenario is illustrated in Figure 4.5(b). All three objects are closer to l_1 than q and o_3 gives the smallest lower-bound $LB_{l_1}(q, o_3) = |10 - 10| = 0$ even though it is the furthest object from q (assume objects are embedded in the Euclidean plane relative to their network distance from q). Next, we propose a technique that is far less beholden to the density of landmarks.

Extending to Moving Objects

Thus far, we have considered static objects, but some object sets may be considered to be moving, e.g., taxis. Since we have assumed that objects occur on road network vertices, we shall consider objects moving from one vertex to another. Such vertices are often not very far apart, especially in real-world maps like OpenStreetMap [Ope] which use additional vertices between intersections to capture properties such as road curvature. When an object p moves, the landmark distances for p will be invalidated in each OL. Handling this scenario is a simple matter of looking up the landmark distance in ALT for the new vertex associated with p, achieved in $O(1)$ per landmark. This update may affect the ordering of each landmark's OL, but since at most only one list element is out of order, each OL can be re-ordered in at most $O(|O|)$ time.

4.3.2 Network Voronoi Diagrams and Landmarks

We propose another approach to efficiently generate candidates through the novel combination of LLBs and a Network Voronoi Diagram (NVD) [OBS00]. As we describe next, candidates are generated in an "object-first" manner using NVDs, which is more suitable

for kNN queries, and are even capable of tightening lower-bounds above and beyond those provided by ALT.

We first define the *Voronoi node set* $V_{ns}(o_i)$ of o_i by (4.4), which identifies the vertices in V for which o_i is the first nearest neighbor by their network distances to o_i. This is the key difference to the Euclidean Voronoi diagram, in that generator o_i occurs on the road network and distances are computed by shortest paths.

$$V_{ns}(o_i) = \{v | v \in V, d(v, o_i) \leq d(v, o_j) \forall o_j \in O \setminus o_i\} \tag{4.4}$$

For any edge (u, v) where $u \in V_{ns}(o_i)$ and $v \in V_{ns}(o_j)$, then $V_{ns}(o_i)$ and $V_{ns}(o_j)$ are *adjacent*. The Network Voronoi Diagram for object set O is the collection of Voronoi node sets for all objects in O. Figure 4.6 shows an example NVD for a graph with unit edge weights and four objects. Each Voronoi node set is surrounded by a dotted container and arrows indicate adjacency.

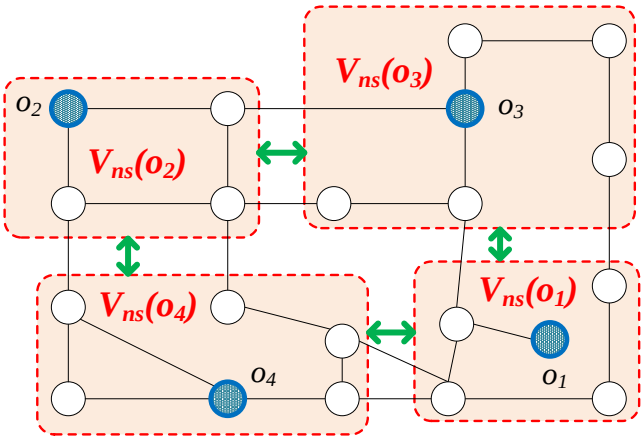


Figure 4.6: Network Voronoi Diagram

Pre-Processing Algorithm and Time Complexity: An NVD can be computed optimally in $O(|V| \log |V|)$ time [EH00] using simultaneous Dijkstra's searches from all objects using a single priority queue. When a vertex v_d inserted by the search from o_i is dequeued and v_d is not assigned to a Voronoi node set, it is assigned to $V_{ns}(o_i)$. This is correct as v_d is the minimum element in the queue and so cannot be closer to another object. However, if v_d *is* assigned to another Voronoi node set $V_{ns}(o_j)$, then $V_{ns}(o_j)$ is added to the list of adjacent sets for $V_{ns}(o_i)$ (the search from o_j creates the reciprocal

entry). The search from o_i is pruned at v_d, i.e., neighbor vertices are not inserted into the queue as they cannot belong to $V_{ns}(o_i)$.

Index Size: A simple method to store NVDs saves the nearest object for every vertex in V, taking $O(|V|)$ space. However, NVDs can be compressed by storing the geometric area of the Voronoi node sets in a quadtree [DS12; SSA08]. As road networks are generally near-planar each set is likely to be contiguous, and quadtree cells with objects from the same set can be merged, reducing the space cost. Given a query vertex q, its Voronoi node set can be determined by a point location query on the quadtree using its Euclidean coordinates.

Revisiting NVD Adjacency: NVD were previously used to answer kNN queries by VN^3 [KS04]. VN^3 uses an NVD to partition the road network. For each partition (i.e., Voronoi node set), the distances from its border vertices to each of its contained vertices is pre-computed and stored. Using the observation that the next NN is contained in a Voronoi node set adjacent to the sets of NNs found so far, VN^3 uses border-to-border and border-to-vertex distances to compute network distances to all adjacent objects and determine the next NN. VN^3 has long since been replaced by far more flexible and efficient techniques, some of which we described in Chapter 3.

However, the observation about adjacent Voronoi node sets containing the next NN was discarded along with VN^3 due to the cumbersome pre-computation of distances which ultimately did not achieve efficient querying. As we detail next, this observation is still quite useful and can be relaxed to consider the next *candidate* NN rather than the next NN. In turn, this allows us to incrementally retrieve candidates by landmark lower-bounds (LLBs) and terminate without retrieving all objects. By combining NVDs with landmarks, we can significantly improve candidate generation for kNN querying, breathing new life into an old idea.

NVD-Based Candidate Generation for kNN Queries

We describe how to retrieve the best candidate using NVDs in Algorithm 4. By their definition, an NVD can quickly return the 1NN by looking up the Voronoi node set (and hence the associated object) containing the query vertex. If $k > 1$, Algorithm 4 returns the next candidate by first retrieving the adjacent Voronoi node sets of the last candidate object. Note that in the first call to Algorithm 4 the last candidate is the 1NN. Each

adjacent set generates a new potential candidate, to which we compute its LB_{max} by (4.2) using the ALT index and insert it into priority queue $\mathcal{PQ}$. We use hash-table or bit-array H to avoid repeated computations for previously evaluated adjacent Voronoi node sets. Once all adjacent sets are processed in this way, we return the element in $\mathcal{PQ}$ with the minimum LB_{max} as the next candidate.

Algorithm 4 Retrieve object with minimum lower-bound using an NVD

```
1: function EXTRACT-MIN-NVD(q, o_last, PQ, H)
2:   for each V_ns(p) adjacent to V_ns(o_last) do
3:     if !H.contains(V_ns(p)) then
4:       PQ.Enqueue(p, ALT.ComputeLBMax(q, p))
5:       H.insert(V_ns(p))
6:   return o_c ← PQ.Dequeue()
```

Figure 4.7 shows a simplified NVD, assume the dotted containers capture the Voronoi node sets of each object and when containers share an edge, they are adjacent. So, for query vertex q in the figure, we can retrieve the 1NN o_1 as q is contained in $V_{ns}(o_1)$. Then the adjacent Voronoi node sets of $V_{ns}(o_1)$ (lightly shaded) are used to retrieve potential candidates, which are inserted into $\mathcal{PQ}$ by their LB_{max} values. Let us say the candidate with the minimum key is now o_7, then o_7 would be returned by the algorithm. In the next call to the algorithm, the Voronoi node sets adjacent to $V_{ns}(o_7)$ would be retrieved and, for sets not already evaluated, new potential candidates inserted into $\mathcal{PQ}$.

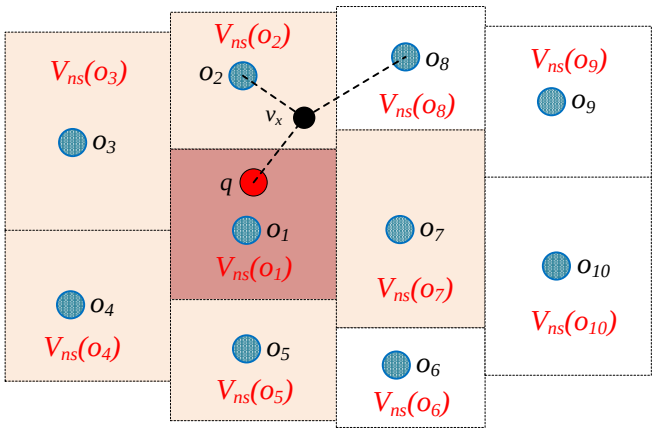


Figure 4.7: Network Voronoi Diagram Query

Recall that ILBR (Section 4.2.3) terminates when the network distance to the kth candidate is less than the lower bound distance to the next candidate. We propose Theorem 1 to show that Algorithm 4 is correct when this occurs.

Theorem 1. *When ILBR terminates the following are true (1) priority queue $\mathcal{PQ}$ does not contain any objects with a distance smaller than the kth candidate (2) no object outside of the Voronoi node sets visited so far (i.e., objects in $\mathcal{PQ}$ or returned as candidates) have distance smaller than the kth candidate.*

Proof. Let D_k be the network distance to the kth candidate. We prove each case of Theorem 1 individually as follows:

Case 1: When ILBR terminates we have $D_k \leq Top(\mathcal{PQ})$. We also have $Top(\mathcal{PQ}) \leq d(q,c)$ for any object c in $\mathcal{PQ}$ as they are inserted using a lower bound distance and $\mathcal{PQ}$ is a minimum priority queue. Thus, we also have $D_k \leq d(q,c)$ and no object c in $\mathcal{PQ}$ has a network distance smaller than the kth candidate.

Case 2: Let $C \subseteq O$ be the set of objects inserted into $\mathcal{PQ}$ and let $S = \{V_{ns}(o) | o \in C\}$ be the set of associated Voronoi node sets. We prove Case 2 by contradiction in a similar but simpler manner to [KS04]. Let us assume there exists an object $p_k \notin C$ such that $d(q,p_k) < D_k$. Algorithm 4 inserts objects into $\mathcal{PQ}$ from adjacent Voronoi node sets beginning with the set containing q, thus all Voronoi node sets in S are adjacent to at least one other set in S. So, the shortest path $P(q,p_k)$ must pass through some Voronoi node set $V_{ns}(x) \in S$ since $p_k \notin C$. Thus $P(q,p_k)$ must contain at least one vertex $v_x \in V_{ns}(x)$, as illustrated in Figure 4.7 with $x = o_2$ and $p_k = o_8$. By the definition of an NVD we have $d(v_x,x) \leq d(v_x,p_k)$ as all vertices in $V_{ns}(x)$ are closer to x than any other object. Adding $d(q,v_x)$ to both sides results in $d(q,v_x) + d(v_x,x) \leq d(q,v_x) + d(v_x,p_k)$. This simplifies to $d(q,x) \leq d(q,p_k)$ as v_x is on the shortest path $P(q,p_k)$ and $d(q,x) \leq d(q,v_x) + d(v_x,x)$. Since x is in $\mathcal{PQ}$, we have $Top(\mathcal{PQ}) \leq d(q,x)$, so we must have $D_k \leq d(q,x)$. This implies $D_k \leq d(q,p_k)$, contradicting our assumption. □

NVD-Based Landmark Lower-Bounds

An added advantage of using NVDs is that we can compute a landmark lower-bound (LLB) using only the NVD. This allows us to either (a) avoid using the ALT index altogether or better yet (b) use both to obtain a tighter lower-bound by choosing the maximum of the NVD-based lower-bound (NVD-LLB) and ALT-based lower-bound (ALT-LLB). First, to enable NVD-LLBs, during NVD pre-processing we also compute the network distances between adjacent objects. Alternatively, an upper-bound between adjacent objects will also suffice. In fact, during NVD construction, we naturally compute an upper bound

distance between objects of adjacent Voronoi node sets when parallel Dijkstra's searches meet. Simply saving the smallest such upper-bound as searches meet comes at no additional pre-processing overhead. This is an upper bound and not an exact distance because searches are pruned (e.g., shorter paths may exist through other adjacent Voronoi node sets).

Now to compute the NVD-LLB we can use the objects themselves as landmarks and then apply the triangle inequality. During querying, we naturally compute the network distance from q to the last candidate object, which is an input to Algorithm 4. For example, in Figure 4.8, let o_2 be the last candidate returned with network distance $d(q, o_2)$. While evaluating the adjacent set $V_{ns}(o_8)$, we use the pre-computed upper bound distance $UB(o_2, o_8)$ between o_2 and o_8 to compute a lower bound $LB_{nvd}(q, o_8) = d(q, o_2) - UB(o_2, o_8)$. Note that we do not use the absolute value due to the upper bound. In Algorithm 4, we insert o_8 into Q keyed by $LB_{nvd}(q, o_8)$ if $LB_{nvd}(q, o_8) > LB_{max}(q, o_8)$. The new lower bound may be tighter than the one computed using ALT, especially when objects are further away from q and comes at a cheap pre-processing and query time cost.

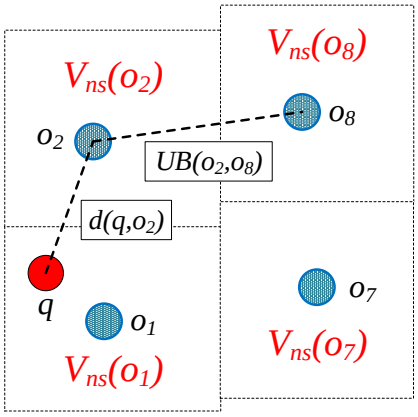


Figure 4.8: NVD-Based LLBs

Extending to Moving Objects

Extending NVDs for moving objects is more complicated than Object Lists. Any change to a single object may potentially invalidate the entire NVD, so updating the NVD may require it to be reconstructed from scratch in $O(|V| \log |V|)$ time. Unlike OLs, this is dependent on the size of the road network (in vertices $|V|$), making it significantly more costly. This suggests that OLs are more suitable for object sets involving moving objects.

We verify this experimentally in Section 4.4.2 in terms of indexing construction time. Note that a more optimistic approach than complete re-computation to update NVDs is described in Section 5.6.2 and is potentially applicable here.

4.4 Experiments

4.4.1 Experimental Settings

Environment: All experiments were run on a 3.2GHz Intel Core i5-4570 CPU with 32GB RAM running 64-bit Linux (kernel 4.2). All code was written in single-threaded C++ and compiled using g++ 5.2 with the O3 flag. We implemented ILBR, ALT and the candidate generation techniques ourselves. We re-used the implementations of existing techniques, experimental scripts, and datasets from our work in Chapter 3. All queries were executed in-memory for fast performance.

Datasets: We use 10 travel time road networks as listed in Table 3.1 with the largest dataset, for the continental US, the default. We use a combination of synthetic and real object sets as described in Section 3.4.2. We choose synthetic objects uniformly at random based on density d where d=$|O|/|V|$ and the 8 real POI sets listed in Table 3.2 used in the experiments of Chapter 3.

Parameters: We vary object set density from 0.0001 to 1 and k from 1 to 50. We use the same default parameters as in Table 3.6.1 with default density $d = 0.001$ and $k = 10$. We generate 25 uniform object sets and execute methods for 200 randomly selected query vertices, averaging running time over 5000 queries.

Techniques: Like IER, ILBR uses a different road network index to compute network distances. We combine ILBR with Pruned Highway Labeling (PHL) [Aki+14] as it is one of the fastest techniques. We use an ALT [GH05] index with 16 random landmarks to compute lower bounds and construct Object Lists. Finally, we compare our techniques against the current fastest state-of-the-art technique, IER (similarly using PHL) as per our insights from Chapter 3. For real-world object sets we also compare against two other prominent techniques G-tree [Zho+15] and INE [Pap+03] for comparison with Chapter 3.

4.4.2 Index Performance

Table 4.1 details the index pre-processing time and space costs. PHL is the road network index employed by ILBR and IER. While PHL is faster to construct for travel time road networks, G-tree consumes less space making it more suitable when there is limited memory. The index size of ALT is small, but this is dependent on m, the number of landmarks used (16 in our case). It can be reduced by using fewer landmarks at the expense of looser lower bounds. We also observe the performance of object indexes used by ILBR (Object Lists and Network Voronoi Diagrams) and IER (R-trees) for the default density $d = 0.001$. Object Lists and R-trees are fast to construct, and their index sizes are small. However, since the space cost is a function of object set size, we expect it to increase with density. NVDs take longer and occupy more space as the time and space complexity are functions of $|V|$. But both costs are still significantly smaller than road network indexes making it feasible to compute an NVD for each object set. Although not shown in the results in Table 4.1, using the compression scheme described in Section 4.3.2 to store NVDs in a quadtree significantly reduces the space cost, for example, by 70% from 92MB to 28MB for the US object set with negligible impact on query performance.

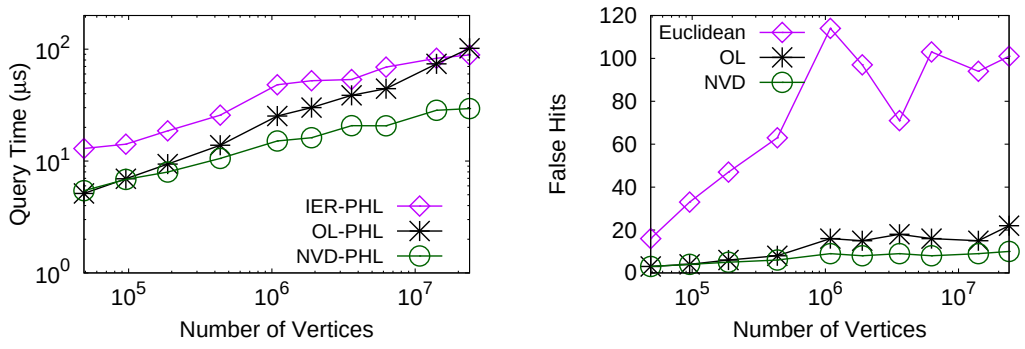
Road Network		**PHL**	**G-tree**	**ALT** (m=16)	**OL** (d=0.1%)	**NVD** (d=0.1%)	**R-tree** (d=0.1%)
NW	Time	16s	47s	2s	0.8ms	264ms	0.2MS
	Space	325MB	104MB	67MB	136KB	4.2MB	44KB
US	Time	30m	71m	60s	15ms	12s	4ms
	Space	15.8GB	2.9GB	1.43GB	1.8MB	92MB	0.9MB

Table 4.1: Road Network and Object Index Statistics

4.4.3 Query Performance

We evaluate the query performance of each technique on two metrics, namely running time and false hits per query. A *false hit* occurs when a candidate NN is not a real kNN. The greater the number of false hits, the more unnecessary network distance computations ILBR must perform. Thus, false hits are an indication of a heuristic's performance irrespective of the experimental setting (disk-based or main memory) or the network distance technique used. We refer to the two ILBR methods as NVD-X and OL-X as variants employing Network Voronoi Diagrams and Object Lists respectively (and X is the road network index used).

Effect of Network Size: Figure 4.9 shows query performance as the number of road network vertices $|V|$ increases. In Figure 4.9(a), NVD-PHL is consistently the best performing method and is 2-3× faster than IER-PHL. OL-PHL is comparable to NVD-PHL for the first few datasets after which its advantage over IER-PHL narrows until being on par with it for the largest dataset. With increasing $|V|$, the total number of objects increases for the same density causing Object Lists to become larger. For example, we might expect there to be more fast food outlets in larger regions. OL is susceptible to objects that appear close when they are similar distances from the landmark as the query vertex. When $|V|$ increases, landmarks become more distant from query vertices on average (as the number of landmarks m is constant), so the probability of such objects appearing increases. While these are only *potential candidates* and are not reflected in false hits, OL must still compute their LB_{max} values. This is evident in Figure 4.9(b) as the number of false hits for OL is still lower than Euclidean distance.

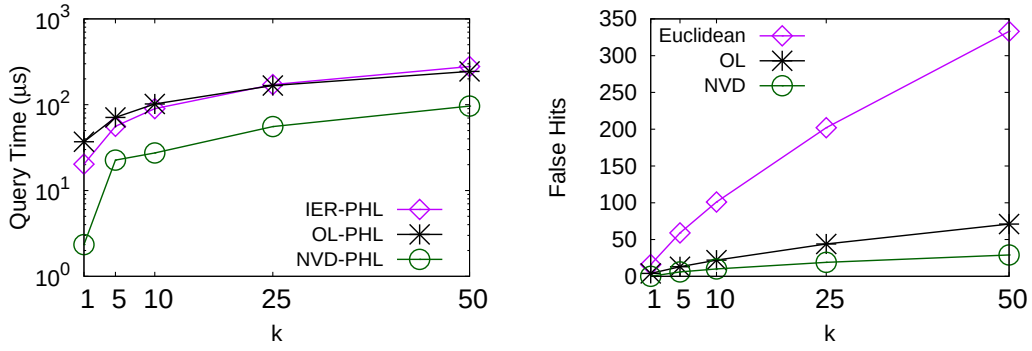


(a) Query Time

(b) False Hits

Figure 4.9: Effect of Road Network Size $|V|$ (d=0.001,k=10,uniform objects)

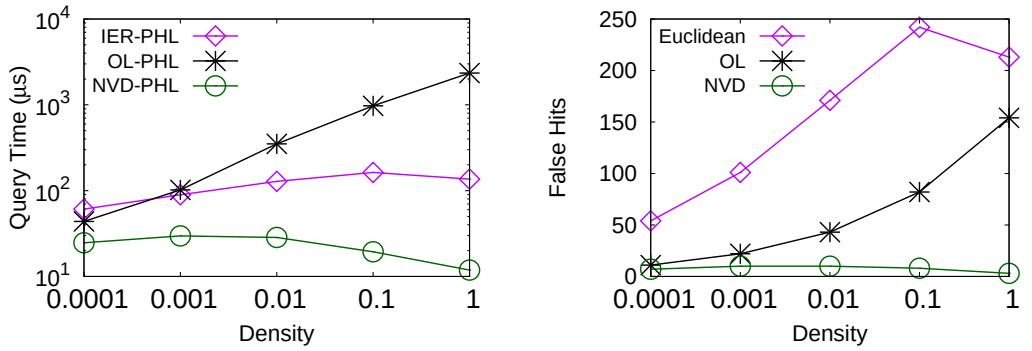
Effect of k: Figure 4.10 shows the query performance with increasing k. For $k = 1$, NVD-based methods are essentially optimal as only a single look-up operation is needed. NVD-PHL once again outperforms all other methods, being at least 2-3× faster than IER-PHL over all k, again showing that it *is* possible to efficiently use landmarks for kNNs. Landmarks display significant improvement on false hits over Euclidean distance in Figure 4.10(b). But earlier trends are also seen here and OL-PHL's query time does not improve on IER-PHL. NVDs incur noticeably fewer false hits than OL-PHL despite both using LLBs, which indicates that the improvement of lower bounds using NVD-based LLBs is working.



(a) Query Time (b) False Hits

Figure 4.10: Effect of k (US,d=0.001,uniform objects)

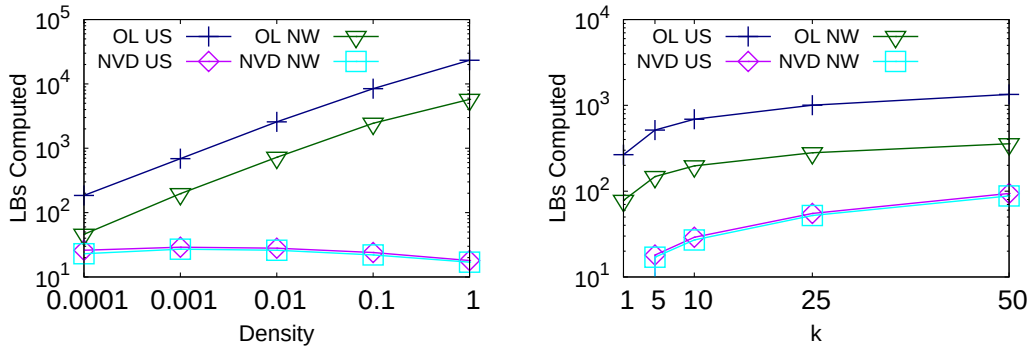
Effect of Density: We observe query performance with increasing object set density in Figure 4.11. As density increases, the average distance between objects decreases. This makes kNNs appear closer to the query vertex and they should be easier to find. IER-PHL is an exception, as objects become closer and more numerous, they become more difficult to differentiate using Euclidean distance. NVD-PHL shows this problem can be remedied using landmarks as it is an order of magnitude better than IER-PHL in Figure 4.11(a). OL-PHL, on the other hand, degrades with increasing density to the point that its running time is an order of magnitude worse than IER-PHL. With more objects, more of them will produce inaccurate lower bounds similar to the scenario depicted in Figure 4.5(b), making distant objects appear close to the query vertex. NVD-PHL does not suffer from this as using adjacent Voronoi node sets acts as a filter avoiding objects that "seem" close by inaccurate lower bounds. As a result, NVD-PHL experiences far fewer false hits in Figure 4.11(b).



(a) Query Time (b) False Hits

Figure 4.11: Effect of Density (US,k=10,uniform objects)

Lower Bounds Computed: Figure 4.11(b) showed that with increasing density, OL experiences fewer false hits than Euclidean distance even at its worst. This suggests that the poor running time of OL for high densities in Figure 4.11(a) is not caused by ILBR making additional network distance computations due to false hits. In fact, it is due to the number of lower bounds computed by OL, which increases with density, as illustrated in Figure 4.12(a). NVD computes very few lower bounds thanks to its filtering property. The final evidence of this is the behavior of OL on the two datasets in Figure 4.12. The US road network with 24 million vertices requires more lower-bounds to be computed than the smaller NW dataset with 1 million vertices. The US has more objects for the same density, resulting in a larger Object List and hence a larger search space to find the best object. We note, however, that computing all lower bounds would require significantly more computations than OL. While OL is a substantial improvement, its utility is still dependent on the number of objects.



(a) Varying Density

(b) Varying k

Figure 4.12: Number of Lower Bounds Computed (US,d=0.001,k=10,uniform objects)

Real-World Object Sets: We verify our observations on real-world POIs in Figure 4.13 with increasing object set sizes from left to right. Trends seen in previous figures are also observed for real-world POIs. NVD-PHL is the overall best performing method, while OL-PHL is competitive except on larger object sets like parks and schools. For the smallest object sets like courts, IER-PHL remains competitive as there are so few objects that Euclidean distance has a smaller probability of making a false hit. A more typical object set such as fast food outlets demonstrates the significant superiority of NVD-PHL.

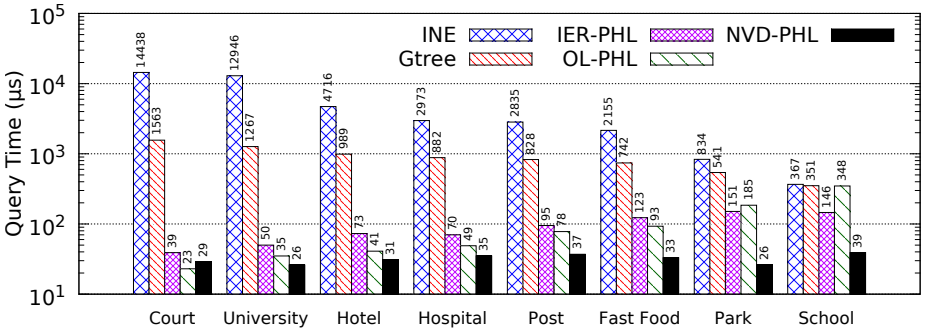


Figure 4.13: Varying Real-World Object Sets (US,k=10)

4.5 Summary

In this chapter, we present two techniques to efficiently use landmark-based lower bounds as the decoupled heuristic in the ILBR kNN algorithm. We empirically compare the heuristic performance of landmark lower bounds (LLBs) and Euclidean distance on kNN search for the first time. In doing so, we show that both of our LLB-based methods significantly improve on the number of false hits (by up to an order of magnitude) incurred in candidate generation than the Euclidean distance heuristic used by IER.

In our experimental investigation on travel time road networks, the Object List technique illustrates the difficulties in using landmarks but outperforms IER for smaller datasets. However, the technique employing a Network Voronoi Diagram outperforms IER by at least 2-3× on query time across all datasets and parameters. Thus, we show that it is indeed possible to use landmark-based lower bounds to improve kNN search. Moreover, we provide further evidence for the effectiveness of decoupled heuristics in POI search, which we first observed in Chapter 3. In addition, our novel combination of NVDs and LLBs proves extremely useful as a versatile and efficient heuristic for other POI search queries as we demonstrate next in Chapter 5.

Chapter 5

Spatial Keyword Querying by Keyword Separation

> Divide each difficulty into as many parts as is feasible and necessary to resolve it.
>
> Rene Descartes

In this chapter, we present our K-SPIN framework to answer spatial keyword queries on road networks. Building on the insights gained in Chapters 3 and 4, we find new insights specific to the spatial keyword problem and propose novel techniques to answer a range of spatio-textual queries efficiently. This chapter is based on work published in [ACK19].

5.1 Overview

Finding the nearest relevant points of interest (POIs) to a user's location is among the most popular queries in map-based services [CJ16]. These POIs are often associated with rich textual descriptions in addition to their spatial locations. Consider the example road network in Figure 5.1 with unit edge weights and eight objects (POIs) each associated with a set of keywords. A *spatial keyword* query retrieves objects that are close to the query location (e.g., in terms of travel time over the road network) and are textually relevant. The following two types of spatial keyword queries have been studied on road networks.

Boolean kNN Query [JFW15]. Given a set of query keywords, a Boolean kNN (BkNN) query returns the k objects closest to the query location among those that satisfy the keyword criteria. The criteria may be disjunctive (contain *any* query keyword) or conjunctive (contain *all* query keywords). For example, a user may want to find the closest object that contains either "restaurant" *or* "takeaway". In Figure 5.1, the answer is o_8 because no object closer to query location q contains either "restaurant" or "takeaway". Another user may wish to find the closest POI containing both "Thai" *and* "restaurant". In Figure 5.1, the result would then be o_6.

Top-k Spatial Keyword Query [RN12; Zho+15]. A top-k spatial keyword query returns k objects with the best scores. The score of an object is computed using a function combining the object's network distance from q and the relevance of the object's textual description with the query keywords. Section 5.2 provides a formal description.

20 billion Google searches with a location component are performed every quarter (including 13.9 billion from mobile devices) [Ste15]. This translates to ≈2500 spatial keyword queries per second on average. Using network distance affords greater accuracy and flexibility (e.g., using travel-time rather than the distance "as-the-crow-flies"). However, efficiently indexing road networks and keyword information to meet such high throughput demands is a challenging problem. Moreover, the indexing strategy used by current road network spatial keyword techniques, called keyword aggregation, leaves substantial room for improvement.

5.1.1 Motivation

Keyword aggregation is the idea of summarizing keyword occurrences over geographical regions. Spatial keyword queries are then answered by searching the most promising regions first while pruning regions that cannot contain results. This technique is used extensively by spatial keyword query techniques in Euclidean space [CJW09; WCJ12; Che+13; ZCT14]. Notably, *all* existing techniques for road networks also use the idea of keyword aggregation. The disadvantage of keyword aggregation is the generation of many false positives. Whenever a candidate is encountered, its distance from the query must be computed to confirm if it is relevant or not. Computing distance in Euclidean space is a quick arithmetic operation, but in road networks computing distance is a complex graph operation and far more expensive. Consequently, the penalty paid for incurring

false positives in road networks is significantly higher than in Euclidean space. So, while keyword aggregation is useful for Euclidean space, it is far less effective for road networks. We illustrate this problem in an example using a state-of-the-art spatial keyword technique for road networks [Zho+15].

Consider again the objects and road network with unit edge weights shown in Figure 5.1. The existing techniques first groups objects, e.g., G-tree [Zho+15] may form four groups G_1, G_2, G_3, and G_4 (shown by rectangles with broken lines). Keywords are then aggregated by creating a pseudo-document for each group that is the union of keyword occurrences over all the contained objects. For example, G_1's pseudo-document contains keywords "Italian", "restaurant", "takeaway", "Thai", and "grocer" each with one occurrence in the group. Note that the frequency of each keyword in the new pseudo-document is the sum of frequencies over all objects contained in the group. Consider a Boolean 1NN query to find the closest object with keywords "Thai" *and* "restaurant". The group G_4 can be pruned, as its pseudo-document does not contain "restaurant". The other three groups may contain the result. The algorithm computes minimum network distances to each group (e.g., the network distance from q to the closest border vertex in the group). These groups are inserted into a priority queue so that they can be accessed in ascending order of their minimum network distances from q (e.g., G_1, G_2, and then G_3). When G_1 is accessed, the algorithm prunes both o_1 and o_2 because neither object contains both query keywords. The algorithm then accesses G_2 and prunes the objects o_3 and o_4. But object o_5 contains both query keywords, so the algorithm computes its network distance from q. The algorithm can terminate if the minimum network distance of the next entry in the queue is greater than the network distance from q to o_5. However, since the minimum network distance of G_3 is smaller, the algorithm accesses G_3 and computes the network distances from q to o_6 and o_7 and determines o_6 to be the closest object satisfying the keyword criteria. Therefore, o_6 is returned as the result since the queue is empty.

In the above, costly minimum network distances needed to be computed to groups even when 1) a group does not contain any objects satisfying the keyword criteria (e.g., G_1) because the aggregated group appeared to contain such an object; or 2) the relevant object in the group is actually quite far from the query and is not a result (e.g., o_5 in G_2) because it appeared to be close as the query was close to the aggregated group. Furthermore, when a group is accessed, the algorithm needs to compute network distances from q to all objects

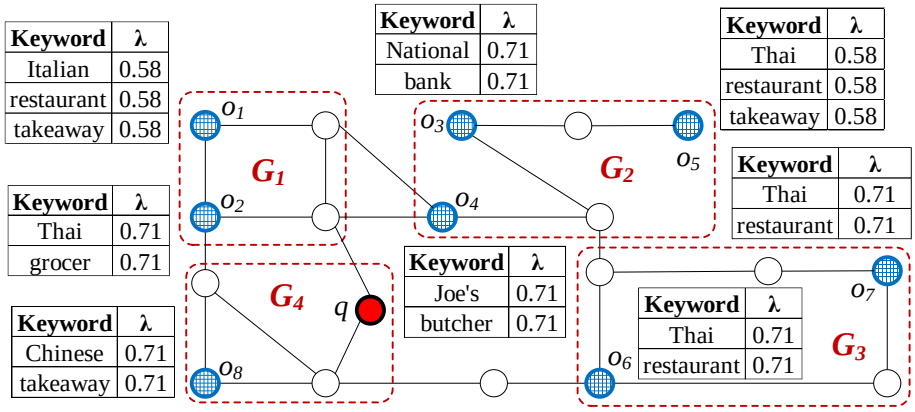


Figure 5.1: Example Road Network and Objects with Textual Information

satisfying the criteria in the group even if they are not results (e.g., o_7 in G_3). Similar issues are faced in top-k queries because network distances must be computed to groups (and objects within) that have low textual similarity (e.g., G_1 and its objects) or large network distance (e.g., o_5 and o_7). Moreover, while we use a simple example here for easier exposition, keyword aggregation is hierarchical and that exacerbates these problems. It is important to note that these problems arise from the hierarchical aggregation of objects and their constituent keywords. They cannot be solved in straight-forward ways due to the permanent loss of discriminating information that results from aggregation. We confirm this difficulty by attempting to improve G-tree in Section 5.7.4. The other road network techniques (described in Section 2.3) use similar ideas and face similar problems. Due to the similar partitioning scheme, ROAD [RN12] experiences the same exact problematic scenarios that G-tree does as presented above. Similarly, the bit-arrays used by FS-FBS [JFW15] aggregate keyword occurrences due to the use of hashing and the resulting collisions lead to false positives and wasted network distance computations.

Spatial Proximity: Utility of Network Distance

While we have identified the weaknesses of network distance-based spatial keyword query techniques, a vast majority of other techniques [CJW09; WCJ12; Che+13; ZCT14] use Euclidean distance as the measure for spatial proximity. In previous chapters, the utility of using network distance was obvious. For example, the application of kNN queries in time-sensitive tasks like finding the nearest drivers for a ride-hailing app requires the highest possible accuracy. However, in the context of spatial keyword queries, we are essentially trading spatial proximity for textual relevance (and vice versa). Naturally, this begs the

question of whether such a high level of accuracy for spatial proximity is even required in practice by most users for such spatial keyword queries. To the best of our knowledge, no user study has been conducted to determine what level of accuracy is preferred by users in the context of spatial keyword queries. The decreased accuracy of Euclidean distance comes with the benefit of cheaper computation than network distance computation. Our goal of closing this gap by improving the query performance of using network distance will make it a more palatable choice when high accuracy for spatial proximity is necessary. Moreover, once top-k results are retrieved, users are likely to compute the shortest path to it before commencing their travel. In most existing techniques, like G-tree [Zho+15], computing network distance allows inexpensive retrieval of the shortest path itself. These added benefits extend beyond the higher accuracy and versatility afforded by using network distance and makes improving query performance an especially worthwhile goal for spatial keyword queries in road networks.

5.1.2 Contributions

We present the **K**eyword **S**e**p**arated **In**dexing (K-SPIN) framework. K-SPIN employs the idea of creating a separate index for each keyword. However, as we detail next, doing this without incurring prohibitive pre-processing cost is challenging. Here we describe how K-SPIN overcomes the problems in our motivating example and our solutions for reducing pre-processing costs.

Efficient Querying: Separate keyword indexes allow us to obtain an *on-demand inverted heap* for each keyword filled with candidate objects specifically relevant to that keyword. Each candidate object in the heap is ranked by lower-bound network distance between it and the query. We then use these heaps to avoid false positives and reduce unnecessary network distance computations during spatial keyword query processing. We explain our method using the running example in Figure 5.1.

For the Boolean 1NN query to find the POI containing "Thai" *and* "restaurant", our algorithm creates an on-demand inverted heap for a single keyword. We obtain a heap for the least frequent keyword as it contains fewer objects. Using the heap for "Thai" (e.g., o_2, o_6, o_5, and then o_7), the first object o_2 is pruned because it does not satisfy the keyword criteria. When o_6 is accessed, its exact network distance is computed as it contains both query keywords. If the network distance of o_6 is smaller than the lower-bound distance

of the next object (i.e., o_5), the algorithm terminates reporting o_6 as the result. The use of a cheap lower-bound heuristic avoids or delays computing expensive network distances to candidate objects (e.g., we avoid it for o_2, o_5, and o_7). Notably, this approach avoids generating false positive groups (e.g., G_1 in keyword aggregated indexing). In Section 5.5 we describe how heaps need only be populated partially and maintained in an iterative lazy manner, thus avoiding computing lower-bounds to all objects that contain the keyword. Other spatial keyword queries also benefit from inverted heaps. For example, we propose the idea of a *pseudo lower-bound* to retrieve more relevant candidates for top-k queries in Section 5.4.2.

This translates into significantly better query throughput (the number of queries processed per second) for K-SPIN based techniques in practice, as summarized in Table 5.1. Note that FS-FBS cannot be constructed on this dataset due to prohibitive pre-processing cost, but on smaller datasets, FS-FBS performs worse than our method K-SPIN (Section 5.7). We even show that K-SPIN is able to use G-tree's road network index more efficiently than G-tree's own query algorithm, confirming the reduction in false positives (Section 5.7.4).

Lightweight Separated Index: Creating a separate index for each keyword involves processing the objects and road network repeatedly. At first glance, doing this on road networks appears untenable. Nevertheless, every cloud has a silver lining. We make several smart, yet simple, observations (Section 5.6), which K-SPIN exploits to make the pre-processing more than viable, even lightweight. For example, we observe that the number of objects associated with a keyword is predictably small for most keywords in datasets that follow Zipf's law. Exploiting the nature of K-SPIN, we leverage this observation to significantly reduce construction time with theoretical and experimental justification. We also introduce a novel data structure, the ρ-Approximate Voronoi Diagram, to reduce the index size by over an order of magnitude. K-SPIN is applicable on even continental scale datasets, occupying less than 600MB and built in under 2 hours for the entire US road network dataset. These come at a small and theoretically bounded penalty in query performance and we still return *exact* query results.

Flexibility: As the lightweight keyword indexes are decoupled from the network distance index, K-SPIN can be combined with any network distance technique. This enables significant performance gains, e.g., the K-SPIN variant using Pruned Highway Labeling

(PHL) [Aki+14] in Table 5.1. Even the variant with the smallest memory footprint using Contraction Hierarchies (CH) [Gei+08] is much faster than the state-of-the-art in Table 5.1. Moreover, K-SPIN can be integrated into any system already processing road network queries and any future improved network distance technique can be plugged into the framework. This versatility of K-SPIN is unique among spatio-textual techniques.

Technique	**Index Size (GB)**	**Queries/second**	
		Top-k	**BkNN**
K-SPIN [Our Method] + CH [Gei+08]	0.6 + 0.6	865	1021
K-SPIN [Our Method] + PHL [Aki+14]	0.6 + 15.8	3942	9869
Spatial Keyword G-tree [Zho+15]	2.7	266	178
ROAD [Lee+12]	4.5	83	✗
FS-FBS [JFW15]	Dataset too large to build index		

Table 5.1: Comparison of index size and throughput (# of queries processed per second) on US road network dataset

5.2 Preliminaries

Road Network: The definition of the road network was provided in Section 1.1. Similar to previous chapters and almost all related studies (e.g., [JFW15; Zho+15]), we consider query locations and POIs occurring on vertices to make exposition simpler. As queries are graph operations, this does not change the asymptotic behavior. K-SPIN can easily be extended for other cases, e.g., POIs on edges would still be generated as candidates in on-demand inverted heaps.

Objects and Textual Information: The road network is also associated with a set of object vertices $O \subseteq V$ (i.e., POIs). Each object $o \in O$ contains a set of keywords known as the document, $doc(o)$, of object o. Each keyword $t \in doc(o)$ is drawn from a corpus of keywords W. For simplicity, we shall refer to $t \in doc(o)$ as $t \in o$ when the context is clear. We note that a keyword t may occur multiple times in $doc(o)$, the number of occurrences is denoted as its frequency $f_{t,o}$. Finally, the inverted list $inv(t)$ for keyword t is the set of objects whose document contains t. Next, we formally state our problem definitions.

Boolean kNN Queries: A Boolean k Nearest Neighbor (BkNN) query takes the form (q, k, ψ, op), where q is the query vertex, k is the number of results, ψ is a set of query keywords, and op specifies a logical operand ($\wedge$ or $\vee$) [JFW15]. The result of this query is the k nearest objects by their network distance to q, which satisfy the criteria. In the conjunctive case ($\wedge$), the result objects must contain all query keywords; and in the

disjunctive case ($\vee$), they contain at least one keyword from ψ. We remark that our proposed framework can be used to handle a combination of $\wedge$ and $\vee$ operators, e.g., find k closest POIs that contain "Thai" and ("takeaway" or "restaurant").

Top-k Spatial Keyword Queries: A top-k query is of the form (q, k, ψ), where q is the query vertex, k is the number of results, and ψ is a set of query keywords. The result is the set of k objects with the smallest scores. The score of each object is computed by combining its network distance from q and its textual relevance. We employ *weighted distance* [Wu+11; RN12] to compute the spatio-textual score for object o, as below.

$$ST(q, o) = \frac{d(q, o)}{TR(\psi, o)} \tag{5.1}$$

Here, $d(q, o)$ is the network distance from q to o and $TR(\psi, o)$ is the textual relevance. We adopt *cosine similarity* [ZM06] for computing $TR(\psi, o)$.

$$TR(\psi, o) = \frac{\sum_{t \in \psi}(w_{t,o} \cdot w_{t,\psi})}{\sqrt{\sum_{t \in o}(w_{t,o})^2 \cdot \sum_{t \in \psi}(w_{t,\psi})^2}} \tag{5.2}$$

In (5.2) above, $w_{t,o} = 1 + ln(f_{t,o})$ with $f_{t,o}$ being the frequency of keyword t in the document of o. Also, $w_{t,\psi} = ln(1 + \frac{|O|}{|inv(t)|})$, where $|O|$ is the total number of objects and $|inv(t)|$ is the size of the inverted list of t (i.e., the number of objects that contain t in their documents). While we do not dwell on the specifics of the textual relevance computation, $w_{t,o}$ represents a measure of the *term frequency* (TF), and $w_{t,\psi}$ represents the *inverse document frequency* (IDF).

As derived in past work [ZM06], (5.2) can be re-written in terms of *impacts*, or $\lambda_{t,x} = \frac{w_{t,x}}{\sqrt{\sum_{t \in x}(w_{t,x})^2}}$, as below.

$$TR(\psi, o) = \sum_{t \in \psi} [\lambda_{t,\psi} \cdot \lambda_{t,o}] \tag{5.3}$$

It is important to note that the object impact values $\lambda_{t,o}$ do not depend on the query and can be pre-computed offline. We emphasize that our indexing algorithms can support other textual relevance methods, such as language models, BM25, and other TF×IDF formulations, e.g., in [ZM06]. Similarly, our techniques are orthogonal to the scoring method and can be applied when *weighted sum* [Che+13] is used to combine $d(q, o)$ and $TR(\psi, o)$ in (5.1) instead of weighted distance, which we use as the example in our experiments.

5.3 K-SPIN: Framework Overview

Figure 5.2 depicts the modules that compose the K-SPIN framework. Here we briefly describe each module and how they interact before delving deeper into the design of specific modules in subsequent sections.

1. Lower Bounding Module. This module computes a lower-bound network distance between any two vertices using selected heuristics. For example, a lower-bound can be obtained using landmarks as in the ALT [GH05] index. ALT pre-computes network distances between some chosen landmark vertices and all vertices in the graph then uses the triangular inequality to obtain a lower-bound network distance between any two vertices. In fact, multiple heuristics can be considered to allow the module to return the tightest lower-bound network distance overall. Depending on the application and indexes available, the module may use more or fewer lower-bound heuristics. We combine K-SPIN with ALT as it provides effective lower-bounds on road networks for POI search, as we demonstrated in Chapter 4.

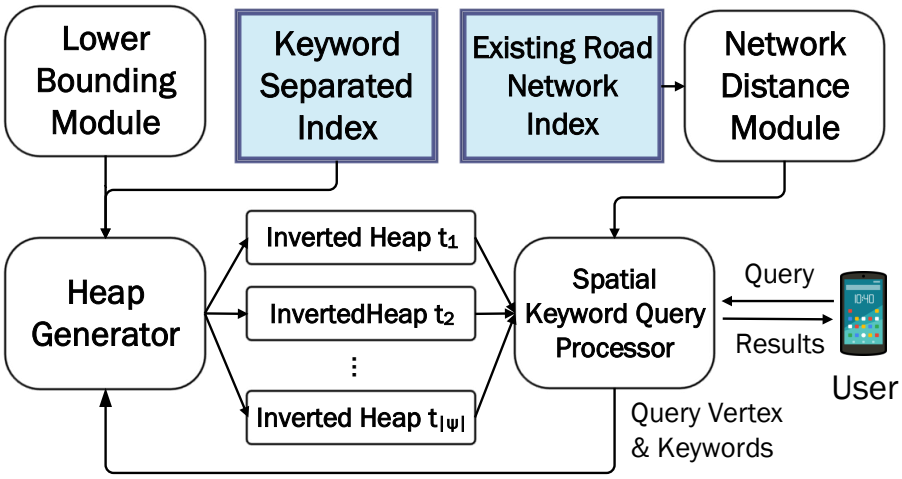


Figure 5.2: Keyword Separated Indexing (K-SPIN) Framework

2. Network Distance Module. This module is employed to compute the exact network distance between any two given vertices. As stated in Section 5.1, this module can use *any* existing technique to compute the network distance. The system administrator may choose a technique based on its efficiency and/or index size or may simply choose the techniques already being used by the system to answer other queries. In Section 5.7, we show the effect of choosing three different network distance techniques namely Contraction Hierarchies [Gei+08], G-tree [Zho+13], and Pruned Highway Labeling [Aki+14].

This module is the bottleneck as network distance computations are the most expensive operation performed for an object.

3. Heap Generator. The Heap Generator is responsible for creating and maintaining the *on-demand inverted heaps*. An on-demand inverted heap for a particular keyword t satisfies the following property at any point in time (i.e., when the heap is first created and whenever a heap element is extracted).

Property 1. *Given the current top object o in inverted heap $\mathcal{H}$ for keyword t and its lower-bound distance $LB(q, o)$ from query vertex q; any object o_t containing t, not yet extracted from $\mathcal{H}$, has network distance $d(q, o_t) \geq LB(q, o)$.*

Property 1 allows our query algorithms to access objects associated with a particular keyword t in order of their lower-bound network distances from q. To efficiently create and maintain an inverted heap, the Heap Generator utilizes a *Keyword Separated Index* that indexes $inv(t)$ for each keyword t in corpus W where $inv(t)$ is the set of all objects associated with t. For example, for keyword "Thai" in Figure 5.1, inv("Thai") consists of o_2, o_5, o_6, and o_7. The inverted heap $\mathcal{H}_{Thai}$ allows access to these objects in ascending order of their lower-bounds, e.g., $(o_2, 1), (o_6, 2), (o_5, 4)$, and then $(o_7, 5)$. Property 1 allows the heap to be populated lazily, i.e., objects are added incrementally such that the property is met. For example, heap $\mathcal{H}_{Thai}$ may initially contain only $(o_2, 1)$ and $(o_7, 5)$ to satisfy Property 1. When $(o_2, 1)$ is extracted, the object $(o_6, 2)$ may be inserted in the heap to ensure it satisfies Property 1. We present our Heap Generator algorithm and Keyword Separated Index data structure in Sections 5.5 and 5.6, respectively.

4. Query Processor. The Query Processor contains algorithms to answer various spatial keyword queries. Algorithms use on-demand inverted heaps to retrieve relevant candidate objects. The challenge lies in deciding which heap to use and how to filter poor candidates using an effective lower-bound score. Hence, the efficiency of the Query Processor is critical in avoiding the false positive problems of existing methods described in Section 5.1. The Query Processor uses the *Network Distance Module* to compute the network distances between the query vertex and the filtered candidate objects. The network distance module, in turn, employs some *Road Network Index* that can answer network distance queries. Our query algorithms are detailed in Section 5.4.

5.4 The Query Processor Module

We first describe the algorithms for Boolean kNN queries in Section 5.4.1, demonstrating how inverted heaps are used. Section 5.4.2 details our top-k algorithm where we introduce the idea of a pseudo lower-bound utilizing a subtle insight to retrieve more relevant candidates and thereby terminate quicker.

5.4.1 Boolean *k*NN Query Processing

Boolean kNN (BkNN) queries retrieve the k nearest objects to q whose associated keywords satisfy some criteria with the set of query keywords ψ. In disjunctive queries, reported objects contain at least one keyword in ψ and in conjunctive queries reported objects contain all keywords in ψ.

Disjunctive Boolean *k*NN Queries

Algorithm 5 begins by initializing an on-demand inverted heap $\mathcal{H}_i$ for each keyword t_i (line 2). Recall that the Heap Generator ensures each heap $\mathcal{H}_i$ satisfies Property 1. Thus we access objects from each heap in order of minimum lower-bound distance from q. Priority queue $\mathcal{PQ}$ is used to choose the heap with the *smallest* minimum lower-bound distance. $\mathcal{PQ}$ is first initialized by inserting the lower-bound distance of the top object in each heap $\mathcal{H}_i$ (lines 3 and 4). The top element in $\mathcal{PQ}$ is extracted (line 6) to identify the heap $\mathcal{H}_s$ whose top object has the smallest lower-bound distance. Candidate object c is then extracted from $\mathcal{H}_s$ (line 7) and LazyReheap is called (line 8) to ensure $\mathcal{H}_s$ continues to satisfy Property 1 (detailed in Section 5.6). c is ignored if it has been extracted from another heap $\mathcal{H}_i$, otherwise network distance $d(q, c)$ is computed (line 11). The set of k best candidates L seen so far is updated if c improves on it and D_k is also updated if it changes (line 13). D_k corresponds to the distance of the kth closest object in L that satisfies the keyword criteria. An element for $\mathcal{H}_s$ is re-inserted into $\mathcal{PQ}$ with its new minimum lower-bound after c is extracted (line 9) to ensure $\mathcal{PQ}$ always chooses the heap whose top element has the smallest lower-bound distance. The algorithm terminates when $\mathcal{PQ}$ is empty or the next candidate object has a lower-bound distance greater than or equal to D_k (line 5).

Algorithm 5 Query Processor module to answer disjunctive BkNN queries

```
1: function GetDisjunctiveBkNNs(k, q, ψ)
2:   Create on-demand inverted heap H_i for each keyword t_i ∈ ψ
3:   Initialize minimum priority queue PQ and set D_k ← ∞
4:   Insert minimum lower-bound distance for each H_i into PQ
5:   while !Empty(PQ) and MinKey(PQ) < D_k do
6:     H_s ← Extract-Min(PQ)
7:     LB(q, c) ← MinKey(H_s) , c ← Extract-Min(H_s)
8:     Call LazyReheap(H_s) to ensure Prop. 1 (see Section 5.6)
9:     Insert(PQ,[t_s,MinKey(H_s)])
10:    if c not already evaluated then
11:      Compute network distance d(q, c)
12:      if d(q, c) < D_k then
13:        Insert(L, [c, d(q, c)]) and update L and D_k if needed
14:  return L
```

Example 3. *Consider a disjunctive B1NN query from vertex q in Figure 5.1 with query keywords "Thai" and "restaurant". On-demand inverted heaps are generated for each keyword, e.g., $\mathcal{H}_{Thai}$=$\{(o_2, 1), (o_6, 2)\}$ and $\mathcal{H}_{rest.}$=$\{(o_1, 2), (o_6, 2)\}$. Each heap's minimum lower-bound is inserted into the priority queue, so $\mathcal{PQ}$=$\{(Thai, 1), (rest., 2)\}$. The top element Thai from $\mathcal{PQ}$ is extracted, identifying $\mathcal{H}_{Thai}$. Now the top candidate object o_2 is extracted from inverted heap $\mathcal{H}_{Thai}$. Naturally, o_2 satisfies the disjunctive criteria, so its network distance is computed, added to the result set L and D_k is set to $d(q, o_2)$=2.* LazyReheap *is called to ensure $\mathcal{H}_{Thai}$ satisfies Property 1, e.g., $\mathcal{H}_{Thai}$=$\{(o_6, 2), (o_7, 4)\}$. Thai is reinserted into $\mathcal{PQ}$ with the minimum lower-bound in $\mathcal{H}_{Thai}$, so $\mathcal{PQ}$=$\{(Thai, 2), (rest., 2)\}$. L cannot be improved as* MINKEY($\mathcal{PQ}$) *is equal to D_k and Algorithm 5 terminates.*

Conjunctive Boolean *k*NN Queries

We also exploit keyword separation to create an algorithm to answer conjunctive BkNN queries in Algorithm 6. First, the algorithm initializes a heap $\mathcal{H}_l$ related to the least frequent keyword t_l, i.e., the keyword with the smallest inverted list (the size of each inverted list is easily stored in the keyword index). This uses the intuition that the least frequent keyword can be considered the bottleneck for conjunctive queries. Furthermore, with lower frequencies, the average distance between candidates increases, making it easier to distinguish them by their lower-bound distances as the error is less likely to create false positives. The algorithm iteratively accesses objects from $\mathcal{H}_l$. If the accessed object c contains all query keywords, its network distance is computed and c is added to L if it

improves on the best k candidates seen so far. D_k is updated if it changes. If the object c does not contain all query keywords, it is ignored. The algorithm continues to iteratively extract objects from $\mathcal{H}_l$ until either $\mathcal{H}_l$ becomes empty or $LB(q, c) \geq D_k$.

Algorithm 6 Query Processor module to answer conjunctive BkNN queries

```
1: function GetConjunctiveBkNNs(k, q, ψ)
2:   Create on-demand inverted heap H_l for least freq. keyword t_l ∈ ψ and set D_k ← ∞
3:   while !Empty(H_l) and MinKey(H_l) < D_k do
4:     LB(q, c) ← MinKey(H_l), c ← Extract-Min(H_l)
5:     Call LazyReheap(H_l) to ensure Prop. 1 (see Section 5.6)
6:     Compute network distance d(q, c)
7:     if d(q, c) < D_k then
8:       Insert(L, [c, d(q, c)]) and update L and D_k if needed
9:   return L
```

Example 4. *Consider a conjunctive B1NN query from vertex q in Figure 5.1 with query keywords t_1 "Thai" and t_2 "grocer". "grocer", occurring once, is the least frequent keyword so an on-demand inverted heap $\mathcal{H}_2$ is generated for t_2. The algorithm then extracts the top candidate object in $\mathcal{H}_2$. This object is o_2, after verifying o_2 contains the other keyword "Thai", the network distance to it is computed, added to the result set L and D_k is set to $d(q, o_2)$=2. Since $\mathcal{H}_2$ is now empty, the algorithm terminates.*

5.4.2 Top-k Query Processing

We propose a novel top-k query algorithm to retrieve the k objects with the best spatio-textual scores by (5.1). Our algorithm computes the top-k objects by utilizing, for each inverted heap, a *pseudo lower-bound score* on only some of the unseen objects in the heap. The algorithm still computes correct results even though the pseudo lower-bound score is not a valid lower-bound score for all unseen objects in the heap. Next, we first describe how to compute a valid lower-bound score on all unseen objects.

Valid Lower-Bound Score on All Unseen Objects: Consider a top-k query for three keywords "Italian", "restaurant", and "takeaway". Let $TR_{max}(\psi, P)$ be the maximum possible textual relevance for query keywords ψ with any object in set P. For simplicity, assume that textual similarity $TR(\psi, o)$ is the number of query keywords present in the object o, so $TR_{max}(\psi, P) = 3$ in this example. Figure 5.3 shows the three inverted heaps $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ created for this query with objects from our running example in Figure 5.1. Since we do not know the textual similarity of unseen objects in any heap $\mathcal{H}_i$, a lower-bound score for all unseen objects in $\mathcal{H}_i$ can be computed using the maximum textual

similarity and minimum lower-bound distance in the heap as $ST_{all}(\psi, \mathcal{H}_i) = \frac{\text{MinKey}(\mathcal{H}_i)}{TR_{max}(\psi,P)}$. For example, the best possible score for any unseen object in $\mathcal{H}_1$ is $\frac{LB(q,o_1)}{3} = \frac{2.7}{3} = 0.9$, $\mathcal{H}_2$ is $\frac{2.4}{3} = 0.8$, and $\mathcal{H}_3$ is $\frac{1.8}{3} = 0.6$. But, as we explain next, it is possible to obtain a pseudo lower-bound score tighter than this without losing any top-k results.

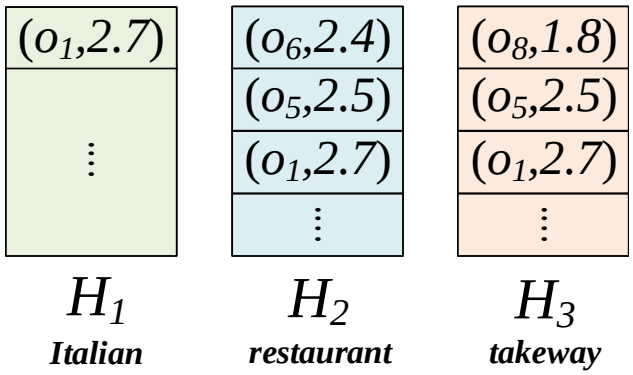


Figure 5.3: Computing Pseudo Lower-Bounds on Inverted Heaps

A Key Insight: Heap $\mathcal{H}_3$ in Figure 5.3 has the smallest lower-bound distance and its top element is object o_8. Since the lower-bound distance $LB(q, o_8) = 1.8$ is smaller than the lower-bound distance of $\mathcal{H}_1$ (i.e., $LB(q, o_1) = 2.7$), this implies that either o_8 has been extracted from $\mathcal{H}_1$ or o_8 does not contain the keyword "Italian". For the same reason, o_8 has either been extracted from $\mathcal{H}_2$ or does not contain the keyword "restaurant". In other words, either o_8 has already been processed by the algorithm (i.e., extracted from another heap) or o_8 only contains the keyword "takeaway". Similarly for heap $\mathcal{H}_2$, top element o_6 has either been extracted from $\mathcal{H}_1$ or contains only the keywords "restaurant" and "takeaway" at best (o_6 may contain "takeaway" because o_6 may still be in $\mathcal{H}_3$ as it has a smaller top lower-bound distance).

Pseudo Lower-Bound Score: Using the insight above, we compute a *pseudo lower-bound score* that is a lower-bound score on a subset of the objects in the inverted heap and hence is tighter than the valid lower-bound score. Let $\text{MinKey}(\mathcal{H}_j)$ be the minimum lower-bound network distance for an element in heap $\mathcal{H}_j$. If $\mathcal{H}_j$ has become empty, $\text{MinKey}(\mathcal{H}_j)$ is assumed to be infinite. The pseudo lower-bound score for a heap $\mathcal{H}_i$ is computed by assuming that every unseen object in $\mathcal{H}_i$ contains a keyword t_j only if $\text{MinKey}(\mathcal{H}_i) \geq \text{MinKey}(\mathcal{H}_j)$ where t_j is the query keyword associated with heap $\mathcal{H}_j$. We next describe the algorithm to compute pseudo lower-bound scores.

Algorithm 7 computes the pseudo lower-bound score for inverted heap $\mathcal{H}_i$ denoted by $ST_{pLB}(\mathcal{H}_i)$. First, it computes a pseudo textual relevance $TR_p(\psi, \mathcal{H}_i)$ following (5.3) by

considering only the keywords that satisfy the condition described above (lines 4-5). Note that we use the real maximum impact $\lambda_{t_j,max}$ of t_j in any object, which can be cheaply computed offline for all keywords. The pseudo lower-bound score is computed using the pseudo textual relevance $TR_p(\psi, \mathcal{H}_i)$ and then returned (line 6).

Algorithm 7 Compute pseudo lower-bound score for inverted heap $\mathcal{H}_i$

1: **function** PSEUDOLB($\psi, \mathcal{H}_i$)
2: $TR_p(\psi, \mathcal{H}_i) \leftarrow 0$
3: **for each** keyword $t_j \in \psi$ **do**
4: **if** MINKEY($\mathcal{H}_i$)$\geq$MINKEY($\mathcal{H}_j$) **then**
5: $TR_p(\psi, \mathcal{H}_i) \leftarrow TR_p(\psi, \mathcal{H}_i) + \lambda_{t_j,\psi} \times \lambda_{t_j,max}$
6: **return** $ST_{pLB}(\mathcal{H}_i) \leftarrow \frac{\text{MINKEY}(\mathcal{H}_i)}{TR_p(\psi,\mathcal{H}_i)}$

Example 5. *Consider again the example in Figure 5.3. The pseudo lower-bound score of* $\mathcal{H}_2$ *is computed assuming that all unseen objects in* $\mathcal{H}_2$ *can only include two keywords "restaurant" and "takeaway", i.e.,* $TR_p(\psi, \mathcal{H}_2) = 2$ *and* $ST_{pLB}(\mathcal{H}_2) = \frac{2.4}{2} = 1.2$. *Similarly,* $\mathcal{H}_3$ *includes only the keyword "takeaway" and* $ST_{pLB}(\mathcal{H}_3) = \frac{1.8}{1} = 1.8$. $\mathcal{H}_1$ *includes all three keywords and* $ST_{pLB}(\mathcal{H}_1) = \frac{2.7}{3} = 0.9$. *Note that pseudo lower-bound scores are not valid lower-bound scores, e.g., the spatio-textual score of* o_1 *in* $\mathcal{H}_2$ *is* $\frac{d(q,o_1)}{TR(\psi,o_1)} = \frac{3}{3} = 1$ *which is smaller than the pseudo lower-bound* $ST_{pLB}(\mathcal{H}_2) = 1.2$.

Next, we show how the Query Processor can use pseudo lower-bounds to answer top-k queries instead of valid lower-bounds. We then prove that it still computes correct results and elaborate on why the pseudo lower-bound score is useful.

Query Processor: The top-k algorithm (Algorithm 8) is quite similar to the algorithm for computing disjunctive BkNN queries. The main difference is that pseudo lower-bound scores of heaps are used in $\mathcal{PQ}$ (see line 4) to access the heap with the best candidate object. If the extracted candidate object c has not already been processed, a lower-bound score is cheaply computed using its *actual* textual relevance and lower-bound network distance (line 10), i.e., $\frac{LB(q,c)}{TR(\psi,c)}$. If this lower-bound score is smaller than D_k, then its actual score is computed using its exact network distance $d(q, c)$ (lines 11 and 12). If its actual score is smaller than D_k, the result list L and D_k are updated accordingly (line 14). The algorithm terminates when $\mathcal{PQ}$ is empty or the top of $\mathcal{PQ}$, representing the smallest pseudo lower-bound score of any heap, is greater than or equal to D_k as L can no longer be improved.

Algorithm 8 Query Processor module to answer top-k queries

1: **function** GETTOPKOBJECTS(q, k, ψ)
2: Create on-demand inverted heap $\mathcal{H}_i$ for each keyword $t_i \in \psi$
3: Initialize minimum priority queue $\mathcal{PQ}$ and set $D_k \leftarrow \infty$
4: Insert pseudo lower-bound score for each $\mathcal{H}_i$ into $\mathcal{PQ}$
5: **while** !EMPTY($\mathcal{PQ}$) **and** TOP($\mathcal{PQ}$) $< D_k$ **do**
6: $n \leftarrow$ EXTRACT-MIN($\mathcal{PQ}$)
7: $LB(q, c) \leftarrow$ MINKEY($\mathcal{H}_n$) , $c \leftarrow$ EXTRACT-MIN($\mathcal{H}_n$)
8: LAZYREHEAP($\mathcal{H}_n$)
9: INSERT($\mathcal{PQ}$,[n,PSEUDOLB($\psi, \mathcal{H}_n$)])
10: **if** c not already processed or $\frac{LB(q,c)}{TR(\psi,c)} \leq D_k$ **then**
11: Compute network distance $d(q, c)$
12: $ST(q, c) \leftarrow \frac{d(q,c)}{TR(\psi,c)}$ ▷ Compute actual score
13: **if** $ST(q, c) < D_k$ **then**
14: INSERT($L, (c, ST(q, c))$), update L and D_k if needed
15: **return** L

Example 6. *Consider a top-*1 *query for our running example in Figure 5.1 and Figure 5.3 with keywords "Italian", "restaurant", and "takeaway". If the heaps are accessed considering the actual lower-bounds,* o_1 *(which is the result) will be the last accessed object. However, our algorithm accesses the heaps based on their pseudo lower-bound scores and* $\mathcal{H}_1$ *has smaller pseudo lower-bound scores than the other two heaps (as seen in Example 5). Thus* $\mathcal{H}_1$ *is accessed first. So Algorithm 8 extracts candidate* o_1*, computes its spatio-textual score* $\frac{d(q,o_1)}{TR(\psi,o_1)} = 1$*, re-inserts an element into* $\mathcal{PQ}$ *for* H_1 *with its new* MINKEY *(i.e., infinity as* $\mathcal{H}_1$ *is now empty). After updating the result set with* o_1*, the algorithm then terminates because the next best pseudo lower-bound score in* $\mathcal{PQ}$ *is* $ST_{pLB}(\mathcal{H}_2) = 1.2$ *which is higher than the score of the current top-*1 *object* o_1*.*

Implementation Notes: While the same candidate may be extracted from multiple heaps (i.e., when associated with multiple query keywords), these can be ignored by using a hash-table or bit-array to track evaluated candidates. In any case, this only entails a small query overhead as the lower-bound computation is cheap and the heap only contains a small number of objects (due to being lazily populated) resulting in a small update cost. Also note that query impacts $\lambda_{t,\psi}$ need only be computed once for the query and $TR(\psi, c)$ need only be computed once for each candidate.

Benefits of Pseudo Lower-Bound Scores: We propose Lemma 2 to show that a pseudo lower-bound score is never worse than the valid lower-bound score for all unseen objects.

Lemma 2. *For any heap $\mathcal{H}_i$, the pseudo lower-bound is always greater than or equal to the valid lower-bound for all unseen objects in $\mathcal{H}_i$, i.e., $ST_{pLB}(\psi, \mathcal{H}_i) \geq ST_{all}(\psi, \mathcal{H}_i)$.*

Proof. Since $ST_{pLB}(\psi, \mathcal{H}_i) = \frac{\text{MinKey}(\mathcal{H}_i)}{TR_p(\psi, \mathcal{H}_i)}$ and $ST_{all}(\psi, \mathcal{H}_i) = \frac{\text{MinKey}(\mathcal{H}_i)}{TR_{max}(\psi, P)}$, it suffices to show that $TR_p(\psi, \mathcal{H}_i) \leq TR_{max}(\psi, P)$. The maximum possible textual relevance for any object in set P can be computed by $TR_{max}(\psi, P) = \sum_{t \in \psi} \lambda_{t,\psi} \times \lambda_{t,max}$ where $\lambda_{t,max}$ is maximum impact of keyword t in any object. By Algorithm 7, we have $TR_p(\psi, \mathcal{H}_i) = \sum_{t_j \in \psi} \lambda_{t_j,\psi} \times \lambda_{t_j,max}[\text{MinKey}(\mathcal{H}_i) \geq \text{MinKey}(\mathcal{H}_j)]$ where t_j is the keyword associated with heap $\mathcal{H}_j$. Clearly the maximum value of $TR_p(\psi, \mathcal{H}_i)$ is $TR_{max}(\psi, P)$, occurring when condition $[\text{MinKey}(\mathcal{H}_i) \geq \text{MinKey}(\mathcal{H}_j)]$ evaluates to true for all heaps $\mathcal{H}_j$. Thus $TR_p(\psi, \mathcal{H}_i) \leq TR_{max}(\psi, P)$, thereby completing the proof. □

From Lemma 2, it can be seen that the textual relevance used for a pseudo lower-bound depends on the condition $[\text{MinKey}(H_i) \geq \text{MinKey}(H_j)]$ over all j. This condition is likely to result in decreasing textual relevance for each subsequent heap in descending order of their MinKey values. This entails increasing pseudo lower-bounds, which in turn allows Algorithm 8 to avoid accessing heaps and terminate sooner. Conversely, the pseudo lower-bound assigns higher textual relevance to larger MinKey values. This allows Algorithm 8 to access more promising candidates, e.g., those that are far from q but contain all keywords and have high textual relevance. Furthermore, K-SPIN is likely to filter out any bad candidates using their actual textual relevance without computing expensive network distances. Pseudo lower-bounds can be applied to any textual model that computes similarity per query keyword, as many popular methods do, including language models, TF×IDF, and BM25.

Proof of Correctness: As stated earlier, pseudo lower-bound scores are not valid lower-bounds, e.g., $ST_{pLB}(\mathcal{H}_2) = 1.2$ is higher than the score of o_1 (1) which is also present in $\mathcal{H}_2$. While it may seem like this can lead to missing objects, the algorithm still produces correct results, e.g., because o_1 is also present in $\mathcal{H}_1$ and its score cannot be better than $ST_{pLB}(\mathcal{H}_1)$. We propose Lemma 3 to express this formally.

Lemma 3. *When Algorithm 8 terminates, every object o that has not been seen has $ST(q, o) \geq D_k$.*

Proof. The algorithm terminates when $\text{Top}(\mathcal{PQ}) \geq D_k$. This implies that, for every heap $\mathcal{H}_i$, $ST_{pLB}(\mathcal{H}_i) \geq D_k$ when the algorithm terminates. Let $\mathcal{H}_{max}$ be the heap with

the largest MinKey. Next, we show that $ST(q,o) \geq ST_{pLB}(\mathcal{H}_{max})$ which implies that $ST(q,o) \geq D_k$ for every unseen object o.

Recall that $ST(q,o) = \frac{d(q,o)}{TR(\psi,o)}$ and $ST_{pLB}(\mathcal{H}_{max}) = \frac{\text{MinKey}(\mathcal{H}_{max})}{TR_p(\psi,\mathcal{H}_{max})}$. Since $\mathcal{H}_{max}$ is the heap with the largest MinKey, Algorithm 7 (lines 4-5) computes $TR_p(\psi,\mathcal{H}_{max})$ assuming it contains all query keywords. Therefore, $TR(\psi,o) \leq TR_p(\psi,\mathcal{H}_{max})$. Furthermore, since o has not been seen by Algorithm 8, $LB(q,o) \geq \text{MinKey}(\mathcal{H}_{max})$ otherwise it would have been extracted from at least one heap $\mathcal{H}_i$. Thus, $d(q,o) \geq \text{MinKey}(\mathcal{H}_{max})$. Hence, $ST(q,o) \geq ST_{pLB}(\mathcal{H}_{max})$.

□

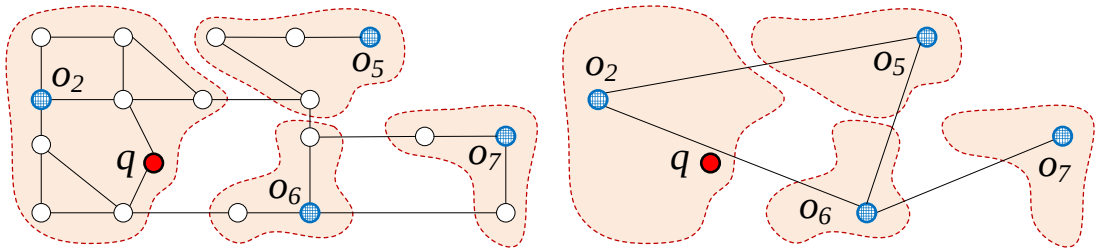
5.5 Heap Generator Module

A Heap Generator creates an on-demand inverted heap for query keyword t. This inverted heap satisfies Property 1, i.e., allows access to objects containing keyword t in ascending order of their lower-bound network distances from query location q. A simple approach to ensure Property 1 is to insert all objects from the inverted list of t (i.e., $inv(t)$) in the heap with their lower-bound distances. However, this is not feasible as it would be required for *every* query. In this section we describe a Heap Generator based on the Network Voronoi Diagram (NVD) [KS04] that instead allows inverted heaps to be populated lazily. However NVDs possess high pre-processing costs, which we describe below before proposing a solution (with low pre-processing cost) in Section 5.6.

Network Voronoi Diagrams: Given a set of objects $inv(t)$ containing keyword t, an NVD is a disjoint partitioning of the road network vertices V for each object in $inv(t)$. A partition for object o_i is the Voronoi node set $Vns(o_i) \subseteq V$ which contains every vertex for which o_i is its closest object by network distance. After computing all Voronoi node sets, the NVD stores the nearest object o_i for every vertex in V.

Figure 5.4(a) shows the NVD for the set of objects containing keyword "Thai" from our running example. The shaded containers indicate the vertices belonging to each Voronoi node set. Note that the NVD does not depend on the query vertex q. Two Voronoi node sets $Vns(o_i)$ and $Vns(o_j)$ are considered *adjacent* if there is an edge $(u,v) \in E$ connecting $u \in Vns(o_i)$ and $v \in Vns(o_j)$. For simplicity, we also say that o_i and o_j are adjacent. So,

in Figure 5.4(a), o_2 and o_5 are adjacent as there is a graph edge connecting the shaded containers. Similarly, o_2 and o_6 are also adjacent.



(a) NVD for "Thai" in Fig. 5.1 (b) Adjacency Graph for Fig. 5.4(a)

Figure 5.4: Example Network Voronoi Diagram

NVD-Based kNN Algorithms The 1NN of a query vertex can be found using an NVD as it stores the nearest object o_i for each vertex, e.g., o_2 is the 1NN of q in Figure 5.4(b) as q is in its Voronoi node set (shaded container). Kolahdouzan and Shahabi [KS04] presented a useful property to find kNNs.

Property 2. *k-th nearest object of q must be an object adjacent (in the NVD) to the first $k-1$ nearest objects of q.*

For example, the 2nd NN of q must be among the objects adjacent to o_2 (i.e., o_5 and o_6). This is because the shortest path from q to the 2nd NN must leave $Vns(o_2)$ and enter one of the adjacent Voronoi node sets. Existing techniques, such as VN^3 [KS04] and the NVD-based decoupled heuristic we proposed in Chapter 4, exploit this property to incrementally answer kNN queries.

Heap Generation via NVD: Property 2 can also be used to create and lazily maintain an on-demand inverted heap for any keyword t. Specifically, a heap can be initialized by inserting 1NN of q obtained from the NVD. Then, whenever an object o is extracted, the adjacent objects of o in the NVD are inserted into the heap with their lower-bound network distances.

When an NVD is computed, we also create an *adjacency graph* representing the relationships between objects that are adjacent to each other. Each node in the adjacency graph is an object o_i and an edge between two nodes o_i and o_j is created if o_i and o_j are adjacent in the NVD. Figure 5.4(b) shows an adjacency graph for the NVD shown in Figure 5.4(a).

Algorithm 9 describes how to maintain an inverted heap $\mathcal{H}$. After the heap is initialized with the 1NN as described earlier, LazyReheap is called whenever an object o_c is extracted from $\mathcal{H}$. Then the adjacent objects of o_c that were not previously inserted are now inserted in $\mathcal{H}$ with their lower-bound network distances using the NVD's adjacency graph.

Algorithm 9 Lazily maintain inverted heap $\mathcal{H}$ to satisfy Property 1

```
1: function LazyReheap(H, q, o_c)
2:   for each o_a adjacent to o_c in adjacency graph do
3:     if o_a has not been inserted into H then
4:       Compute lower-bound network distance LB(q, o_a)
5:       H.Insert(o_a, LB(q, o_a))
6:       Mark o_a as "inserted" into H
```

Limitations: While NVDs allow efficient creation and maintenance of on-demand inverted heaps, it comes at the expense of higher pre-processing cost. This is exacerbated by building an NVD for each keyword t. An NVD takes $O(|V|log|V|)$ time and $O(|V|)$ space [EH00] and building one for each keyword multiplies them by $|W|$. For example, $|V|$ is 24 million and $|W|$ is 106,000 for the US road network dataset and, even with existing optimizations, the resulting index requires 3-days to build and occupies 90GB of memory! Updating NVDs when an object is added/deleted or changed also comes at a sizable cost. Next, we make several important observations and propose a space-efficient NVD with significantly reduced pre-processing and update costs.

5.5.1 Query Processor Complexity

Based on this heap generator module, we may now derive expressions for query time. We perform our analysis for BkNN queries, but a similar analysis can be performed for top-k queries. For a BkNN query, let us say the loop in Algorithm 5 runs for $\kappa \geq k$ iterations. The value of κ depends on the efficiency of the candidate generation heuristic. The inverted heap $\mathcal{H}$ contains at most $|O|$ objects, thus extracting from a binary heap implementation takes $O(\log|O|)$ time. For an NVD graph with maximum degree Δ, LazyReHeap computes a lower-bound for each adjacent object and inserts them into heap $\mathcal{H}$ at cost $O(\log|O|)$. Using the ALT index to compute a lower-bound takes $O(m)$ time where m is a small constant (typically 16) and in practice Δ is also a small constant both in our experiments and past studies [KS04]. Lastly, a single network distance computation

is performed per iteration with time, denoted by $O(NDIST)$, depending on the technique used. This operation tends to dominate the iteration's cost, e.g., a Contraction Hierarchies query takes $O(\log^2 n \log^2 D)$ time [Abr+10]. So the total query time is $O(\kappa m \Delta \log |O| + \kappa NDIST)$ for a BkNN query. Top-k queries have an additional small constant time cost per iteration to compute textual relevance. The smallest possible value of κ is k for a perfect heuristic and in practice κ is a small constant multiple of k, at most $3k$ for BkNN and $5k$ for top-k queries over all settings in our experiments.

5.6 Keyword Separated Index

Keyword separation has led to the higher pre-processing cost described above, but a remedy can also be found in keyword separation. Inspired by several simple but smart observations, we propose a novel space-efficient NVD. The resulting Keyword Separated Index is not only viable but also lightweight.

Observation 1: Most keywords have small inverted lists and this is consistent for any Zipfian dataset. Keywords in real-world datasets are known to follow Zipf's law [JFW15]. Let frequency f_t be the size of a keyword t's inverted list and r_t be the rank of t by its frequency in corpus W. Zipf's law states $f_t \propto \frac{1}{{r_t}^\alpha}$ with $\alpha \approx 1$. In simple terms, classic Zipf's law suggests keyword t with rank r_t occurs $\frac{1}{r_t}$ as often as the most frequent keyword.

We can predict the frequency of any keyword using the theoretical basis of Zipf's law. For example, we can predict that 80% of keywords have a frequency of $\frac{f_{max}}{0.2|W|}$ or less, where f_{max} is the maximum frequency of a keyword and $|W|$ is the number of keywords. This predicted 80-th percentile frequency is less than or equal to 5 for all datasets listed in Table 5.2 and closely matches the real values. On reflection, this is not surprising as Zipfian distributions follow a harmonic progression, i.e., are "long-tailed".

K-SPIN can exploit this observation to avoid creating indexes for a vast majority of keywords while only paying a small penalty in query performance. If the number of objects in the inverted list $inv(t)$ of keyword t is at most a small constant ρ, we do not create an NVD at all. For queries involving such keywords, we simply need to initialize the inverted heap with all objects in $inv(t)$, which is at worst only ρ objects and in K-SPIN only costs a cheap lower-bound computation anyway. Using $\rho = 5$, indexes for over 80% of keywords are avoided, substantially reducing pre-processing cost. Moreover, given the

long tail Zipfian distributions, such a ρ will scale slowly for increasing keyword dataset size.

Observation 2a: While the size of an NVD is $O(|V|)$, the adjacency graph takes $O(|inv(t)|)$ space where $|inv(t)| \leq |V|$ is the total number of objects containing keyword t. In general, the average degree in NVD adjacency graphs is a small constant, e.g., 6 as shown in [KS04] over several real-world road networks. Therefore, the adjacency graph's size is linear to the number of objects and is independent of $|V|$. In short, *we only need the small adjacency graph to maintain the heap and not the large NVD*, the latter being the bottleneck for space usage.

Observation 2b: K-SPIN does not actually require an NVD to provide the exact 1NN of q when initializing an inverted heap. The heap would still satisfy Property 1 if we initialize it with $\rho \geq 1$ candidate objects as long as the 1NN of q is among the ρ objects, as proven in Theorem 2.

Theorem 2. *An inverted heap $\mathcal{H}$ initialized as above and maintained by Algorithm 9 satisfies Property 1. Specifically, let o_c be the current top object in $\mathcal{H}$ with lower-bound network distance $LB(q, o_c)$. Every object o_x that is not yet extracted from $\mathcal{H}$ has network distance $d(q, o_x) \geq LB(q, o_c)$.*

Proof. We prove Theorem 2 for each possible case:

At Initialization: Let o_1 be the 1NN of q. At initialization, the heap contains up to ρ objects including o_1. Since o_1 is the 1NN, $d(q, o_1) \leq d(q, o_x)$. And since o_1 is in the heap, it is obvious that $LB(q, o_c) \leq LB(q, o_1) \leq d(q, o_1) \leq d(q, o_x)$ (note that o_1 and o_c could be the same object).

General Case: If o_x is in the heap then clearly, we have $LB(q, o_c) \leq LB(q, o_x) \leq d(q, o_x)$. If o_x is not in the heap, this means o_x is not adjacent to any object that has been extracted from $\mathcal{H}$ (or it would have been inserted it into the heap by Algorithm 9). Therefore, as observed in [KS04], there exists at least one object o_y in the heap such that $d(q, o_y) \leq d(q, o_x)$. Since o_c is the top object in the heap we must have $LB(q, o_c) \leq LB(q, o_y)$, which implies $LB(q, o_c) \leq LB(q, o_y) \leq d(q, o_y) \leq d(q, o_x)$. □

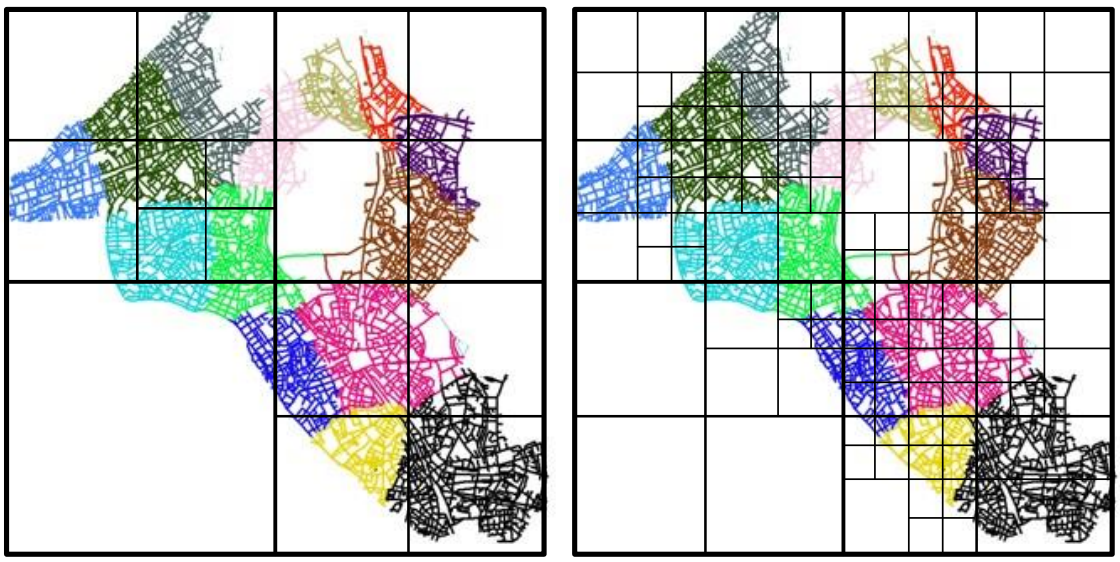
Observation 3: Separated indexing means that building NVDs are independent operations. As an added benefit of the K-SPIN framework, NVD construction is easily parallelized on all available cores to further reduce the construction time.

5.6.1 ρ-Approximate Network Voronoi Diagram

Observations 2a and 2b suggest that an exact NVD is not necessary to initialize or maintain inverted heaps. We propose the ρ-Approximate NVD, defined below, to take advantage of these observations while significantly reducing index size. Furthermore, due to the nature of K-SPIN, it still returns *exact* query results.

Definition 1. *A ρ-Approximate Network Voronoi Diagram allows retrieving, for* every *vertex $v \in V$, up to ρ objects such that one of these ρ objects is the 1NN of v.*

Constructing ρ-Approximate NVDs: We first compute an exact NVD in $O(|V| \log |V|)$ time if there more than ρ objects. We then store a ρ-Approximate NVD in a quadtree as follows. The root node of the quadtree is a minimum bounding box of all vertices in the road network. Each node is recursively divided into four children until all the vertices contained in the node belong to at most ρ different Voronoi node sets. To simplify the explanation, assume that each object $o \in O$ has a unique color and an NVD is represented by assigning each vertex $v \in V$ the same color as the color of its nearest object (see Figure 5.5). The ρ-Approximate quadtree continues dividing nodes into four children until the node contains at most ρ different colors. Figure 5.5(a) shows a 4-Approximate NVD indexed using a quadtree. After each iteration only the ρ-Approximate NVD is kept (the exact NVD is not kept). If there are fewer than ρ objects, the exact NVD does not need to be computed at all, which is quite beneficial as per Observation 1.

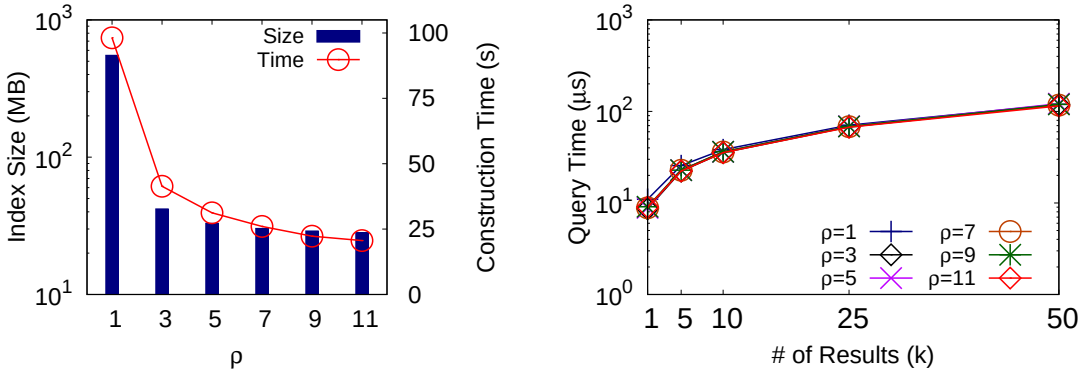


(a) 4-approximate Quadtree (b) Region Quadtree (shown to depth 4)

Figure 5.5: ρ-Approximate Network Voronoi Diagram

The ρ-Approximate NVD indexed using a quadtree significantly reduces space usage compared to an exact NVD indexed using standard techniques such as a region quadtree. By relaxing the need to distinguish the boundaries of different Voronoi node sets, the ρ-Approximate NVD is able to reduce the height of the required quadtree as shown in Figure 5.5(a). On the other hand, an exact NVD's region quadtree continues dividing nodes into four children until the node contains exactly one color, i.e., $\rho = 1$. This results in a deeper tree and hence significantly higher space usage. For example, in Figure 5.5(b) the exact NVD's region quadtree is shown only up to a depth of 4 and there are still quite a few nodes that contain more than 1 color. Voronoi node sets exhibit *spatial coherence* [SAS05], forming largely contiguous regions. This property combined with Observation 2a, the number of adjacent Voronoi node sets being a small constant, suggests ρ-Approximate NVDs will be quite effective even for small values of ρ.

Experimental Index Size and Time: Figure 5.6(a) shows the effect of ρ from 1 to 11 on pre-processing of the Florida road network with 1 million vertices. Observation 2a+b result in an index that is 18 times smaller for $\rho = 5$ than exact NVDs indexed by region quadtrees (i.e., $\rho = 1$) as shown in the bar plots (refer to the left-hand y-scale). The effect of Observation 1 is seen in the index time line-plot (refer to the right-hand y-scale), with a substantial reduction in construction time with increasing ρ. Florida is used as exact indexes (ρ=1) cannot be constructed on larger datasets.



(a) ρ vs. Pre-Processing

(b) ρ vs. Top-k Query Time

Figure 5.6: Effect of ρ on ρ-Approximate NVDs for Florida (# of terms=2,k=10)

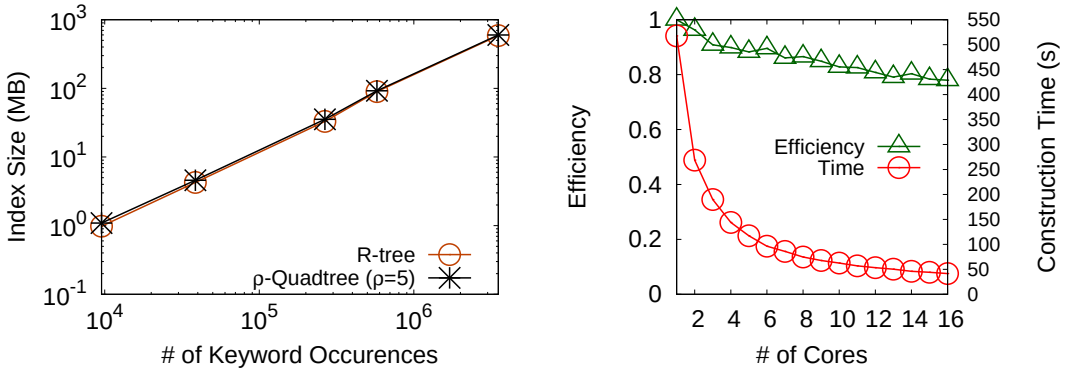
Heap Initialization and Query Penalty Guarantee: The penalty paid for the approximation is during the heap initialization, when a point location query is issued on a ρ-Approximate NVD quadtree to find the cell containing q. Since the cell contains at

most ρ colors, the Heap Generator computes the lower-bound network distances to at most ρ objects (the 1NN is among them) and inserts these in the heap. In the worst-case, when lower-bounds of the $\rho - 1$ objects are smaller than the lower-bound of the 1NN, the algorithm needs to compute network distances to these $\rho - 1$ objects. Thus, the worst-case penalty is $\rho - 1$ network distance computations. However, in practice, these (at most) $\rho - 1$ objects are very likely to be adjacent objects to 1NN of q and thus would normally be evaluated as candidates anyway. This is verified in Figure 5.6(b) as query time does not vary for different ρ.

Space Complexity Theory vs. Practice: Approximate NVDs can alternatively be stored in R-trees. In this case, the leaf nodes of the R-tree contain the Minimum-Bounding Rectangle (MBRs) covering each Voronoi node set. However, R-trees cannot provide the ρ guarantee on the number of 1NN candidates as more than ρ MBRs may overlap and contain the query vertex q. On the other hand, R-trees can provide a worst-case space complexity. If $|inv(t)|$ is the number of objects indexed by the NVD for keyword t, then there will be $|inv(t)|$ such MBRs, so the total space complexity for all keywords is $O(\sum_{t \in W} |inv(t)|)$. In other words, the space cost is linear in the total number of keyword occurrences, which is the space cost of the input keyword dataset.

Figure 5.7(a) shows the comparison of index size for a number of real-world road network datasets (Table 5.2), with the number of keyword occurrences increasing from left to right. As expected, the index size of Approximate NVDs stored in R-trees increases linearly with the number of keyword occurrences. Remarkably, storing in quadtrees also displays comparable and linearly increasing index size. While it remains to be seen whether quadtrees also theoretically take space linear in the number of keyword occurrences (the input), we see that this is true in practice on real datasets. So R-trees provide a worst-case guarantee on index size, while ρ-Approximate NVDs stored in quadtrees provide a guarantee on the number of 1NN candidates. Given the candidate guarantee, slightly faster construction and flexibility offered by ρ, we choose quadtrees in our experiments and represent them as Morton lists [Sam05] which display better locality of reference.

Parallelized NVD Construction: Figure 5.7(b) exhibits significant speed-up using multi-core processing, with NVD construction time reduced by a factor of 12.5 with 16-cores. Efficiency ($\frac{T_1}{p * T_p}$ where T_p is the time for p cores) barely drops below 80%, suggesting



(a) R-trees vs. ρ-Quadtrees (b) Effect of Parallelization

Figure 5.7: ρ-Approximate NVD Indexing for Florida (# of terms=2,k=10)

serial parts of NVD construction are not significant and corroborating Observation 3. We parallelize NVD construction over all available cores in subsequent experiments.

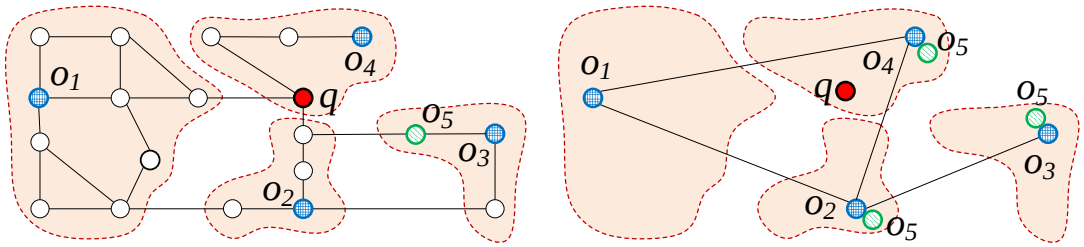
5.6.2 Handling Updates

Our ρ-Approximate NVD index (called APX-NVD hereafter) can handle various types of object and keyword updates. Both insertion and deletion of either objects or keywords are ultimately handled in the same way, i.e., by adding/deleting objects to/from the APX-NVD of the affected keyword(s). For example, to incorporate a new object o with keyword set ψ, o is added to the NVD of each keyword $t \in \psi$. Similarly, adding/deleting a keyword t to/from an existing object o involves adding/deleting o to/from the NVD for t. In this section, we present techniques to support these basic operations efficiently using *lazy updates*. Generally adding/deleting an object involves full or part re-computation of the NVD, which is a relatively expensive operation, e.g., requiring up to 1 second per NVD on the Florida dataset. We delay this re-computation by allowing a certain threshold of lazy updates to the APX-NVD while still supporting exact querying *and* amortizing the re-computation cost over multiple updates.

Object Deletion: Deleting object o from an APX-NVD is handled simply by marking o as deleted. If an extracted object is marked as deleted, the Heap Generator does not return it to the Query Processor. Its adjacent objects are still added to the heap as usual.

Object Insertion: Inserting an object o is more complicated and requires knowing the objects that might be affected by inserting o. We define *affected set* $A(o)$ as the objects whose Voronoi node sets may change when o is added to the NVD.

A previous study [KS04] reported that the affected set of o consists of the 1NN and its adjacent objects. However, we observe that this is not correct. Consider the example of Figure 5.8(a) where a new object o_5 is added. The shaded containers show the Voronoi node sets of the objects *before* o_5 is inserted. The 1NN of o_5 is the object o_3 and the only adjacent object of o_3 is o_2. However, the newly inserted object o_5 will become the 1NN of vertex q, which means the Voronoi node set of o_4 is also affected even though it is not an adjacent object of o_3 (the 1NN of the newly inserted object). In the example, the Voronoi node sets of objects o_2, o_3, and o_4 are affected by the insertion of o_5. We now describe how to determine a correct affected set of inserted object o.



(a) o_2, o_3, and o_4 are affected (b) Adjacency Graph after insertion

Figure 5.8: Updating APX-NVD after inserting o_5

Let $MaxRadius(p)$ of an object p be the maximum network distance between p and a vertex v in its Voronoi node set, i.e., $MaxRadius(p) = \arg\max_{v \in Vns(p)} d(p, v)$. Theorem 3 identifies a condition to construct the affected set.

Theorem 3. *An object p is not in the affected set $A(o)$ of o if $d(o, p) \geq 2 \times MaxRadius(p)$.*

Proof. We prove this by contradiction. Assume there exists a vertex $v \in Vns(p)$ for which o is the new 1NN. Since $d(o, p) \geq 2 \times MaxRadius(p)$ and $d(p, v) \leq MaxRadius(p)$, we have $d(o, p) \geq 2 \times d(p, v)$. Subtracting $d(p, v)$ on both sides gives, $d(o, p) - d(p, v) \geq d(p, v)$. By triangular inequality, $d(o, p) - d(p, v) \leq d(v, o)$. Therefore, $d(v, o) \geq d(p, v)$ and o cannot be the 1NN of v which contradicts the assumption. □

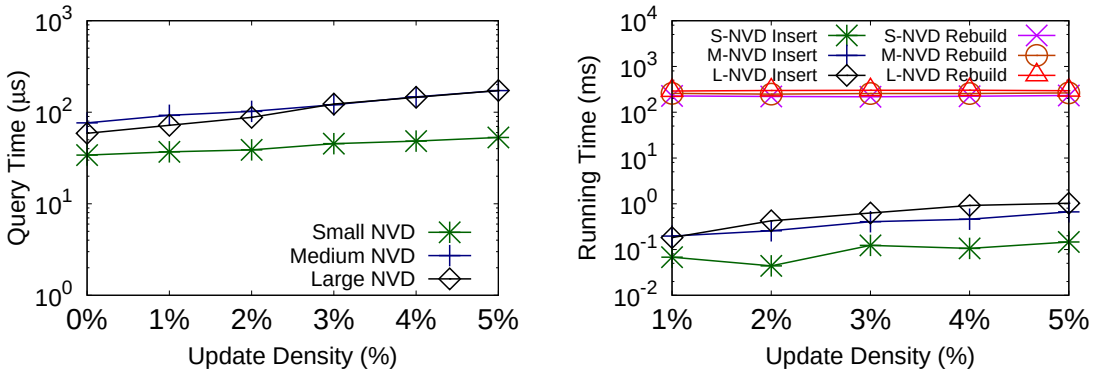
This theorem is used to compute the affected set of o as follows. First, we find the 1NN p of o and initialize the affected set $A(o)$ with p. Then, we conduct a breadth-first search (BFS) on the adjacency graph from p. For any expanded object o_e, if it satisfies the condition in Theorem 3, it is pruned (i.e., the BFS is not expanded from o_e). Otherwise, it is included in the affected set. Note that $A(o)$ may contain some objects that are not affected, but this does not affect correctness.

$MaxRadius(p)$ for every object p can be computed essentially for free during NVD construction. Storing these values incur a small storage overhead linear to the input $O(|inv(t)|)$ (the size of the inverted list for the indexed keyword t). Also $d(o, p)$ can be conveniently computed using the Network Distance Module already available in the K-SPIN framework.

Once the affected set is computed, we perform lazy insertion of object o. Rather than inserting o into the quadtree, we insert o in the adjacency graph. Specifically, we add o to the node of each object in its affected set. For the example of Figure 5.8, we add o_5 to the nodes of o_2, o_3, and o_4 as shown in Figure 5.8(b). The nodes of the adjacency graphs are now assumed to contain one or more objects, e.g., the Heap Generator initializes the heap by inserting the 1NN of q and all the objects stored in the node. In the example, q is located in the Voronoi node set of o_4. Since o_5 was also added to the node of o_4, the heap is initialized with both o_4 and o_5.

Figure 5.9 shows the effect of our proposed techniques to handle updates. Specifically, we chose three keywords distributed in the lower, middle and higher thirds of the frequency distributions and the corresponding APX-NVDs are called large, medium and small, respectively. For each NVD, we insert $x\%$ of the total objects in it using lazy updates and study the effect of lazy updates on the query processing time in Figure 5.9(a). As expected, the processing time has increased but the results are still impressive. In Figure 5.9(b), we report the average time per insertion as well as the total time to rebuild the NVD after the lazy updates. The lazy update cost is only 1ms even when 5% objects are inserted in the large NVD and the cost to rebuild NVDs is under one second. Lazy updates allow the system to continue processing incoming queries while a new APX-NVD may be built in parallel.

Non-NVD Updates: It is possible that an NVD does not exist when inserting an object or adding a keyword to an existing object. This may occur when a keyword is new, or there are fewer than ρ objects in a keyword's inverted list. However, we do not need to construct a new APX-NVD until there are at least ρ objects plus the additional threshold for lazy updates. For deletion, however, if an APX-NVD is no longer required because there are fewer than ρ objects, then NVD updates are unnecessary, and the objects only need to be removed from the inverted list. Handling updates in the road networks (e.g., a new edge, or a deleted edge) is much more complicated and may invalidate NVDs as well



(a) Query Time (b) Update Time

Figure 5.9: Handling Updates on Florida Road Network

as the network distance module. This is a challenge for all existing techniques including shortest path algorithms and requires further research. We remark that such updates occur less frequently.

5.6.3 Improved Pre-Processing Scalability

The ideas proposed in Sections 5.6.1 and 5.6.2 combine in an elegant way to address the pre-processing woes described in Section 5.5. Utilizing ρ-Approximate NVDs instead of exact NVDs reduce the space requirement by more than an order of magnitude. Exploiting the Zipfian nature of keywords to eliminate keyword indexes for a vast majority of them substantially reduces the construction time. For example, the 90GB index size and 3-day build time for exact NVDs for the US road network dataset are reduced dramatically to 584MB and 1.5 hours, respectively. Additionally, updating approximate NVDs for changes to objects is considerably cheaper. Finally, these benefits incur only a small bounded penalty in query performance while still returning exact results.

5.7 Experiment Results

5.7.1 Experimental Setting

Competing Methods: We compare three variants of K-SPIN. All variants use our ρ-Approximate Network Voronoi Diagram (Section 5.6.1) for the Keyword Separated Index and the ALT index [GH05] for the Lower Bounding Module. ALT computes lower-bound network distances between any two vertices using pre-computed distances to "landmark" vertices and the triangle inequality. The variants differ in their Network Distance Module.

KS-CH employs Contraction Hierarchies (CH) [Gei+08], KS-GT uses G-tree [Zho+15] and KS-PHL utilizes Pruned Highway Labeling (PHL) [Aki+14]. Each index offers a different trade-off between pre-processing cost and query performance. Generally speaking, PHL offers fast queries at the expense of high space cost, while CH offers considerably less space cost but relatively slower queries.

K-SPIN answers both top-k and BkNN queries (Section 5.4) and returns *exact* results. We compare with state-of-the-art top-k techniques ROAD [RN12] and G-tree [Zho+15], and BkNN technique FS-FBS [JFW15]. We also adapt G-tree to answer BkNN queries. We exclude network expansion methods as past results [JFW15] (that we verified) showed them to be orders of magnitude slower. Note that G-tree can also answer network distance queries.

We re-used the code for G-tree and ROAD from our implementations for the experiments in Chapter 3. We modified this code for spatial keywords and implemented the query algorithms. The code for PHL and FS-FBS was obtained from the authors. The parameters for the G-tree, FS-FBS and ROAD indexes were chosen as in past studies [Zho+15; Lee+12; JFW15] for best query performance with practicable index construction.

Environment: We conduct experiments on a Linux (64-bit) dedicated Amazon Web Services c4.8xlarge instance with two Intel Xeon E5-2666v3 2.9GHz 10-core CPUs and 60GB DDR4-1866 memory. All code was written in C++ and compiled by g++ v5.4 with O3 flag. Query algorithms use a single thread. All experiments were conducted using memory-resident indexes. This setting is preferred given the high query throughput demands and viable given the affordability of RAM. This was very apparent when a disk-based variant of FS-FBS performed slower than Dijkstra's algorithm using only the input graph in memory [JFW15].

Datasets: We used five real-world road network graphs as listed in Table 5.2. The DE (Delaware), ME (Maine), FL (Florida), E (Eastern United States), and US (United States) datasets were created for the 9th DIMACS Challenge [Pat06] and used widely in recent studies [Zho+15; Wu+12] and in our experiments in Chapters 3 and 4. We extracted points of interest (POIs) and their descriptors from OpenStreetMap (OSM) [Ope]. Each POI was mapped to the closest road network vertex and keywords were extracted from its descriptors. Table 5.2 lists the statistics for the keyword dataset of each road network,

where $|O|$ is the number of object vertices (POIs), $|W|$ is the number of unique keywords, and $|doc(V)|$ is the number of keyword occurrences in all objects.

Region	$\|V\|$	$\|E\|$	$\|O\|$	$\|doc(V)\|$	$\|W\|$
DE	48,812	119,004	2,369	9,539	2,103
ME	187,315	412,352	7,827	38,590	5,289
FL	1,070,376	2,687,902	48,560	265,769	17,628
E	3,598,623	8,708,058	111,085	725,944	33084
US	23,947,347	57,708,624	688,918	3,517,112	106,559

Table 5.2: Real-World Road Network and Keyword Datasets

Query Parameters: We investigate the effect of varying (a) the number of results k, (b) the number query keywords, (c) size of the dataset, and (d) the frequency of the query keywords. Table 5.3 lists the parameter values with defaults in bold. We create a set of query keyword vectors by first choosing several popular search terms including "hotel", "restaurant", "supermarket", "bank", and "school". For each term, we select an object o that contains the keyword. We select further keywords associated with o to create query keyword vectors of length 1 to 6. This ensures that combinations of query keywords are correlated because they exist for a real-world object and is similar to the process used in a recent spatial keyword experimental study [Che+13]. We repeat this until we have selected 10 objects for each of the five terms, generating a total of 50 vectors for each length. Each vector is combined with 100 uniformly selected query vertices for a total of 5,000 queries over which we report the average query time. The query time for K-SPIN includes the creation and maintenance of the on-demand inverted heaps.

Parameter	**Values**
Road Networks	DE, ME, FL, E, **US**
Number of Results (k)	1, 5, **10**, 25, 50
Number of Terms	1, **2**, 3, 4, 5, 6

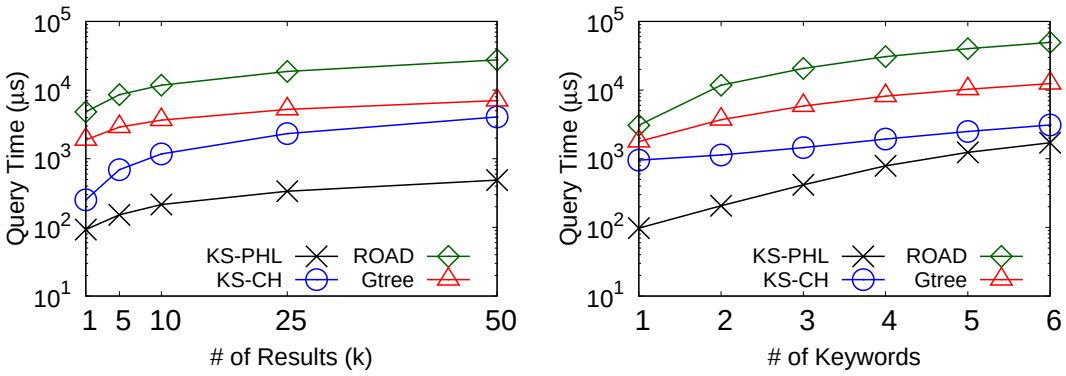
Table 5.3: Spatial Keyword Experimental Parameters (Defaults in Bold)

5.7.2 Query Performance

Top-k Queries

For increasing k (Figure 5.10(a)) and numbers of query keywords (Figure 5.10(b)) on the US dataset, both K-SPIN methods significantly outperform the next best competitor by at least several times on all settings. KS-PHL, in particular performs, up to several orders of

magnitude faster, demonstrating the advantages of K-SPIN's modular nature by allowing the faster network distance technique, PHL, to be used. While the performance gap between KS-CH over G-tree is consistent in Figure 5.10(b), the performance gap between KS-PHL and other methods decreases with additional keywords. PHL is a significantly faster network distance method than CH. As a result, with increasing keywords, the cost of maintaining additional inverted heaps takes a bigger proportion of the query time in KS-PHL. However, in real terms, both KS-PHL and KS-CH are increasing by the same margin despite appearing otherwise due to the logarithmic scale, thus KS-PHL will never "catch-up" to KS-CH. Note that all K-SPIN query times in all experiments include the cost of lazy heap initialization and maintenance (described in Algorithm 9 and in Section 5.6).



(a) Varying k

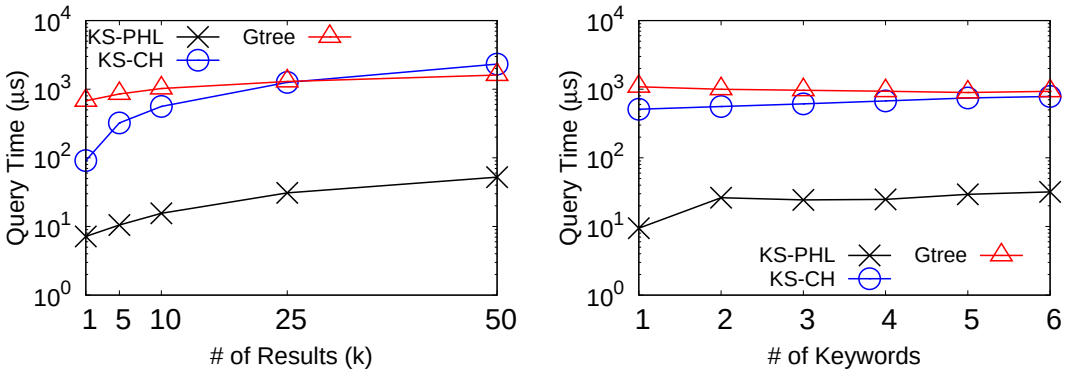
(b) Varying # of Keywords

Figure 5.10: Top-k Queries (US,# of terms=2,k=10)

Boolean *k*NN Queries

Boolean *k*NN Disjunctive Queries: Figure 5.11 shows the query performance for disjunctive BkNN queries. KS-PHL again significantly outperforms the other techniques irrespective of k or the number of keywords. Interestingly, KS-CH does not improve over G-tree as significantly as for top-k queries in some cases, e.g., for $k = 50$ in Figure 5.11(a). The reason for this is two-fold. First, disjunctive queries are easier to answer, in a sense, than top-k and conjunctive queries because the criteria only require an object to have any single query keyword. Thus, in general, we can expect result objects to be found closer to the query location, which means they appear in nearby G-tree nodes that are less costly to evaluate. This also explains why G-tree improves marginally with increasing

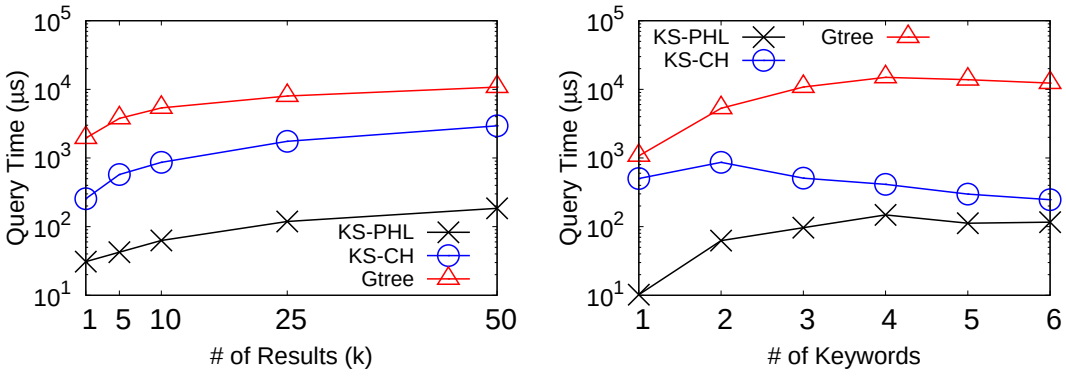
query keywords (objects are easier to match). Second, G-tree is able to re-use intermediate network distance computations, hence scales efficiently for increasing k because many of them can be re-used. However, we note that a similar strategy has been applied to Dijkstra-based hierarchical methods in the past [Kno+07] by saving and re-using the forward search between network distance computations which may also be applied to CH. Nonetheless, we note that KS-CH uses less memory than G-tree but is still able to match or beat its performance on disjunctive queries without this improvement. FS-FBS is not shown, as it cannot be built for the large US dataset.



(a) Varying k (b) Varying # of Keywords

Figure 5.11: Disjunctive BkNN (US,# of terms=2,k=10)

Boolean kNN Conjunctive Queries: Figure 5.12 depicts performance on conjunctive BkNN queries. The advantage of K-SPIN methods over G-tree is even more pronounced than disjunctive queries, e.g., with several times to orders of magnitude improvement for varying k in Figure 5.12(a). Keyword aggregation used by G-tree is more susceptible to false positives for *conjunctive* queries as the hierarchy must be evaluated deeper before false positives can be identified. K-SPIN, on the other hand, can quickly eliminate objects not satisfying the criteria, avoiding computation of expensive network distances. We also see that increasing query keywords results in improving query times for K-SPIN methods in Figure 5.12(b). With additional keywords, fewer objects match the conjunctive criteria. Consequently, the least frequent keyword is more likely to have an even lower frequency. This gives K-SPIN an advantage as it has to consider fewer candidates (i.e., only those that contain the least frequent keyword), explaining the observed improvement.

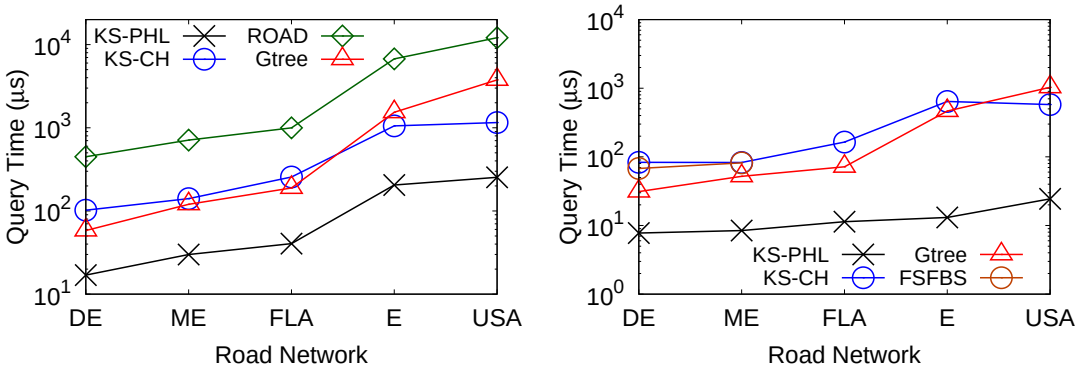


(a) Varying k (b) Varying # of Keywords

Figure 5.12: Conjunctive BkNN (US,# of terms=2,k=10)

Varying Road Network

Figures 5.13(a) and 5.13(b) depicts the query time of each technique for top-k queries and disjunctive BkNN queries, respectively, for varying road network size. The number of vertices in the network increases from left to right. First, KS-PHL significantly outperforms the other techniques on all datasets for both types of queries. Second, we generally see that the performance improvement of K-SPIN techniques over competing methods increases with dataset size. This shows keyword separation scales better with dataset size as the occurrence of false positives is reduced. This can be explained by the fact that, in a bigger graph, higher levels of the G-tree and ROAD hierarchies aggregate more keyword occurrences. This results in degraded pruning power and hence more false positives and redundant network distance computations in G-tree and ROAD.



(a) Top-k (b) BkNN (Disjunctive)

Figure 5.13: Varying Road Network (# of terms=2,k=10)

Varying Keyword Frequency

Figure 5.14 illustrates the effect of increasing keyword frequency. We express frequency in terms of keyword object density $|inv(t)|/|V|$, where $|inv(t)|$ is the number of objects which contain keyword t and $|V|$ is the total number of vertices in the road network. Each tic on the x-axis represents a "bucket" of keywords in the density range greater than or equal to the current tic but less than the next tic (the last tick includes keywords of all densities larger than 0.01). We execute single-keyword BkNN queries to isolate the impact of frequency. Once again, K-SPIN outperforms G-tree, with KS-PHL more than an order of magnitude faster. KS-CH improvement over G-tree is smaller as only a single query keyword is involved, allowing G-tree to avoid the false positive problems seen earlier with the more realistic multi-keyword disjunctive BkNN and top-k queries.

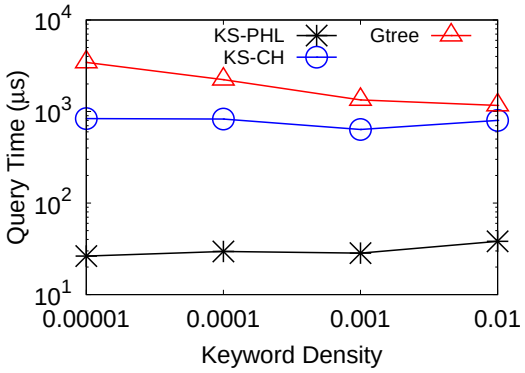


Figure 5.14: Varying Frequency BkNN (US,# of terms=1,k=10)

5.7.3 Index Performance

Figure 5.15 shows the size of each index. "Input" is the input graph and keyword dataset. Contraction Hierarchies entails the smallest footprint out of all indexes at 2.6GB compared to 2.8GB for G-tree on the largest dataset (US). KS-PHL entails an index size of 17.9GB compared to 4.5GB for ROAD for the US. The FS-FBS index could only be constructed for the two smallest datasets. The 2-hop labeling index used to build the FS-FBS index requires a vertex order. As described in the original study [JFW15], we tested several vertex orders generated by Contraction Hierarchies [Gei+08] (including reverse order), but could not build an index for FL in less than 24-hours, not to mention the prohibitive scaling of index size. Unlike K-SPIN, FS-FBS does not provide an easy way to replace the road network index used. Apart from FS-FBS, the pre-processing time of each technique

in Figure 5.16 is comparable. K-SPIN received a useful speed-up from parallelization (Section 5.6.1), while other techniques cannot be as easily parallelized.

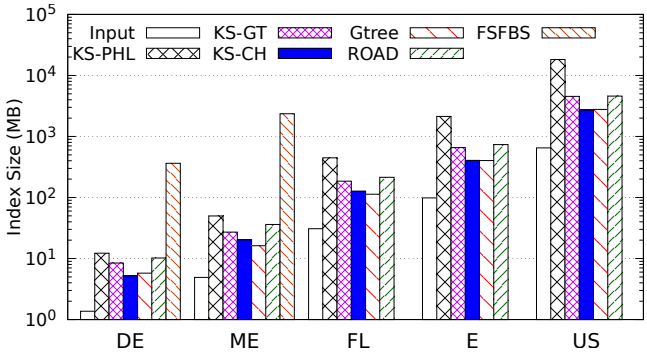


Figure 5.15: Index Size

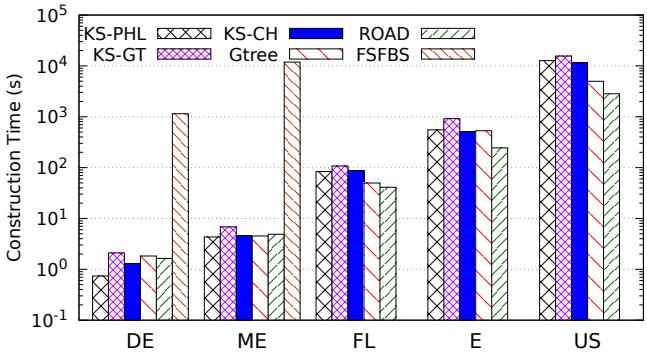


Figure 5.16: Pre-Processing Time

5.7.4 Heuristic False Positive Performance

In Section 5.1, we presented examples of how existing spatial keyword algorithms that use keyword aggregation, like G-tree, incur costly additional work due to false positive candidates. Thus far, we have shown empirical evidence that shows K-SPIN outperforms its keyword aggregation counterparts, often quite significantly. To give further credence to our claim that K-SPIN does indeed reduce the occurrence of false positives, we present a deep-dive experimental comparison using the G-tree index.

We use the G-tree index as the network distance module in K-SPIN (referred to as KS-GT). So KS-GT and G-tree's spatial keyword query algorithms use the same underlying road network index (G-tree) to compute network distances. This occurs in exactly the same manner, e.g., already computed partial network distances are re-used for later computations, described as *materialization* by Zhong *et al.* [Zho+15]. As a result, we perform

an apples-to-apples comparison and thus obtain a truer understanding of the reduction in false positives achieved by K-SPIN. Before presenting our findings, we first describe how to apply keyword separation principles to G-tree's spatial keyword query algorithms using.

Applying Keyword Separation Principles to G-tree

Given the disadvantages of keyword aggregation and advantages of keyword separation described so far, two pertinent follow-up questions may arise: (1) can principles of keyword separation be applied to existing spatial keyword algorithms and (2) does applying such principles mitigate the drawbacks of keyword aggregation discussed earlier. We answer (2) in the subsequent section (the short answer being "no"), but first, describe how an answer to (1) can be implemented for the top-k query algorithm proposed by Zhong *et al.* [Zho+15] using the G-tree index.

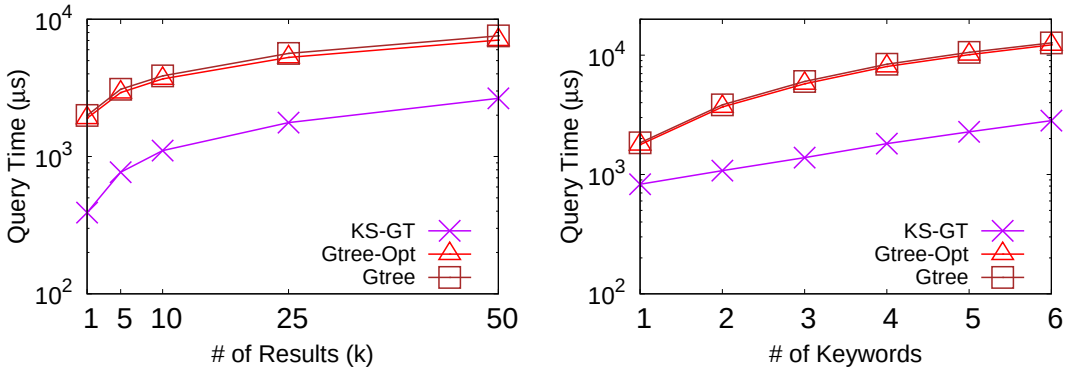
As described in Section 3.3.5, G-tree is a tree data structure where each tree node represents a road network subgraph. Starting with the entire road network as the root, each child node is a partitioning of the parent node's subgraph as illustrated earlier in Figure 3.3. G-tree's top-k algorithm finds candidates by traversing this subgraph hierarchy up from the leaf node containing the query and down towards other leaf nodes containing objects. Each tree node is associated with an *occurrence list*, which lists all child nodes with an object. Occurrence lists can then be used to avoid searching G-tree nodes (i.e., subgraphs) without objects. In the case of top-k queries, the set of objects consists of all vertices that contain a keyword. Thus, each tree node's occurrence list indicates which child nodes contain an object with *any* keyword. Note that this is in addition to the pseudo-document associated with each tree node, which contains all the keywords present in the subgraph, as usual for the keyword aggregation approach. By using an occurrence list, child nodes without objects can be pruned immediately without consulting their pseudo-documents.

We observe that keyword separation principles can be applied to occurrence lists. Rather than building one occurrence list for a tree node, build a separate occurrence list for each keyword in the tree node's pseudo-document. Now G-tree's top-k algorithm can be modified to prune child nodes that do not contain objects with any of the query

keywords. Note that we use this optimized version of G-tree spatial keyword algorithms in all previous experiments.

Query Time and Network Distance Cost

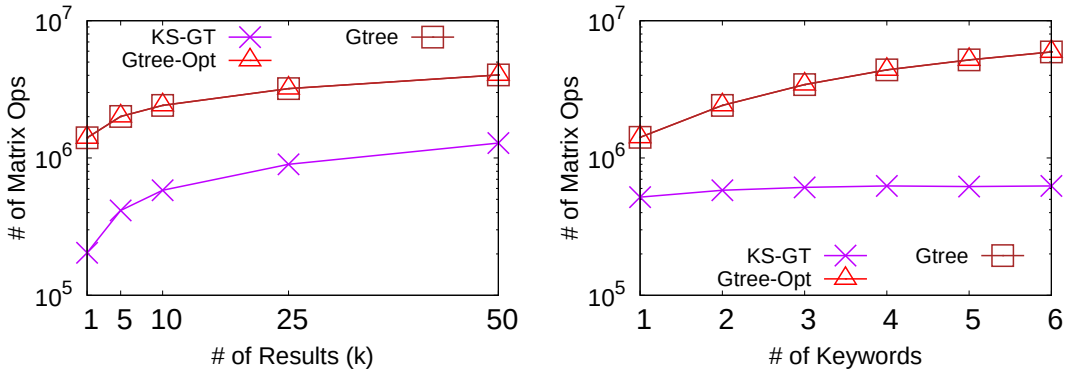
Figure 5.17 displays the query time of KS-GT, optimized G-tree described above (Gtree-Opt), and G-tree's original top-k query. Additionally, we compare methods in terms of *matrix operations* in Figure 5.18. Computing network distance using G-tree involves determining the tree path between source and destination vertices in the G-tree hierarchy. Given this path, distances are computed to each border associated with a tree node on the path by looking up and summing distance matrix elements (described briefly in Section 3.3.5 and in detail by Zhong *et al.* [Zho+15]). We term this look-up and sum as a machine-independent *matrix operation* that accurately captures how costly the network distance was to compute. Most importantly, if fewer false positives occur there will be fewer matrix operations.



(a) Varying k (b) Varying # of Keywords

Figure 5.17: Top-k Query Time (US,# of terms=2,k=10)

In Figure 5.17, we see that Gtree-Opt marginally improves on the original G-tree top-k query algorithm in terms of query time. However, in Figure 5.18 we see little to no improvement in terms of matrix operations. This suggests that the query time improvement is entirely from avoiding pseudo-document look-ups rather than incurring fewer false positives. Identical numbers of matrix operations show that the hierarchy is still being evaluated to the same depth to overcome the effect of aggregation. These observations strongly evince that problems arising from keyword aggregation cannot be easily solved in existing techniques.



(a) Varying k (b) Varying # of Keywords

Figure 5.18: Top-k Matrix Operations (US,# of terms=2,k=10)

In Figure 5.17, KS-GT consistently outperforms G-tree by up to an order of magnitude in terms of query time. This is despite KS-GT query time including extra overheads, e.g., computing lower-bounds and initializing/maintaining inverted heaps. The even greater improvement on matrix operations in Figure 5.18 removes any doubt. The improvement in matrix operations directly shows that K-SPIN utilizes the G-tree index more efficiently, i.e., due to fewer false positives. We cannot apply further keyword separation to G-tree itself due to the permanent loss of discriminating information without reversing the keyword aggregation itself. K-SPIN in fact achieves this, but in a simple and versatile manner.

5.8 Summary

Keyword separation is a viable alternative to keyword aggregation, as evident in the significant improvement in query performance of K-SPIN over competing methods. Furthermore, we show that this does not need to come at a prohibitive pre-processing cost, as shown by the substantial reduction in keyword index size and pre-processing time. In fact, utilizing the long-tail of Zipfian distributions and ρ-Approximate NVDs are useful techniques on their own. Ultimately, K-SPIN provides an efficient and versatile framework for spatial keyword query processing, in addition to the provision for dynamic updates and parallelized index building. Moreover, these achievements further advocate the usefulness of decoupled heuristics, particularly given the huge improvement in utilization of the G-tree road network index by K-SPIN compared to G-tree's own more complex dedicated heuristics.

Chapter 6

Efficient Hierarchical Traversal for Aggregate kNN Queries

> The whole is greater than the sum of its parts.
>
> Aristotle

The queries we have focused on so far involved a single query vertex. However, other POI search problems such as Aggregate kNN (AkNN) queries involve multiple query locations. As a result, the intuitions behind our highly successful heuristics for earlier queries are not necessarily applicable to AkNN queries. As we discuss next, we propose techniques to efficiently answer AkNN queries based on a more appropriate intuition for such queries. Moreover, in the context of the decoupled heuristic paradigm, we show that it is capable of incorporating more sophisticated heuristics to solve varying POI search problems.

6.1 Overview

In the age of ubiquitous mobile computing, finding the nearest relevant points-of-interest (POIs) through the road network is an increasingly important problem. Compared to retrieving POIs in Euclidean space (i.e., "as the crow flies"), using the road network distance computed by graph algorithms enables more accurate metrics for proximity, such as travel time. This is highly useful in popular map-based services like ride-sharing apps, where inaccuracy equals lost profit.

The key to POI search in road networks is in developing heuristics to retrieve POIs that are most promising. The vast majority of related work has focused on finding POIs

by their network distance from a single user's location, such as k Nearest Neighbor (kNN) queries studied in Chapters 3 and 4, which retrieve the k POIs with minimum road network distances. However, POI search is not necessarily limited to this scenario. For example, Aggregate kNN queries [YMP05] retrieve POIs by an aggregate network distance from multiple users' locations.

Heuristics to answer AkNN queries have largely been borrowed from kNN techniques [YMP05; Zhu+10; Yao+18], which is problematic in a number of ways. First, intuitions to find kNNs do not necessarily translate to AkNNs. For example, our highly efficient kNN heuristic in Chapter 4 based on NVDs uses a recurrence rule stating that the k-th nearest POI must be adjacent to the $k-1$ nearest POIs. The rule is exploited by storing the 1st nearest POI of every query location and the adjacency relationships between POIs. However, AkNN queries involve retrieving POIs by an aggregate distance that summarizes network distances from multiple query locations according to some function. Thus, the 1st nearest POI is unlikely to be close to all query locations, so the recurrence rule is no longer true and cannot be applied, rendering the heuristic unsuitable.

A more appropriate heuristic to find AkNNs is to conduct a hierarchical search on the road network. Yiu *et al.* [YMP05] search an R-tree [Gut84] containing the POIs in a top-down manner according to a Euclidean distance heuristic, similar to the IER kNN technique [Pap+03]. When constructed, the R-tree recursively divides POIs into subsets by Minimum Bounding Rectangles (MBRs). During the search, a lower-bound aggregate distance for all POIs in a child R-tree node is computed using the Euclidean distances to its MBR. Now the most promising tree branches can be visited to pinpoint result POIs. However, this is not ideal for road networks as Euclidean distance is only a loose lower-bound especially on metrics like travel time, making the heuristic less efficient. Moreover, the inefficiency is exacerbated for AkNNs as the error will also be aggregated.

Landmark Lower-Bounds (LLBs) are a more accurate alternative to Euclidean distance [GH05]. They involve pre-computing and storing certain distances to *landmarks* in the road network, which can then be used to compute a lower-bound between any two locations using the triangle inequality. However, there is no hierarchical data structure to compute minimum LLBs to groups of POIs in the same way as R-trees using Euclidean distance.

While these diverse factors are challenging, in this chapter, we propose techniques to overcome them simultaneously. First, we present hierarchical data structures, SL-Trees and COLT (Section 6.3). These add significantly more landmarks than previous methods, while still keeping the data structures reasonably small in theory (Section 6.5) and practice. COLT, the main index used for querying, is particularly light-weight. It is also the first index to support hierarchical traversal of the road network to locate POIs by landmark lower-bounds. Utilizing COLT and part of the SL-Tree, we propose a heuristic search algorithm to efficiently answer AkNN queries (Section 6.4). Our algorithm is particularly adept at AkNN queries due to a unique property of COLT for convexity-preserving aggregate functions. Finally, our detailed experimental investigation on real-world datasets shows our techniques achieve orders of magnitude improvement in query time over competing methods (Section 6.6). We also verify the improvement in terms of machine independent heuristic efficiency.

6.2 Preliminaries

Road Network: Similar to previous chapters, we use the same definition of a road network as outlined in Section 1.1 and consider query locations and objects (POIs) to be located on road network vertices for simpler exposition.

Aggregate k Nearest Neighbor (AkNN) Queries: Given a set of query vertices $Q \in V$, a set of object vertices $O \in V$, and an aggregation function agg, an AkNN query retrieves the k objects in O with minimum aggregate distances from set Q. Aggregate distance $d_{agg}(Q, o)$ to object o is computed by aggregating the network distances to o from each query vertex $q \in Q$ by function agg (e.g., their sum). These are also known as Group kNN queries [Pap+04].

Landmark Lower-Bounds (LLBs): We refer the reader to Section 4.2 for a description of Landmark Lower-Bounds and the ALT index that can be used to produce them.

6.2.1 Lower-Bound Aggregate Distances

Yiu *et al.* [YMP05] described how to compute a lower-bound on the exact aggregate distance for object o using lower-bound network distances to o from each query vertex $q_i \in Q$, as in Lemma 4 below.

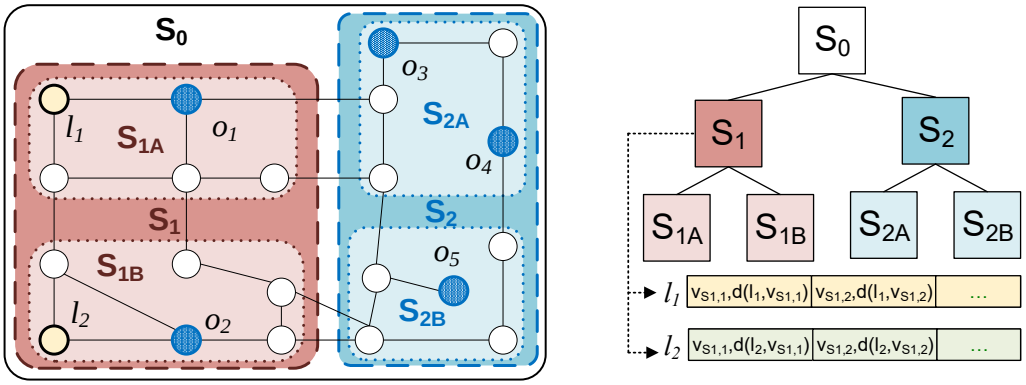
Lemma 4. *Given a monotone aggregate function agg and lower-bound distances $LB(q_i, o)$ from each query vertex $q_i \in Q$ to object o, the aggregation of the lower-bound distances gives a lower-bound on the exact aggregate distance. That is, $LB_{agg}(Q, o) = agg(LB(q_i, o), \cdots, LB(q_{|Q|}, o)) \leq d_{agg}(Q, o)$.*

Lemma 4 was shown to be true by Yiu *et al.* [YMP05] due to the monotonicity of the aggregate function. Notably, their technique used Euclidean distance as the example lower-bound on network distance similar to IER.

6.3 Data Structures

We describe our data structures to index the road network and object set, respectively, to be utilized for hierarchical graph traversal and ultimately locate AkNN objects.

Road Network Index: We first introduce the Subgraph-Landmark Tree (SL-Tree) to index the road network. The SL-Tree is a supporting index that we use to construct our object index efficiently. Each node in the SL-Tree represents a subgraph of the road network with the root being G. G is recursively partitioned into b disjoint subgraphs of equal size, stopping when a subgraph has no more than α vertices. Figure 6.1(a) shows such a partitioning for $b = 2$ and $\alpha = 6$ with the corresponding SL-tree shown in Figure 6.1(b). Note that we refer to tree *nodes* and road network *vertices*.



(a) SL-Tree Subgraph Partitions (b) SL-Tree Node Hierarchy

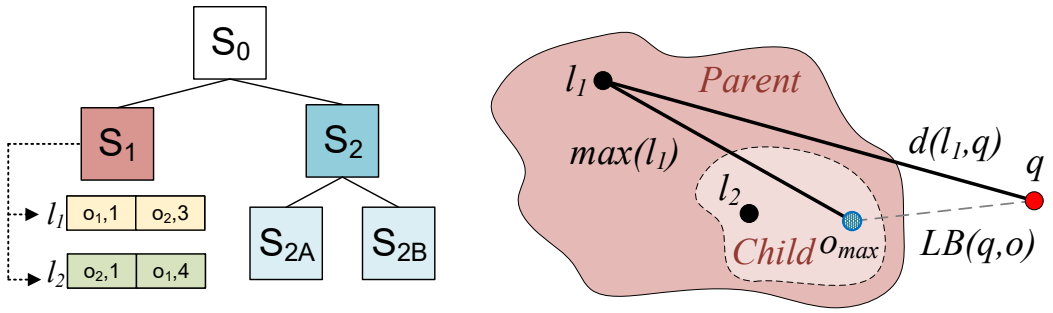
Figure 6.1: Subgraph Landmark Tree (SL-Tree)

For each node n_T, we select m of its vertices as *local landmarks*, e.g., l_1 and l_2 for $m = 2$ for S_1 in Figure 6.1(a) (landmarks for other nodes are omitted for clarity). We then compute distances from each landmark l_i to every vertex in n_T's subgraph using Dijkstra's

search, which are stored in *distance list* DL_i. Figure 6.1(b) shows the distance lists for the landmarks of S_1 (those for other nodes are again omitted). Note that subgraphs are stored implicitly by mapping road network vertices to SL-Tree leaf nodes, and any partitioning scheme can be used, e.g., [KK98].

Object Index: The SL-Tree can then be used to efficiently construct our object index, the Compacted Object-Landmark Tree (COLT). COLT is a carefully compacted version of the SL-Tree for object set O. Compaction ensures that there are m local landmarks for every λ objects, to increase the likelihood of finding a tighter lower-bound for more objects. Note that the SL-Tree is shared between the construction of all COLT indexes, i.e., for many different object sets.

Given SL-Tree T, COLT index C is constructed by visiting nodes in T in a top-down manner and creating corresponding nodes in C. Let n_T be the currently visited node in T (initially the root). A corresponding node n_C is created in C for n_T. Let $\lambda \geq \alpha$ be the maximum number of objects in a leaf node for COLT. If n_T contains more than λ objects, the search expands to its children. Otherwise, the search is stopped at n_C, which becomes a leaf-node of COLT. For the new leaf n_C, an *Object Distance List* ODL_i is created in n_C for each landmark l_i in n_T. These are simply the distance lists of n_T, except with only the distances for object vertices in O. Any interior nodes with only one child are merged with the child (keeping the child's landmarks, which are more local). Figure 6.2(a) shows a COLT index for $\lambda = 2$ constructed from the SL-tree in Figure 6.1(b) based on the 5 object vertices in Figure 6.1(a). Note that S_{1A} and S_{1B} were removed as the search was pruned at S_1 due to its number of objects (the ODLs of other nodes are not shown for clarity).



(a) COLT Hierarchy (b) COLT Lower-Bounds

Figure 6.2: COLT Index

Each object distance list ODL_i of leaf node n_{leaf} in C is sorted on distance. In non-leaf nodes n_C, we only store the minimum distance min_{n_C,l_i} and maximum distance max_{n_C,l_i}) to any object in the node from each of its landmarks l_i. These can be computed using distance lists in corresponding SL-Tree nodes. Next, we use this information to compute lower-bounds to nodes and traverse the hierarchy.

6.3.1 Lower-Bound Heuristic for Graph Traversal

Mouratidis *et al.* [Mou+15] proposed computing a lower-bound distance for a group of vertices in a social graph using landmarks similar to the use of the triangle inequality in (4.1) from Section 4.2.1. A similar idea can be considered to compute a lower-bound for all objects contained within a node n_C in COLT index C using (6.1) for one landmark $l_i \in n_C$. (6.2) gives the best lower-bound over all m landmarks of n_C.

$$LB_{l_i}(n_C, q) = \begin{cases} d(l_i, q) - max_{n_C,l_i} & \text{if } d(l_i, q) \geq max_{n_C,l_i} \\ min_{n_C,l_i} - d(l_i, q) & \text{if } d(l_i, q) \leq min_{n_C,l_i} \\ 0 & \text{else} \end{cases} \tag{6.1}$$

$$LB_{max}(n_C, q) = \max_{l_i \in n_C}(LB_{l_i}(n_C, q)) \tag{6.2}$$

min_{n_C,l_i} and max_{n_C,l_i} for n_C are already available in COLT. However, for non-root nodes, $d(l_i, q)$ in (6.1) is only available if l_i and q are in the same subgraph. Pre-computing this distance for all V and landmarks is infeasible given the space implications. The COLT index must be kept compact, because unlike the single set of users indexed by Mouratidis *et al.* [Mou+15], there are potentially thousands of object sets each requiring a COLT index. Alternatively, computing $d(l_i, q)$ on the fly using another technique is expensive and may be wasteful if the node does not contain results. Interestingly, (6.1) still holds if we replace the distances with lower-bound $LB(l_i, q)$ and upper bound $UB(l_i, q)$, as in (6.3). The distance lists of the root or lowest common ancestor SL-Tree node can be conveniently used to compute $LB(l_i, q)$ and $UB(l_i, q)$ by (4.1) and its upper-bound equivalent (by adding rather than subtracting distances), respectively. Choosing the tightest over all landmarks of n_C gives an inexpensive and accurate bound, as previously mentioned, for

even a small number of landmarks.

$$LB_{l_i}(n_C, q) = \begin{cases} LB(l_i, q) - max_{n_C, l_i} & \text{if } LB(l_i, q) \geq max_{n_C, l_i} \\ min_{n_C, l_i} - UB(l_i, q) & \text{if } UB(l_i, q) \leq min_{n_C, l_i} \\ 0 & \text{else} \end{cases} \tag{6.3}$$

Landmark Choices: Additional landmarks closer to the objects naturally improves the tightness of produced lower-bounds. Past work has shown that landmarks on the fringe of the graph also give better lower-bounds [GW05]. Inspired by this, we choose landmarks from border vertices of subgraphs. We partition the borders of the SL-Tree node into m slices around the Euclidean center of the subgraph. We choose 1 border as a landmark from each slice. If a slice has no borders, we choose the slice vertex furthest from the Euclidean center. If a slice has no vertices, we randomly choose a subgraph vertex to ensure m landmarks.

Effective Lower-Bounds: The accuracy of COLT's inexpensive lower-bounds increases as we delve deeper into the hierarchy. In Figure 6.2(a), let us say we use l_1 and its maximum object distance to compute a lower-bound for the child node. At the lower level, we may use the child's landmarks like l_2, which are local to the objects and more likely to produce a better lower-bound. This lets us differentiate tree branches and pinpoint the most promising candidates. Next, we utilize this as a decoupled heuristic for AkNN querying.

6.3.2 Extending to Moving Objects

We now briefly describe how COLT can incorporate updates for moving objects, such as taxis, as they move from vertex to vertex in the road network. The simple approach is to reconstruct the whole COLT index, which we show is still fast both in theory (Section 6.5) and practice (Section 6.6). However, COLT can be updated more efficiently through a careful approach. Given an object p that has moved, its distance in the ODLs of its leaf node will no longer be accurate. These values can be updated using the distance lists in the equivalent SL-Tree node with a single $O(1)$ lookup for each ODL. Since the elements corresponding to p may now be out of order in the ODLs, these can be sorted by simply finding the element's correct position in worst-case $O(\lambda)$ time for each ODL. Lastly, the

maximum and minimum values of each ancestor node from the leaf node in COLT containing p must also be updated. Doing so may involve iterating over the landmark distances to every object contained within each ancestor node containing p. However, this can be avoided by also storing the objects associated with the minimum and maximum distances for each landmark (achievable for free during pre-processing). Given a landmark l for an ancestor node n_A of p, if $d(l,p)$ is smaller than the minimum distance $min_{n_A,l}$ (respectively greater than the maximum distance $max_{n_A,l}$), then $min_{n_A,l}$ (respectively $max_{n_A,l}$) can safely be updated. If $d(l,p)$ is greater than $min_{n_A,l}$ and smaller than $max_{n_A,l}$), no update is necessary if p is not associated with $min_{n_A,l}$ or $max_{n_A,l}$. Otherwise, $min_{n_A,l}$ (respectively $max_{n_A,l}$) must be re-computed by iterating over all objects contained in node n_A. This process is repeated for each non-root ancestor node n_A of object p.

6.4 Query Algorithm for AkNN Search

We have already seen the effectiveness of decoupled heuristics on POI search in previous chapters. Accordingly, we develop a decoupled heuristic that traverses COLT in a hierarchical manner to pinpoint and retrieve the best AkNN candidate objects by their lower-bound aggregate distances. Interestingly, COLT considers fewer AkNN candidates to find the object with the minimum lower-bound using a novel property of COLT's Object Distance Lists (ODLs).

6.4.1 Object Distance Lists and Convexity

From Section 4.2.1, (4.1) can be expressed as an absolute value function of form $f(x) = |C - x|$ for some landmark l. Here, C is the constant distance $d(l,q)$ between the landmark l and the query point q, and x is a variable distance $d(l,o)$ depending on the object $o \in O$. Since absolute-value functions are convex, $f(x)$ is minimized for x closest to C. This property is useful to find the minimum lower-bound in an Object Distance List ODL for the landmark l for a single query vertex q. Since ODL essentially stores the domain of x for all objects in the node, the minimum lower-bound for the landmark l can be found by searching ODL for $d(l,c)$ closest to $d(l,q)$ for some object $c \in ODL$. Since ODL is sorted, this is possible using a modified binary search, similar to Object Lists in Section 4.3.1, in only $O(\log \lambda)$ time.

Finding the minimum lower-bound *aggregate* distance is complicated by the presence of multiple query locations $q_i \in Q$. As described in Lemma 4 in Section 6.2.1, the aggregation of *lower-bound* distances from each query vertex q_i is a lower-bound on the aggregation of exact network distances for monotonic functions [YMP05]. Therefore, the function to minimize becomes $f(x) = agg(|C_1 - x|, \cdots, |C_n - x|)$ for a monotonic aggregate function agg where C_i is $d(l, q_i)$ for the given landmark l and a query $q_i \in Q$. At first glance, this might suggest we need to search ODL for multiple values (i.e., once for each C_i) to find the object with minimum aggregate lower-bound. Surprisingly, it is not necessary for aggregate functions that preserve convexity. Moreover, we find that the most widely used functions [Pap+05], *max* and *sum*, *do* preserve convexity. [BV04] prove convexity preservation for a range of functions.

Specifically, once the minima x^* of the function $f(x)$ is found, iteratively retrieving the object that gives the next smallest lower-bound simply requires, due to the convexity of the function, checking the element to the right or left of x^* in ODL. However, unlike the single query case, finding the minima of $f(x)$ is not obvious for aggregate kNN queries. Below, we show how to find minima for two common aggregate functions, *max* and *sum*.

Lemma 5. *Consider the aggregate function defined by the sum of a set of absolute functions* $f(x) = sum(|C_1 - x|, \cdots, |C_n - x|)$. *The minima* x^* *of* $f(x)$ *is the median value of the constants* $C_1, ..., C_n$.

Proof. Let constants be sorted such that $C_1 \leq C_2 \leq ... \leq C_n$. Let x^* be the median of these constants. We show that $f(x^*) \leq f(x')$ for all x'. Let $d = |x^* - x'|$. Without loss of generality, assume $x' < x^*$. For each $C_k < x^*$, the difference between $|C_k - x^*|$ and $|C_k - x'|$ is at most d, i.e., $|C_k - x^*| - |C_k - x'| \leq d$. On the other hand, for each $C_j \geq x^*$, the difference between $|C_j - x'|$ and $|C_j - x^*|$ is exactly d, i.e., $|C_j - x'| - |C_j - x^*| = d$. Since x^* is median of the constants, the number of constants $C_j \geq x^*$ is at least $\lceil \frac{n}{2} \rceil$. In other words, for at least half of the constants, $|C_i - x'| - |C_i - x^*| = d$ and for each of the remaining constants, $|C_i - x^*| - |C_i - x'| \leq d$. Thus, $f(x^*) \leq f(x')$. □

Lemma 6. *Consider the aggregate function defined by the maximum of a set of absolute functions* $f(x) = max(|C_1 - x|, \cdots, |C_n - x|)$. *The minima* x^* *of* $f(x)$ *is* $\frac{C_{min}+C_{max}}{2}$, *i.e., the average of the minimum and maximum constants.*

Proof. For sorted constants, let $C_1 = C_{min}$ and $C_n = C_{max}$. Note that $f(x) = max(|C_1 - x|, \cdots, |C_n - x|) = max(|C_1 - x|, |C_n - x|)$. Thus, the minimum value of $f(x)$ is $\frac{C_1+C_n}{2}$, i.e., minima x^* of $f(x)$ is $\frac{C_{min}+C_{max}}{2}$. □

Of the commonly used aggregate functions in previous studies, only the *min* function does not preserve convexity. Since the resulting function is not convex, we are not able to find a minimizing value in $ODLs$ in the same manner as for the *sum* and *max* functions. However, to solve AkNN queries for the *min* function, we may simply use non-aggregate kNN queries for each query vertex in Q, and then merge the results with minimum distances to any query vertex. In Section 7.2.3, we provide some preliminary experimental results for kNN queries utilizing COLT compared to existing kNN techniques, which can be used to infer AkNN query performance for the *min* function.

6.4.2 Query Processing

Algorithm 10 uses hierarchical graph traversal on the COLT index to guide us towards ODLs most likely to contain AkNN results. The algorithm maintains a priority queue $\mathcal{PQ}$ containing objects and COLT nodes keyed by their aggregate lower-bound distances from Q. The loop iteratively extracts the minimum lower-bound queue element. If an object is extracted, its exact aggregate distance is computed and the result set R and D_k, the distance to the k-th nearest candidate, is updated (lines 7-10). If a non-leaf node is extracted then an aggregate lower-bound score is computed according to (6.3) and (6.2) for each of its child nodes (lines 12-14). If it is a leaf-node, it is initialized if encountered for the first time (lines 16-21). A landmark is chosen to determine the object list to process, the constants in the absolute value functions are computed, and a binary search is performed to find the minimizing list index given by Lemma 5 or 6. Pointers RP and LP are initialized to the index of minimizing value. Then objects are retrieved from the list using RetrieveObjectsOL and, if the list is not completely searched, the node is re-inserted into $\mathcal{PQ}$ with minimum lower-bound computed by RP or LP. We set $c.l$ (line 17) to the landmark furthest from the query vertices (on average) by a lower-bound computed using the SL-Tree and (4.1), to obtain tighter lower-bounds similar to fringe landmarks in Section 6.3.1. Line 29 computes the maximum (best) lower-bound aggregate distance for object p using available network distances for any landmark (e.g., landmarks in the root SL-Tree node or $c.l$ in the leaf).

Algorithm 10 Answer AkNN queries using hierarchical travel of COLT index

1: **function** GetAkNNsByCOLT($k, Q, agg, COLT, SLTree$)
2: $\mathcal{PQ} \leftarrow \phi$ ▷ Priority queue keyed by lower-bound distance
3: $R \leftarrow \phi$ ▷ Max priority queue containing k best candidates
4: Insert($\mathcal{PQ}, COLT.root, 0$) , $D_k \leftarrow \infty$
5: **while** MinKey($\mathcal{PQ}$) $< D_k$ and $\mathcal{PQ}$ not empty **do**
6: $c \leftarrow$ Extract-Min($\mathcal{PQ}$)
7: **if** c is an object **then**
8: Compute agg. dist $d_{agg}(Q, c)$ using $d(q_i, c) \forall q_i \in Q$
9: **if** $d_{agg}(Q, c) < D_k$ **then**
10: Insert($R, c, d_{agg}(Q, c)$) ▷ Check/update R and D_k
11: **else if** c is a non-leaf node, i.e., with no ODL **then**
12: **for each** child node e of c **do**
13: Compute lower-bound $LB_{agg}(Q, e)$ for e by (6.2)
14: Insert($\mathcal{PQ}, e, LB_{agg}(Q, e)$)
15: **else if** c is a leaf node **then**
16: **if** c not seen before **then**
17: Set $c.l$ to furthest landmark by LLB from Q
18: Compute network distances $d(q, c.l) \forall q \in Q$
19: Initialize array $c.d[]$ using query distances to $c.l$
20: Binary search $c.l$'s ODL for minima of $f(x)$
21: Initialize $c.RP/c.LP$ to index of minima
22: RetrieveObjectsOL($\mathcal{PQ}, Q, c$)
23: Set best lower-bound $LB_{agg}(Q, c)$ using $c.RP/c.LP$
24: Insert($\mathcal{PQ}, c, LB_{agg}(Q, c)$)
25: **return** R
26: **function** RetrieveObjectsOL($\mathcal{PQ}, Q, c$)
27: **while** $LB_{agg,c.l}(Q, p) < \mathcal{PQ}.Top()$ for p at $c.RP/c.LP$ **do**
28: $p \leftarrow$ object at $c.RP$ or $c.LP$ with smaller $LB_{agg,c.l}$
29: Insert($\mathcal{PQ}, p, LB_{agg,max}(Q, p)$)
30: Increment $c.RP$ or decrement $c.LP$ used at Line 28

6.5 Complexity Analysis

COLT is converted from the SL-Tree, which is a complete b-ary tree. There are at most $O(|O|)$ leaf nodes in COLT. Even though COLT may be unbalanced (as the SL-tree structure is fixed), since we merge nodes with only one child, the total number of nodes will be the same as a complete b-ary tree, i.e., $O(|O|)$. However, COLT's space complexity is dominated by the m object distance lists stored in each leaf node, resulting in $O(m|O|)$ total space. We remark that m is a small constant. Due to similar reasoning, the average depth in COLT will be $O(\log |O|)$. Given the top-down conversion and $O(1)$ look-ups of SL-Tree vertex distances lists as hash-tables, propagating objects to build object distance lists and computing required values takes $O(|O| \log |O|)$ time. Sorting all object distances lists takes $O(|O|\lambda \log \lambda)$ time where λ is a small constant. Thus, the total time complexity is $O(|O| \log |O|)$. We remark that only the root of the SL-Tree is required for querying, requiring $O(m|V|)$ space and $O(|V| \log |V|)$ time to build similar to the ALT index [GH05].

6.6 Experiments

6.6.1 Experimental Settings

Environment: We conduct experiments on a Linux (64-bit) Amazon Web Services r5a.2xlarge instance with eight AMD EPYC 2.5GHz CPU cores and 64GB of memory. All code was written in single-threaded C++ and compiled by g++ v5.4 with O3 flag. All experiments were conducted using memory-resident indexes for fast query processing.

Datasets: We use the real road network graph for the continental United States with $23,947,347$ vertices and $57,708,624$ travel time edge weights as listed in Table 3.1. We also use the 8 real POI sets for the US listed in Table 3.2. For sensitivity analysis, we generate synthetic POI sets chosen uniformly at random for density d where d=$|O|/|V|$ as in previous chapters.

Parameters: We use parameter A to define a connected subgraph of G with $A\%$ of the total vertices $|V|$. Query vertices are then chosen uniformly at random from the $A\%$ subgraph. Similar to past studies [YMP05], this represents how "local" a group of query locations are. We test the sensitivity of techniques to varying numbers of results k, density d, query vertices $|Q|$, and subgraph percentages A. We also test two popular aggregate

functions *max* and *sum*. Parameter values are listed in Table 6.1 with defaults in bold if applicable. We report query time over 500 queries, e.g., with 10 synthetic object sets and 50 uniformly random query vertex sets.

Parameter	Values
k	1, 5, **10**, 25, 50
d	1, 0.1, 0.01, **0.001**, 0.0001
A (%)	1, 5, **15**, 50, 100
$\|Q\|$	2, 4, **8**, 16, 32
Aggregate Functions	***max***, *sum*

Table 6.1: AkNN Experimental Parameters (Defaults in Bold)

Techniques: We compare our algorithm against the Incremental Euclidean Restriction (IER) AkNN algorithm proposed by [YMP05]. We also use their concurrent expansion approach to adapt our highly efficient kNN heuristic based on Network Voronoi Diagrams (NVDs) from Section 4.3.2 to answer AkNN queries. Each technique uses Pruned Highway Labeling (PHL) [Aki+14] implemented by its authors to compute network distance. This ensures a level playing field, and as PHL is one of the fastest techniques, it will better show the differences in overheads. Nonetheless, we also compare algorithms on heuristic performance in terms of "false positives", which is independent of the network distance technique. SL-Trees and COLT use branch factor $b = 4$, maximum object distance list size $\lambda = 128$ (which is also α) and $m = 4$ landmarks per node for ideal performance vs. index size. NVD uses the ALT index [GH05] with $m = 16$ random landmarks to compute LLBs, which we also use as it is essentially the root of an SL-Tree.

6.6.2 Real-World Query Performance

Figure 6.3 depicts query time on real-world POI datasets, with the number of objects increasing from left to right. COLT significantly outperforms the other methods across the board, with up to two orders of magnitude improvement. COLT tends to improve more on larger POI sets, where it is more difficult to distinguish between objects. Next, our sensitivity analysis delves deeper into this and other nuances of query performance for varying parameters (Section 6.6.1).

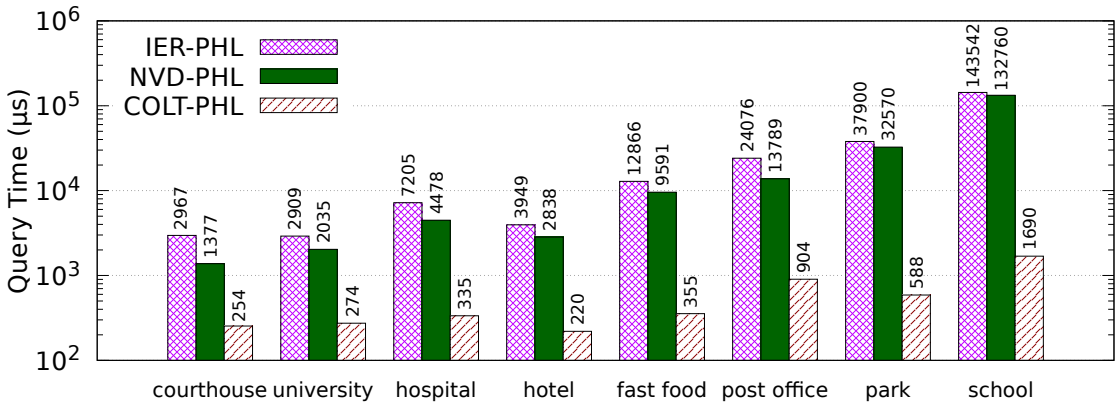


Figure 6.3: Varying Real-World Object Sets (US,k=10,$|Q|$=8,A=15%,max)

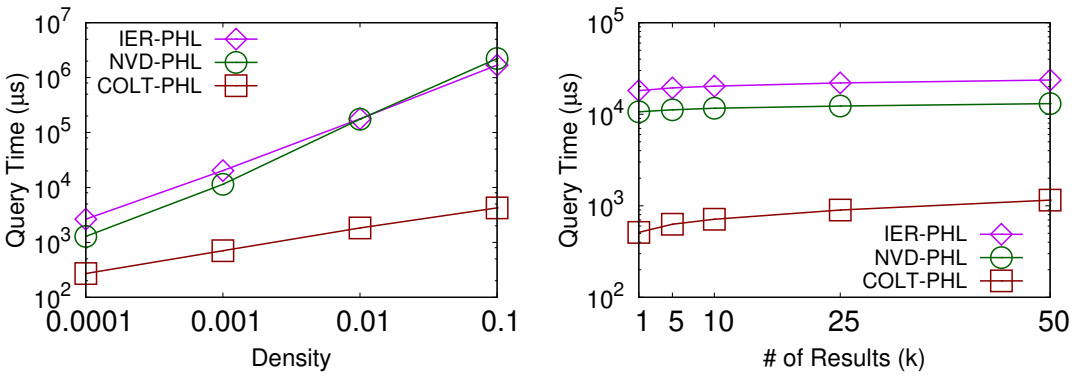
6.6.3 Sensitivity Analysis

Effect of *d*: The trend seen for real-world POIs is confirmed by using increasing object density d in Figure 6.4(a). Both IER and NVD scales poorly with increasing d. With more objects, NVD-PHL must expand more adjacent objects to find a common candidate for the same query vertices. While IER-PHL finds it harder to distinguish objects using its less accurate Euclidean lower-bound. In contrast, COLT's tighter lower bounds and hierarchical traversal is more effective in pruning subgraphs and pinpointing likely candidates.

Effect of *k*: COLT significantly outperforms the other methods for varying k in Figure 6.4(b). NVD-PHL and IER-PHL query times do not vary significantly compared to COLT (note the logarithmic scale). This suggests the same amount of work is done irrespective of k, and strongly implies the competing techniques cannot effectively identify good candidates and terminate quickly. For example, NVD-PHL expands so many candidates to find the first AkNN, that subsequent candidates have already been encountered.

Effect of $|Q|$: We investigate query time as the number of query vertices increases in Figure 6.5(a). Increasing query vertices involves computing additional lower-bounds and network distances to candidates, thus query time increases for all methods. However, COLT scales better because it conducts a single binary search on its object distance lists irrespective of the number of query vertices. On the other hand, NVDs require additional concurrent expansions as AkNN results are less likely to be close to any one query vertex.

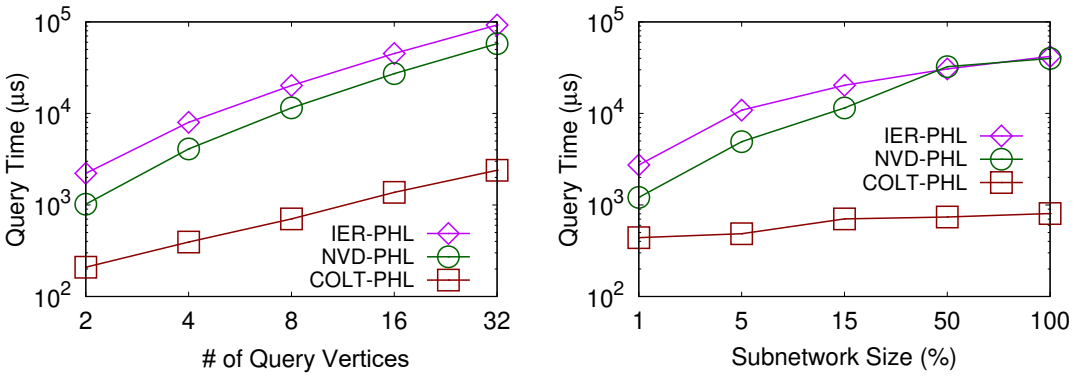
Effect of *A*: Recall that A is the percentage of graph vertices in a subgraph from which we choose query vertices. With increasing A query vertices become further apart, e.g.,



(a) Varying d (b) Varying k

Figure 6.4: *max* Function Performance (US,k=10,$|Q|$=8,A=15%,uniform objects)

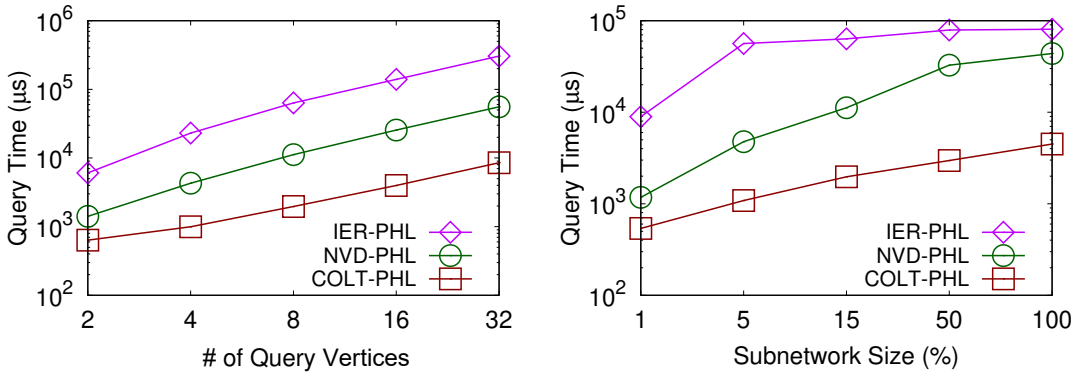
to represent an AkNN query by a logistics company placing depots nation-wide. For $A = 1\%$, NVD-PHL performance is relatively close to COLT in Figure 6.5(b). For NVDs, this scenario is similar to kNN queries where it excels, in that more query vertices share the same 1NN and concurrent expansions overlap. IER does not benefit from this as its lower-bounds are still inaccurate. In a sense, queries become "harder" with increasing A and COLT scales extremely well in that case.



(a) Varying $|Q|$ (b) Varying A

Figure 6.5: *max* Function Performance (US,k=10,$|Q|$=8,A=15%,uniform objects)

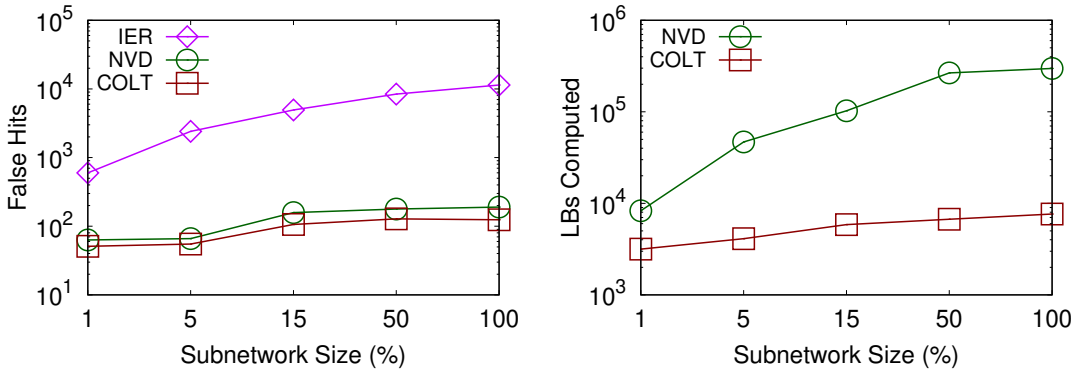
Effect of Aggregate Function: We evaluate another popular aggregate function, *sum*, in Figure 6.6. In absolute terms, both IER and COLT have higher query times for *sum* than *max*. This is because *sum* also sums the error of the lower-bound. The detrimental effect is amplified by the use of hierarchies by IER and COLT, explaining the relative improvement of NVD-PHL. However, COLT is more robust to this than IER as its lower-bounds are tighter and still significantly outperforms both methods.



(a) Varying $|Q|$ (b) Varying A

Figure 6.6: *sum* Function Performance (US,k=10,$|Q|$=8,A=15%,uniform objects)

Heuristic Efficiency: In addition to query time, we also measure the efficiency of the heuristics in Figure 6.7 using machine independent metrics. Figure 6.7(a) shows the number of network distances computed to non-results. The poor query time performance of IER is explained by the significantly higher (and costly) network distances computed. However, NVDs do not compute much more network distances than COLT. As both methods use landmark lower-bounds (LLBs), that are more accurate than IER's Euclidean distance, they both avoid computing network distances. The poor query time of NVDs can be explained by Figure 6.7(b), which shows NVDs computing a significantly higher number of LLBs than COLT. This confirms expansion in NVDs encounters a significantly higher number of objects than COLT's hierarchical search.



(a) False Hits (b) Landmark Lower-Bounds

Figure 6.7: AkNN Heuristic Performance (US,k=10,$|Q|$=8,A=15%,max,uniform objects)

6.6.4 Pre-Processing Costs

Table 6.2 details the road network pre-processing costs in terms of time and space. As expected, SL-Tree has higher pre-processing than ALT. However, the space cost is not significantly higher. The indexing time is still comparable to PHL, which has one of the fastest pre-processing times for high-performance indexes [Aki+14]. Moreover, the SL-Tree is discarded after COLT construction and only its root node is kept for query processing, which has the same index size as ALT.

	ALT (m=16)	**SL-Tree (m=4)**	**PHL**
Time	71s	25m	25m
Space	1.43GB	4.6GB	15.8GB

Table 6.2: Road Network Index Statistics (US)

Table 6.3 lists the pre-processing costs for object indexes used by each technique for default density $0.001 \times |V|$. Note that object indexes are constructed for each object set (e.g., the set of restaurants). COLT is significantly smaller and faster to construct than NVDs and is comparable to R-trees as both have space complexity linear to the input. Note that the NVD index has been compressed using merged quadtrees [DS12; SSA08], which only store the geometric area of Voronoi node sets.

	COLT (m=4)	**NVD**	**R-tree**
Time	63ms	11s	6ms
Space	0.9MB	28MB	0.9MB

Table 6.3: Object Index Statistics (US,uniform objects,d=0.001)

6.7 Summary

COLT elegantly combines several properties that benefit AkNN search. First, AkNN queries involve multiple query locations. This means result objects are likely to be found further from all query vertices, which is easier to locate using the hierarchical subgraph traversal in COLT. Second, COLT can compute better and inexpensive lower-bounds using localized landmarks at each level of the hierarchy. Combined with its novel property for convexity preserving aggregate functions, we can retrieve more promising candidates and terminate search sooner. This is demonstrated in our experiments with COLT significantly outperforming competing techniques on AkNN queries. Moreover, the data structures used

for querying are light-weight in both theory and practice. These results contribute to the main hypothesis of this study by showing an additional POI search query for which a decoupled heuristic is still the most effective technique by a wide margin. Furthermore, it shows that decoupled heuristics can increasingly be more sophisticated to better capture the intuitions for different POI search queries.

Chapter 7

Final Remarks

> Never measure the height of a mountain until you have reached the top. Then you will see how low it was.
>
> Dag Hammarskjöld

At a time where smartphones are omnipresent and backed by cheap network bandwidth, point of interest (POI) search is a highly relevant problem. Given the surge in demand for map-based services, such as ride-hailing applications, users are increasingly demanding greater accuracy and capabilities from these services. Consequently, many of these map-based services employ indexing methods and query processing algorithms to find POIs through the road network. While closely related to the shortest path problem in road networks, POI search has not received nearly as much attention.

In this thesis, we study heuristics used to guide searches towards the POIs we are most interested in (and by extension, avoid those we are not interested in). We find that a simple paradigm to solve POI search problems using *decoupled heuristics* has been highly underrated. This approach involves retrieving likely candidate POIs according to some heuristic and then computing their shortest paths using another technique. By proposing new techniques employing this paradigm, we show that it is demonstrably effective on a wide range of POI search problems. Moreover, we show that our techniques significantly improve on the state-of-the-art in practice on real-world datasets and under many experimental settings. In the process, we also make several useful ancillary contributions. We summarize the main contributions and conclusions of this thesis as follows.

7.1 Main Outcomes

Our thorough experimental investigation (Chapter 3) into the state-of-the-art for the kNN problem on road network provides a much clearer picture of the relative advantages and disadvantages of the most advanced existing techniques. Our key insight, and the linchpin of our subsequent hypotheses, is the performance of the neglected decoupled heuristic based on Euclidean distance. We show that this heuristic, through a simple improvement, is actually the best performing method on almost all settings. This is even true in cases where it is expected to perform extremely poorly, such as travel time. As it turns out, this remarkable observation has far reaching consequences for other POI search problems and the overall effectiveness of the decoupled heuristic paradigm.

While Euclidean distance was effective compared to other techniques in Chapter 3, we observe that there is still room for improvement in Chapter 4. This is especially true for road networks with travel-time edge-weights, where Euclidean distance can only be used to form a loose lower-bounding heuristic on travel time. We revive a long-discarded data structure, the Network Voronoi Diagram (NVD), and re-purpose it through a novel observation on its heuristic capabilities when combined with Landmark Lower-Bounds (LLBs), which are more accurate than Euclidean distance lower-bounds. In the process, we overcome several technical challenges faced by previous studies that made it difficult to take advantage of LLBs in a scalable manner. Our technique significantly improves on the Euclidean heuristic in terms of running time. We verify that the improvement is also significant in terms of false positive candidates generated. This machine-independent metric directly shows that we improve heuristic efficiency.

In Chapter 5, we identify a flaw in existing approaches to solve spatial keyword problems on road networks, due to the use of *keyword aggregation*. We propose the use of an alternative *keyword separation* approach, which is capable of employing decoupled heuristics. Our flexible and extensible framework, named K-SPIN, exhibits significant improvement in query time and heuristic efficiency over existing techniques. Keyword separation has dire implications for pre-processing cost, which perhaps explains why it has never previously been used. However, by making smart observations, we propose techniques to significantly reduce the pre-processing cost, to the point that our framework is light-weight for even continent sized road network datasets. Our novel observation on

Zipfian distributions and new ρ-Approximate Network Voronoi Diagram are contributions that may find applications in other problems related to spatial keyword queries and POI search. Moreover, applying our NVD-based heuristic from Chapter 4 demonstrates its utility to POI search problems in general.

In Chapter 6, we observe that the intuition that makes our heuristics proposed in Chapter 4 highly effective is not necessarily true for other types of POI search queries, such as Aggregate k Nearest Neighbor (AkNN) queries. A hierarchical approach seems more appropriate, but we also observe that no solutions exist to do this efficiently. In response, we propose the SL-Tree and COLT indexes to enable efficient hierarchical graph traversal. Based on COLT, we propose a more sophisticated decoupled heuristic that quickly and efficiently homes in on likely AkNN candidates wherever they may be located. Furthermore, our novel observation on convexity-preserving aggregate functions reduces the number of candidates we must consider. In terms of the overarching message of this thesis, our results strengthen the case regarding the effectiveness of the decoupled heuristic by showing that it can successfully incorporate more complex heuristics to answer POI search problems. Moreover, our light-weight COLT index can be applied to other hierarchical graph search problems, which we leave as future work.

We have demonstrated our significant query processing improvements on a wide range of POI search problems using real-world datasets, numerous varying and realistic settings, and highly efficient implementations. In the process, we made critical observations regarding the drastic impact of implementation on experimental performance. Consequently, we provide case studies and advice in Appendix A to guide future researchers towards efficient implementations. Although this is provided in the context of kNN queries, our insights are applicable to the implementation of any algorithm. Moreover, our implementations have already been released as open-source [Abe16] or will be released shortly, for other researchers to reproduce our results or to use in future studies. Our experimental framework has already been used by several noteworthy works [Yao+18; Sha+16; Xu+18; Luo+18].

7.2 Directions for Future Work

Our overarching contribution has been to demonstrate the effectiveness of decoupled heuristics. The strong performance of the techniques we propose in this study indicates

a potential better direction for POI search algorithms. The low-hanging fruit here is to see whether existing decoupled heuristics, such as the simple Euclidean heuristic, can be applied to other POI search problems. In fact, our insight into the Euclidean distance heuristic has already influenced work on Flexible Aggregate Nearest Neighbor queries [Yao+18] and kNN queries on navigation meshes [ZTH18]. However, if the goal is to improve query performance generally, one option is to develop faster network distance techniques. Given the decoupling of network distance computation, any of our techniques will benefit from such advances. The other, perhaps more interesting, option is to develop new more sophisticated decoupled heuristics in the same vein as our COLT index. Nonetheless, in this section, we discuss several concrete paths for future work.

7.2.1 More Diverse Settings

We have investigated the "purest" POI search queries, such as kNN or spatial keyword queries, which are the relatively well-studied POI search queries for road networks. There is a plethora of variations on these settings (as surveyed in Section 2.5) for which decoupled heuristics may be effective and could be investigated. For example, one variation is to retrieve objects when they are moving [Luo+18]. Or perhaps the objective could be to, instead, retrieve objects near a query location that is moving along a path [Che+09]. While we have studied static road networks, accuracy can be improved by considering road networks that change due to various factors. One way is to consider edge-weights as time-varying functions [DBS10] and retrieve points of interest for the current (or a future) time to capture factors like peak hour traffic. Another approach is to consider edge-weights as real-time values to incorporate live events such as accidents [DW15]. The variations are endless and many of these settings have not been well studied.

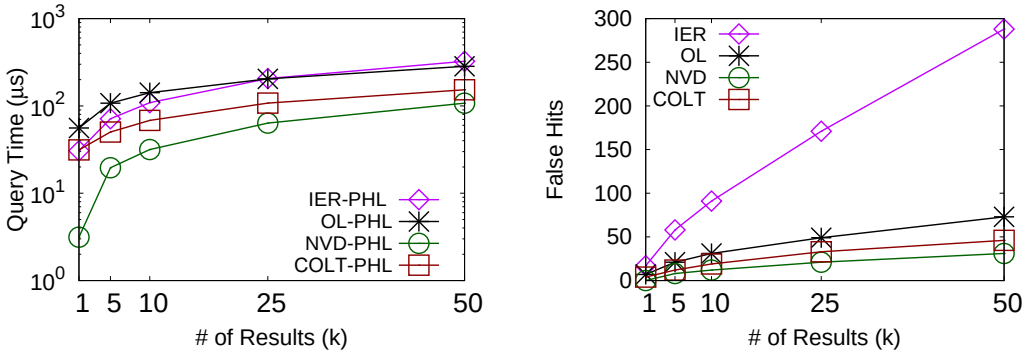
7.2.2 Theoretical Candidate Guarantees

While we have shown that our heuristics are efficient in practice, it has not yet been possible to provide theoretical guarantees on the number of candidates generated. For example, in Section 5.5.1 we derive an expression for the query complexity in terms of a variable κ, which indicates the maximum number of candidates retrieved. While we obtain experimental results for real-world datasets showing the maximum value of κ is a small constant in practice, it is not immediately clear whether this can also be proven true

in theory. In fact, to the best of our knowledge, no other work has been able to provide meaningful guarantees in the context of POI search in road networks. Thus, theoretical guarantees on heuristic efficiency remains an open problem.

7.2.3 Further Application of COLT

In Chapter 6, we presented the COLT index to enable efficient hierarchical traversal of a graph by an accurate lower-bound to find POI search candidates. Our technique was particularly adept at answering AkNN queries, especially due to an interesting property it displayed for convexity-preserving aggregate functions. For comparison against COLT, we adapted our NVD-based heuristic from Chapter 4, to demonstrate that a different type of heuristic was required for AkNN queries. Specifically, the intuition that made NVDs exceptional at kNN queries, that the next nearest object is among the current nearest objects, is not necessarily true for AkNNs. In contrast, the hierarchical search using the COLT index is more suitable for AkNN search, as verified in our experimental analysis. However, it would be interesting to see if the converse is also true, i.e., a hierarchical approach is less effective on kNN queries. If true, as a follow-up to that, it would be interesting to also see if there is a way to use COLT efficiently for kNN querying. Figure 7.1 shows some initial experimental results for varying k, with COLT-PHL using the same top-down hierarchical search described in Chapter 6 to answer kNN queries. While NVD-PHL is still the most efficient heuristic on both query time and false positives, COLT has made noticeable improvement on OL-PHL. It would be interesting to see if a bottom-up algorithm, rather than the top-down search suitable for AkNN queries, would further reduce this gap.



(a) Query Time (b) False Hits

Figure 7.1: Effect of k on kNN Queries (US,d=0.001,uniform objects)

References

[Abe16] Tenindra Abeywickrama. *Road Network kNN Experimental Framework*. 2016. URL: https://github.com/tenindra/RN-kNN-Exp.

[Abr+10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms". In: *SODA*. 2010, pp. 782–793.

[Abr+11] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. "A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks". In: *SEA*. 2011, pp. 230–241.

[AC] Tenindra Abeywickrama and Muhammad Aamir Cheema. "Hierarchical Graph Traversal for Aggregate kNN Search in Road Networks". In: *To Be Submitted*.

[AC17] Tenindra Abeywickrama and Muhammad Aamir Cheema. "Efficient Landmark-Based Candidate Generation for kNN Queries on Road Networks". In: *DASFAA*. 2017, pp. 425–440.

[ACK19] Tenindra Abeywickrama, Muhammad Aamir Cheema, and Arijit Khan. "K-SPIN: Efficiently Processing Spatial Keyword Queries on Road Networks". In: *IEEE Trans. Knowl. Data Eng.* (2019). DOI: 10.1109/TKDE.2019.2894140.

[ACT16a] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. "k-Nearest Neighbors on Road Networks: A Journey in Experimentation and In-Memory Implementation". In: *CoRR* abs/1601.01549 (2016). URL: https://arxiv.org/abs/1601.01549.

[ACT16b] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. "k-Nearest Neighbors on Road Networks: A Journey in Experimentation and In-memory Implementation". In: *PVLDB* 9.6 (2016), pp. 492–503.

[ACT18] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. "k-Nearest Neighbors on Road Networks: Euclidean Heuristic Revisited". In: *SoCS*. 2018, pp. 184–185.

[AIY13] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. "Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling". In: *SIGMOD*. 2013, pp. 349–360.

[Aki+14] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. "Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labeling". In: *ALENEX*. 2014, pp. 147–154.

[And16] Monica Anderson. *More Americans using smartphones for getting directions, streaming TV*. 2016. URL: https://web.archive.org/web/20190117104508/http://www.pewresearch.org/fact-tank/2016/01/29/us-smartphone-use/ (visited on 01/19/2019).

[Bas+07] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. "In Transit to Constant Time Shortest-path Queries in Road Networks". In: *WEA*. 2007, pp. 46–59.

[Bas+15] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. "Route Planning in Transportation Networks". In: *CoRR* abs/1504.05140 (2015). URL: https://arxiv.org/abs/1504.05140.

[Bau+10] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. "Combining Hierarchical and Goal-directed Speed-up Techniques for Dijkstra's Algorithm". In: *JEA* 15.2.3 (2010), 2.3:2.1–2.3:2.31.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[Cao+18] B. Cao, C. Hou, S. Li, J. Fan, J. Yin, B. Zheng, and J. Bao. "SIMkNN: A Scalable Method for in-MemorykNN Search over Moving Objects in Road Networks". In: *IEEE Transactions on Knowledge and Data Engineering* 30.10 (2018), pp. 1957–1970.

[CC05] Hyung-Ju Cho and Chin-Wan Chung. “An Efficient and Scalable Approach to CNN Queries in a Road Network”. In: *VLDB*. 2005, pp. 865–876.

[Che+09] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, and Jeffrey Xu Yu. “Monitoring Path Nearest Neighbor in Road Networks”. In: *SIGMOD*. 2009, pp. 591–602.

[Che+13] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. “Spatial Keyword Query Processing: An Experimental Evaluation”. In: *PVLDB* 6.3 (2013), pp. 217–228.

[CJ16] Gao Cong and Christian S. Jensen. “Querying Geo-Textual Data: Spatial Keyword Queries and Beyond”. In: *SIGMOD*. 2016, pp. 2207–2212.

[CJW09] Gao Cong, Christian S. Jensen, and Dingming Wu. “Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects”. In: *PVLDB* 2.1 (2009), pp. 337–348.

[Cor+01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. 2nd. McGraw-Hill, 2001.

[CXC17] Peng Cheng, Hao Xin, and Lei Chen. “Utility-Aware Ridesharing on Road Networks”. In: *SIGMOD*. 2017, pp. 1197–1210.

[DBS10] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. “Efficient K-nearest Neighbor Search in Time-dependent Spatial Networks”. In: *DEXA*. 2010, pp. 432–449.

[Del+11] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. “PHAST: Hardware-Accelerated Shortest Path Trees”. In: *IPDPS*. 2011, pp. 921–931.

[DGW13] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. “Hub Label Compression”. In: *SEA*. 2013, pp. 18–29.

[Dij59] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[DS12] Ugur Demiryurek and Cyrus Shahabi. “Indexing Network Voronoi Diagrams”. In: *DASFAA*. 2012, pp. 526–543.

[DW15] Daniel Delling and Renato F. Werneck. “Customizable Point-of-Interest Queries in Road Networks”. In: *IEEE Trans. on Knowl. and Data Eng.* 27.3 (2015), pp. 686–698.

[EH00] Martin Erwig and Fernuniversitat Hagen. “The Graph Voronoi Diagram with Applications”. In: *Networks* 36 (2000), pp. 156–163.

[Gei+08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. In: *WEA*. 2008, pp. 319–333.

[GH05] Andrew V. Goldberg and Chris Harrelson. “Computing the Shortest Path: A Search Meets Graph Theory”. In: *SODA*. 2005, pp. 156–165.

[Guo+19] Fangda Guo, Ye Yuan, Guoren Wang, Lei Chen, Xiang Lian, and Zimeng Wang. “Cohesive Group Nearest Neighbor Queries over Road-Social Networks”. In: *ICDE*. 2019, pp. 434–445.

[Gut84] Antonin Guttman. “R-trees: A Dynamic Index Structure for Spatial Searching”. In: *SIGMOD*. 1984, pp. 47–57.

[GW05] Andrew V. Goldberg and Renato Fonseca F. Werneck. “Computing Point-to-Point Shortest Paths from External Memory”. In: *ALENEX*. 2005, pp. 26–40.

[He+19] Dan He, Sibo Wang, Xiaofang Zhou, and Reynold Cheng. “An Efficient Framework for Correctness-Aware kNN Queries on Road Networks”. In: *ICDE*. 2019, pp. 1298–1309.

[HJŠ05] Xuegang Huang, Christian S. Jensen, and Simonas Šaltenis. “The Islands Approach to Nearest Neighbor Querying in Spatial Networks”. In: *SSTD*. 2005, pp. 73–90.

[HLL06] Haibo Hu, Dik Lun Lee, and Victor C. S. Lee. “Distance Indexing on Road Networks”. In: *VLDB*. 2006, pp. 894–905.

[HLX06] Haibo Hu, Dik Lun Lee, and Jianliang Xu. “Fast Nearest Neighbor Search on Road Networks”. In: *EDBT*. 2006, pp. 186–203.

[HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE Trans. Syst. Sci. Cybern* 4.2 (1968), pp. 100–107.

[Iou+07] Elias Ioup, Kevin Shaw, John Sample, and Mahdi Abdelguerfi. "Efficient AKNN Spatial Network Queries Using the M-Tree". In: *GIS*. 2007, 46:1–46:4.

[JFW15] Minhao Jiang, Ada Wai-Chee Fu, and Raymond Chi-Wing Wong. "Exact Top-k Nearest Keyword Search in Large Networks". In: *SIGMOD*. 2015, pp. 393–404.

[Jia+14] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. "Hop Doubling Label Indexing for Point-to-point Distance Querying on Scale-free Networks". In: *VLDB*. 2014, pp. 1203–1214.

[KK98] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (1998), pp. 359–392.

[Kno+07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. "Computing Many-to-many Shortest Paths Using Highway Hierarchies". In: *ALENEX*. 2007, pp. 36–45.

[Kri+07] Hans-Peter Kriegel, Peer Kröger, Peter Kunath, and Matthias Renz. "Generalizing the Optimality of Multi-step K-nearest Neighbor Query Processing". In: *SSTD*. 2007, pp. 75–92.

[Kri+08] Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Tim Schmidt. "Hierarchical Graph Embedding for Efficient Query Processing in Very Large Traffic Networks". In: *SSDBM*. 2008, pp. 150–167.

[KS04] Mohammad Kolahdouzan and Cyrus Shahabi. "Voronoi-based K Nearest Neighbor Search for Spatial Network Databases". In: *VLDB*. 2004, pp. 840–851.

[LCW19] Zijian Li, Lei Chen, and Yue Wang. "G*-Tree: An Efficient Spatial Index on Road Networks". In: *ICDE*. 2019, pp. 268–279.

[Lee+12] Ken C. K. Lee, Wang-Chien Lee, Baihua Zheng, and Yuan Tian. "ROAD: A New Spatial Object Search Framework for Road Networks". In: *IEEE Trans. Knowl. Data Eng.* 24.3 (2012), pp. 547–560.

[Li+18] Chuanwen Li, Yu Gu, Jianzhong Qi, Jiayuan He, Qingxu Deng, and Ge Yu. "A GPU Accelerated Update Efficient Index for kNN Queries in Road Networks". In: *ICDE*. 2018, pp. 881–892.

[Li05] Feifei Li. *Real Datasets for Spatial Databases: Road Networks and Points of Interest*. 2005. URL: http://www.cs.utah.edu/~lifeifei/SpatialDataset.

[Lia+15] Bailong Liao, Leong Hou U, Man Lung Yiu, and Zhiguo Gong. "Beyond Millisecond Latency KNN Search on Commodity Machine". In: *IEEE Trans. on Knowl. and Data Eng.* 27.10 (2015), pp. 2618–2631.

[LLZ09] Ken C. K. Lee, Wang-Chien Lee, and Baihua Zheng. "Fast Object Search on Road Networks". In: *EDBT*. 2009, pp. 1018–1029.

[Luo+18] Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, and Yudian Zheng. "TOAIN: A Throughput Optimizing Adaptive Index for Answering Dynamic kNN Queries on Road Networks". In: *PVLDB* 11.5 (2018), pp. 594–606.

[Luo+19] Siqiang Luo, Ben Kao, Xiaowei Wu, and Reynold Cheng. "MPR — A Partitioning-Replication Framework for Multi-Processing kNN Search on Road Networks". In: *ICDE*. 2019, pp. 1310–1321.

[LWZ19] Lei Li, Sibo Wang, and Xiaofang Zhou. "Time-Dependent Hop Labeling on Road Network". In: *ICDE*. 2019, pp. 902–913.

[Mou+06] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias, and Nikos Mamoulis. "Continuous Nearest Neighbor Monitoring in Road Networks". In: *VLDB*. 2006, pp. 43–54.

[Mou+15] Kyriakos Mouratidis, Jing Li, Yu Tang, and Nikos Mamoulis. "Joint Search by Social and Spatial Proximity". In: *IEEE Trans. Knowl. Data Eng.* 27.3 (2015), pp. 781–793.

[New18] Newzoo. *Global Mobile Market Report*. 2018. URL: https://web.archive.org/web/20181128134855/https://newzoo.com/insights/articles/newzoos-2018-global-mobile-market-report-insights-into-the-worlds-3-billion-smartphone-users/ (visited on 02/01/2019).

[OBS00] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. 2nd. John Wiley & Sons, Inc., 2000.

[Ope] OpenStreetMap. URL: http://www.openstreetmap.org.

[Ouy+18] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. "When Hierarchy Meets 2-Hop-Labeling: Efficient Shortest Distance Queries on Road Networks". In: *SIGMOD*. 2018, pp. 709–724.

[Pap+03] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. "Query Processing in Spatial Network Databases". In: *VLDB*. 2003, pp. 802–813.

[Pap+04] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. "Group Nearest Neighbor Queries". In: *ICDE*. 2004, pp. 301–312.

[Pap+05] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. "Aggregate Nearest Neighbor Queries in Spatial Databases". In: *ACM Trans. Database Syst.* 30.2 (2005), pp. 529–576.

[Pat06] 9th DIMACS Implementation Challenge - Shortest Paths. 2006. URL: http://www.dis.uniroma1.it/%7Echallenge9/.

[Poh71] Irah Pohl. "Bi-directional Search". In: *Machine Intelligence* 6 (1971), pp. 127–140.

[Qia+13] Miao Qiao, Lu Qin, Hong Cheng, Jeffrey Xu Yu, and Wentao Tian. "Top-K Nearest Keyword Search on Large Graphs". In: *PVLDB* 6.10 (2013), pp. 901–912.

[RN12] João B. Rocha-Junior and Kjetil Nørvåg. "Top-k Spatial Keyword Queries on Road Networks". In: *EDBT*. 2012, pp. 168–179.

[Sam05] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2005.

[SAS05] Jagan Sankaranarayanan, Houman Alborzi, and Hanan Samet. "Efficient Query Processing on Spatial Networks". In: *GIS*. 2005, pp. 200–209.

[Sha+16] Zhou Shao, Muhammad Aamir Cheema, David Taniar, and Hua Lu. "VIP-Tree: An Effective Index for Indoor Spatial Queries". In: *PVLDB* 10.4 (2016), pp. 325–336.

[ŠJ14] Darius Šidlauskas and Christian S. Jensen. "Spatial Joins in Main Memory: Implementation Matters!" In: *PVLDB* 8.1 (2014), pp. 97–100.

[SK98] Thomas Seidl and Hans-Peter Kriegel. "Optimal Multi-step K-nearest Neighbor Search". In: *SIGMOD*. 1998, pp. 154–165.

[SKS03] Cyrus Shahabi, MohammadR. Kolahdouzan, and Mehdi Sharifzadeh. "A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases". In: *GeoInformatica* 7.3 (2003), pp. 255–273.

[SSA08] Hanan Samet, Jagan Sankaranarayanan, and Houman Alborzi. "Scalable Network Distance Browsing in Spatial Databases". In: *SIGMOD*. 2008, pp. 43–54.

[SSA09] Jagan Sankaranarayanan, Hanan Samet, and Houman Alborzi. "Path Oracles for Spatial Networks". In: *PVLDB* 2.1 (2009), pp. 1210–1221.

[Ste15] Greg Sterling. *Data Suggest Local-Intent Queries Nearing Half of All US Search Volume*. 2015. URL: https://web.archive.org/web/20180410190608/http://screenwerk.com/2015/05/11/data-suggest-that-local-intent-queries-nearly-half-of-all-search-volume (visited on 01/19/2019).

[WCJ12] Dingming Wu, Gao Cong, and Christian S. Jensen. "A Framework for Efficient Spatial Web Object Retrieval". In: *VLDB J.* 21.6 (2012), pp. 797–822.

[Wu+11] Dingming Wu, Man Lung Yiu, Christian S. Jensen, and Gao Cong. "Efficient Continuously Moving Top-k Spatial Keyword Query Processing". In: *ICDE*. 2011, pp. 541–552.

[Wu+12] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. "Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation". In: *PVLDB* 5.5 (2012), pp. 406–417.

[Xu+18] Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, and Lars Kulik. "Finding All Nearest Neighbors with a Single Graph Traversal". In: *DASFAA*. 2018, pp. 221–238.

[Yao+18] Bin Yao, Zhongpu Chen, Xiaofeng Gao, Shuo Shang, Shuai Ma, and Minyi Guo. "Flexible Aggregate Nearest Neighbor Queries in Road Networks". In: *ICDE*. 2018, pp. 761–772.

[YMP05] Man Lung Yiu, Nikos Mamoulis, and Dimitris Papadias. "Aggregate Nearest Neighbor Queries in Road Networks". In: *IEEE Trans. Knowl. Data Eng.* 17.6 (2005), pp. 820–833.

[YR19] Pranali Yawalkar and Sayan Ran. "Route Recommendations on Road Networks for Arbitrary User Preference Functions". In: *ICDE*. 2019, pp. 602–613.

[ZCT14] Dongxiang Zhang, Chee-Yong Chan, and Kian-Lee Tan. "Processing Spatial Keyword Query As a Top-k Aggregation Query". In: *SIGIR*. 2014, pp. 355–364.

[Zha+17] J. Zhao, Y. Gao, G. Chen, C. S. Jensen, R. Chen, and D. Cai. "Reverse Top-k Geo-Social Keyword Queries in Road Networks". In: *ICDE*. 2017, pp. 387–398.

[Zha+18] J. Zhao, Y. Gao, G. Chen, and R. Chen. "Why-Not Questions on Top-k Geo-Social Keyword Queries in Road Networks". In: *ICDE*. 2018, pp. 965–976.

[Zhe+16] Bolong Zheng, Kai Zheng, Xiaokui Xiao, Han Su, Hongzhi Yin, Xiaofang Zhou, and Guohui Li. "Keyword-aware continuous kNN query on road networks". In: *ICDE*. 2016, pp. 871–882.

[Zho+13] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. "G-tree: An Efficient Index for KNN Search on Road Networks". In: *CIKM*. 2013, pp. 39–48.

[Zho+15] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, Lizhu Zhou, and Zhiguo Gong. "G-Tree: An Efficient and Scalable Index for Spatial Search on Road Networks". In: *IEEE Trans. Knowl. Data Eng.* 27.8 (2015), pp. 2175–2189.

[Zhu+10] Liang Zhu, Yinan Jing, Weiwei Sun, Dingding Mao, and Peng Liu. "Voronoi-based Aggregate Nearest Neighbor Query Processing in Road Networks". In: *GIS*. 2010, pp. 518–521.

[Zhu+13] Andy Diwen Zhu, Hui Ma, Xiaokui Xiao, Siqiang Luo, Youze Tang, and Shuigeng Zhou. "Shortest Path and Distance Queries on Road Networks: Towards Bridging Theory and Practice". In: *SIGMOD*. 2013, pp. 857–868.

[ZM06] Justin Zobel and Alistair Moffat. “Inverted Files for Text Search Engines”. In: *ACM Comput. Surv.* 38.2 (2006).

[ZTH18] Shizhe Zhao, David Taniar, and Daniel Damir Harabor. “Fast k-Nearest Neighbor on a Navigation Mesh”. In: *SoCS*. 2018, pp. 124–132.

Appendix A

Implementation in Main Memory

During our experimental investigation into the state-of-the-art of kNN queries, detailed in Chapter 3, we encountered some discrepancies in our results when different implementation choices were made. In this appendix, we describe our journey towards efficient implementations of kNN query algorithms and provide guidelines for future implementers to avoid common pitfalls. Notably, our findings are not limited to POI search queries but applicable to all in-memory query processing algorithms. The insights described in this appendix were published in [ACT16b; ACT16a].

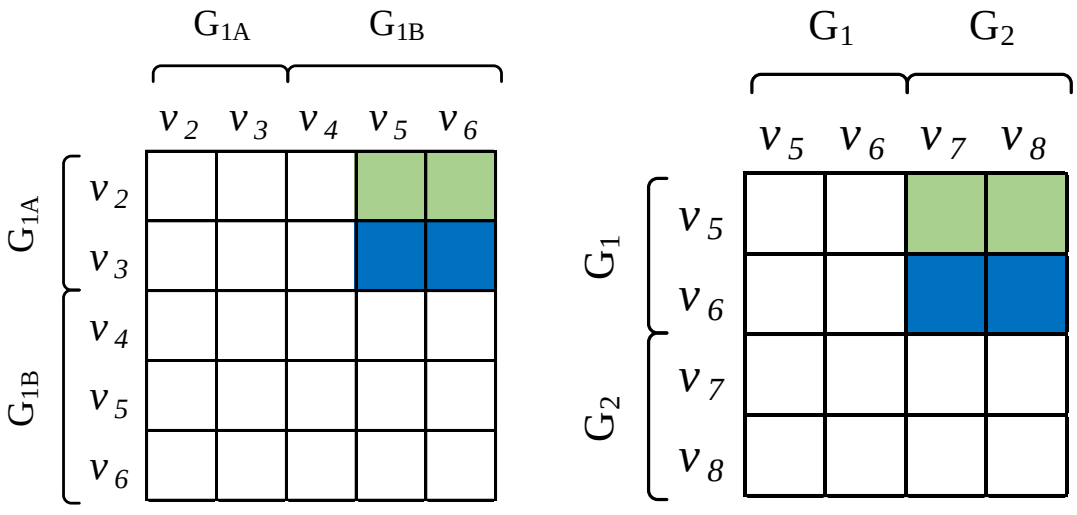
A.1 Overview

Given the affordability of memory, the capacities available and the demand for high performance map-based services, memory-resident query processing is a realistic and often necessary requirement. However, we have seen in-memory implementation efficiency can affect performance to the point that algorithmic efficiency becomes irrelevant [ŠJ14]. Pertinently for us, in this study, it can potentially make some heuristics look better than others purely due to different implementations. First, this identifies the need to understand how this can happen so that guidelines for efficient implementation may be developed. Second, it implies that some algorithms may possess intrinsic qualities that make them superior in-memory. The utility of the latter cannot be ignored. We first illustrate both points using a case study and then outline typical choices and our approach to settle them.

A.2 Case Study: G-tree Distance Matrices

G-tree's distance matrices store certain pre-computed graph distances (between borders of subgraphs), allowing "assembly" of longer distances in a piece-wise manner. We first describe the G-tree assembly method below, then show how the implementation of distance matrices can significantly impact its performance.

Every G-tree node has a set of borders. From our running example in Figure 3.3, v_5 and v_6 are borders of G_1. Each non-leaf node also has a set of children, for example, G_{1A} and G_{1B} are the children of G_1. These, in turn, have their own borders, which we refer to as "child borders" of G_1. A distance matrix stores the distances from every child border to every other child border. For example, for G_1, its child borders are v_2, v_3, v_4, v_5, v_6, and its distance matrix is shown in Figure A.1(a). But recall that a border of a G-tree node must necessarily be a border of a child node, e.g., the borders of G_1, v_5 and v_6, are also borders of G_{1B}. This means the distance matrix of G_1 repeatedly stores some border-to-border distances already in the distance matrix of G_{1B}, a redundancy that can become quite large for bigger graphs. To avoid this repetition and utilize, in general, $O(1)$ random retrievals, a practitioner may choose to implement the distance matrix as a hash-table. This has the added benefit of being able to retrieve distances for any two arbitrary borders.



(a) G_1 (b) G_0

Figure A.1: Distance Matrices

Given a source vertex s and target t, G-tree's assembly method first determines the *tree path* through the G-tree hierarchy. This is a sequence of G-tree nodes starting from

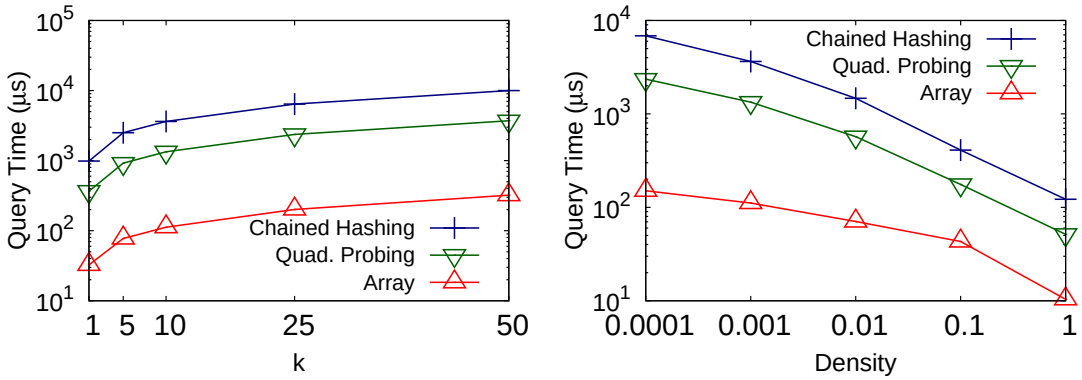
the leaf node containing s through its immediate parent and each successive parent node up to the least common ancestor (LCA) node. From the LCA, the path traces through successive child nodes until reaching the leaf node containing t. The assembly method then computes the distances from all borders from the ith node in the path, G_i, to all borders in i+1th node, G_{i+1}. These two nodes are necessarily either both children of the LCA or have a parent-child relationship. In either case, the parent node's distance matrix contains values for all border-to-border distances. Assuming we have computed all distances from s to the borders of G_i, we compute the distances to the borders of G_{i+1} by iterating over each border of G_i and computing the minimum distance through them to each border of G_{i+1}.

From our running example in Figure 3.3, let v_1 be the source and v_{12} be the target. In this case the beginning of the tree path will contain the child node G_{1A} and then its parent node G_1. Assume we have computed the distances to the borders of G_{1A} (easily done by using the distance matrix of leaf node G_{1A}, which stores leaf vertex to leaf border distances). Now we compute the distance to each border of G_1 from v_1, by finding the minimum distance through one of G_{1A}'s borders. To do this, for each of G_{1A}'s borders, we iterate over G_1's borders, retrieving distance matrix values for each pair (updating the minimum when a smaller distance is found). This is shown by the shaded cells in Figure A.1(a). Similarly, G_1 and its sibling G_2 are the next nodes in the tree path, and we again retrieve distance matrix values by iterating over two lists of borders. These values are retrieved from the matrix of the LCA node, G_0, and the values accessed are shaded in Figure A.1(b).

As we are iterating over lists (i.e., arrays) of borders, the distance matrix does not need to be accessed in an arbitrary order, as we observed in the G-tree authors' implementation. This is made possible by grouping the borders of child nodes as shown in Figure A.1 and storing the starting index for each child's borders. Additionally, we create an offset array indicating the position of the nodes' own borders in its distance matrix. For example, the offset array for G_1 indicates its borders (v_5 and v_6) are at the 3rd and 4th index of each row in its distance matrix shown in Figure A.1(a). While Figure A.1 shows the distance matrix as a 2D array, it is best implemented as a 1D array. This and the previously described access method, allow all shaded values to be accessed from sequential memory locations, thus displaying excellent spatial locality. This is shown in Figure A.1 as the shaded cells

are either contiguous or very close to being so. Spatial locality makes the code cache-friendly, allowing the CPU to easily predict and pre-fetch data that will be read next into the CPU cache. Otherwise, the data would need to be retrieved from memory, which is 20−200× slower than CPU cache (depending on the level). This effect is amplified in real road networks as they contain significantly larger numbers of borders per node.

We compare three implementations of distance matrices, including the 1D array described above and two types of hash-tables: chained hashing [Cor+01] (STL `unordered_map`); and quadratic probing [Cor+01] (Google `dense_hash_map`). In Figure A.2, chained hashing is a staggering 30 times slower than the array. While quadratic probing is an improvement, it is still an order of magnitude slower. Had we used either of the hash-table types, we would have unfairly concluded that G-tree was the worst performing algorithm.



(a) Varying k (b) Varying Density

Figure A.2: Distance Matrix Variants (NW,d=0.001,k=10)

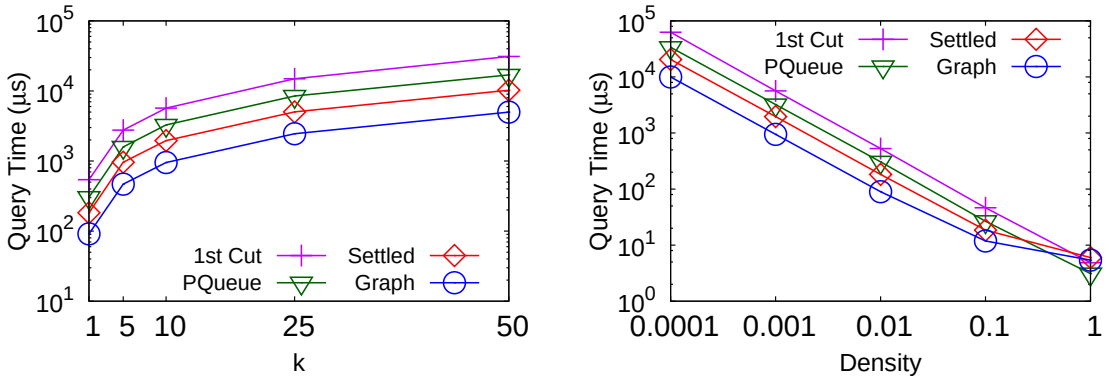
We investigate the cache efficiency of each implementation in CPU cache misses at each level in billions in Table A.1 (also showing INS, the number of instructions in billions) using `perf` hardware profiling of 250,000 varied queries on NW. Chained hashing uses indirection to access data, resulting in poor locality and the highest number of cache misses. Quadratic probing improves locality at the expense of more costly collision resolution, hence it uses more instructions than chained hashing. However, it cannot achieve better locality than storing data in an array sorted in the order it will be accessed. This ordering means the next value we retrieve from the array is far more likely to be in some level of cache. Unsurprisingly, it suffers from the fewest cache misses. This is a unique strength of G-tree's distance matrices and shows, while in-memory implementation is challenging, it is still possible to design algorithms that work well.

Distance Matrix	INS	Cache Misses (Data)		
		L1	L2	L3
Chained Hashing	953 B	28.8 B	20.5 B	13 B
Quadratic Probing	1482 B	11.2 B	7.5 B	5.3 B
Array	151 B	1.5 B	0.4 B	0.3 B

Table A.1: Hardware Profiling: 250,000 Queries on NW Dataset

A.3 Guidelines for Implementation Choices

In-memory implementation requires careful consideration, or experimental outcomes can be drastically affected as seen with G-tree's distance matrices and in [ŠJ14]. Many choices are quite simple, but their simplicity can lead to them being overlooked. Here we outline several choices and options to deal with them to assist future implementers. To illustrate the impact of these choices we progressively improve a first-cut in-memory implementation of INE. Each plot line in Figure A.3 shows the effect of one improved choice. Each roughly halves the query time, with the final implementation of INE being 6−7× faster.



(a) Varying k

(b) Varying Density

Figure A.3: Successive INE Improvement (NW,d=0.001,k=10)

1. Priority Queues. All methods in our study employ priority queues. In particular, INE and ROAD involve many queue operations and thus rely on their efficient implementation. Binary heaps are most commonly used, but we must choose whether to allow duplicate vertices in the queue or not. Without duplicates, the queue is smaller and queue operations involve less work. But this means the heap index of each vertex must be looked up to update keys, e.g., through a hash-table. On degree-bounded graphs, such as road networks, the number of duplicates is small, and removing them is simply not worth the lost locality and increased processing time incurred with hash-tables. As a result, we see a

2× improvement when INE is implemented without decreasing keys (see *PQueue* in Figure A.3). Note that we use this binary heap for all methods.

2. Settled Vertex Container. Recall INE and ROAD must track vertices that have been dequeued from their priority queues (i.e., settled). The scalable choice is to store vertices in a hash-table as they are settled. However, we observe an almost 2× improvement, as shown by *Settled* in Figure A.3 by using a *bit-array* instead. This is despite the need to allocate memory for $|V|$ vertices for each query. The bit-array has the added benefit of occupying 32× less space than an integer array, thus fitting more data in cache lines. This does add a constant pre-allocation overhead for each query, which is proportionally higher for small search spaces (i.e., for high density). But the trade-off is worth it due to the significant benefit on larger search spaces (i.e., low density).

3. Graph Representation. A disk-optimized graph data structure was proposed for INE in [Pap+03]. In main memory, we may choose to replace it with an array of graph vertex objects, with each object containing an adjacency list array. However, by combining all adjacency lists into a single array we can obtain another 2× speed-up (refer to *Graph* in Figure A.3). First, we assign numbers to vertices from 0 to $|V|-1$. An *edges* array stores the adjacency list of each vertex consecutively in this order. The *vertices* array stores the starting index of each vertex's adjacency list in *edges*, also in order. Now for any vertex u, we can find the beginning of its adjacency list in *edges* using $vertices[u]$ and its end using $vertices[u+1]$. This contiguity increases the likelihood of a cache hit during expansion. We similarly store ROAD's shortcuts in a global shortcut array, with each shortcut tree node storing an offset to this array. The principle demonstrated here is that the data structures recommended by past studies cannot be used verbatim. It is necessary to replace IO-oriented data structures recommended in the originally disk-based DisBrw and ROAD with in-memory performance in mind, e.g., we replaced the B^+-trees with sorted arrays.

4. Language. C++ allows more low-level tuning, such as specifying the layout of data in memory for cache benefits, making it preferable in high-performance applications. Implementers may consider other languages such as Java for its portability and design features. But when we implemented INE with all aforementioned improvements in Java (Oracle JDK 7), we found it was at least 2× slower than the equivalent C++ implementation. One possible reason is that Java does not guarantee contiguity in memory for collections

of objects. Also, the same objects take up more space in Java. Both factors lead to lower cache utilization, which may penalize methods that are better able to exploit it.

A.4 Summary

We investigated the effect of implementation choices using G-tree's distance matrices and data structures in INE. By investigating simple choices, we show that even small improvements in cache-friendliness can significantly improve algorithm performance. As such, there is a need to pay careful attention when implementing and designing algorithms for main memory. Furthermore, our insights are applicable to any technique not just those we study. In our setting, this is another factor to be wary of in the development efficient heuristics for POI search.

Appendix B

Improved kNN Algorithms

Our goal in Chapter 3, in comparing the state-of-the-art for the kNN problem, was to ascertain the true performance of each technique. To this end, we carefully inspected and applied numerous optimizations to each existing technique to ensure they performed as efficiently as possible. In contrast to implementation issues in Appendix A, these algorithmic improvements are applicable in any setting (in-memory or otherwise). In this appendix, we describe all modifications including pseudocode and, for major changes, experiments to demonstrate the margin of improvement. The improvements outlined in this appendix appeared in [ACT16b; ACT16a].

B.1 Distance Browsing

We have made several major changes to the DisBrw algorithm proposed in [SSA08]. Here we discuss several minor optimizations, correction of edge cases, and major improvements to DisBrw. Note that we propose an alternative algorithm to use the SILC index in Section 3.5.2 but describe these improvements to the original DisBrw algorithm for completeness. The updated pseudocode from [SSA08] is shown in Algorithm 11. Please refer to Section 3.3.3 and [SSA08; SAS05] for a detailed description of the algorithm and definition of subroutines.

One of the main improvements of DisBrw over the SILC kNN algorithm proposed in [SAS05] was the pruning of inserts by computing an upper bound D_k for the kth object. However, for any encountered object, DisBrw would still compute a distance interval (necessarily involving an $O(\log V)$ operation), before the insertion of that object

could be pruned. Since DisBrw already uses Euclidean distance to compute intervals for Object Hierarchy nodes, we additionally compute the Euclidean distance as a cheaper $O(1)$ initial lower bound for newly encountered objects (as in line 25). Thus, we are able to prune the insertion of many encountered objects, which cannot be better candidates, without computing distance intervals. Samet *et al.* [SSA08] also omitted computation of distance interval upper bounds for Object Hierarchy nodes. However, this is only a small additional expense when computing lower bounds. Instead, we compute upper bounds for nodes and use them to compute D_k sooner. Our Object Hierarchies also store the number of objects contained in each node (a simple additional pre-processing step in object index construction), which allows us to update D_k as shown in line 39. In this way, we also prune the insertion of nodes and avoid needless lower bound evaluations to prune objects in regions that cannot contain objects.

The DisBrw algorithm of [SSA08] does not handle several minor but possible edge-case scenarios, which we have corrected in Algorithm 11, as follows:

- In the original DisBrw algorithm, the if condition at line 12 was $UB_e \geq Front(Q)$, i.e., test if the upper bound of the dequeued element (UB_e) is greater than or equal to the next smallest lower bound in Q ($Front(Q)$). If true, DisBrw attempts to refine the bounds for e and re-insert it into Q. However, if the interval for e is fully refined (i.e., $LB_e = UB_e$) but we still have $UB_e = Front(Q)$, then we re-insert e into Q only to dequeue it and immediately re-insert it, leading to an infinite loop. Therefore, we change the condition at line 12 to $UB_e > Front(Q)$. The second part of the condition ensures that objects with the same upper bound are refined further, otherwise elements may be out of order in L.

- Let x be the object associated with D_k. Consider the scenario where x is at the front of Q (i.e., has the smallest lower bound) with $LB_x < UB_x$. Now when x is dequeued, if $UB_x > Front(Q)$ it will be refined. If $Front(Q)$ is associated with another object p, then p may be a kNN (as $LB_p = Front(Q)$ and before refinement $UB_x = D_k$, so $LB_p < UB_x$). If x is refined such that we then have $LB_x = UB_x = D_k$ (i.e., UB_x did not change), by the original algorithm, x will not be re-inserted into Q because the if condition at line 19 was $LB_e < D_k$. However, when p is dequeued next, we may have $UB_p < Front(Q)$ (as the next smallest lower bound is unknown). Then p

is dropped implicitly, potentially losing a correct kNN. This means when $LB_e = D_k$ it must still be inserted into Q, to ensure x is the last element dequeued before termination, and we change line 19 in the algorithm accordingly.

- We check L contains the dequeued object e before deleting it (line 13). If there are objects with the same upper bound as D_k, then e may not be in L.

B.1.1 Exploiting Vertices with Outdegree ≤ 2

Real road network graphs consist of large numbers of degree-2 and degree-1 vertices. Generally, 30% of vertices have degree-2 for road networks in Table 3.1, e.g., on the US dataset 30.3% of vertices (another 19.9% have degree-1). These may exist to capture details such as varying speed limits or curvature. Not considering this can have a significant impact on computing shortest paths, and we demonstrate the potential improvement on DisBrw.

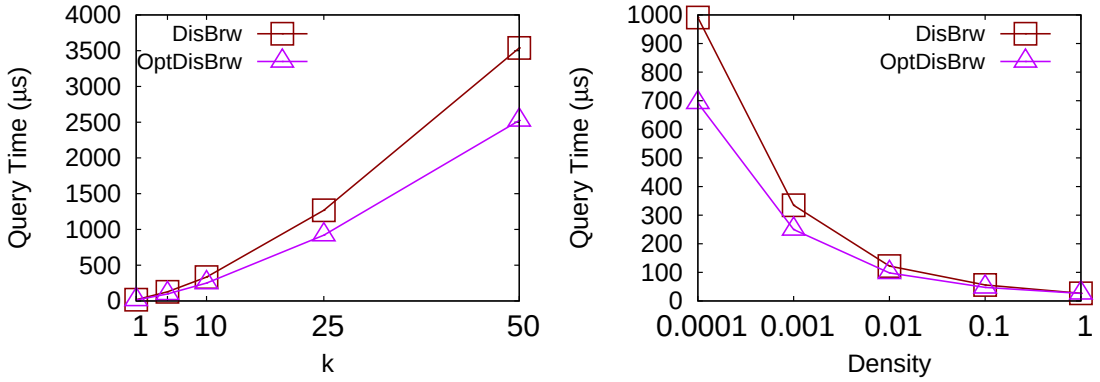
SILC uses the quadtrees and coloring scheme described in Section 3.3.3 to iteratively compute the vertices in a shortest path, at a cost of $O(\log |V|)$ for each vertex. We use *chain* to refer to a path consisting only of vertices with degree-2 or less, e.g., a section of motorway with no exits. Let v be the current vertex in the shortest path from s to t and u be the previous vertex in the shortest path. If v is on a chain, we do not need to consult the quadtree because the next vertex in the shortest path *must* be the neighbor of v that is not u. This saves $O(\log |V|)$ for each degree-2 vertex in the shortest path. In fact, if target t is not on the chain, we can directly "jump" to the last vertex in the chain saving several $O(\log |V|)$ lookups. This observation can be easily exploited by storing the two ends of the chain for each vertex with degree less than 2.

This optimization can significantly improve DisBrw query times. We refer to this version as OptDisBrw. For our default NW dataset this results in a 30% improvement as in Figure B.1, coinciding with the number of degree-2 vertices quoted above.

However, some road networks have an even larger proportion of degree-2 vertices, such as the highway road network for North America used in past studies [LLZ09; Lee+12; SSA08] with 175,813 vertices [Li05], 95% of which are degree-2. In this case, OptDisBrw is up to an order of magnitude faster than DisBrw as shown in Figure B.2, as the average chain length is significantly higher resulting in longer jumps. Accordingly, future work

Algorithm 11 Improved version of DisBrw kNN algorithm by Samet *et al.* [SSA08]

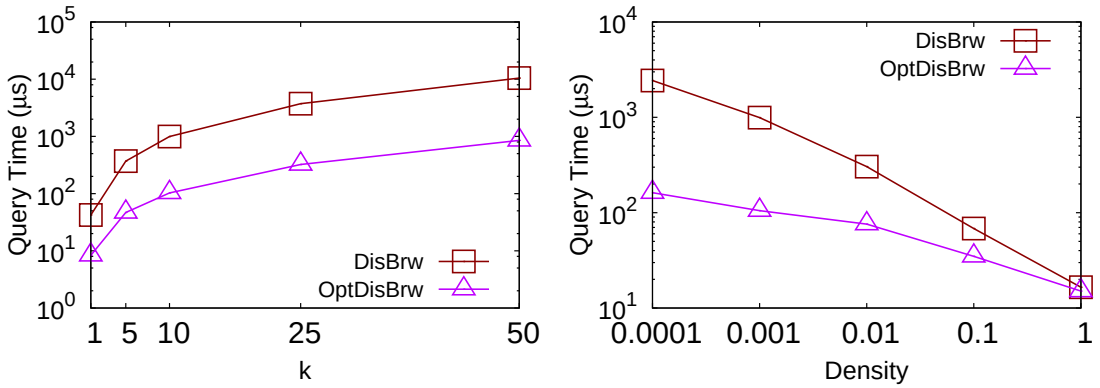
function GetKNNByDisBrw($v_q, k, SILC, OH$)
  Input: $SILC$ is SILC quadtree for v_q and OH is Object Hierarchy for object set
  Initialize min priority queue Q and max priority queue L
  Set upper bound on the kth neighbor $D_k \leftarrow \infty$
  $Enqueue(Q, ([OH.Root, 0, \_, \_], 0))$
  Note: Q elements also stores upper bound UB_e and (for objects) v_n the next intermediary vertex in the shortest path from v_q and d the distance to v_n from v_q
  while $Q \neq \phi$ **do**
    $([e, UB_e, v_n, d], LB_e) \leftarrow Dequeue(Q)$
    if $UB_e \geq D_k$ **then**
      break
    else if $IsObject(e)$ **then**
      if $UB_e > Front(Q)$ **or** $(UB_e = Front(Q)$ **and** $UB_e \neq LB_e)$ **then**
        if $UB_e \leq D_k$ **and** $Contains(L, e)$ **then**
          $Delete(L, e)$
        $(v_n, d, LB_e, UB_e) \leftarrow Refine(v_n, d, LB_e, UB_e)$
        Note: $Refine$ tightens bounds, and updates the next vertex and its distance (this only involves one binary search on the Morton List for current v_n)
        if $UB_e \leq D_k$ **then**
          $Enqueue(L, [e, UB_e)])$ and update L and D_k if needed
        if $LB_e \leq D_k$ **then**
          $Enqueue(Q, ([e, UB_e, v_n, d], LB_e))$
      else ▷ e dropped implicitly, no further refinement needed
    else ▷ Note: e must be an Object Hierarchy node
      if $IsLeaf(e)$ **then**
        for each object $v_o \in e$ **do**
          $LB_o \leftarrow EuclideanDistance(v_q, v_o)$
          if $LB_o < D_k$ **then**
            $(v_n, d, LB_o, UB_o) \leftarrow Refine(v_q, 0, LB_o, \text{inf})$ ▷ Initial v_n is v_q with d=0
            if $LB_o < D_k$ **then**
              $Enqueue(Q, ([o, UB_o, v_q, 0], LB_o))$
              if $UB_o < D_k$ **then**
                $Enqueue(L, [e, UB_e)])$ and update L and D_k if needed
      else
        for each child node $c \in e$ **do**
          if $NumObjects(c) > 0$ **then**
            $(LB_c, UB_c) \leftarrow ComputeInterval(LB_e, UB_e)$
            if $LB_c < D_k$ **then**
              $Enqueue(Q, ([c, UB_c, \_, \_], LB_c))$
              if $NumObjects(c) >= k$ **and** $UB_c < D_k$ **then**
                $D_k \leftarrow UB_c$
  $Populate(R, L)$ ▷ Deqeue from L to populate R so results are in distance order
  return R



(a) Varying k (b) Varying Density

Figure B.1: Deg-2 Optimization (NW,d=0.001,k=10)

must keep degree-2 vertices in mind for potential optimizations. Given these results, we use chain optimized refinement for DisBrw in our experiments.



(a) Varying k (b) Varying Density

Figure B.2: Deg-2 Optimization (NA-HWY,d=0.001,k=10)

B.2 G-tree

The G-tree kNN algorithm is largely unchanged from the more recent G-tree study [Zho+15], except for an improved leaf search algorithm we describe in Appendix B.2.1 below. Algorithm 12 includes an additional guard condition at line 10 as the *UpdateT* subroutine does not guarantee that Q will not be empty (i.e., when the new T_n's only child with an occurrence is the previous T_n, which we do not re-insert). Similar to how Association Directories are used by ROAD, we make a small modification to the algorithm to accept an Occurrence List OL rather than the set of objects O. The original algorithm implies that OL must be constructed for each query, which is not a trivial cost (as we

verify in Section 3.7.3) or how it would be used in practice. Note that again node refers to G-tree nodes and vertex refers to road network vertices. Please refer to Section 3.3.5 and [Zho+15] for descriptions of the data structures and explanation of the main algorithm.

Algorithm 12 Modified version of G-tree kNN algorithm by Zhong *et al.* [Zho+15]

```
1: function GetKNNByGtree(v_q, k, Gt, OL)
2:   Input: Gt is a G-tree and OL is an Occurrence List for an object set
3:   Initialize min priority queue Q and R ← φ
4:   if |OL(Leaf(v_q))| > 0 then
5:     R ← R ∪ GtreeLeafSearch(v_q, k, OL, Q, leaf(v_q))
6:   T_n ← Leaf(v_q) and set T_min for T_n
7:   while |R| < k and (Q ≠ φ or T_n ≠ Gt.Root) do
8:     if Q = φ then
9:       UpdateT(T_n, T_min, OL, Q
10:    if Q ≠ φ then
11:      (e, d) ← Dequeue(Q)
12:      if d > T_min then
13:        UpdateT(T_n, T_min, OL, Q)
14:        Enqueue(Q, (e, d))
15:      else if e is a vertex then
16:        R ← R ∪ e
17:      else if e is a node then
18:        for each node or vertex c ∈ OL(e) do
19:          Enqueue(Q, SPDist(v_q, c))
20:  return R
21: function UpdateT(T_n, T_min, OL, Q)
22:  Tn ← Tn.father and update T_min for T_n
23:  for each node c ∈ OL(T_n) do
24:    Skip loop if c is previous T_n
25:    Enqueue(Q, SPDist(v_q, c))
```

B.2.1 G-tree Leaf Search Improvement

We note that G-tree's query performance plateaus and sometimes increases for very high densities. Given a query vertex v_q, let G_q be the leaf subgraph containing v_q. Now the network distance to any object in G_q is the minimum of two possible shortest paths (a) one consisting of only vertices within G_q; and (b) one that leaves and re-enters G_q through some of its borders. To ensure correctness, the original G-tree algorithm performs Dijkstra's search limited to G_q until all objects are found, capturing network distances of type (a). For each object found it then computes network distances for type (b) paths by using the distance matrix to compare distances through borders. Recall that a leaf node contains at most τ vertices (e.g. τ=256 for NW and τ=512 for US datasets as in Table 3.1). For

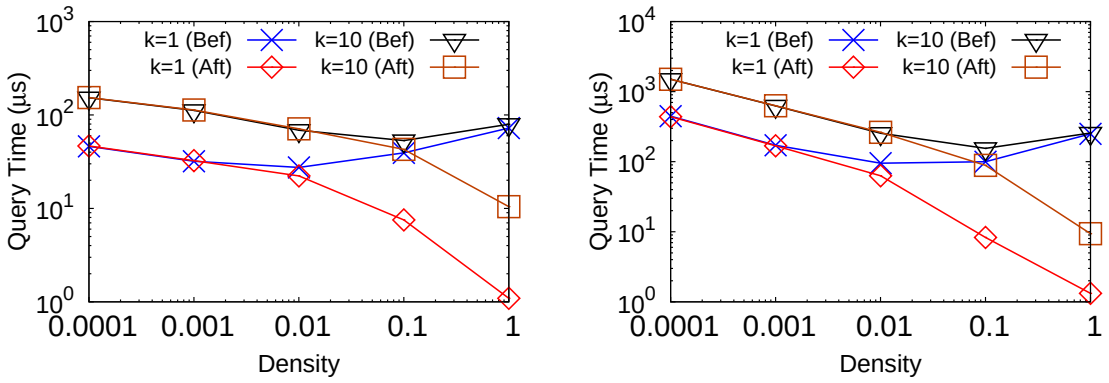
objects with density d there are on average $d \times \tau$ objects in each leaf node. If $d \times \tau > k$, G-tree computes distances of each type for more objects than necessary. The penalty is worse with increasing τ and decreasing k. INE cannot be applied to G_q because the kNNs within it may not be the global kNNs and some objects within G_q may be closer through paths that travel outside it.

We modify Dijkstra's search within G_q to capture paths of both types as shown in Algorithm 13. Let L be the priority queue used by this search. Our search continues until the first k leaf objects are settled (i.e. dequeued from L). Until the first border is settled, all settled objects are kNNs (like INE). This is correct as, without a closer border, no objects can have a shortest path that leaves G_q. But any subsequent object may not be a kNN, so we instead insert them into the priority queue Q used by the main G-tree algorithm. To ensure the distances consider paths that leave G_q, whenever we settle a border v_b we insert every other unsettled border v'_b of G_q into L. The distance to v'_b from v_q can be computed with the distance matrix.

Algorithm 13 Improved leaf-search subroutine for G-tree kNN algorithm

```
1:  function GtreeLeafSearch(v_q, k, OL, Q, R, G_q)
2:    Input: Q: queue used by Algorithm 12, G_q leaf node containing v_q
3:    Initialize min priority queue L
4:    Enqueue(L, (v_q, 0)), targetsFound ← 0, borderFound ← false
5:    while L ≠ φ and |R| < k and < k do
6:      (v_e, d) ← Dequeue(L)
7:      if not IsVisited(v_e) then
8:        if v_e ∈ OL(G_q) then
9:          targetsFound++
10:         if not border_found then
11:           R ← R ∪ v_e
12:         else
13:           Enqueue(Q, (v_e, d))
14:       borderFound ← RelaxLeafVertex(v_e, d, L, G_q))
15:       IsVisited(v_e) ← true
16:   return R
17: function RelaxLeafVertex(v_e, d, L, G_q)
18:   for each vertex v_a ∈ AdjacencyList(v_e) do
19:     if not IsVisited(v_a) and v_a ∈ G_q then
20:       Enqueue(L, (v_a, d + w(v_e, v_a)))
21:   if v_e ∈ Borders(G_q) then
22:     for each border v_b ∈ Borders(G_q) do
23:       if not IsVisited(v_b) then
24:         Enqueue(L, (v_b, d + G_q.DistMatrix(v_e, v_b)))
25:     return true
26:   return false
```

In Figure B.3 we see a significant speed-up for $k = 10$ and over an order of magnitude improvement for $k = 1$ on both datasets for the highest density. The improvement is even noticeable at lower densities for $k = 1$ on NW and both k on the US dataset as the leaf still contains far more objects than k. Note that this improvement is also applicable to other object distributions with the same density as leaf nodes will contain the same number of objects, on average.



(a) Varying Density on NW

(b) Varying Density on US

Figure B.3: Improved G-tree Leaf Search (k=10)

B.3 ROAD

The kNN search (Algorithm 14) and supporting algorithm (Algorithm 15) using the ROAD index are largely unchanged from [LLZ09]. We simplify these for the scenario where objects occur on vertices. In addition, we made a minor improvement by preventing unnecessary priority queue inserts for Rnet borders that have already been visited (see line 12 of Algorithm 15). The original algorithm re-inserts all borders into the priority queue Q, but then discards each of them immediately after dequeuing (as they are "visited"). This can be particularly expensive for larger Rnets (as they tend to have more borders). Refer to Section 3.3.4 and [LLZ09] for detailed descriptions and explanations of the main algorithm.

Algorithm 14 Modified version of ROAD kNN algorithm by Lee *et al.* [LLZ09]

```
1:  function GetKNNByROAD(v_q, k, RO, AD)
2:    Input: RO is a Route Overlay, AD is an Association Directory for an object set
3:    Initialize min priority queue Q and R ← φ
4:    Enqueue(Q, (v_q, 0))
5:    while Q ≠ φ and |R| < k do
6:      (v_e, d) ← Dequeue(Q)
7:      if not IsVisited(v_e) then
8:        if IsObject(AD, v_e) then
9:          R ← R ∪ v_e
10:       RelaxShortcuts(Q, RO, AD, v_e)
11:       IsVisited(v_e) ← true
12:   return R
```

Algorithm 15 Modified algorithm to relax shortcuts in ROAD based on [LLZ09]

```
1:  function RelaxShortcuts(v_e, d, Q, RO, AD)
2:    Input: Q queue used by Algorithm 14
3:    Initialize stack S
4:    T ← RO.GetShortcutTree(v_e)
5:    Push(S, T.Root)
6:    while S ≠ φ do
7:      n ← Pop(S)
8:      if ¬IsLeaf(n) then
9:        for each R ∈ Rnets(n) do
10:         if ¬HasObject(AD, R) then          ▷ Then this Rnet can be bypassed
11:           for each shortcut S(v_e, v_b) ∈ R do
12:             if ¬IsVisited(v_b) then
13:               Enqueue(Q, (v_b, d + |S(v_e, v_b)|))
14:         else
15:           for each child tree node c of n do
16:             Push(S, c)
17:     else
18:       R ← Rnets(n)                         ▷ Leaves have only one Rnet
19:       for each edge e(v_e, v_a) ∈ R do
20:         if ¬IsVisited(v_a) then
21:           Enqueue(Q, (v_a, d + w(v_e, v_a)))
22:   return
```