Scalable ride-sharing through geometric algorithms and multi-hop matching

by

Yixin Xu

ORCID:0000-0002-3135-7163

A thesis submitted in total fulfillment for the
degree of Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

February 2021

Abstract

Thanks to the ubiquitous access to the Internet, on-demand ride-sharing service has emerged to provide timely and convenient rides to passengers. Ride-sharing creates a win-win situation for all participants involved and brings substantial environmental benefits, thus becoming a prevalent transportation mode. A fundamental problem of ride-sharing is determining how to dispatch vehicles to passengers.

Efficiency and optimality are two core requirements for dispatching algorithms. Long response times would significantly impact the user experience, while sub-optimal matching results would substantially lower the operational efficiency. Finding optimal matches in a short time is challenging due to a large number of involved participants, the highly dynamic scenario, and the expensive road network distance computation.

This thesis proposes efficient and scalable algorithms to enable fast and high-quality dispatching. We observe that vehicles and passengers can only visit limited areas to ensure their latest arrival times. The limited areas can be used to quickly filter out infeasible vehicles. Such areas can be easily indexed and updated if represented as rectangles. Inspired by this key observation, we first propose an efficient algorithm to solve a basic problem in ride-sharing – how to quickly prune infeasible vehicles for passengers. Our algorithm ensures the finding of all possible candidates since we compute and index the confined visiting areas using ellipses, and the ellipses provide tight bound regardless of the underlying network. The effective pruning leads to only a small number of candidates remained, which substantially reduces the complexity of selecting the optimal matches and the overall matching time. We further investigate multi-hop ride-sharing algorithms that allow transfers on a user's trip. The algorithms search for possible transfers between vehicles by detecting whether the visiting areas of vehicles overlap. The proposed algorithms prune the combinations of vehicles and transfer points, thus overcoming the efficiency bottleneck and achieving real-time responses. We propose exact algorithms that guarantee the finding of optimal matches. We further improve the matching time using speed-up strategies. The enabled multi-hop trips largely enhance the flexibility of real-world ride-sharing, increase the number of successfully matched requests, and reduce the travel distance of vehicles. Lastly, we facilitate quick assigning of nearest pick-up/drop-off locations for passengers by proposing an efficient and scalable all nearest neighbor algorithm in road networks. We exploit the property of the problem such that finding a nearest neighbor only consumes constant time while only one graph traversal is required during the preprocessing phase.

The proposed algorithms in this thesis achieve orders of magnitude faster running time compared to the state-of-the-art algorithms. The auxiliary indices are lightweight and eliminate the high preprocessing and update cost for the highly dynamic scenarios.

Declaration of Authorship

- The thesis comprises only my original work towards the degree of Doctor of Philosophy;
- due acknowledgement has been made in the text to all other material used; and
- the thesis is fewer than the maximum word limit in length, exclusive of tables, maps, bibliographies and appendices as approved by the Research Higher Degrees Committee.

Signed: _____

Date: _____

Preface

Below is the list of publications and manuscripts arising from this thesis. I was the principle author of all papers and contributed more than 50% on each paper. I was responsible for designing the algorithms, collecting and pre-processing the datasets, implementing the comparing algorithms, running the experiments, analyzing the experimental results, and writing the paper drafts. My co-authors on each paper gave critical suggestions on the study design (e.g., defining the research problem, designing the algorithms and experiments), and contributed to the revisions of the manuscripts.

- Part of the contents of Chapter 4 has been published in the following paper:
  Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, Lars Kulik. *GeoPrune: Efficiently Matching Trips in Ride-sharing Through Geometric Properties*, International Conference on Scientific and Statistical Database Management (**SSDBM**) 2020.
- Part of the contents of Chapter 5 has been accepted and will appear in the following paper:
  Yixin Xu, Lars Kulik, Renata Borovica-Gajic, Abdullah Aldwyish, Jianzhong Qi. *Highly Efficient and Scalable Multi-hop Ride-sharing*, accepted by the International Conference on Advances in Geographic Information Systems (**SIGSPATIAL**) 2020.
- Part of the contents of Chapter 6 has been published in the following paper:
  Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, Lars Kulik. *Finding All Nearest Neighbors with a Single Graph Traversal*, International Conference on Database Systems for Advanced Applications (**DASFAA**) 2018.

Acknowledgements

It has been a long, challenging, and rewarding journey that would have been impossible without the aid and support of many people.

First and foremost, I would like to express my sincere gratitude to my supervisors Lars Kulik, Jianzhong Qi, and Renata Borovica-Gajic, for their continuous support, patient guidance, and enthusiastic encouragement during the journey. The suggestions I received from my meetings are precious for me. I can't forget the moments that I left from our meetings with full faith in my research and life while I was deeply frustrated before the meetings. I am extremely grateful to Lars for helping me out at my hardest time and smoothing my journey with wise advice and full support. His insightful ideas and constructive suggestions are invaluable for me to open my mind and light my research path in the right direction. He always encouraged me to look at the bright side in hard times and reward myself to celebrate success. His cheerful and great supports help me through many difficult times and more importantly teach me to always keep a positive attitude towards life. I am thankful to Jianzhong for his timely and detailed help on polishing my papers and helping me to write clearly, concisely, and precisely. The publications and this thesis would have been impossible without his careful polish. His rigorous scientific attitude teaches me to always be careful and thoughtful. I am grateful to Renata for her tremendous encouragement, care, and help. I am always impressive by her strong research ability and charming personality, and regard her as my role model. She gave me heaps of valuable suggestions on every writing, presentation, and discussion. I felt very heartwarming when she strove for every opportunity for me and encouraged me for even small achievements. Her cheering and encouraging words always inspire me to persist and strongly motivate me throughout the journey.

I am particularly grateful for the support and help from my Advisory Committee Chairs Justin Zobel and Richard Sinnott. I would have given up in the first year without Justin's help and wise advice. I am thankful to Richard for always giving me thoughtful suggestions during my progress reviews .

I feel lucky to meet many lovely friends who always support me during this journey. Special thanks to Abdullah and Shima for spending more than three years with me in the lab and creating a wonderful office environment. I would like to thank Ge Yao, Wei Gao, Li Li, Nairu Geng for the fun hangouts and warm moments. I would also like to thank other fellow students, Alex, Aref, Tabinda, Oscar, Estrid, Soheila, Somayeh, Daniel, for sharing the ups and downs of our Ph.D. journeys and giving me helpful advises. I would like to express my gratitude to the University of Melbourne for offering me the scholarship for covering my living expenses and conference travels.

I would like to offer my special thanks to my parents and husband for their unconditional love, faith, and support. I am indebted to my parents for encouraging me to pursue my dream and giving me full support whenever and wherever I am. I owe my deepest gratitude to my husband, Dong, for always giving me full support and be my side. Although being 700km away geographically, I feel we were never apart.

Contents

List of Figures

List of Tables

Acronyms

NN Nearest Neighbor Query

ANN All Nearest Neighbor Query

kNN k Nearest Neighbor Query

GNN Group Nearest Neighbor Query

PCD Pre-computed Cluster Distance

Hiti Hierarchical Multi

TNR Transit Node Routing

CH Contraction Hierarchy

CPD Compressed Path Database

HL Hub-based Labeling

PLL Pruned Landmark Labeling

PHL Pruned Highway Labelling

MLP Multilayer Perceptron

SILC Spatially Induced Linkage Cognizance

COLT Compacted Object-Landmark Tree

NN-join Nearest Neighbor Join

MNN Multiple Nearest Neighbor

BNN Batched Nearest Neighbor

SL-Tree Subgraph-Landmark Tree

DARP Dial-A-Ride-Problem

DSA Dual-Side Search Algorithm

DAG Directed Acyclic Graph

DSB-Score Demand-Supply Balance Score

TEG Time-Expanded Graphs

MBR Minimum Bounding Rectangle

NYC New York City

CD Chengdu

UTM Universal Transverse Mercator

MSE Mean Square Error

IoT Intersection over True

VIVET **Vi**rtual **ve**rtex **t**raversal

Chapter 1

Introduction

1.1 Background

We are witnessing a transportation revolution led by an innovative transportation model – ride-sharing. Due to the widespread use of mobile devices, user locations can now be shared in real-time between passengers and drivers, which gives rise to on-demand transportation such as ride-sharing. In ride-sharing, passengers request rides via an app or website and get served shortly after requesting, with the possibility to share the ride with other passengers heading towards the same or similar directions.

The ride-sharing market is experiencing a rapid growth in recent years. Uber, the largest ride-sharing service provider, has expanded its geographical footprint from less than 100 cities in 2014 to more than 900 cities nowadays [1, 2]. The company completes 50 times more trips annually in 2018 than in 2014. A leading competitor, Lyft, has expanded from 60 cities in 2014 to 656 cities today [3]. In New York City, Uber and Lyft provided more than double rides compared to traditional taxi services in January 2019 [4]. Ride-sharing has become a popular transport option, and the market is rapidly growing. The compound annual growth rate of the ride-sharing market will reach 19.2% over the forecast period 2020-2025 [5]. Shared ride service is an important service provided by the ride-sharing companies. In 2016, 20% rides of Uber are completed by the shared ride service UberPool [6].

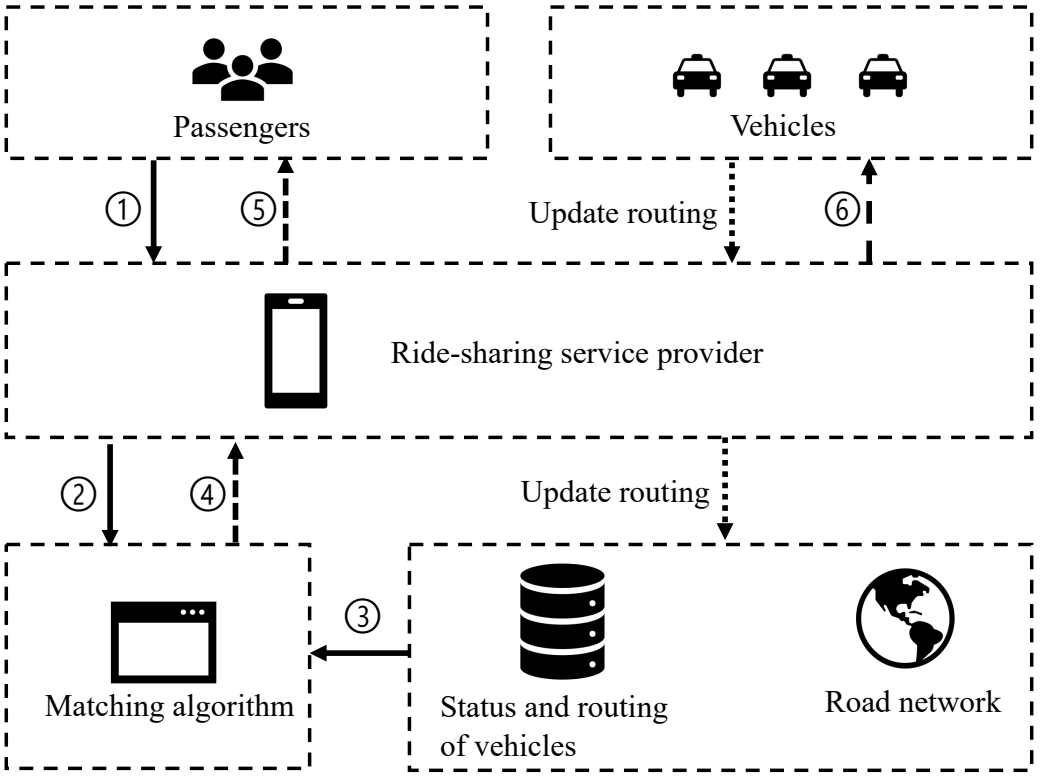


FIGURE 1.1: Framework of the dynamic ride-sharing. ① Passengers send requests to the service provider. ② The service provider proceeds the request to the matching algorithm. ③ The matching algorithm retrieves information of vehicle and the road network. ④ The optimal matches are computed and returned by the matching algorithm. ⑤ The passengers get notification. ⑥ The vehicle is informed about the new requests and the new scheduled routing.

Figure 1.1 shows how requests are served in ride-sharing. New passengers send their requests to the service provider via an app or website. Then, the service provider invokes a matching algorithm (also called dispatching algorithm in the literature) to compute the optimal matches (dispatches) of passengers, considering both the request constraints (such as their latest arrival times) and the status and routing of vehicles. The optimal matches are sent back to the passengers, and vehicles are notified with the information of the new requests along with the new scheduled routing. The records for vehicles need updates when the routing of vehicles changes due to the newly assigned requests or the continuous object movement.

Ride-sharing provides key advantages over traditional transportation models. Ride-sharing enables passengers to be quickly served by simply swiping or tapping on the mobile phones, instead of waiting on the street for long periods to hail a car. Besides, sharing the route with other passengers helps to reduce the cost per passenger and makes

rides more affordable. As for the global benefits, ride-sharing can be an effective way to address urban traffic problems and environmental issues. Shared vehicles can reduce the number of required vehicles, which helps to reduce traffic congestion, lower pollutant emissions, and free unnecessary parking spaces [7, 8].

1.2 Motivation and Challenges

Providing on-demand services to users while achieving optimal operating cost is challenging for ride-sharing service providers, and there remain many key problems to be solved. One fundamental task is to determine how to match vehicles with passengers after receiving the service requests. The matching results not only impact the experience of passengers but also influence distribution of vehicles in the future. Thus, matching algorithms are the key to the system performance. They are the core theme of this thesis.

Adopting a brute-force approach, a matching algorithm can enumerate the matching between every vehicle and every request. However, the ride-sharing process involves a large number of vehicles and requests and the number of combinations between them is huge. For example, in 2019, the average number of active vehicles is more than 70,000 and the number of trips exceeds 900,000 per day in New York City [9]. Exhaustively checking every vehicle-request pair is too expensive to meet the real-time requirement. Besides, a vehicle may be scheduled to serve several requests. It is costly to examine the detour times of all affected requests due to expensive computations when computing the detour cost and assessing the constraints. Thus, the matching process is computationally hard and the brute-force approach requires a long running time before responding to passengers. In another extreme scenario, a matching algorithm can randomly match vehicles with requests without comparing the matching quality. However, random matches may be sub-optimal and lead to poor riding experience of users, e.g., lower matching possibilities, longer waiting times, and delayed arrival times. Besides, such a method cannot fully exploit the capacity of vehicles and the fewer matched requests and longer travel distance may increase the operating cost of the service provider and in turn for the passengers.

Advanced matching algorithms are needed to meet two core requirements: *efficiency* and *optimality*. The efficiency requirement specifies that the running time of the matching algorithms should be short. Achieving optimal matching results, on the other hand, helps the service provider to utilize the limited resources, save the operating cost, and eventually profit higher. Typical metrics measuring the optimality include the number of satisfied requests, the travel distance of vehicles, and the revenue of the service provider.

Designing efficient matching algorithms with high quality results is challenging due to the following reasons.

- **Challenge 1: Expensive road network distance computations.** In ride-sharing, the movement of vehicles is constrained by the road network. The travel time/distance between two locations cannot be determined simply by their Euclidean distance but depends on their shortest paths that consider the connecting roads. The matching process needs to call the shortest path computation numerous times due to the large number of vehicles and passengers, which requires frequent and costly traversal on the network structure. Traditional shortest path algorithms such as Djikstra's algorithm [10] are too slow due to the high computational complexity $O(|E| + |V| \log |V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices in the graph. Although many speed-up algorithms have been proposed to reduce the search space [11–24], these algorithms typically rely on large indices to store a big volume of distance information in exchange for shorter query times, thus suffering from poor scalability. An efficient matching algorithm should eliminate unnecessary shortest path computation and avoid expensive traversal on the network.
- **Challenge 2: Highly dynamic scenarios.** Matching vehicles in ride-sharing is a highly dynamic process. First, the service provider receives requests frequently and the information of future requests is unknown. It is difficult to find the optimal matches for all requesting passengers in a short time. Further, serving these continuously incoming requests incurs frequent changes to the routing schedules of vehicles. It is challenging to update the routing schedule in real-time. Besides, the continuous movement of vehicles and constant updates in their occupancy status require real-time maintenance in ride-sharing algorithms.

- **Challenge 3: Multi-party constraints.** Matching in ride-sharing needs to consider various constraints. One prominent constraint is the service time of passengers. Due to the sharing nature, passengers may need to reach the pick-up/drop-off locations of other passengers while on-board, which causes detours and delays. A ride-sharing system should guarantee reasonable detour time of all served passengers such that they can arrive at their destinations on time. A vehicle may already be committed to serve several requests and must visit pick-up/drop-off locations of committed requests on time. Therefore, the time constraints to be considered not only result from the passengers but also vehicles. Another consideration is the capacity limit of the vehicles. The number of on-board passengers cannot exceed the capacity of the vehicles at any time. The matching algorithm needs to assess the time and capacity constraints of all passengers and vehicles in a short time. One solution is to assess every vehicle individually [25–27]. However,the exhaustive search requires a long running time and suffers from low efficiency. Existing studies show that using indices helps to quickly identify potential vehicles and improve the efficiency [28, 29]. However, these indices cannot accurately evaluate the constraints and may miss potential vehicles. Designing an index that can efficiently and accurately assess the constraints of all vehicles so as to filter out infeasible vehicles is thus a non-trivial task.
- **Challenge 4: Complexity of optimization.** The optimal operating of ride-sharing largely depends on the matching strategy. An optimal matching algorithm should improve the satisfaction of passengers. Besides, it is important to reduce the resource consumption of ride-sharing so as to bring environmental benefits and lower the operating cost of the service provider, e.g., reduce the number of required vehicles and shorten the travel distance of vehicles. Improving the performance of a ride-sharing system is a key and challenging problem since ride-sharing is a complicated process. Ride-sharing involves a large number of passengers and vehicles, and each of them has its own benefit and constraint. Besides, the matches of requests affect each other. A match influences the status of vehicles and passengers, which further impacts the matching results of other passengers.
- **Challenge 5: Various optimization goals.** The matching should consider various factors such as the waiting and arrival times of passengers, the income of drivers, and the revenue of the platform. Existing studies are usually proposed based on specific optimization goals [26–28, 30–32], making it challenging for them

to be adopted to other optimization objectives. A matching algorithm should be generic and applicable to various optimization goals considering the needs in different scenarios.

Although various matching algorithms have been proposed in the literature to address the discussed challenges, there still remain important research gaps for highly efficient and effective ride-sharing.

1.3 Research Problems

We investigate three critical research problems in ride-sharing match, aiming to improve both efficiency and matching quality.

Research problem 1. *Efficiently finding vehicles for every passenger in ride-sharing.* The real-time response to passengers is critical in ride-sharing. Typical matching algorithms run in two phases: pruning and selection. The pruning phase prunes infeasible vehicles and the selection phase determines the match results among the remaining vehicles. The pruning phase substantially reduces the number of vehicles to be checked in the selection phase, thus is crucial for improving the overall matching time. A superior pruning algorithm must support fast pruning of potential vehicles and achieve low update cost for the highly dynamic scenarios. Existing pruning algorithms need to store and update a large size of information, thus requiring expensive update cost. We aim to design an efficient and scalable pruning algorithm such that potential vehicles can be quickly identified so as to improve the matching efficiency. To tackle Challenge 1, we aim to design a pruning criteria that prunes vehicles without expensive shortest path computation. To address Challenge 2, the pruning algorithm achieves low updating costs for the highly dynamic scenarios. Challenge 3 is overcome by quickly assessing the time constraints of vehicles and passengers. We overcome Challenge 5 since the proposed pruning algorithm is applicable to various optimization goals.

Research problem 2. *Developing efficient multi-hop matching algorithms.*
Most of existing studies assume passengers can be served by only one vehicle during a trip. However, finding direct trips for passengers may be difficult due to the limited supply and constrained movement of vehicles, leading to sub-optimal matching and low flexibility. One effective strategy to overcome the limitation is the multi-hop ride-sharing

that allows transfers between vehicles. Enabling multi-hop creates more matching options for the passenger and hence improves the chance to be served. Moreover, allowing transfers generates shorter trips that are more likely to find shared routes, leading to reduced detour cost and travel distance. Multi-hop ride-sharing extends the benefits of single-hop ride-sharing but also adds extra computational complexity, resulting from the high number of combinations of vehicles and transfer points. Existing multi-hop ride-sharing matching algorithms exhaustively check all matching possibilities, thus lacking efficiency and are impractical in real-world scenarios. We aim to propose efficient and scalable algorithms that find multi-hop ride-sharing trips for passengers in real-time on large networks. This research problem aims to improve the performance of ride-sharing systems and address Challenge 4.

Research problem 3. *Efficiently identifying the pick-up/drop-off location of passengers.*

In research problem 1 and research problem 2, we study a general scenario similar to most of existing ride-sharing studies [8, 25, 25–27, 27–29, 33–37] in which passengers can be picked up and dropped off at any locations, e.g., their original source and destination locations. In real-world ride-sharing applications such as UberPool and LyftLine, the original source and destination of passengers may be inaccessible or at prohibit parking spots. Assigning pick-up and drop-off locations enables safer boarding and helps to reduce the detour costs of vehicles and save the service time of passengers. A critical problem in such a setting is how to efficiently find the nearest pick-up and drop-off locations for every passenger. Such a problem is an application of a fundamental query problem in road networks, i.e., all nearest neighbor query. Given a set of data points O and a set of query points Q, ANN aims to find the nearest data point $o_j \in O$ for every query point $q_i \in Q$. Finding the nearest neighbors is essential in various location-based service applications. Despite its importance, studies on the ANN query mainly focus on the Euclidean distance. To the best of our knowledge, no previous works investigate the ANN query in road networks that model the movement of vehicles more realistically. We propose the first efficient and scalable ANN algorithm in the literature. Our proposed ANN algorithm achieves fast query time and incurs low pre-processing cost. It overcomes Challenge 1 since it takes simple look-ups in the query phase and only traverses the graph once in the pre-processing phase, thus avoiding expensive shortest path computation.

1.4 Contributions

In this thesis, we make the following contributions.

Contribution 1. *An efficient and scalable match pruning algorithm.*
To address the first research problem, we propose an algorithm to support efficient and effective pruning of infeasible vehicles in ride-sharing in Chapter 3. Instead of storing and maintaining the status and routing of a large amount of vehicles as existing studies, our algorithm saves the computation and updating overhead by taking advantage of geometric objects (ellipses and circles). We introduce a concept called the *reachable area* to indicate the area the vehicles/passengers can possibly visit without violating their time constraints. We then prove that the reachable areas are bounded by ellipses that can be quickly computed and retrieved. The bounding ellipses enable possible vehicles to be quickly identified with low update costs for the highly dynamic scenarios. Existing pruning algorithms can only provide approximation results, i.e., they may falsely filter out some possible vehicles and cause sub-optimal matches. In contrast, our pruning algorithm guarantees finding of all possible vehicles for every passenger as the ellipses bound the reachable areas. The experiments on real-world data show that our algorithm prunes an order of magnitude more vehicles and reduces the update time by two to three orders of magnitude compared to state-of-the-art algorithms.

Contribution 2. *Efficient and scalable multi-hop ride-sharing.*
In Chapter 4, we design efficient and scalable algorithms to offer passengers multi-hop ride-sharing options in real-time, addressing the second research problem. Existing multi-hop algorithms are inefficient as they need to enumerate all possible matching possibilities, thus not being applicable to real-world scenarios. To fill the research gap, we design efficient and scalable multi-hop ride-sharing algorithms to reduce the large search space and improve the query time. The experiments show that the proposed algorithms are two orders of magnitude faster than the state-of-the-art multi-hop algorithms. We further speed up the proposed algorithms by another order of magnitude using techniques such as deep learning, while preserving a comparable matching quality of the system. Our algorithms help to verify the benefits of multi-hop ride-sharing in real-world scenarios experimentally. Allowing transfers in ride-sharing brings up to 10% more requests to be matched that would have been abandoned otherwise. This

corresponds to more than ten thousand requests to be matched that were previously discarded in big cities such as Chengdu and NYC everyday.

Contribution 3. *Efficient and scalable all nearest neighbor algorithms in road networks.* In Chapter 5, we study the ANN query in road networks, tackling the third research problem. Although an ANN query can be solved by applying a separate nearest neighbor algorithm on each query point, existing nearest neighbor algorithms either require long query times or large indices. Besides, such a method causes redundant computations as the same areas may be visited multiple times. We propose a novel algorithm that accelerates both the pre-processing cost and the query time by orders of magnitude. We carefully analyze the properties of the proposed problem such that the pre-processing phase only requires one traversal over the network to obtain the nearest neighbor of each vertex in the road network. We then use a lightweight index to store the nearest neighbor and distance label for each vertex. The memory consumption of the index is thus only linear to the number of vertices in the road network and much lower than other indices in road networks. Based on the index, answering an ANN query only requires simple look-ups without extra computations, yielding constant query time. The experiments confirm that our proposed algorithm outperforms the state-of-the-art algorithms by one to two orders of magnitudes in terms of both query time and pre-processing cost.

1.5 Thesis Organization

The organization of this thesis is outlined below.

- **Chapter 2** surveys related work on computing the road network distance, finding nearest neighbors, and finding ride-sharing matches.
- **Chapter 3** defines the ride-sharing matching problem and the related concepts.
- **Chapter 4** proposes a pruning algorithm for ride-sharing matching, which leverages the geometric objects to efficiently retrieve potential vehicles while requiring low updating costs for the highly dynamic scenarios.

  This work is summarized in the paper: Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, Lars Kulik. *GeoPrune: Efficiently Matching Trips in Ride-sharing Through Geometric Properties*, International Conference on Scientific and Statistical Database Management (**SSDBM**) 2020.

- **Chapter 5** proposes efficient and scalable algorithms to find multi-hop trips for passengers in ride-sharing, which achieve real-time responses and largely enhance the flexibility of the ride-sharing system.

  This work is summarized in the paper: Yixin Xu, Lars Kulik, Renata Borovica-Gajic, Abdullah Aldwyish, Jianzhong Qi. *Highly Efficient and Scalable Multi-hop Ride-sharing*, accepted by the International Conference on Advances in Geographic Information Systems (**SIGSPATIAL**) 2020.
- **Chapter 6** proposes a novel algorithm for the ANN queries in road networks, which only requires one traversal on the graph in the pre-processing phase and answers the nearest neighbors in constant time.

  This work is summarized in the paper: Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, Lars Kulik. *Finding All Nearest Neighbors with a Single Graph Traversal*, International Conference on Database Systems for Advanced Applications (**DASFAA**) 2018.
- **Chapter 7** summarizes our key findings and provides potential directions for future research.

Chapter 2

Related Work

In ride-sharing, the movement of vehicles is constrained by the road network. The shortest path determines the distance between locations and impacts the routing of vehicles, thus is essential in ride-sharing. In Section 2.1, we discuss shortest-path algorithms in road networks. We further survey nearest neighbor algorithms in road networks in Section 2.2. The nearest neighbor algorithms aim to find the nearest objects for query locations, which have various applications in ride-sharing. An example of the nearest neighbor query is finding the nearest pick-up/drop-off points for a passenger. We discuss three typical types of nearest neighbor queries in road networks that are related to our work: Nearest Neighbor Query (NN), Group Nearest Neighbor Query (GNN), and All Nearest Neighbor Query (ANN). We further discuss skyline queries in road networks in Section 2.2. A skyline query aims to find a set of dominant locations in respect of given criteria, for example, finding hotels with close distance to the beach and cheap price. In Section 2.3, we discuss dispatching algorithms of ride-sharing. We review existing techniques to accelerate the matching time of ride-sharing, and then survey algorithms to improve the matching quality based on different optimization objectives. In Section 2.4, we review flexible ride-sharing that brings more flexibility to passengers by optimizing meeting points or allowing transfers between vehicles.

2.1 Shortest Paths in Road Networks

In this section, we review shortest-path algorithms in road networks. Given a road network $G = \langle V, E \rangle$, where V denotes a set of vertices, E denotes a set of edges, and the edge weight between two vertices denotes their travel cost (travel time or travel distance), the shortest path problem aims to find a path with the minimum sum of edge weights from a given source vertex s to a given destination vertex d. We denote the shortest path distance (also called network distance) following the road network from s to d as $d_n(s, d)$ and the shortest travel time as $t(s, d)$. We also represent the Euclidean distance between s to d as $d_e(s, d)$.

2.1.1 Classic Shortest-path Algorithms

We first review three classic shortest-path algorithms. Many of the speed-up techniques are proposed on the basis of these algorithms.

Dijkstra's algorithm. The *Dijkstra's* algorithm [10] gradually expands the search space from the source vertex following the connecting edges and iteratively improves the travel distance of visited vertices. Dijkstra's algorithm records three pieces of information: an array *unsettled* marking whether the shortest-path distance of a vertex is settled (i.e., shortest-path distance finalized) or not; an array *distance* recording the current shortest-path; a heap $\mathcal{H}$ ranking all vertices that are visited but not settled yet, where the keys are their current shortest-path distance values.

In the initial stage, Dijkstra's algorithm marks the source vertex as visited and sets its distance value as zero. All other vertices are marked as unsettled and assigned with infinity distance values. The algorithm initializes the heap $\mathcal{H}$ with only the source vertex (with key value zero).

To find the shortest path, the algorithm gradually expands the search area to reach the destination. At each iteration, it extracts the minimum element $(m, dist)$ from the heap $\mathcal{H}$, where m represents the extracted vertex and $dist$ is the key value. It marks the extracted vertex m as settled, meaning that the shortest-path distance of m is found as $dist$. For every adjacent vertex m' of m, the algorithm checks whether the shortest-path distance of m' can be improved through m and updates the distance value, i.e.,

$distance[m'] = min(distance[m'], distance[m] + w(m, m'))$. The algorithm updates the key value of m' in $\mathcal{H}$ if m' is already in the heap. Otherwise, the algorithm inserts m' into the heap with the found optimal distance $distance[m']$ as the key.

The complexity of Dijkstra's algorithm is $O(|E| + |V| \log |V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices in the graph.

Bellman-Ford Algorithm. The *Bellman-Ford* [38] algorithm finds the shortest path from a source vertex to all other vertices. The algorithm records an array *distance* to update the optimal shortest travel distance of each vertex. Initially, the algorithm sets the *distance* of the source as zero and that of all other vertices as infinity. The algorithm needs to iterate for $|V| - 1$ times to get the shortest path distances. Each iteration traverses all edges. When traversing an edge $e(u, v)$, it updates the distance of v as $distance[v] = min(distance[v], distance[u] + w(u, v)$. After the ith iteration, the algorithm is guaranteed to find the shortest path with at most i edges on the path. The shortest paths consist of at most $|V| - 1$ edges, which is guaranteed to be found after $|V| - 1$ time iterations.

The complexity of the Bellman-Ford algorithm is $O(|V||E|)$, which is higher than the Dijkstra's algorithm. One benefit of the Bellman-Ford algorithm is that it can handle graphs with negative edge weights.

Floyd-Warshall Algorithm. The Floyd-Warshall [39] algorithm computes the shortest paths between all pairs of vertices. It maintains a matrix *distance* to record the pairwise distance of vertices and initializes *distance* as the graph adjacency matrix. It updates the matrix by gradually adding the intermediate vertices. When examining vertex k, the algorithm already finds shortest paths considering vertices $0, ..., k-1$ as the intermediate points. It checks whether considering k as the intermediate vertex will improve the shortest path distance, i.e., $distance[i][j] = min(distance[i][j], dist[i][k] + dist[k][j])$.

The Folyd-Warshall algorithm conducts $|V|$ iterations and each iteration checks the distance improvement of all pairs of vertices ($|V|^2$). Thus, the complexity of the Folyd-Warshall algorithm is $O(|V|^3)$.

The most commonly used graph shortest-path algorithm is Dijkstra's algorithm, due to its simple procedure. As we will illustrate later, many of the speed-up algorithms are developed on the basis of the Dijkstra's algorithm. Other speed-up techniques cache the

shortest paths between every pair of vertices such that the shortest path query can be looked up through a distance table. In such cases, the Floyd-Warshall may be applied in the pre-processing phase to obtain the shortest path distance between all pairs of vertices.

2.1.2 Index-free Speed-up Techniques

The classic shortest path algorithms need to detect all edges connecting the source and destination locations and thus require expensive graph traversals. Hence, these algorithms are too slow to meet the real-time requirement in location-based services. Many speed-up techniques are proposed to improve the response time. These algorithms can be classified into two types: index-free and index-based. The index-free algorithms use heuristics to reduce the search space and achieve shorter query time without requiring index and pre-processing cost. In contrast, the index-based algorithms pre-compute distance information and store them in indices to enable quick retrieval and save the computational cost. In what follows, we first review the index-free speed-up algorithms and then discuss the index-based algorithms.

Bidirectional Dijkstra's algorithm [40] runs two Dijkstra's algorithms from the source and destination concurrently. When the two Dijkstra's algorithms hit each other, i.e., their search scopes overlap, the algorithm terminates and the shortest path is guaranteed to be scanned. Note that the first vertex scanned by both sides may not be on the shortest path. Therefore, to retrieve the shortest path, the algorithm needs to check all vertices scanned from both sides. The bidirectional search helps to reduce the search space by roughly half [41].

*A** [42] applies an important property of shortest path distance: the road network distance between two locations must be no shorter than their Euclidean distance, i.e., $d_n(v_i, v_j) \geq d_e(v_i, v_j)$. It uses the Euclidean distance as the lower bound to guide the search towards the destination in the Dijkstra's algorithm. Specifically, when the expansion reaches a vertex v, the sub-path from the source s to v $(s - v)$ is explored while the sub-path from v to the destination d $(v - d)$ remains unknown. Although undetermined, the network distance from v to d $(v - d)$ must be longer than their Euclidean distance, i.e., $d_n(v, d) \geq d_e(v, d)$). A lower bound of the whole path passing through v is thus $d_n(s, v) + d_e(v, d)$. A* uses this lower bound to determine the key

value for v in the priority queue $\mathcal{H}$, i.e., the key value of v in the heap $\mathcal{H}$ is set as $d_n(s, v) + d_e(v, t)$. The search is thus guided to visit areas close to the shortest path instead of spreading from the source in the original Dijkstra's algorithm. The lower bound is computed by dividing the Euclidean distance by the maximum speed value if the edge weights represent travel times.

2.1.3 Index-based Speed-up Techniques

The index-free speed-up techniques apply the heuristics to reduce the search space and accelerate the running time. However, the required graph traversal is too expensive to meet the real-time requirement in large road networks. To achieve faster query time, index-based algorithms are proposed to avoid the graph traversals. The index-based algorithms involve a pre-computation phase and build an index to trade-off quicker query time. The distance information stored in indices can be directly retrieved without the need for re-computation, thus saving the computation cost in the query phase.

Landmark-based Algorithms.

ALT [43] uses a similar idea of the A* algorithm [42]. Motivated by the limitation that the Euclidean distance may only provide loose lower bound, ALT determines the lower bound using a set of pre-determined landmarks. Its pre-computation phase selects a set of vertices as landmarks and pre-computes the distance from every landmark to all vertices in the road network. The road network distance from a vertex v_i to another vertex v_j must be larger than both $d_n(v_i, l_i) - d_n(v_j, l_i)$ and $d_n(l_i, v_i) - d_n(l_i, v_j)$ for any landmark l_i. Compared to using Euclidean distance as lower bounds, ALT provides tighter lower bounds as it considers the road network distance between locations. The advantage of ALT is more obvious when the edge weights represent the travel times. However, ALT consumes larger memory consumption as it requires a larger space to store the distance between landmarks and vertices. The selection of landmarks is crucial to the performance of ALT [44, 45].

Partition-based Algorithms.

Partition-based algorithms pre-compute distance between partitions on the space. *Arc-flags* [11] partitions the space into k cells. Each cell has roughly the same number of vertices. It then records a k-sized vector for each edge to mark the leading cells of

edges. For the vector of an edge e, its i_{th} bit marks whether e is a correct edge to expand heading to the i_{th} cell, i.e., the i_{th} bit of e is settled as true if e lies on a shortest path to any vertex within the i_{th} cell. The search algorithm of Arc-flags is similar to the Dijkstra's algorithm. However, rather than expanding all connecting edges from the source s, it only expands edges that settle the bit value indicating the cell of the destination d as true.

Pre-computed Cluster Distance (PCD) [12] partitions the network into k cells in a similar way of Arc-flags. However, instead of indicating the heading cells of edges as in Arc-flags, PCD pre-computes the minimum shortest path distance between every pair of cells. Meanwhile, PCD identifies boundary vertices that are linked to areas outside of the cell through connecting edges. It then pre-computes the distance from all internal vertices (no direct connection to outside areas of the cell) to all boundary vertices. PCD then computes the distance lower bound through a vertex u as $d_n(s,u) + d_n(C(u), C(d) + d_n(v,d)$, where $C(u)$ and $C(d)$ are cells containing u and d respectively and v is the closest boundary of d. The search algorithm of PCD is similar to A* and ALT with a different lower bound computation.

Hierarchical Multi (Hiti) [13, 14] also partitions the network into cells and identifies border vertices. Hiti creates a smaller overlay graph that contains only border vertices of cells and edges connecting these border vertices. The search algorithm of Hiti is then conducted on the overlay graph together with the cells containing the source and destination vertices.

Transit Vertex-based Algorithms

The *Transit Node Routing (TNR)* [15, 16] algorithm selects a set of transit nodes and pre-computes the pairwise distance between the transit nodes. Then, for every other vertex v, TNR computes its access nodes $A(v)$ and the distance to these access nodes. A transit node u is an access node of a node v if there exists a shortest path from v on which u is the first transit node. The shortest path between s and d can then be referred as the path minimizing the distance $s - A(s) - A(d) - d$. When s and d are close to each other, their shortest path may not hit any transit nodes. Such a query is called a local query and will be checked by TNR using a locality filter before searching. If the query is likely to be a local query, the searching applies Dijkstra's algorithm.

Hierarchy-based Algorithms

Hierarchy-based approaches are motivated by the observation that vertices (or edges) traversed by more shortest paths are more likely to lie on a shortest path and thus are more important than other vertices in the shortest path search. For example, the shortest path between two distant locations is very likely to pass through a highway than other roads. The hierarchy-based approach ranks vertices and roads and leads the search to visit areas that are more likely to be the shortest paths.

The *Contraction Hierarchy (CH)* [17] adds shortcuts on the original graph and assigns a rank value $r()$ for every vertex to mark the importance of vertices. A shortcut is added from a vertex u to another vertex v if 1) the rank of v is higher than that of u, i.e., $r(v) > r(u)$; 2) the rank of all intermediate points on the shortest path from u to v is lower than the rank of u. The weight of the added shortcut is the shortest path distance from u to v in the graph. The graph with the attached shortcuts is called the contracted graph. CH answers a shortest path query by applying a modified bidirectional Dijkstra's algorithm on the contracted graph. The algorithm expands the search space from both source and destination simultaneously and only scans neighborhood vertices that are ranked not lower. The shortest path distance is determined by the node with minimum added distance from s and to t among all nodes settled by both side. The ranking of vertices largely impacts the performance of CH. A good ranking of vertices should minimize the number of edges added during the preprocessing.

The *Reach* algorithm [18] records the upper bound of any pass-through shortest paths of every vertex v. Given a path P from s and d, the reach of a vertex v (v is on P) with respect of P is defined as $min(d_n(s, v), d_n(v, d))$. The global reach value of v (denoted as $r(v)$) is the maximum reach value considering all shortest paths that pass through v. The search phase runs bidirectional Dijkstra's algorithm and prunes the visiting of a vertex v if $d_n(s, v) > r(v)$ and $d_n(v, t) > r(v)$ both hold.

Labeling-based Algorithms.

The labeling-based approaches record a label for each vertex to guide the expansion and save the search time.

Compressed Path Database (CPD) [19, 20] pre-computes the shortest paths between all pairs of vertices and implicitly stores the all-pairs shortest path in the index. Every

vertex stores the first move (adjacent edge) on the shortest paths to every other vertex. The search for the shortest path from s to d only needs to follow the shortest path guided by the stored labels, while avoiding the visiting on other areas. CPD implicitly stores the pair-wise shortest path between vertices. It suffers from the high pre-processing cost and large index size especially in large networks.

One typical type of labeling based algorithms is hop-based algorithm. The idea is similar to adding shortcuts to the graphs but the virtual shortcuts are referred to the labeled information. The hop-based approach records a distance table $L(v)$ for every vertex v to store the travel distance from v to several vertices or paths. The shortest path between every pair s and d can then be determined by the distance table of s ($L(s)$) and that of d ($L(d)$). The principle of constructing the distance tables is to guarantee that $L(s) \cap L(d)$ must contain at least one vertex on their shortest path for every pair of s and d. To search for the shortest path between s and d, the algorithm simply adds the label of intersecting vertices from s and d, i.e., $d_n(s,d) = min(d_n(s,v) + d_n(v,t) | v \in L(s), v \in L(d))$.

Abraham et al. propose the *Hub-based Labeling (HL)* [21, 22] for the shortest path calculation. HL labels the distances to the hub vertices for a vertex v through the upward search of a CH query starting from v. The vertex order of CH thus impacts the quality of labeling. A good strategy observed is to select the vertices with the most passing-through shortest paths greedily.

The *Pruned Landmark Labeling (PLL)* [23] focuses on complex networks such as social networks. It conducts a breadth-first search from each vertex u. When a vertex v is reached during the breadth-first starting from u, PHL adds a distance label $d_n(v,u)$ to the label $L(v)$. If the distance from u to v ($d_n(u,v)$) exceeds or equals to a labeled path distance through a vertex w and the algorithm already conducted a breadth-first search from w, i.e., $d_n(u,v) \geq d_n(u,w) + d_n(w,v)$, the algorithm prunes the labeling by not adding the label $(u, d_n(v,u)$ to $L(v)$ and stops traversing from u.

The *Pruned Highway Labelling (PHL)* [24] combines the labeling algorithm with the transit nodes algorithm. Instead of using hubs for labeling, PHL first decomposes the road network into disjoint shortest paths. It then uses these paths to label vertices. A label $L(v)$ on a vertex v consists of a set of triples $(i, d_n(p_{i,1}, p_{i,j}), d_n(v, p_{i,j}))$, where i represents an indexed path P_i, $d_n(p_{i,1}, p_{i,j})$ represents the travel distance from the

starting point of P_i $(p_{i,1})$ to a vertex $p_{i,j}$, and $d_n(v, p_{i,j})$ represents the travel distance from v to the vertex $p_{i,j}$. Given the source s and the destination d, finding their shortest path is then reduced to the problem of minimizing $d_n(s, p_{i,j}) + d_n(p_{i,j}, p_{i,k}) + d_n(p_{i,k}, d)$, where$(i, d_n(p_{i,1}, p_{i,j}), d_n(s, p_{i,j})) \in L(s)$, $(i, d_n(p_{i,1}, p_{i,k}), d_n(d, p_{i,k})) \in L(d)$. The $d_n(p_{i,j}, p_{i,k})$ equals to $|d_n(p_{i,1}, p_{i,j}) - d_n(p_{i,1}, p_{i,k})|$, which can be easily computed referring the $L(s)$ and $L(d)$.

Zhang et al. [46] extend the hub labeling algorithm to count the number of shortest paths between a source and a destination. Considering that the cover scheme of an one shortest path algorithm is not sufficient to find all shortest paths, they propose a new covering and labeling scheme.

Index-based approaches assume that the road network is static. However, the road network may be dynamically changing in real-world, for example, when the edge weight represents the travel time. The index needs to be rebuilt or updated for dynamic road networks. Several studies extend the index-based approaches to the dynamic scenarios. For example, when applying ALT to dynamic road networks, instead of rebuilding the index from scratch, we can keep the landmarks and recompute the distance to the landmarks [44, 47].

2.1.4 Learning-based Algorithms

In recent years, the data-driven approaches have been developed rapidly thanks to the growing size of data collected and higher computing capability. By finding patterns of historical trajectories or training deep learning models, the travel cost can be quickly approximated without the expensive graph traversal. The key problem of these data-driven approaches is to improve the prediction accuracy.

Existing deep learning methods mainly apply Multilayer Perceptron (MLP) for shortest path prediction. Jindal et al. [48] predict the shortest path distance using the coordinates of source and destination. The input layer consists of four coordinate values and is connected to an MLP (three-layer fully connected network) that outputs the predicted travel distance. Using the predicted travel distance and the time-of-the-day information, they further propose to predict the travel times by another MLP (also a three-layer fully connected network).

Instead of using coordinates, several studies embed the vertices to latent space and represent the source and destination as vectors. Rizi et al. [49] propose to learn the node embedding using node2Vec [50] and then use a fully connected layer to predict the distance. Qi et al. [51] add an embedding layer between the input layer and the MLP. The embedding layer has $|K|$ neurons to embed the vertices into a k-dimensional space. The embedding is similar to the landmark-based shortest path approaches that first select $|K|$ landmarks and then associate every vertex with a distance vector to record its travel distance to all $|K|$ landmarks. The embedding layer is further connected to an MLP to predict the shortest path distance between the source and destination. Their experiments show that their algorithm achieves better accuracy compared to [48] and [49].

Another research topic closely related to this thesis is the travel time prediction. Predicting the travel times is more challenging than predicting the travel distances as it needs to consider various factors such as traffic, weather, and departure time. One popular framework is the segment-based method. It first identifies the route between locations and then estimates the travel times of individual road segments or sub-paths [52–54]. The travel time of the whole path is computed by summing the travel times of all road segments in the path. Such a method overlooks the travel cost at road intersections and the impact of traffic lights, leading to inaccurate predictions. Recent studies focus more on directly predicting the whole travel time. Wang et al. [55] propose a non-learning method that averages the travel times of similar historical trajectories as the prediction of a trip. Two trajectories are similar if their source and destination are close. The main limitation of the method is that it may fail to find similar routes of the trajectories. Their experiments show that the proposed method outperforms the basic learning method (linear regression) and segment-based methods. However, the algorithm is revealed to be outperformed by a simple MLP as shown in the experiments of [48]. In recent years, many more advanced deep learning models are proposed to improve the prediction accuracy [56–60]. They propose embedding techniques for more accurate capture and representation on the spatial and temporal features (the locations of source and destination, the departure time) and integrate these embedded features and external information using more advanced deep models instead of simple MLPs.

2.1.5 K Shortest-path Algorithms

The k shortest-path problem has two variations: loopy and loopless. The loopy k shortest-path problem allows the same node to be visited more than once in a path, while the loopless k shortest-path problem prohibits loops in paths. The loopless k shortest-path problem is more complicated to solve due to the extra constraint.

The Dijkstra's algorithm [10] can be extended to find the k shortest paths by using a variable *count* to record the number of found shortest paths to the destination vertex d. When the extracted vertex from the heap is the destination vertex d, we increase the value of *count* by one, meaning that the next shortest path to d is found. We keep extracting nodes from the heap until the value of *count* equals to k. The complexity of this algorithm is $O(k(|E| + |V|log|V|))$, where $|V|$ and $|E|$ are the number of nodes and edges, respectively.

For loopy k shortest-path problem, the best time complexity is achieved by Eppstein's algorithm [61] with $O(k+|E|+|V|log|V|)$. Eppstein observes that every k shortest path from s to d can be represented as a sequence of sidetrack edges. An edge is a sidetrack edge if it is a network edge and does not lie on the shortest path from s to d. Following a sidetrack edge represents a deviation from the shortest path. Eppstein's algorithm first computes the shortest path tree of the graph such that the shortest path distance from each node to the destination d is obtained. The weight of a sidetrack edge $e(v, u)$ is then computed as the difference between the shortest distance of u and that of v, which represents the detour cost when following this sidetrack. After computing the weight of sidetrack edges, Eppstein's algorithm builds a graph to store all sidetracks edges and the corresponding deviation paths. The next shortest path can therefore be easily retrieved by popping elements from the built graph.

Several other algorithms are developed on the basis of Eppstein's algorithm. Jiménez and Marzal [62] propose an algorithm that is inferior to Eppstein's algorithm with respect to the time complexity but shows better performance when applied to practical problems. They also propose to construct only parts of the path graph to speed-up Eppstein's algorithm [63]. Instead of pre-computing the path graph, K^* builds the path on-the-fly and avoids visiting distant area unless necessary [64]. K^* is preferable in large networks since it avoids exhaustive search on the original network.

For loopless k shortest-path problem, the best time complexity is achieved by Yen's algorithm [65]. Yen's algorithm exploits the idea that the k_{th} shortest path usually shares edges and sub-paths of the $(k-1)_{th}$ shortest path. It first computes the shortest path from source s to destination d. Then it finds the deviation paths from the found $(k-1)_{th}$ shortest paths.

Akiba et al. [66] extend their PHL [24] algorithm for shortest-path queries to answer k shortest-path distance queries. It is the first index-based approach for k shortest-path distance queries. The built index significantly improves the response time. However, their method can only return the top-k shortest path distance but unable to restore the k shortest paths.

2.1.6 Batch Shortest-path Algorithms

Several algorithms have been proposed to compute multiple shortest path queries in a batch and reduce the computational cost by reusing shareable computation. Most of the proposed batch processing algorithms are index-free algorithms. The research focuses are twofold: effective clustering scheme for similar queries; efficient computation of queries in each cluster. The main property applied is the *path-coherence property*: shortest path queries with similar sources and destinations share a large portion of shortest path computation.

Mahmud et al. [67] propose to group similar queries with nearby sources and nearby destinations using the straight line connecting the source and destination of each query. For each cluster,they split the search into three parts: from the sources to an intermediate point, from the first intermediate point to the second intermediate point, from the second intermediate point to the destinations. The shortest path distance can then be retrieved by concatenating these sub-path distances.

Zhang et al. [68] propose an algorithm named $A^* - 1N$ to find shortest paths from a source s to a set of target vertices T. $A^* - 1N$ shares the same idea as the A^* algorithm but chooses an arbitrary node as the representative node. When the target nodes are sparse, they cluster the target nodes considering two criteria: the angle and distance to the source node. Running $A^* - 1N$ algorithm on these clusters separately helps to reduce the search space and speed-up the search time. The shortest paths from a set of

source nodes S to a set of target nodes T can be answered by running one $A^* - 1N$ from each source node $s \in S$. They further observe that the search space of different clusters may overlap, thus causing redundant computation. In [69], the authors propose to reuse a priority queue of clusters to avoid the overlap search and develop three scheduling methods to determine the order of processing clusters.

Li et al. [70] propose more advanced query decomposition methods to group queries that are likely to share the search space. The *1-N Zigzag Decomposition* method clusters queries with the same source or destination. The *Search Space Estimation* method estimates the search space of a query using an ellipse and groups queries with overlap search space. The *Coherence-Aware Co-Clustering* method clusters queries based on the coordinates of the source and destination of queries.

2.1.7 Discussion

Many algorithms have been proposed to accelerate the computation of the travel distance/time between locations in road networks. A typical strategy is to build a large index to store distance information so as to achieve faster query time. In the extreme case, a query can be responded in constant time if the shortest paths between all pairs of vertices are pre-computed and cached. However, such a method is not scalable to large networks as the index size increases when the road network grows. It is important to achieve a balance between the query time and pre-computation cost when designing a shortest path algorithm.

Existing speed-up algorithms exploit the network structure from various perspectives. Combing the ideas of different speed-up techniques may further taking their advantages and enhance the query time. For example, the state-of-the-art shortest algorithm achieves a fast query time by combining the hierarchical-based approach and the hop-based approach [56].

Existing shortest path algorithms inspired the development of many other problems in road networks. For example, by applying the idea of tree decomposition proposed in [56], Ouyang et al. [71] achieve the fastest query time for the k nearest neighbor problem in comparison with other proposed algorithms. In the future, it is worth exploring the

advantages of other advanced shortest path indices on more query problems in road networks.

The learning-based methods discover patterns of the shortest path computation in road networks. They achieve better balance of the query time and pre-computation cost thanks to the elimination of expensive graph traversals. The learning-based methods are also more flexible to dynamic factors such as traffic conditions when computing the travel costs. However, they only provide approximate results, thus are inapplicable when accurate travel costs are required.

2.2 Nearest Neighbor and Skyline Queries

Given a set of data points, a nearest neighbor query aims to find the closest data point(s) to the given query point(s). The distance computation between locations depends on the underlying metric and distance measurement. The distance between locations is the straight-line distance in Euclidean space, whereas the distance is determined by the shortest path between locations in road networks. Nearest neighbor queries are necessary in many real-world problems. For example, passengers are more likely to be served by nearby taxis. A driver may want to find a closest parking space in the street.

A simple brute-force solution for nearest neighbor queries exhaustively checks the distance from the query point(s) to every data point. Such a method needs to run shortest path algorithms multiple times, thus suffering from poor efficiency and cannot meet the real-time requirement in location-base services. The search time of nearest neighbor queries can be significantly reduced if the algorithm can quickly locate the optimal data points instead of unnecessarily visiting unpromising areas. We survey the studies on nearest neighbor queries in road networks. We focus on three types of nearest neighbor queries in road networks as they are most relative to our work: k Nearest Neighbor Query (kNN), Group nearest neighbor GNN, and all nearest neighbor ANN. We further survey skyline queries in road networks. Given a set of points in a road network P and a set of criteria, a skyline query aims to find the skyline points from P such that every returned point is not dominated by other points in P.

2.2.1 K Nearest Neighbor Queries

Given a set of data points O, a query point q, and a positive integer k, a k nearest neighbor query aims to return $top-k$ data points from O with the smallest road network distance to q.

The kNN query has been extensively studied in the Euclidean space and other metrics. Various spatial data structures are proposed for efficient kNN computation in the Euclidean distance, e.g., R-tree [72]. Next, we mainly discuss the kNN algorithms in road networks as the movement of vehicles is restricted by the road network structure. Compared to other metrics, the main challenge of processing kNN query in road networks is the expensive shortest path computation. Exhaustively checking the shortest path distances from each data point to the query point is too slow to meet the real-time requirement. To achieve a real-time response time, a kNN algorithm needs to quickly locate the potential nearest neighbors and avoids unnecessary computation on remote areas.

Papadias et al. [73] propose two kNN algorithms in road networks: *IER* and *INE*. The IER algorithm is based on the observation that the road network distance between two data points must be no smaller than their Euclidean distance. The algorithm first computes the data point o_i with the smallest Euclidean distance $d_e(o_i, q)$ to the query point q. Data point o_i is considered as an NN candidate, and the network shortest path distance between o_i and q, i.e., $d_n(o_i, q)$ is then computed. Distance $d_n(o_i, q)$ reduces the search space into an Euclidean circle around q. Only data points within this circle may be nearer to q than o_i in the network. To help identify data points that are near to q in the Euclidean space, spatial indices such as the R-trees [72] may be used. Abeywickrama et al. [74] further propose to calculate the lower bound using landmarks, which is similar to the idea of ALT algorithm [44, 45]. Their experiments show that landmarks provide tighter lower bound than the Euclidean distance, especially when the edge weights represent travel times. The INE algorithm gradually expands the search region from the query points so that the first data point reached when expanding is the query answer.

Kolahdouzan et al. [75] pre-compute a *network Voronoi diagram* of the road network. Their algorithm partitions the network into disjoint *network Voronoi cells*. Each network Voronoi cell encloses one data point o_i. All other vertices and edges within the Voronoi

cell consider o_i as the nearest neighbor. In other words, for a vertex v, its network distance to any other data points is no smaller than the network distance between v and the owner of the Voronoi cell covering v. A nearest neighbor query of a query point q can then be answered by simply locating the network Voronoi cell covering q. To find the next nearest neighbors, the algorithm gradually checks the adjacent Voronoi cells of the nearest neighbors that are already found. The algorithm needs to pre-compute all border-to-border distance of adjacent Voronoi cells. The large size of borders may lead to high pre-processing costs, hence making the algorithm inapplicable to large networks.

The *ROAD* [76] algorithm hierarchically partitions the road network. It identifies border vertices that are directly connected to other partitions through road network edges. For every partition, it pre-computes the shortest path distance between all borders within this partition. The search phase of ROAD is similar to INE [73], i.e., gradually expands from the query point. However, when ROAD reaches a partition that covers no object, it skips visiting inner vertices but directly reaches the next partition using the pre-computed border-border distance.

The *G-tree* [77] algorithm also partitions the network but differs from ROAD in respect of the tree structure and searching paradigm. G-tree builds a balanced tree and it uses a best-first search algorithm to only access partitions containing objects. It maintains a priority queue to rank partitions and objects based on their road network distance such that the nearby partitions and objects will be traversed first. The distance between locations is computed by assembling the border-border distance and vertex-border distance.

An experimental paper [78] compares the performance of various nearest neighbor algorithms in road networks. Previously, the nearest neighbor algorithm IER was widely acknowledged to perform slower than other algorithms such as INE. Their study combines the IER algorithm with the state-of-the-art shortest path algorithm PHL [24]. The new version is called IER-PHL algorithm. Surprisingly, they found that IER-PHL is faster than other NN algorithms in most cases. G-tree is also competitive to compute the shortest paths. It requires lower pre-computation costs than PHL while ranks the second in terms of the query time. INE is the most efficient algorithm and outperforms IER when the data points are densely distributed.

The *distance browsing* algorithm [79] proposes an index called Spatially Induced Linkage Cognizance (SILC) and stores the network shortest path distance between every pair of

vertices. The algorithm is not scalable to large networks due to the high pre-processing time and large index size (only evaluated on networks with less than 1 million vertices in the literature). Luo et al. [80] propose an algorithm called TOAIN that leverages CH for the k nearest neighbor queries. For every vertex u, TOAIN stores the top-k nearest neighbor in $u's$ downhill objects. An object located at vertex v is a downhill object of u if v reaches u in the contracted graph. He et al. [81] propose a grid-based approach. For each grid, they record all objects located in the grid. In the query phase, the algorithm gradually expands the search to nearby grids from the query point. The expansion terminates when the lower bound distances of all objects within unexplored grids are larger than the found results.

2.2.2 All Nearest Neighbor Queries

Given a set of data points O and a set of query points Q, the all nearest neighbor query aims to find the closest data point $o_j \in O$ for every query point $q_i \in Q$. Although the ANN query is a fundamental query type in spatial database, the studies on ANN query mainly focus on the Euclidean space and other metrics. Efficient and scalable ANN algorithms in road networks still remain unexplored. Next, we review ANN algorithms in the Euclidean space.

ANN algorithms in the Euclidean space can be grouped into two types: index-free and index-based algorithms. We start with the index-free algorithms. Clarkson et al. [82] consider the case where query and data points belong to the same set. They split the space into small cubic cells of equal size. The distance from a query object to its nearest neighbor is hence bounded by the distance to the nearest cell occupied by a data object. Their algorithm complexity is $O(n \log \delta)$, where n is the size of input data and δ is the ratio of the diameter of the point set to the distance between the closest pair of input points. Vaidya et al. [83] use a similar idea but optimize the splitting scheme. When query objects and data points are in two different sets, the ANN query is also called the Nearest Neighbor Join (NN-join) query. Xia et al. [84] propose the *Gorder* algorithm to process the ANN-join query. Gorder divides query objects and data points into several blocks and schedules the searching order of data points' blocks so that promising nearest neighbor candidates are visited first. Zhang et al. [85] propose a hash-based algorithm that hashes the query objects together with the data points and

divides them into buckets. For query objects in a bucket, nearest neighbors only need to be searched from data points in the same or overlapping buckets. Chen et al. [86] use the Hilbert curve to hash the data points into grid cells. The index-free algorithms discussed above cannot be applied to road network settings. They partition the space based on the Euclidean distance and only search for the nearest neighbor for a query object in nearby partitions. Such partitioning is difficult to form on a road network.

Index-based ANN algorithms compute the query with a traversal over a pre-computed index structure. Böhm et al. [87] propose an R-tree based algorithm named *MuX*. They optimize the I/O cost by organizing input data using large pages and take advantage of a secondary search structure within pages to optimize the efficiency. Zhang et al. [85] propose two algorithms for the case where the data points are indexed with an R-tree. Their first algorithm named Multiple Nearest Neighbor (MNN) finds the nearest neighbor of every query object by computing an NN query on the R-tree of data points. The processing order of the query objects is optimized so that close query objects can be handled consecutively. Their second algorithm named Batched Nearest Neighbor (BNN) finds nearest neighbors of multiple query objects at a time. BNN first groups multiple query objects and traverses the R-tree of data points once for finding the nearest neighbors of a group. Chen et al. [88] use a Quad-tree variant called the *MBRQT* to index the data points and propose a metric called *NXNDIST* (MINMAXMINDIST) to prune the search space during index traversal. Sankaranarayanan et al. [89] propose another pruning metric called *MAXMAXDIST*. Yu et al. [90] use *iDistance* [91] as the index structure. They propose an algorithm named *iJoin* that takes advantage of the data partitioning strategy of iDistance. Emrich et al. [92] propose to index the data points with an *SS-tree* and use trigonometric relationships to prune the search space during index traversal. The index structures used in these studies are built based on the Euclidean space. They are not applicable to the road network settings.

2.2.3 Group Nearest Neighbor Queries

Group nearest neighbor query is also called aggregate nearest neighbor query in the literature. Given a set of data points P and a set of query points Q, the group nearest neighbor aims to find a data point from P that minimizes the aggregate distance (sum

or max) to all query points in Q. Many algorithms are proposed for group nearest neighbor. Next, we discuss the group nearest neighbor algorithms in road networks.

Yiu et al. [93] are the first to investigate the group nearest neighbor query in road networks. They propose three algorithms to answer a GNN query: IER, TA, and CE. The proposed IER algorithm shares a similar idea to the IER algorithm in kNN query problem but the details are different. The IER algorithm for the GNN query considers the Euclidean aggregate distance as the lower bound of the road network aggregate distance. To save the computational cost, the algorithm checks data points in ascending order of the Euclidean aggregate distance. The checking terminates when the Euclidean aggregate distances of all remaining vertices exceed the found minimum network aggregate distance. The algorithms TA and CE, on the other hand, are similar to the INE algorithm proposed for kNN query problem. They expand the search space from the query points simultaneously and gradually visit vertices close to the query points. The expansion maintains a priority queue. The key of a vertex v in the queue is the minimum road network distance from the vertex to any point from the set of query points. TA computes the aggregate road network distance for a data point once it is scanned by the expansion of any data point. Denoting the found road network aggregate distance as $best\_dist$, the expansion terminates when the minimum key in the queue is larger than or equal to the threshold $best\_dist/|Q|$, where $|Q|$ is the size of the query points. In contrast to TA, CE computes the aggregate road network distance of a data point when it is scanned by the expansion of a data point. CE maintains a set $|S|$ to record all scanned vertices during the expansion. Vertices are removed from $|S|$ if their aggregate road network distances are computed. The termination of CE must satisfy two criteria: 1) a distance threshold criteria that is similar to TA, i.e., the minimum key of the queue must be no smaller than $best\_dist/|Q|$; 2) the set $|S|$ becomes empty. Their experiments show that IER algorithm outperforms TA and CE in most cases unless the data points are dense.

Zhu et al. [94] apply the network Voronoi diagram on the GNN query. The authors propose to gradually reach the next nearest neighbor of each query point using the algorithm VN^3 based on the Voronoi diagram. Every query point q_i maintains a queue to record nearest neighbors reached by it. Note that gradually retrieving the next nearest neighbors through VN^3 can help to determine the road network distance between some locations. Specifically, if o is scanned by q, the road network distance from q to o is

determined. On the other hand, if o has not been reached by q yet, the distance between o and q is unknown but must be greater than the distance from q to its scanned nearest neighbor (lower bound). When a data point is scanned by all query points, the aggregate nearest neighbor is proved to be reached by at least one query point. The algorithm thus terminates the expansion and computes the lower bound of aggregate road network distance for all reached data point points from any query point. They gradually check the reached data points in ascending order of their aggregate road network distance lower bounds. A data point is pruned if its aggregate distance lower bound is larger than the optimal distance found. Their algorithm is similar to the CE algorithm proposed in [93]. However, they apply VN^3 algorithm while CE apply INE to find the nearest neighbors. Besides, CE needs to check the aggregate road network distance of all scanned vertices while the algorithm proposed in [94] uses the lower bounds to prune unpromising vertices. They further propose an approximation algorithm for quicker response time. Their approximate algorithm first clusters adjacent query points and then computes the geometric center of the formed clusters. The closest data point of the geometric center is returned as the approximated group nearest neighbor.

Yan et al. [95] consider all locations in the the space (rather than only vertices of the road network) as potential query points and data points. They prove that it is sufficient to only check query points and vertices in the road network as potential group nearest neighbor point. They further prove that the group nearest neighbor point must be within a convex hull that bounds all query points and the shortest paths between query points. These two observations help to largely prune the search space and improve algorithm efficiency. They further propose an approximation algorithm that first checks the nearest road network vertex to the geometric center of the query points and then iteratively checks the neighborhood vertices of the checked area. The group nearest neighbor result is updated if a neighbor obtains a smaller GNN distance, otherwise they terminate the expansion and return the found results.

Yao et al. [96] study a variation of group nearest neighbor, i.e., flexible group nearest neighbor. Instead of minimizing the aggregate distance to all query points, the problem aims to minimize the aggregate distance to $\phi|Q|$ query points, where ϕ is a parameter ranging from 0 to 1. They propose an IER-based algorithm that gradually traverses the R-tree indexing all data points from top to down. The lower bound of any vertex indexed under an R-tree node is the Euclidean aggregate distance considering $\phi|Q|$ data

points. They compute the road network aggregate distance of data points reached by the traversal and record the optimal distance, which helps to prune unnecessary visit to R-tree nodes with larger lower bounds. They further propose an approximate approach that only checks data points that are nearest neighbors of the query points.

Abeywickrama et al. [97] hierarchically partition the graph into a tree and employs landmark-based lower bound calculation to find aggregate k nearest neighbors. They first partition the space into subgraphs. For each subgraph, they select m vertices as landmarks and compute the distance from inner vertices to the landmarks. They call this graph index as the Subgraph-Landmark Tree (SL-Tree). Based on the SL-Tree, they further construct an index called Compacted Object-Landmark Tree (COLT) to store the distance between objects to landmarks belonging to the same subgraph in the SL-Tree. Their search algorithm is similar to the IER algorithm, i.e., traverse the hierarchical structure and guide the search with lower bounds. However, instead of traversing the R-tree, they traverse COLT from top to bottom. Besides, the lower bound of a COLT node is obtained using the landmarks instead of the Euclidean distance.

Guo et al. [98] incorporate the social relationships between query points. Given a social network, a road network, a query user u_q, the number of attendees c, a keyword w, and a set of data points, they aim to find c attendees (contains keyword w) and top-k data point such that the total travel time of these attendees is minimized and the closeness of them is maximized.

2.2.4 Skyline Queries

Deng et al. [99] aim to find skyline locations given multiple query points in a road network such that each returned skyline location is not dominated by other locations considering both spatial (e.g., road network distance) and non-spatial attributes (e.g., price of the location). The authors propose three algorithms. The Collaborative Expansion algorithm gradually expands the search space from each query point and updates the skyline results during the expansion. The Euclidean Distance Constraint algorithm uses the Euclidean skyline points to guide the search so as to reduce the search space. The Lower Bound Constraint algorithm uses the Euclidean distance as the lower bound of a road network distance to terminate the search in an early stage.

Zou et al. [100] study skyline queries in graphs considering the network distance between vertices. Their algorithm first filters out data points that cannot be included in the skyline results. They then select the skyline results among remaining vertices. The distances between locations are quickly computed by taking advantage of a shortest path tree.

Kriegel et al. [101] aim to find skyline paths of users with respect to multiple criteria. Each edge of the road network is associated with a set of attributes such as the length and the maximum speed of the street. They define a user's preference as a weighted sum over all considered edge attributes. They propose a pruning strategy similar to the A^* algorithm to filter out routes that are guaranteed not to be included in the skyline results.

Huang et al. [102] aim to find skyline locations when a user is moving along a predefined path. They consider two criteria for skyline results: 1) the network distance to a query point; 2) the detour road network distance from the predefined route.

The approaches discussed above assume the query points and data points are static. Several other studies investigate skyline queries when the query points are moving. Jang et al. [103] assume the query points are moving along a path. Since the distances between skyline points and query points keep changing dynamically, the skyline results need frequent updating. To reduce the updating cost, they pre-compute a safe area for each data object o such that a query point q will consider o as a skyline point if q is located within the safe area of o. Huang et al. [104] design a grid index to manage the data objects. Assuming the route of a query point is known in advance, they calculate the skyline results at the starting location and compute locations along the route where the skyline results change . They mark these locations and update the skyline results when the query point passes through the marked locations.

Fu et al. [105] represent the query location as a a spatial range area and consider the case when the query objects and data objects are both moving continuously. The authors propose two algorithms. The landmark-based algorithm computes a landmark location between every pair of data objects to denote the change of dominance relationship. When the query range passes a landmark, the algorithm updates the skyline results. The index-based algorithm computes the skyline scope for every data object and index these scopes using a group of B+-trees [106]. The skyline scope of a data object o is a

spatial region such that for any query object q in the scope of o, o is closer to q than any of its dominators. The skyline results can be obtained by detecting the intersecting skyline scopes.

Miao et al. [107] propose several algorithms to answer why-not skyline queries when the queries are represented as range areas. A why-not range-based skyline query aims to explore the reasons why a data object is not included in the skyline results. They propose several modification strategies to include the missing data object such as modifying the query range and improving the non-spatial attributes of the missing data object.

2.2.5 Discussion

Nearest neighbor queries and skyline queries in road networks are essential in location-based services. Various algorithms are proposed in the past few years to improve the query time. The general idea of these algorithms is to gradually expand the search space from the query points until the results are guaranteed to found. Similar to the shortest path algorithms, nearest neighbor algorithms usually involve a pre-computation phase and build an index to speed up the search time. The balance between the pre-computation cost and the query time is crucial for nearest neighbor algorithms. Reducing the cost of graph traversal is a common goal of both nearest neighbor query and shortest path query. Thus, the shortest path indices have great potential to shorten the query time of the nearest neighbor query problem. Preliminary studies already verified this potential [71, 80].

Despite of the increasing attention, nearest neighbor algorithms in road networks are still developing and there remains many interesting research problems to address. ANN query is a fundamental query type in the spatial database. However, ANN has not been investigated in road networks in the literature. Designing efficient and scalable ANN algorithms in road networks is an important research gap to fill.

2.3 Dynamic Ride-sharing

Ride-sharing helps to relieve traffic congestion, reduce emission of vehicles, and provide convenient transportation service to passengers. In ride-sharing, passengers with similar

routes are grouped and a vehicle may carry multiple passengers at the same time. Next, we survey the evolvement of the dynamic ride-sharing algorithms.

2.3.1 Dial-a-Ride Problem

Dynamic ride-sharing is originated from the Dial-A-Ride-Problem (DARP) [108–111]. Given a set of requests with their source and destination specified, the DARP aims to schedule and route a fixed set of vehicles to serve these requests while achieving optimization goals (e.g., maximize the number of served requests) and satisfying a set of constraints (e.g., time constraints and capacity limit). Studies on DARP can be classified into two types: static DARP and dynamic DARP. The static DARP assumes the information of all requests is known in advance whereas the dynamic DARP considers a more realistic scenario that future requests are unknown and appear progressively.

The setting of dynamic ride-sharing is more similar to that of the dynamic DARP. However, dynamic ride-sharing models the problem differently than the dynamic DARP. DARP assumes that the set of vehicles is fixed. However, in ride-sharing, vehicles may join the leave the system at any position and any time. Besides, DARP has only been evaluated in small instances, while ride-sharing needs to be implemented in large networks with large size of requests and vehicles. Achieving real-time response in ride-sharing is challenging.

Exiting algorithms on DARP model the problem as integer linear programming, e.g., they use a binary to indicate whether a vehicle is scheduled to traverse an edge. DARP algorithms can be classified into three categories: exact, heuristic, and meta-heuristic. Exact DARP algorithms provide accurate solutions. However, the high computational cost makes them inapplicable to large-scale instances. As an illustration, Cordeau [112] applies the branch-and-cut algorithm and solves up to 48 requests.

Due to the computational overhead, heuristic and meta-heuristic algorithms are proposed for larger size of instances. Classical heuristics include *insertion* and *cluster-first route-second*. The insertion heuristic first orders requests and then sequentially inserts ordered requests to vehicles [113–118]. The cluster-first route-second heuristic first constructs clusters and then schedules the route of vehicles to chain the generated clusters [119, 120]. These heuristics help to largely reduce the search space. The insertion

heuristic is also widely applied in dynamic ride-sharing [26, 36, 121]. However, due to the different problem modelling, the algorithms on DARP cannot be directly applied to the dynamic ride-sharing problem.

Meta-heuristic methods first construct an initial solution and then gradually improve the solution. For example, Tabu search records the previously visited solutions such that revisiting can be avoided [122–125]. Variable neighborhood search iteratively checks the neighbor of the current solutions [124, 126–128]. The large neighborhood search destroys parts of the solution and rebuilds the complete solution at each iteration. The meta-heuristics help to achieve more globally optimized solutions [129–131]. The iterations required by meta-heuristic algorithms bring computational overhead and cannot achieve response time to passengers in dynamic ride-sharing.

Constraints such as time windows and capacity limits are essential considerations in both DARP and dynamic ride-sharing. The constraints increase the complexity of the DARP due to the employed integer linear programming methods. Instead, in dynamic ride-sharing, the constraints are critical indicators to prune infeasible dispatches and reduce the search space.

2.3.2 Speeding up the Matching Process

A critical challenge in dynamic ride-sharing is how to improve the matching time and achieve real-time response for on-demand requests. The main bottleneck for the efficiency is how to quickly determine firstly the feasible and subsequently the optimal vehicles to serve the new requests, which depends on efficient and scalable ride-sharing indices to store and manage the ride-sharing data. Existing indices on the shortest path query and nearest neighbor query cannot be applied to ride-sharing as they only store the current vehicle location but fail to consider the scheduled paths of ride-sharing vehicles. Next, we survey the ride-sharing indices and other speed-up strategies proposed in the literature to accelerate the matching process.

Tshare. Tshare is the first index proposed for the dynamice ride-sharing [28].
Index: Tshare partitions the space into grids and the distance between locations is computed based on the centroid points of their enclosed grids. Tshare maintains three pieces of information for every grid: 1) a spatial list that ranks other grids ordering by

their travel distances; 2) a temporal list that ranks other grids ordering by their travel times; 3) a vehicle list that records vehicles that are scheduled to enter the grid and their scheduled arrival times.

Match: Tshare sequentially dispatches requests ordering by their issue times. Besides, it assumes the new requests are inserted to the current routing of vehicles, i.e., the insertion heuristic. The aim of Tshare is to sequentially dispatch every new request with a vehicle that incurs minimum extra travel distance. They propose two searching algorithms: single-side search and dual-side search. The single-side search checks vehicles from the located grid of the source and expands the search space to nearby grids (according to the temporal list) until a feasible vehicle is found. The dual-side search expands the search space from both sides of the source and destination simultaneously and terminates once a vehicle is scanned from both sides.

Update: If the system assigns a vehicle to a new request, it first erases the recording of the assigned vehicle, i.e., unmark all grids of the vehicle's previous schedule. Then, it updates the assigned vehicle's trip schedule by adding the new request. The grids passed by the new scheduled routing are then marked in the index. When the vehicle moves, it monitors the position of moving vehicles and unmarks grid passed by the vehicles.

Kinetic Tree. Tshare only records one optimal schedule for every vehicle and assumes that passengers are inserted into the existing vehicles' schedules. In contrast, the Kinetic tree [25] may re-order the existing schedules to obtain a better-optimized route.

Index: Although a vehicle has multiple possible routes to schedule, these possible routes usually share a large part of sub-routes while only a few stops have different orders. This is the main observation of the Kinetic tree. It represents the possible paths as a tree and a path as a branch of the tree. Every vehicle maintains a tree to record all possible vehicle routes instead of only recording the optimal one as in Tshare [28]. The root of the tree represents the current location of the vehicle. The algorithm branches the tree if there are more than one locations to be visited as the next stop. Every branch of the tree indicates a feasible route of the vehicle.

Match: When a new request arrives, it checks all possible edges to insert the source and destination. It then obtains the possible insertions to all possible routes instead of only the currently scheduled route. The optimal insertion position will then be selected and form the new schedule.

Update: When a vehicle is dispatched to a new request, the algorithm updates the

kinetic tree of the dispatched vehicle by inserting the source and destination vertices to all possible edges. The vehicles then follow the routes with the minimum travel distance. When the vehicle moves, the algorithm deletes all obsolete vertices from the tree since visiting these points becomes impossible.

Discussion: The kinetic tree needs to check the matching to all vehicles. For each vehicle, it examines all possible routes instead of only the optimal one, which may achieve better-optimized schedules but requires more computation. The kinetic tree suffers from the poor scalability due to the large search space. Meanwhile, as shown in [28], reordering the existing stops may only bring marginal improvement to the matching quality.

Xhare: Xhare [29] partitions the road network hierarchically.

Index: Xhare partitions the network into a three-level hierarchical structure. In the first level, it partitions with grids (much smaller than Tshare [28]) such that every location is enclosed by a grid. In the second level, it extracts a set of landmarks using popular locations such as transportation stations. It then links every grid with a nearest landmark. In the highest level, it clusters landmarks such that every landmark is associated with one cluster. For each vehicle, Xhare computes all clusters the vehicle is going to pass through and marks the vehicle id in the record of every pass-though cluster.

Match: when a new request arrives, Xhare first locates the cluster of the source and destination following the hierarchical mapping (location→grid→landmark→cluster). Then it checks the pass-through vehicles of the source cluster and the destination cluster. Vehicles found from both clusters are returned as the vehicle candidates for the request as they can potentially visit both the source and destination.

Update: when a vehicle is dispatched to a new request. Xhare first unmarks the pass-through clusters of the vehicle's previous schedule. It then updates the schedule of the newly dispatched vehicle by adding the new request. The new pass-through clusters are then computed and marked. When a vehicle moves, Xhare unmarks obsolete clusters that have already been passed through and impossible to be visited.

Discussion: Tshare [28] terminates once a vehicle candidate is found. The returned vehicle from Tshare is a possible match but may not optimal. Xhare, on the other hand, returns more than one vehicle candidate and then select the optimal one to serve the new request. Thus, the matching quality of Xhare may be better optimized.

DSA [27]: The Dual-Side Search Algorithm (DSA) algorithm is proposed to return skyline query results for a query considering two factors: pickup time and price. Given

a set of criteria (e.g., arrival time and trip price), a skyline query aims to return a set of matching results and ensure that every returned result is not dominant by any other results. A result dominates another result if it outperforms/not worse on every criterion. DSA combines the idea of Kinetic tree [25] and Tshare [28].

Index: Similar to Tshare, DSA partitions the road network into grids. For every grid, DSA records five pieces of information: 1) a border vertex list that records vertices connecting to other vertices outside of the grid; 2) an inner vertex list that records in-grid vertices together with their shortest path distance to the borders; 3) a grid cell list that ranks other grids based on the travel times; 4) an empty vehicle list that records in-grid empty vehicles; 5) a non-empty vehicle list that records non-empty vehicles currently in or are scheduled to enter the grid. The distance between borders helps to identify the distance lower bounds and upper bounds between locations, which provides quick determination on whether an insertion position is possible.

Match: DSA uses kinetic trees to index the vehicles' schedules. When a new request arrives, it first prunes vehicles violating the waiting time constraint by only checking nearby grids. For each remaining vehicle, it checks all possible insertion positions and prunes an insertion if the lower bound distance is larger than the maximum allowed distance.

Update: When a vehicle is dispatched a new request, DSA updates the kinetic tree of the dispatched vehicle (the same as [25]). Then, it updates the pass-through grids of previous trip schedule similar to Tshare [28].

Discussion: The DSA algorithm applies the Kinetic tree to index the schedule of vehicles. The same as the Kinetic tree [25], DSA suffers from poor scalability, which limits its application on highly scalable scenarios when there are a large number of vehicles and requests.

Luo et al. [30] use lower bound distance to prune insertion positions by using hierarchical road network partitions.

Index: They partition the road network into hierarchical structured subgraphs similar to G-tree [77]. The distance lower bound between two vertices is determined by the borders of their located sub-graphs, which is pre-computed and stored in the pre-computation phase. They mark a set of currently located vehicles for each sub-graph and record only one optimal route for each vehicle.

Match: They match requests and vehicles to achieve two optimization goals: maximizing the served rate and minimizing the total additional increased distance. When a new request arrives, the algorithm first filters out vehicles that are too far away and violate the time constraints of the request. Then they check the insertion probability to all remaining vehicles. They accelerate the determination on whether a vehicle is too far away or a insertion position is feasible by using the obtained lower bound distance. They propose two algorithms to maximize the served rate. The Distance-first algorithm processes requests in the order of their issue times. The Greedy algorithm first computes the matching feasibility of every pair of vehicle and request and then dispatches the pair in ascending order of the utility value.

Update: They update the located grids of vehicles when vehicles move.

Discussion: The algorithm accelerates the matching time using the lower bound distance obtained from the hierarchical data structure. However, they only compare their algorithm with [27] that is slower than other state-of-the-art such as Tshare [28] and X-hare [29]. The performance comparison of their algorithm and other state-of-the-art remains further study.

	Fix order of existing stops	Fix vehicles' destination	Index structure
Tshare [28]	✓	✗	Grid
Kinetic tree [25]	✗	✗	Kinetic tree
Xhare [29]	✓	✓	Clustering
DSA [27]	✗	✗	Grid + kinetic tree
Luo et al. [30]	✓	✗	Hierarchical partition
Ta et al. [31]	✓	✓	Hierarchical partition

TABLE 2.1: Comparison of ride-sharing indices.

Ta et al. [31] study the scenario when drivers have a pre-defined route and passengers can only be picked up or dropped off along the route. They partition the graph into hierarchical structure similar to G-tree [77]. Considering that the shortest path calculation is expensive, they partition the graph into sub-graphs similar to G-tree [77]. They then utilize two bounds to accelerate the matching process: a lower bound determined by the Euclidean distance between two vertices and an upper bound obtained by the distance between sub-graphs in the tree. Their optimization goal is to maximize the ratio of shared routes among passengers and drivers. They consider two assignment strategies: greedy assignment and batch assignment. The greedy algorithm processes passengers in

the order of their arrival times. It first computes all possible vehicle-passenger pairs and then processes pairs in ascending order to their lower bounds until the lower bounds of remaining pairs exceed the shared ratio of the found optimal match. The batch assignment matches multiple passengers and multiple drives simultaneously using the bipartite weighting graph. Drivers and passengers are placed in two sides of the bipartite graph, respectively. A driver is connected to a passenger if the matching is feasible and the connecting edge between them represents their shared ratios.

Tong et al. [26] aim to quickly determine the optimal insertion positions of a request to a vehicle. Their optimization goal focuses on minimizing the total increased travel distance. They improve the complexity of checking the optimal insertion positions of a new request into a vehicle into $O(1)$ by applying dynamic programming. The idea is extended to other optimization goals such as minimizing the maximum flow time of all requests [121] and minimizing the demand-supply score [132]. To accelerate the matching time, they propose to use the Euclidean distance to compute the lower bounds of a detour cost, which helps to quickly determine whether an insertion is feasible. The idea of using Euclidean distance for pruning largely improves the algorithm efficiency and is widely applied in the subsequent works [121, 132]. However, Euclidean distance may only provide loose lower bound and the bound may be further tightened using techniques such as landmarks or graph partitions (as shown in some shortest path algorithms [43, 77]). The matching time may be further enhanced using other lower bound computations.

2.3.3 Improving the Matching Quality

Many other algorithms aim to improve the matching quality such that more passengers are satisfied and less travel distance is required.

Personalized Ride-sharing

Duan et al. [32] aim to maximize the satisfaction of passengers considering three factors: payment, travel time, and waiting time. They group the involved participants in ride-sharing into three types: in-vehicle passengers, waiting passengers, and vehicles. They measure the request's satisfaction as the combination of time and payment, while the driver's satisfaction is measured by the income difference. During the pre-computation phase, they partition the space into grids and then pre-compute the maximum distance

and minimum distance between grids. The pre-computed distances are used as lower bounds and upper bounds to prune unsatisfied matches, which helps to prune infeasible vehicles and improve the matching time. However, their algorithm is based on the assumption that a vehicle can be committed to at most two passengers, which is unrealistic in real-world scenarios.

Cheng et al. [133] study the dispatching problem of maximizing the satisfaction of requests. They model the satisfaction of a rider on a vehicle considering three factors: the vehicle-related utility, the rider-related utility, and the trajectory-related utility. The vehicle-related utility refers to the vehicle quality such as brand and models. The rider-related utility is related to the service provided by the drivers and the social interest of shared requests, and the trajectory-related utility considers the detour cost. The linear combination of these three utilities forms the final utility value of a vehicle-request pair. They propose three matching algorithms. The first algorithm BilateralArrangement randomly selects requests to match. For every request, if searches for the optimal vehicle that maximizes the utility value. If the optimal vehicle is feasible to insert the request, the system arranges the request to it. On the other hand, if the optimal vehicle is infeasible, the algorithm tries to replace an existing request in the optimal vehicle's schedule with the new request so as to achieve higher utility value. The second algorithm Greedy first computes the utility gain of all potential request-vehicle pairs. It then assigns each pair in ascending order to the utility gain. The third algorithm first clusters trips based on their time duration and distributions. It then classifies trips into long trips and short trips and group all long trips and short trips in the same region. Requests are then processed as groups in the order of the number of requests in each group.

Cao et al. [134] consider the maximum waiting time constraint and maximum price constraint of requests. They propose to return skyline matches to requests considering two factors: waiting time and price. They formulate the price by combining the travel cost of the request and the detour cost of a vehicle. They assume drivers have pre-defined source and destination, and that requests can only be served on the route. Their algorithm first prunes vehicles with Euclidean distance larger than the maximum waiting distance. The lower bounds of the detour cost and price are computed also using Euclidean distance. A vehicle is pruned if the lower bound of its price exceeds the maximum acceptable price. They then construct a matching table that ranks all remaining pairs based on the lower bounds of their detour costs. Pairs are extracted gradually from the matching table.

	Dispatching strategy	Considerations	Optimization goal	Pre-defined destination
Duan et al. [32]	greedy	price, travel time, waiting time	satisfaction of passengers	✗
Cheng et al. [133]	greedy & batch	vehicle quality, social interest, detour cost	satisfaction of passengers	✗
Cao et al. [134]	greedy	price & waiting time	skyline results	✓
Chen et al. [27]	greedy	price & waiting time	skyline results	✗

TABLE 2.2: Summary of personalized ride-sharing dispatching algorithms.

They compute the nearest driver for all pairs remaining in the matching table and use this as a lower bound of their waiting times. The lower bounds (of the detour costs and waiting times) help to quickly determine whether the extracted pair is a skyline result or not. The algorithm terminates if the lower bounds of remaining pairs exceeds the skyline results. They assume that the vehicles have pre-defined destinations and hence new stops can only be inserted between the existing schedules. Chen et al. [27] relax this assumption while also aiming to return skyline results to requests. They partition the space into grids and pre-compute the lower bound distance and upper bound distance between grids. During the matching process, they use the pre-computed lower bounds to quickly determine whether an insertion position is feasible, eliminating the expensive shortest path calculation and improving the matching time.

Table 2.2 summarizes the considerations of exiting personalized ride-sharing algorithms.

Price-aware ride-sharing

As ride-sharing is a commercial business, pricing is a crucial component for all ride-sharing stakeholders, i.e., passengers, drivers, and the service provider. Setting appropriate pricing should consider various factors such as acceptance of passengers, revenue of the service provider, and attractiveness to drivers. On one hand, the service provider earns less revenue when orders are under-priced. On the other hand, overpricing orders will make the ride-sharing service less appealing and attract fewer passengers, thus decreasing the system profit.

Pricing in ride-sharing is complicated. First, the pricing is dependent on the dispatching and routing process. Second, the pricing on different times and regions should be different and adjusted dynamically considering the various spatial and temporal distribution

of vehicles and requests. Compared to other spatial crowd-sourcing tasks, pricing in ride-sharing is more challenging due to the latest arrival time constraints of requests. A vehicle can only serve several nearby requests considering their spatial and temporal distributions. Besides, the movement of vehicles are restricted within specific areas and it is non-trivial to check the detour constraints of vehicles.

Most of the price-aware ride-sharing studies aim to maximize the revenue of the service provider. Asghari et al. [35] formulate the profit by subtracting the payments to drivers from the fares paid by passengers. The income of a driver only depends on their total travel distance. The fare of a request, on the other hand, is determined by its shortest path trip and compensated by its detour distance. When a new request arrives, the algorithm computes the expected revenue from each driver and dispatch the request to a driver that earns the highest income. Asghari et al. [35] considers no preference of vehicles. In contrast, Zhao et al. [135] enable drivers to value each order according to their preferences such that drivers are more likely to serve their preferred requests, e.g., passengers located in familiar areas of the driver.

Asghari et al. [35] dispatch requests sequentially by the order of their arrival times. The fare of an order depends on the dispatch result. Zheng et al. [36], on the other hand, dispatch a batch of requests during every step and assume upfront order fares regardless of the dispatching and routing results. They assume the new stops are inserted into the vehicle schedule and propose two matching algorithms: greedy and bipartite graph. The greedy algorithm first computes all feasible request-vehicle pairs and order these pairs by their profits. It then greedily dispatches pairs with the highest profit until no feasible request-vehicle pair remains. They further propose another algorithm that constructs a bipartite graph considering the vehicles and requests as vertices in the two sides of the graph. The edge weight is larger than zero if the vehicle-request pair is feasible, otherwise the edge weight equals to zero. They then solve the problem using maximum weighted matching algorithms.

In [35], vehicles bid for the orders. Zheng et al. [136] consider a reverse case when passengers submit bids for their offers. Passengers with higher bids will thus get higher priority to be served. Their optimization goal is to maximize the overall utility for passengers, drivers, and the system profit. They propose two dispatching strategies: the greedy algorithm and the ranking based algorithm. The greedy algorithm first finds all

	Insertion heuristic	Dispatching strategy	Fixed order price	Optimization goal
Asghari et al. [35]	✓	greedy	✗	revenue
Zheng et al. [36]	✓	greedy & batch	✓	revenue
Zheng et al. [136]	✓	greedy & batch	✗	overall utility

TABLE 2.3: Summary of price-aware ride-sharing dispatching algorithms.

feasible vehicles of each request and computes the utility value for each vehicle-request pair. Then, it greedily dispatches pairs with the highest utility values. The ranking based algorithm first locates the nearest vehicle to every request. Then, it computes a pack for every request. The pack for request r_n combines r_n with at most $|C| - 1$ other requests such that dispatching the packed requests to their nearest vehicles achieves the highest utility, where $|C|$ is the number of vehicles. They dispatch packs sequentially by the order of their utility value.

Table 2.3 summarizes the assumptions and methods of existing price-aware dispatching algorithms in ride-sharing.

Except for optimizing the dispatching results, several studies develop pricing strategies for every order. The attractiveness of passengers depends on the pricing strategy. Lowering the price attracts more passengers while increasing the price will attract fewer passengers. Several studies focus on surging the price in high demand areas such that more drivers will be attracted [137–139]. However, They only consider the pricing at a single region at a particular time. Bimpikis et al. [140] set the pricing of trips considering the spatial distribution of vehicles in the future but overlook the effect of future demand. Asghari et al. [141] study dynamic pricing based on the request origin and destination while considering the future demand. They first apply a local optimization algorithm to compute the optimal price at the current time t and the next timestamp $t + 1$. The revenue increase and decrease in each region is then considered to modify the optimal price. Although they consider future demand, they set the price at the current time t by only referring to the supply and demand imbalance in the next timestamp $t + 1$ but overlook the potential revenue in all subsequent time periods.

Tong et al [34] propose dynamic pricing schemes to consider supply and demand in spatial crowdsourcing, aiming to maximize the total revenue of the service provider. They partition the space into grids and compute the optimal unit price for each grid.

Their setting is different from the ride-sharing scenarios. They assume a worker (vehicle) can only serve one request, but vehicles may carry multiple requests at a time in ride-sharing. Besides, they only define the source, destination and the maximum accepted unit price for requests but ignore their time budgets.

Demand-aware ride-sharing

Most of the online dispatching algorithms assume that future requests are unknown and dispatch requests sequentially ordering by their issue times. Such a dispatching strategy considers a realistic scenario and achieves high efficiency. However, it only achieves sub-optimal results as the previous assignments affect the routes of vehicles and the assignment of succeeding requests. In contrast, the offline dispatching algorithms assume an awareness of all upcoming requests and may reschedule the trips. The offline dispatching strategy can achieve global optimization while the online dispatching strategy only achieves local optimization. However, the offline dispatching strategy is unrealistic as it is impossible to know information of all future requests in advance. Besides, they usually consider all matching possibilities between vehicles and passengers, which requires a long response time.

An idea developed rapidly in the last few years is to consider future requests during the matching and routing process [33, 132, 142–144] so as to achieve a better optimization while still preserving the real-time response time. The future request information can be analyzed through the historical data or predicted by deep learning models. Existing algorithms usually represent the traffic demand in the near future using transition probabilities between regions (nodes).

Lin et al. [142] aim to optimize the routing of ride-sharing vehicles to maximize the system's revenue while considering the service constraints of passengers. They assume a vehicle can serve at most two requests and pick up the two requests sequentially. They propose two routing algorithms that apply to two stages respectively. The first stage (planning stage) focuses on route planning when only the first request is picked up. The algorithm aims to find a path that maximizes the probability of encountering a second request. They design a dynamic programming algorithm from the source to the destination based on the probability of picking up a request at a vertex. The routing only considers the direction of the second request but overlooks their future directions. If an

encountered second request is infeasible, the vehicle rejects the request and keeps following the planned path. On the other hand, if the vehicle encounters a feasible request on the path, the vehicle picks up the second request and changes the route according to the second stage. The second stage calculates a path after picking up both passengers. The goal is to plan a route that maximizes the service provider's total revenue while considering their time constraints. Their algorithm only considers the routing of a single vehicle. They further propose a framework to jointly consider the routing and matching process to maximize the matching ratio [143]. They formulate the problem using a time-expanded graph and propose an approximation strategy to solve the problem. Their work provides solid theoretical analysis. However, the proposed algorithm is too expensive to achieve real-time responses. Besides, the assumption that only two passengers can be served is unrealistic in the real-world taxi ride-sharing systems.

Yuen et al. [144] aim to recommend the routes of vehicles to maximize the possibility of picking passengers along the route. They observe that vehicles usually move closer to the destinations following the road network. Therefore, instead of searching for all possible routes, they reduce the search space by constructing a Directed Acyclic Graph (DAG) that only consists of forwarding edges (edges that decreases the distance to the destination). They further propose a polynomial-time algorithm using dynamic programming that processes vertices ordering by their topological order. They also add backward edges (edges that increase the distance to the destination) that satisfy the DAG's time constraints to achieve better approximation performance. Similar to [142, 143], they overlook the destination direction of the requests but only consider the origins.

Wang et al. [132] match ride-sharing requests considering the potential profit in the near future by an indicator *Demand-Supply Balance Score (DSB-Score)*. They construct a demand map and a supply map to represent the traffic pattern. Instead of using historical data, the demand map is constructed by applying state-of-the-art travel demand prediction deepST [145] that predicts the number of riders in a region at different times. The supply map is calculated by considering the shortest path of vehicles and their estimated arrival time at each area. They proposed a dynamic programming algorithm to calculate the optimal insertion positions to a vehicle's schedule in linear time, following the insertion assumptions. When assigning a new request, they first derive the lower bounds of each vehicle in terms of the DSB-score and then sequentially check the vehicles. The optimal match that maximizes the demand-supply balance score will be

	dispatching	routing
Lin et al. [142]	✗	✓
Lin et al. [143]	✓	✓
Yuen et al. [144]	✗	✓
Wang et al. [132]	✓	✗
Liu et al. [33]	✓	✓

TABLE 2.4: Research focus of demand-aware ride-sharing.

dispatched and the supply map will be updated accordingly. They assume the vehicles follow the shortest path between every two consecutive stops. Integrating the routing algorithm proposed in [144] may further enhance the matching quality.

Liu et al. [33] propose a novel indexing technique considering the travel directions of vehicles and requests. They partition the road network into regions and group rides and taxis with similar directions, rather than only considering their current locations. When a new request arrives, only the vehicles moving to similar directions will be returned as candidates. Among these candidates, they dispatch the new request with a vehicle that requires the least detour time to serve the request. They further propose a routing algorithm to recommend routes between stops. They reduce the search space to accelerate the path finding in a similar way of [144] by only considering the possible partitions satisfying the time constraints and represent each partitions using landmarks. The route is then computed by applying Dijkstra's algorithm on the smaller graph.

2.3.4 Discussion

Most of the dispatching algorithms in ride-sharing first retrieve potential vehicles to serve new requests (pruning phase) and then select the optimal matches according to the optimization objectives (selection phase). Retrieving possible vehicles efficiently is the key to reduce the matching time and improve the performance of the ride-sharing system. However, it is challenging to handle the large number of vehicles and passengers involved in the matching process and design an efficient and scalable pruning algorithm. First, dynamic ride-sharing needs to consider many constraints and benefits of different stakeholders, e.g., passengers, vehicles, and the service provider. The index needs to consider all these constraints so as to effectively prune infeasible matches. Second, ride-sharing is a highly dynamic scenario as the status of vehicles and passengers are not

static but changing rapidly. The algorithm needs to have low update cost. Most of the existing algorithms use simple graph partition (e.g., grids) or Euclidean distance for pruning, which suffers from poor pruning effectiveness. Although several advanced indices are proposed, they need to store and maintain a large size of information, which falls short on the updating efficiency. Besides, they can only provide approximation results and may miss potential vehicles for the new requests. A critical research gap to fill is how to design an efficient and scalable pruning algorithm in ride-sharing considering both the matching time, update cost, and the index size.

2.4 Flexible Ride-sharing

Traditional dynamic ride-sharing described in Section 2.3 assumes passengers are served from their requested locations and a passenger can only be served by one vehicle. Next, we review studies aiming to provide more flexible ride-sharing services to passengers. Popular schemes to enhance flexibility include: 1) computing meeting (pick-up/drop-off) points where passengers get on/off board; 2) allowing passengers to transfer between vehicles.

2.4.1 Flexible Meeting Point

Meeting points enable passengers to meet the vehicles at assigned meeting point instead of their source and destination. Allowing meeting points helps vehicles to reduce the detour cost and increase the matching ratio of passengers. Moreover, meeting points facilitate safer boarding of passengers compared to using the original source and destination. We next review the literature that investigate ride-sharing algorithms with flexible meeting points.

Most of the existing works on meeting points assume that vehicles have their pre-defined destinations and search for possible passengers to share along the pre-defined path. Aissat et al. [146, 147] study selecting the optimal intermediate points between passengers and drivers to minimize the total travel cost subject to the time constraints of passengers and drivers. They consider no restrictions on passengers' effort to arrive at the intermediates and hence passengers may find it hard to arrive at the meeting points. Stiglic et al. [148] study the case when the intermediate points are within reasonable

walking distances of passengers. Their experiments verify that allowing pick-up and drop-off points can substantially increase the matching ratio and reduce the system-wide distance travel distance. In [149], each passenger has a set of feasible meeting points (called close enough points), and they propose to assign passengers to drives at meeting points while satisfying the detour constraints and vehicle capacity of drivers. Their model assumes that all passengers share the same destination but are originated from different places. Li et al. [150] introduce preferable time windows for ride-requests instead of only the earliest and latest times.

A few studies integrate the meeting points with multi-hop ride-sharing. Chen et al. [151] consider a ride-sharing system in that participants with a car can drive to a meeting point to share the route with another driver. The drivers need to go back to the meeting point to drive to their final destinations. Their optimization goal is to minimize the overall travel distance.

Meeting points enlarges the search space to combine vehicles, passengers, and meeting points. Enabling meeting points brings more challenges to the algorithm efficiency. Most of the works uses integer linear programming to solve the problem, which lack efficiency and are inapplicable to large-scale scenarios such as metropolitan cities. Zhao et al. [152] propose to reduce the solution space and then decompose the problem into two sets of sub-problems. They reduce the matching problem and vehicle-routing problem into the knapsack problem and shortest path problem, respectively. Yu et al. [153] consider flexible pickup points within the walking range of passengers. They adopt contraction hierarchies to accelerate the shortest-path computation. Czioska et al. [154] consider real-world taxi ride-sharing scenarios. They first cluster passengers with similar routes and then assign an optimal meeting point for every cluster. They then search for nearby vehicles to visit the meeting points of the clusters.

Another key problem is the selection of suitable meeting points. Czioska et al. [155] identify and rate potential meeting points considering factors such as parking facilities, illumination, sheltering, and seating of the meeting points. In their following work [156], they extend the possible meeting points from nearby locations to areas that are reachable by using public transportation. Their optimization goal is to minimise the total travel time of drivers and the passengers. Goel et al. [157] selects the meeting point using subsets of major street intersections.

2.4.2 Multi-hop Ride-sharing

Multi-hop ride-sharing refers to a ride-sharing service in which passengers can transfer between vehicles. Allowing transfers between vehicles provides higher flexibility to passengers and enables more requests to be served. Multi-hop ride-sharing further brings the opportunity to reduce the travel distance thanks to more shared trips.

Existing multi-hop algorithms mostly model the problem using Time-Expanded Graphs (TEG) [158–160]. The purpose of building TEGs is to record all possible routes of vehicles considering their time budget such that overlap routes can be searched when a new request arrives. In a TEG, the TEG nodes have two keys: location id and time, representing that the vehicle can reach the location at the specified time. The TEG nodes record possible visits of vehicles with time information. These nodes are connected by TEG edges to denote possible connections such that possible routes can be tracked following the edges. A TEG edge that connects two TEG nodes of different points is called a *transfer edge*, whereas a *waiting edge* connects two TEG nodes of the same location but different timestamp.

The first adoption of TEG in multi-hop ride-sharing is [160]. They reduce the problem of multi-objective optimization into a multi-objective shortest path problem and solve it using an evolutionary algorithm after building the TEGs. Several algorithms are proposed to accelerate the search of multi-hop matches on TEG graphs. Drews et al. [158] apply A* algorithm on the TEG graph to find the best multi-hop option. Masoud et al. [159] reduce the number of nodes in a TEG by only considering points within areas bounded by ellipses. They further develop dynamic programming algorithms to accelerate search on TEGs for the best routes [159]. Despite the reduced search spaces, their algorithms are still based on TEGs and thus are only able to handle small road networks and few transfer points (the maximum evaluated network has only 10000 vertices). Besides, they assume a vehicle can at most occupy one more request along a pre-defined path, while we study a real-world taxi ride-sharing scenario in which vehicles are roaming in the street and can serve multiple requests at a time.

Several other studies assign multiple trip requests jointly and focus on improving the defined optimization goals instead of the matching efficiency. Hou et al. [161] aim to optimize the matching ratio. They first enumerate the multi-hop options for all passengers and then assign requests to passengers ordering by optimization strategies such

first assigning requests with less detour required. Coltin et al. [162] also enumerate all multi-hop options before dispatching, aiming to minimize the total distance traveled of all vehicles and the transfer costs of requests. None of these works applies to real-world scenarios due to the overhead of the exhaustive search.

Herbawi et al. [163] model the problem using time windows and reduce the multi-hop dispatching problem to a single-hop dispatching problem. However, they employ genetic algorithms that are only applicable to small instances (only 50 vehicles and 150 requests evaluated in their experiments). Besides, their genetic algorithms assume the vehicles have pre-defined source and destination

Yeung et al. [164] consider the road conditions. The requests may suffer from delayed service in congested roads. They transfer affected requests to other vehicles or compensate affected requests without transferring.

2.4.3 Discussion

By setting up meeting points and allowing transfers between vehicles, flexible ride-sharing brings more flexibility to passengers, improves the matching possibility, and increases the ratio of shared routes. Despite the advantages, adding flexibility further increases the complexity of the problem as more matches are possible. It is challenging to implement the flexible ride-sharing system in real-world due to the large search space.

Existing studies on flexible ride-sharing mainly come from the transportation area and most of them model the problem using integer linear programming. Besides, existing algorithms fail to notice the road network constraints and the expensive cost of network distances. On the other hand, most of the speed-up techniques proposed by the database area focus on the traditional ride-sharing (as described in Section 2.3). Directly applying them on flexible ride-sharing lacks efficiency due to the large set of possible matches. Designing efficient and scalable flexible ride-sharing algorithms is a key method to provide better ride-sharing services. An interesting research problem is to carefully integrate the traditional ride-sharing indices to reduce the search space and prune infeasible matches in an early stage.

2.5 Summary

In this chapter, we reviewed efficient shortest-path and nearest neighbor algorithms in road networks. We found that most of existing algorithms build an index and store pre-computed distance information to speed up the query time. A superior algorithm should achieve fast query time while requiring low pre-computation costs. We further surveyed exiting ride-sharing studies. Many algorithms have been proposed to improve the operation of ride-sharing service provider and the experience of passengers. Efficiency is the main bottleneck of these algorithms as they need to exhaustively compare the dispatching quality of different vehicles.

Although ride-sharing has been widely studied in the literature, several key problems still remain unsolved and affect the effective implementation of ride-sharing in real-world scenarios. To overcome the efficiency bottleneck, in Chapter 4, we propose a novel index to maintain the information of vehicles and passengers in ride-sharing . The index is lightweight and provides short query response. It has low updating cost and low memory consumption. The superior performance is achieved by a key insight that a vehicle can only visit a limited area once committed to a request and this area can be efficiently computed and indexed using ellipses.

As we surveyed in Section 2.4, flexible ride-sharing brings benefits to passengers, vehicles and the ride-sharing companies. Despite the advantages, implementing flexible ride-sharing in real-world scenarios still faces great challenges due to the added complexity and efficiency requirement. Taking advantage of the index proposed in Chapter 4, we further investigate multi-hop ride-sharing that allows transfers between vehicles in Chapter 5. We propose effective pruning strategies to reduce the search space and find multi-hop trips in real-time.

Section 2.4 shows that using meeting points is an effective way to improve the system flexibility and service quality to users. This strategy is commonly adopted by the real-world ride-sharing service such as UberPool [165] where passengers need to walk to their pick-up/drop-off points to meet vehicles. One key problem in this setting is how to efficiently find the nearest pick-up/drop-off (meeting) point for passengers. In Chapter 6, we model this problem as an ANN in road networks. The ANN query is a fundamental problem in spatial databases but has not been studied in the literature. We propose

a simple yet efficient algorithm that achieves constant query time relying on a light lightweight index to answer ANN queries.

Chapter 3

Basic Concepts

In this chapter, we explain basic concepts in road networks and the ride-sharing matching problem.

Road network. The *road network* is modeled as a directed graph $G = \langle V, E \rangle$, where V is a set of vertices and E is a set of edges. An edge $e(v_i, v_j) \in E$ connects two vertices v_i and v_j in V. Such two vertices are called *adjacent vertices*. Every edge $e(v_i, v_j)$ is associated with a *weight*, denoted by $w(v_i, v_j)$, which represents the cost of traveling between v_i and v_j. A *path* between two vertices v_i and v_j is an ordered list of edges between the two vertices, denoted by $P_{i,j}$. We use $|P_{i,j}|$ to denote the number of edges in the path, and $l(P_{i,j})$ to denote the *length* of the path, which is the sum of the weights of the edges in the path. The *shortest path* between v_i and v_j is the path between them with the smallest length. This smallest length is called the *shortest path distance* between v_i and v_j, denoted by $d_n(v_i, v_j)$. We denote the estimated travel time between v_i and v_j as $t(v_i, v_j)$ (which may be calculated based on their shortest path distance or fetched from a navigation service). The estimated travel time between two locations is dynamic and depends on the real-time traffic conditions. We further use $d_e(v_i, v_j)$ to denote the Euclidean distance between v_i and v_j.

Trip request. A *trip request* $r_i = \langle t, s, e, w, \epsilon, \eta \rangle$ consists of six elements: the issue time t, the source location s, the destination location e, the maximum waiting time w, the maximum detour ratio ϵ, and the number of passengers η. A set of trip requests is represented as $R = \{r_1, r_2, ..., r_n\}$.

For a trip request r_i, the issue time $r_i.t$ records the time when the trip request is sent. The maximum waiting time $r_i.w$ limits the *latest pickup time* of the request to be $r_i.lp = r_i.t + r_i.w$. The maximum detour ratio $r_i.\epsilon$ limits the extra detour time of the request. Together with the maximum waiting time, it constraints the *latest drop-off time* of the request to be $r_i.ld = r_i.t + r_i.w + t(s, e) \times (1 + \epsilon)$. Alternatively, a request can directly set the latest pickup and drop-off times or simply set the latest drop-off time and the latest pickup time is then calculated as $r_i.lp = r_i.ld - t(s, e) - r_i.t$. The difference between the latest drop-off time and the issue time, i.e., $r_i.ld - r_i.t$, is its maximum allowed travel time. A request with advanced booking can be represented by setting their issue time as their departure time.

Example 3.1. *Assume two trip requests* $r_1 = \langle$*9:00 am,* s_1*,* e_1*, 5 min, 0.2, 1*$\rangle$ *and* $r_2 = \langle$*9:07 am,* s_2*,* e_2*, 5 min, 0.2, 1*$\rangle$ *in Figure 3.1. The shortest travel times from* s_1 *to* e_1 *and from* s_2 *to* e_2*, i.e.,* $t(s_1, e_1)$ *and* $t(s_2, e_2)$*, are both 15 min. Then, the time constraints of* r_1 *and* r_2 *are:* $r_1.lp$*=9:00 am+5 min=9:05 am,* $r_2.lp$*=9:07 am+5 min=9:12 am,* $r_1.ld$*= 9:05 am+15 min*$\times$*1.2=9:23 am,* $r_2.ld$*=9:12 am+15 min*$\times$*1.2=9:30 am.*

Vehicle. A *vehicle* (car) c_i is represented as $c_i = \langle l, S, u, v \rangle$, where l denotes the location of the vehicle, S represents the *trip schedule* of the vehicle (detailed later), u is the vehicle capacity, and v is the travel speed. We use $C = \{c_1, ..., c_n\}$ denotes a set of vehicles.

Note that vehicles are moving in the street and their locations and travel speeds are changing dynamically. Following previous works [25, 27–29], we update the status of vehicles at specified time intervals, e.g., every second.

We track the occupancy status of the vehicles [166]. A vehicle is *empty* if it has not been assigned to any trip requests. Otherwise, the vehicle is *non-empty* and needs to follow their trip schedules.

Trip schedule. The trip schedule of a vehicle c_i, $c_i.S = \{p^0, p^1, ..., p^m\}$, is a sequence of meeting points (points on the road network) of trip requests, except for p^0 that records the current location of the vehicle, i.e., $p^0 = c_i.l$. We call a meeting point on a trip schedule a *stop*, and the path between every two adjacent stops p^{k-1} and p^k a *segment*, denoted as (p^{k-1}, p^k). We distinguish stops locating at the same place considering their different time constraints and request information.

Example 3.2. *Figure 3.1 shows an example trip schedule. The current time is 9:00 am and the vehicle is at l. Two trip requests (r_1, r_2) are assigned to the vehicle and the vehicle schedule is $(l, r_1.s, r_2.s, r_1.e, r_2.e)$.*

Trip schedule recorder. We follow a previous study [26] and record the *earliest estimated arrival time*, *latest arrival time*, and *slack time* of $c_i.S$ with three arrays $arr[]$, $ddl[]$, and $slk[]$:

(1) Earliest estimated arrival time $arr[k]$ records the estimated arrival time to stop p^k via the trip schedule.

(2) Latest arrival time $ddl[k]$ records the latest acceptable arrival time at stop p^k. If p^k is the pickup point of a request r_j, $ddl[k]$ is the latest pickup time of r_j, $ddl[k] = r_j.lp$. If p^k is the drop-off point of r_j, $ddl[k]$ is the latest drop-off time of r_j, $ddl[k] = r_j.ld$.

(3) Slack time $slk[k]$ records the maximum extra travel time allowed between (p^{k-1}, p^k) to satisfy the latest arrival time of p^k and all stops scheduled after p^k. For stop p^i, it only allows $ddl[i] - arr[i]$ detour time to ensure its latest arrival time. A detour between p^{k-1} and p^k will not only affect the arrival time of p^k but also that of all stops scheduled after p^k. Thus, a detour between p^{k-1} and p^k must guarantee the latest arrival time of p^k and all stops scheduled after p^k, i.e., $slk[k] = min\{ddl[i] - arr[i]\}, i = k, ..., m$. $slk[k]$ can be calculated by referring to $slk[k+1]$, i.e., $slk[k] = min\{(ddl[k] - arr[k]), slk[k+1]\}$. The *maximum allowed travel time* between (p^{k-1}, p^k) is $arr[k] - arr[k-1] + slk[k]$.

Example 3.3. *The arrays of the trip schedule in Figure 3.1 are shown in Table 3.1. The earliest estimated arrival time of the stops is computed based on the arrival time of previous stops and the shortest travel time between stops, e.g., $arr[1]$=9:00 am+3 min= 9:03 min, $arr[2]$ =9:03 am+5 min= 9:08 min. The latest arrival time of the stops is determined by the corresponding trip requests, e.g., the latest arrival time of p^1 is the latest pickup time of r_1, i.e., $ddl[1] = r_1.lp$=9:05 am. $ddl[k] - arr[k]$ represents the allowed detour time before visiting p^k to ensure $ddl[k]$, e.g., p^1 allows 9:05 am-9:03 am=2 mins detour before it and p^2 allows 9:12 am-9:08 am =4 mins detour before it. $slk[k]$ records the minimum allowed detour time of p^k and all stops after p^k, e.g., a detour before p^3 will not only affect the arrival time of p^3 but also that of p^4. Thus, $slk[3] = min\{5 min, 4 min\}$=4 min.*

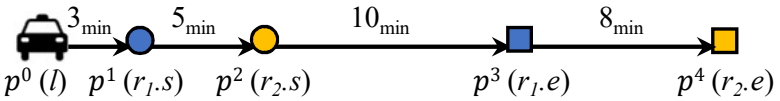


FIGURE 3.1: A vehicle schedule example at 9:00 am.

TABLE 3.1: Recorded data for the trip schedule in Figure 3.1.

p^k	$arr[k]$	$ddl[k]$	$ddl[k] - arr[k]$	$slk[k]$
p^1	9:03 am	9:05 am	2 min	2 min
p^2	9:08 am	9:12 am	4 min	4 min
p^3	9:18 am	9:23 am	5 min	4 min
p^4	9:26 am	9:30 am	4 min	4 min

Valid trip schedule. To form a *valid trip schedule*, the following trip constraints need to be satisfied:

(1) *Point order constraint:* Trip schedule $c_i.S$ must visit the pickup location $r_j.s$ before the drop-off location $r_j.e$, for any trip request r_j assigned to vehicle c_i.

(2) *Time constraint.* Trip schedule $c_i.S$ must meet the constraints for every request r_j assigned to vehicle c_i, i.e., r_j needs to be picked up before $r_j.lp$ and be dropped off before $r_j.ld$.

(3) *Capacity constraint.* At any time when c_i is traveling with trip schedule $c_i.S$, the number of passengers in the vehicle must be within the vehicle capacity.

Insertion heuristic. In our study, we apply the insertion heuristic to schedule route of vehicles. The insertion heuristic assumes that the order of existing trip schedules of vehicles is unchanged and new stops are inserted to the existing schedules. The insertion heuristic is widely adopted in previous studies of ride-sharing algorithms [26, 121, 133, 167]. The *insertion position* k indicates that the new stop is inserted after the stop p^k of the schedule. Note that insertions cannot be before p^0 as it represents the current location of the vehicle.

Vehicle schedule. The trip schedule of vehicle c_i in multi-hop ride-sharing is a sequence of locations $c_i.S = \langle p^0, p^1, p^2, ..., p^m \rangle$, where p^i is a node in the road network representing a *stop* that is a source, destination or transfer point of an assigned request.

Match. A match of a new request r_n assigns a vehicle $c_i \in C$ to serve r_n and add the the pick-up and drop-off location of r_n to the schedules of c_i.

TABLE 3.2: Ride-sharing datasets.

Name	# vertices	# edges	# requests
NYC	166,296	405,460	448,128
CD	254,423	467,773	259,343

TABLE 3.3: Road networks datasets.

Name	# vertices	# edges	Description
NY	264,346	733,846	New York (Undirected)
COL	435,666	1,057,066	Colorado (Undirected)
FLA	1,070,376	2,712,798	Florida (Undirected)
NW	1,207,945	2,840,208	Northwest USA (Undirected)
CAL	1,890,815	4,657,742	California & Nevada (Undirected)
E	3,598,623	8,778,114	Eastern USA (Undirected)
W	6,262,104	15,248,146	Western USA (Undirected)
CTR	14,081,816	34,292,496	Central USA (Undirected)
Europe	18,010,173	42,188,664	Europe (Directed)
USA	23,947,347	58,333,344	Full USA (Undirected)

Feasible match. Given a new trip request r_n, assigning c_i to serve r_n is *feasible* if adding r_n into the trip schedule of c_i yields a valid trip schedule. Vehicle c_i is then a *feasible vehicle* for r_n.

Similar to the previous studies [26, 121, 133], we assume that the source and the destination of the new trip request are inserted or appended to the current schedule of the matching vehicle.

Dataset. In Chapter 4 and Chapter 5, we perform experiments on real-world road network datasets extracted from OpenStreetMap [168], *New York City (NYC)* and *Chengdu (CD)*. We transform the coordinates to Universal Transverse Mercator (UTM) coordinates to support pruning based on Euclidean distance. We use real-world taxi requests on the two road networks [169, 170] and remove unrealistic ones, i.e., duration time less than 10 seconds or longer than 6 hours. There are 448,128 taxi requests (April 09, 2016) for NYC and 259,423 (November 18, 2016) taxi requests for Chengdu. Every request consists of a source, a destination, and an issue time. We map the locations to their nearest road network vertices. Similar to previous studies [25, 27], we assume the number of passengers to be one per request.

In Chapter 6, we run ANN queries on real-world road network datasets as listed in Table 3.3, which are created for the 9^{th} DIMACS Challenge [171]. Each undirected network has two datasets, a travel time dataset and a travel distance dataset, the edge weight of which correspond to the travel distance and the travel time between vertices, respectively. Note that the directed network *Europe* has only the travel time dataset.

Chapter 4

Efficient Single-hop Ride-sharing Matching Algorithm

Efficiency is a main challenge in dispatching algorithms. In Chapter 2, we reviewed existing dispatching algorithms and showed that most of the existing algorithms run in two phases to obtain shorter response time: pruning phase and selection phase. The pruning phase quickly prunes infeasible vehicles, and the selection phase selects the optimal matches among the remaining vehicles. The selection phase is usually expensive as it requires a detailed check on every remaining vehicle, which needs to run the costly shortest path query multiple times. Developing effective and efficient pruning algorithms to reduce the number of potential vehicles is the key to speed up the selection phase and improve dispatching efficiency. Existing pruning strategies, as reviewed in Section 2.3.2, either suffer from poor pruning performance or depend on a large index that requires expensive update cost. To address the limitations, this chapter proposes an efficient and scalable pruning algorithm called GeoPrune. GeoPrune records a set of circles and ellipses during the matching process, which enables fast retrieval for potential vehicles and low update cost for the dynamic scenario.

Part of the contents of Chapter 4 has been published in the following paper:
Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, Lars Kulik. *GeoPrune: Efficiently Matching Trips in Ride-sharing Through Geometric Properties*, International Conference on Scientific and Statistical Database Management (**SSDBM**) 2020.

4.1 Overview

To find matches for trip requests, existing algorithms typically employ two stages: **pruning** and **selection**. The pruning stage filters out infeasible vehicles that cannot meet the service constraints of trip requests, e.g., vehicles that are too far away. From the remaining vehicles, the selection stage selects the optimal vehicles and adds the new trip requests to their routes. The computation time of the selection stage largely depends on the effectiveness of the pruning stage (i.e., the number of remaining vehicles) as it usually requires exhaustive checks on all remaining vehicles regarding the optimization goal. The pruning stage is thus crucial for both the efficiency of the selection stage and the overall matching efficiency.

We study efficient pruning of infeasible vehicles for fast matching. We focus on finding vehicles that satisfy the service constraints of trip requests rather than any particular optimization goal. Thus, our solution is generic and can be easily integrated with selection algorithms for various optimization goals. We consider essential service constraints in ride-sharing studies, the *latest arrival times* of trip requests [8, 25–29, 33–37]. Vehicles violating the constraints are infeasible matches and filtered out.

Pruning infeasible vehicles in real-time is challenging. First, ride-sharing is a highly dynamic process. New requests are arriving frequently and vehicles are moving continuously. A pruning algorithm has to not only effectively prune infeasible vehicles but also quickly update any information needed for future pruning. Second, the pruning process needs to consider the constraints of not only the new trip request but also the trip requests that are currently being served by the vehicles. Checking all these constraints poses significant challenges to the algorithm efficiency.

Existing pruning algorithms maintain dynamic indices over the road network. A simple pruning strategy is to partition the road network space into grid cells and dynamically record the grid cell where each vehicle resides. To match a trip request, only the vehicles in the nearby grid cells of the trip request source location need to be examined [26]. Such a strategy finds nearby vehicles but overlooks the future directions of vehicles and requests. Thus, it may return many infeasible vehicles. To obtain a higher efficiency, two approximate algorithms, *Tshare* [28] and *Xhare* [29], were proposed. Tshare precomputes pair-wise distances between grid cells and records the cells on the route of

each vehicle. To match a request, Tshare checks the cells within a distance threshold of the request source/destination and retrieves vehicles passing these cells in a certain time range. Xhare, on the other hand, clusters the road network and records reachable clusters for vehicles given the time constraints. To match a request, Xhare returns all vehicles that can make a detour to the cluster where the request source/destination resides. Both algorithms may fail to find all feasible vehicles due to approximation errors such as in distance estimation, and their indices may have high storage and update costs.

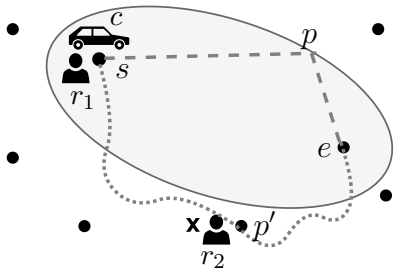


FIGURE 4.1: Illustration of our key idea.

To overcome the limitations above, we propose novel pruning strategies based on geometric properties of service constraints. Our strategies are built upon the following intuition. As Figure 4.1 shows, consider a vehicle c that has been assigned to a request r_1 with a trip from point s to point e. Vehicle c is now at s and needs to reach e within time t_1 (e.g., t_1 minutes), as constrained by r_1's latest drop-off time. To meet the time constraint t_1, vehicle c can visit a point p on its way to e only if $d_e(s,p)/v_{max} + d_e(p,e)/v_{max} \leq t_1$, where d_e is the Euclidean distance between two points, and v_{max} is the maximum vehicle speed. Obviously, the vehicle may need to travel longer than the Euclidean distance as its movement is constrained by the roads. Also, it may not be able to always travel at the maximum speed. Thus, even if point p satisfies this inequality, vehicle c may still be unable to visit p. On the other hand, *if point p does not satisfy this inequality, vehicle c must not visit p.* The above inequality defines an ellipse as shown in the figure. Any point outside this ellipse violates the inequality and must not be visited by c. Thus, if there is another request r_2 from a different user at point p', we can safely prune vehicle c from consideration if p' is not in the ellipse of c. This forms the basis of our pruning strategies.

Following the idea above, we propose an efficient *geometry-based pruning* algorithm (**GeoPrune**) for ride-sharing that bounds the search space for vehicles using ellipses.

We further index these ellipses using efficient data structures such as R-trees for fast search and updates. The construction of the ellipses is independent of the underlying road network and thus our algorithm is applicable to dynamic traffic-aware scenarios when vehicles may travel with different routes and speed. For every new trip request, our algorithm returns the pruning results by applying several point/range queries on the R-trees. Among the candidates, the optimal one is computed and returned with a separate selection algorithm satisfying the optimization goal. Once a trip request is assigned to a vehicle, we insert its source and destination to the vehicle route. Experimental results show that GeoPrune can prune most infeasible vehicles, which substantially reduces the computational costs of the selection stage and improves the overall matching efficiency.

The ellipse idea was explored in several matching problems [159, 172, 173]. However, their exhaustive search is inapplicable in the real-time dynamic ride-sharing settings where the relevant ellipses need to be retrieved and updated frequently and efficiently. Besides, existing algorithms represent the pruning area of every vehicle using a single ellipse to cover its entire route. Such an ellipse may be too loose to achieve effective pruning. In contrast, our algorithm uses multiple ellipses to tightly bound the pruning area of a vehicle, which achieves more effective pruning.

Our main contributions are as follows:

- We propose novel pruning strategies to filter infeasible vehicles for trip requests. Our pruning strategies are based on geometric properties, which eliminate expensive precomputation and update costs, making them suitable for large networks and highly dynamic scenarios.
- Based on the pruning strategies, we propose an algorithm named GeoPrune that can filter out most infeasible vehicles. It significantly reduces the computational costs of the selection stage and the overall matching process. Our theoretical analysis shows that the running time of GeoPrune is $O(\sqrt{|S||C|} + |S||C| \log(|S||C|))$, where $|S|$ is the maximum number of stops of the vehicle schedules and $|C|$ is the number of vehicles. GeoPrune takes $O(|S| \log^2(|S||C|))$ time to update the states for a newly assigned trip request. During every time slot, GeoPrune takes $O(|S| \log(|S||C|) + |C| \log^2 |C|)$ time to update for moving vehicles.
- Experiments on real datasets confirm the effectiveness and efficiency of our algorithm. Comparing with the state-of-the-art, it reduces the number of potential

vehicles in nearly all cases by an order of magnitude and the update time by two to three orders of magnitude.

4.2 Preliminaries

4.2.1 Matching Objective

Problem definition. Given a road network G, a set of vehicles C, a set of requests R, and an optimization objective O, we aim to match every request $r \in R$ with a feasible vehicle $c \in C$ to optimize O.

We examine a popular optimization objective, *minimizing the total increased travel distance (time)* [8, 25, 26, 28, 29]. Suppose that the total travel time of the current trip schedules of all vehicles is T, and the total travel time becomes T' after assigning vehicles in C to serve requests in R, our optimization goal O is to minimize $T' - T$.

Minimizing the total increased distance for all vehicles is NP-complete [28], and the future trip requests are unknown. A common solution is to greedily assign each trip request to an optimal vehicle [26–28, 121] ordering by their issue time. For every trip request, we assign it to a feasible vehicle such that the increased distance of the vehicle trip schedule is minimized.

4.2.2 Pruning and Selection

We take a two-stage approach to solve the problem:

(1) **Pruning.** Given a new request r_n, the pruning stage filters out infeasible vehicles and returns a set of vehicle candidates C'.

(2) **Selection.** Given a set of vehicle candidates C', the selection stage finds the optimal feasible vehicle in C'.

In what follows, we develop algorithms for the pruning stage. Observing that empty vehicles can be pruned by applying existing spatial network algorithms [73, 78], we distinguish non-empty/empty vehicles and focus on pruning non-empty vehicles.

4.3 Geometric-based Pruning

When a new trip request arrives, we find an optimal feasible vehicle and add the source and destination of the new trip request to the vehicle trip schedule. As discussed before, the trip schedule of the vehicle must satisfy the service constraints of all trip requests assigned to it including the new trip request. This is the basis of our pruning strategies.

There are two possibilities to add a stop to a trip schedule, either inserting it into a segment of the schedule or appending it to the end. For example, to add a new stop p to the trip schedule in Figure 3.1, we can either insert it to a segment to form a new schedule such as $(p^0, p, p^1, p^2, p^3, p^4)$ (we cannot insert before p^0 because p^0 is the current location of the vehicle) or append it to the end where the schedule becomes $(p^0, p^1, p^2, p^3, p^4, p)$. We say that a stop is *added* to a schedule if it is either inserted or appended to the schedule and the adding is *valid* if it still generates a valid trip schedule.

We first detail the criteria to determine whether adding the source or the destination of a new trip request is valid. These are based on constraints of the new trip and the existing trip schedule. Then, we summarize these criteria into three pruning rules.

4.3.1 Constraints Based on Existing Trip Requests

Given a segment (p^{k-1}, p^k), if we insert a new stop p to it, the path from p^{k-1} to p^k becomes (p^{k-1}, p, p^k). The travel time from p^{k-1} to p^k becomes $t(p^{k-1}, p) + t(p, p^k)$, which must be no larger than the maximum allowed travel time of the segment $arr[k] - arr[k-1] + slk[k]$ to satisfy the constraints of exiting trip requests.

The maximum allowed travel time limits the area that the vehicle can reach between p^{k-1} and p^k. Our key observation is that such a reachable area can be bounded using an **ellipse** $vd[k]$, and we call it the *detour ellipse* of the segment.

Definition 4.1. The *detour ellipse* $vd[k]$ *of a segment* (p^{k-1}, p^k) is an ellipse with p^{k-1} and p^k as its two focal points, and the major axis length $vd[k].major$ equals to the maximum allowed travel time multiplied by the vehicle speed v, i.e., $vd[k].major = (arr[k] - arr[k-1] + slk[k]) \cdot v$

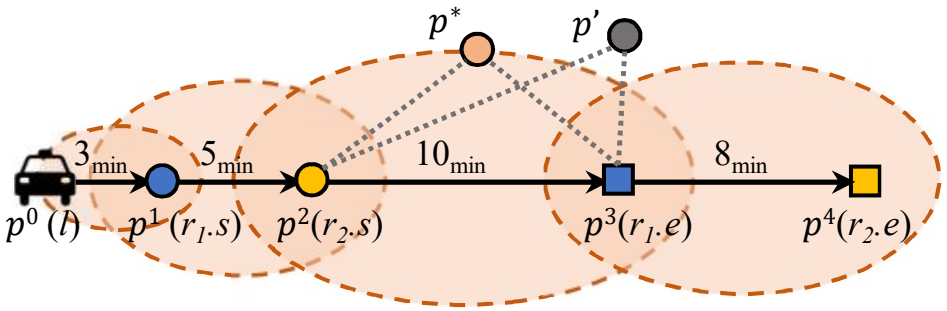


FIGURE 4.2: Detour ellipses of the trip schedule in Figure 3.1.

Lemma 4.2. *For a segment (p^{k-1}, p^k), if a point p is outside of $vd[k]$, $t(p^{k-1}, p) + t(p, p^k)$ will exceed the maximum allowed travel time. The segment is therefore invalid for inserting p.*

Proof. According to the definition of ellipses, if a point p is outside of the ellipse, the sum of the Euclidean distances $|p^{k-1}p| + |pp^k|$ must be greater than $vd[k].major$. Since any road network distance between two points is no smaller than their Euclidean distance (triangle inequality), the sum of road network distances $d_n(p^{k-1}, p) + d_n(p, p^k)$ is at least as large as $|p^{k-1}p| + |pp^k|$ and thus must also be greater than $vd[k].major$. The time required to travel such a distance thus exceeds the maximum allowed travel time and violates the latest arrival time of existing stops. □

Example 4.1. *Figure 4.2 shows the detour ellipses of the trip schedule illustrated in Figure 3.1. For segment (p^2, p^3), the slack time is 4 min and thus the maximum allowed travel time from p^2 to p^3 is 10 min + 4 min = 14 min. We make an ellipse with p^2 and p^3 as the two focal points and the major axis length being 14 min multiplied by the vehicle speed, i.e., $|p^2p^*| + |p^*p^3| = (14\,\text{min} \cdot v)$ for a point p^* on the ellipse. If a point p' is outside this ellipse, then the Euclidean distance $|p^2p'| + |p'p^3| > (14\,\text{min} \cdot v)$. The road network distance $d_n(p^2, p') + d_n(p', p^3)$ will also be greater than $(14\,\text{min} \cdot v)$ and the corresponding travel time with speed v will exceed 14 min, which violates the service constraint of exiting trip requests. Therefore, it is invalid to insert p' between (p^2, p^3).*

Lemma 4.2 shows that any point outside of the constructed ellipse is unreachable and thus the ellipse provides an upper bound of reachable areas. Next, we further show that the ellipse is also a lower bound of the reachable area regardless of the underlying road network, i.e., the ellipse is tight.

Lemma 4.3. *The detour ellipse $vd[k]$ tightly bounds the points that the vehicle can reach between segment (p^{k-1}, p^k) without violating the constraints of its existing tripe schedule.*

Proof. We prove by contradiction. Suppose that there is a smaller ellipse $vd[k]'$ with the same foci as ellipse $vd[k]$ and a major axis length of $vd[k].major - \epsilon$ ($\epsilon > 0$ is a sufficiently small value), which bounds all reachable points. This ellipse is fully enclosed by $vd[k]$. Now consider a point p^* on the boundary of $vd[k]$, which is outside $vd[k]'$ by definition. If the underlying road network happens to contain two straight routes from p^{k-1} to p^* and from p^* to p^k, which allows the vehicle to travel with the maximum speed. Then, p^* is reachable and it is outside $vd[k]'$. This contradicts the claim that $vd[k]'$ bounds all reachable points and completes the proof. □

Most existing ride-sharing matching algorithms do not consider the variations in traffic, and they assume a constant travel speed [37]. We replace the constant speed assumption with a **maximum** speed when computing the ellipses. This enables our approach to avoid false negatives if vehicles travel at varying speeds: all feasible vehicles are kept (by Lemma 4.2) as long as they do not exceed the maximum speed. We later show that using the maximum speed still preserves pruning efficiency. In practical implementation, we may also use different maximum speeds for different areas, e.g., in Victoria (a state in Australia), the speed limit in most built-up areas is under 60 km/h while that in rural areas is under 120 km/h.

The ellipse construction is independent of the vehicle trajectories. It only relies on the maximum allowed travel time and the endpoints of a trip segment. *Vehicles are not restricted to follow the shortest paths but are flexible to take any dynamic routes at varying speeds.* We record the ellipses of vehicles and update them only if the corresponding segments change. Specifically, when a trip request is assigned to a vehicle, we update the vehicle trip schedule and recompute the ellipses. Meanwhile, when the vehicles reach stops on their trip schedules, the corresponding segments become obsolete. We remove the ellipses of such obsolete segments.

Due to the real-time traffics and dynamic paths of vehicles, the actual arrival times at stops may be delayed and thus affect the vehicles' reachable area. $arr[]$ records the earliest arrival times and the ellipses always bound the reachable area. These allow lazy

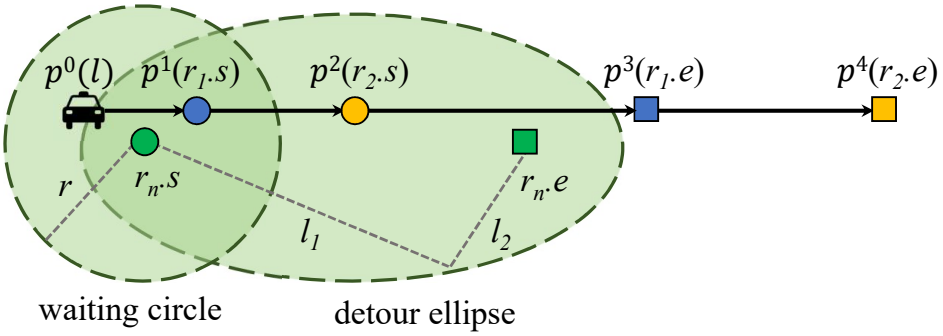


FIGURE 4.3: Waiting circle and detour ellipse of r_n, $r = r_n.w \cdot v$, $l_1 + l_2 = (r_n.ld - r_n.t) \cdot v$.

updates to vehicle ellipses when vehicles move, i.e., we do not need to recompute the ellipses when the actual arrival time is delayed.

4.3.2 Constraints Based on the New Request

Next, we analyze the service constraints of new requests.

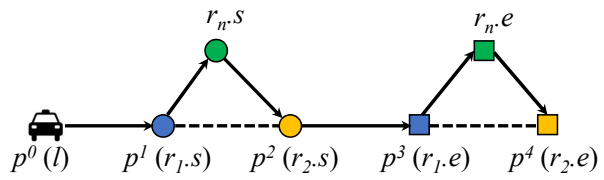
Latest pickup time constraint. Recall that $r_n.w$ denotes the maximum waiting time to ensure the latest pickup time of the new request r_n. We define a *waiting circle* with $r_n.w$.

Definition 4.4. The *waiting circle of* r_n, denoted by $r_n.wc$, is a circle centered at $r_n.s$ and with $r_n.w \cdot v$ as its radius.

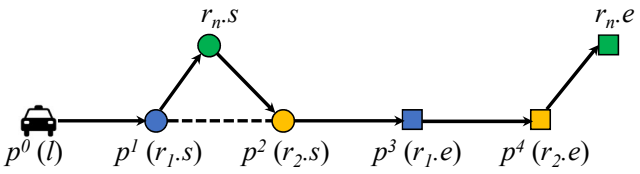
Lemma 4.5. *If it is valid to add $r_n.s$ after a stop p^k in $c_i.S$, then p^k and all stops before p^k must be covered by $r_n.wc$.*

Proof. The waiting circle bounds the area a vehicle can reach before picking up r_n to ensure the latest pickup time of r_n. Points outside of $r_n.wc$ have Euclidean distances (and hence network distances) to $r_n.s$ greater than $r_n.w \cdot v$. If a vehicle needs to visit a point outside of $r_n.wc$ before reaching $r_n.s$, it cannot pickup r_n before the latest pickup time $r_n.lp$. □

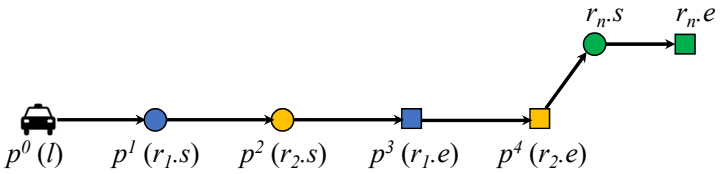
Example 4.2. *Figure 4.3 shows the waiting circle of a new request r_n. The source $r_n.s$ can only be added after the stops in the waiting circle $r_n.wc$, i.e., p^0 or p^1. If the vehicle visits p^2 (outside of the waiting circle) before $r_n.s$, it will not pick up r_n before the latest pickup time of r_n. Thus, it is invalid to add $r_n.s$ after p^2 or any stops afterwards.*



(A) insert source and insert destination.



(B) insert source and append destination.



(C) append source and append destination.

FIGURE 4.4: Cases to add a new trip request to a trip schedule.

Latest drop-off time constraint. Similar to the detour ellipses of segments, we define a detour ellipse for a new request r_n to ensure the latest drop-off time of r_n.

Definition 4.6. The *detour ellipse* $r_n.rd$ *of a new trip request* r_n is an ellipse with $r_n.s$ and $r_n.e$ as the two focal points. The major axis length is the maximum allowed travel time of r_n multiplied by the speed v, i.e., $r_n.rd.major = (r_n.ld - r_n.t) \cdot v$.

The detour ellipse of r_n restricts the area that a vehicle can visit while serving r_n. After picking up r_n (reaching $r_n.s$), if the vehicle visits any stop outside of the detour ellipse of r_n, it will not be able to reach the destination $r_n.e$ before the latest drop-off time $r_n.ld$.

Lemma 4.7. *Let* $r_n.s$ *be added after stop* p^s *in the trip schedule* $c_i.S$ *of a vehicle* c_i*. If it is valid to add* $r_n.e$ *after* p^k *in* $c_i.S$*, then* p^k *and all stops scheduled between* p^s *and* p^k *must be covered by* $r_n.rd$*.*

Example 4.3. *The detour ellipse of* r_n *is shown in Figure 4.3. If* $r_n.s$ *is added after* p^0*, then* $r_n.e$ *can only be added after either* p^0 *or stops inside of the detour ellipse, i.e.,* p^1 *and* p^2*. Adding* $r_n.e$ *after later stops (e.g.,* p^3*) will violate the latest drop-off time of* r_n*.*

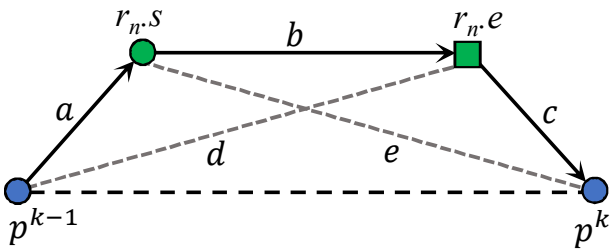


FIGURE 4.5: The special case of insert-insert.

4.3.3 Pruning Rules

There are three cases as shown in Figure 4.4 when adding a new trip request r_n to the trip schedule $c_i.S$ of a vehicle c_i:

(1) **insert-insert:** insert $r_n.s$ into a segment of $c_i.S$ and insert $r_n.e$ into the same or another segment of $c_i.S$.

(2) **insert-append:** insert $r_n.s$ into a segment of $c_i.S$ and append $r_n.e$ to the end of $c_i.S$

(3) **append-append:** append $r_n.s$ and $r_n.e$ to the end of $c_i.S$.

We next analyze the conditions that c_i needs to satisfy so that adding r_n to $c_i.S$ is valid for each case.

Insert-insert. Figure 4.4a shows the insert-insert case, where both $r_n.s$ and $r_n.e$ are inserted into some segments of the trip schedule $c_i.S$. According to Lemma 4.5, a segment is valid for inserting a stop only if the stop is inside its detour ellipse. Thus, both $r_n.s$ and $r_n.e$ must be inside the detour ellipse of at least one segment of $c_i.S$.

A special case is to insert both $r_n.s$ and $r_n.e$ to the same segment of $c_i.S$, as shown in Figure 4.5. In this case, both $r_n.s$ and $r_n.e$ must be inside the detour ellipse of the segment.

Lemma 4.8. *A segment (p^{k-1}, p^k) is valid to insert both $r_n.s$ and $r_n.e$ only if $r_n.s$ and $r_n.e$ are both included in the detour ellipse of the segment $vd[k]$.*

Proof. We use Figure 4.5 to illustrate our proof. The Euclidean distances among the stops are represented by a, b, c, d, e. Suppose that (p^{k-1}, p^k) is valid to insert both $r_n.s$ and $r_n.e$, and the schedule becomes $(p^{k-1}, r_n.s, r_n.e, p^k)$ after the insertion. Traveling

between (p^{k-1}, p^k) must satisfy the maximum allowed travel time constraint. Thus, $d_n(p^{k-1}, r_n.s) + d_n(r_n.s, r_n.e) + d_n(r_n.e, p^k) = (t(p^{k-1}, r_n.s) + t(r_n.s, r_n.e) + t(r_n.e, p^k)) \cdot v \leq (arr[k] - arr[k-1] + slk[k]) \cdot v_{max} = vd[k].major$. Since the Euclidean distance between two stops is no larger than their road network distance, $a + b + c \leq d_n(p^{k-1}, r_n.s) + d_n(r_n.s, r_n.e) + d_n(r_n.e, p^k) \leq vd[k].major$. According to the triangle inequality, $e < b + c$. Thus, $a + e < a + b + c \leq vd[k].major$. The Euclidean distance sum from $r_n.s$ to p^{k-1} and p^k is smaller than $vd[k].major$ and $r_n.s$ must be inside $vd[k]$. Similarly, $d < a + b$, and $d + c < a + b + c \leq vd[k].major$. $r_n.e$ must be inside $vd[k]$. □

The pruning rule for the insert-insert case is as follows:

Lemma 4.9. *A vehicle c_i may be matched with r_n in the insert-insert case only if it satisfies:*

(1) there exists a segment of $c_i.S$ with the detour ellipse that covers $r_n.s$, i.e., $r_n.s \in vd[k]$, $k = 1, ..., m$; and

(2) there exists a segment of $c_i.S$ with the detour ellipse that covers $r_n.e$, i.e., $r_n.e \in vd[k]$, $k = 1, ..., m$.

Insert-append. Figure 4.4b illustrates the insert-append case. According to Lemma 4.2, to insert $r_n.s$, there must be a segment in the trip schedule of c_i whose detour ellipse cover $r_n.s$. Meanwhile, any stop between $r_n.s$ and $r_n.e$ needs to be covered by the detour ellipse of r_n (see Lemma 4.7).

Checking all the stops between $r_n.s$ and $r_n.e$ against the detour ellipse of r_n is non-trivial. For fast pruning, we only check the ending stop of the current trip schedule: if the ending stop is outside of the detour ellipse of r_n, it is invalid for appending $r_n.e$. Take Figure 4.4b as an example. We only check if p^4 is inside the detour ellipse of r_n. This simplified rule may bring in a small number of infeasible vehicles, which will be filtered later as explained in the next paragraphs. The pruning rule for the insert-append case is:

Lemma 4.10. *A vehicle c_i may be matched with r_n in the insert-append case only if it satisfies:*

(1) there exists a segment of $c_i.S$ with the detour ellipse that covers $r_n.s$, i.e., $r_n.s \in vd[k]$, $k = 1, ..., m$; and

(2) the ending stop of the vehicle schedule, p^m, is covered by the detour ellipse of r_n, i.e., $p^m \in r_n.rd$.

Append-append. Figure 4.4c illustrates the append-append case, where we append both $r_n.s$ and $r_n.e$ to the end of the trip schedule. In this case, r_n will not affect any exiting stops. Only the service constraints of r_n need to be considered. No stop is scheduled between $r_n.s$ and $r_n.e$, and hence the detour constraint of r_n is satisfied already. We only need to check is the waiting time constraint of r_n. According to Lemma 4.5, all stops scheduled before $r_n.s$ must be covered by the waiting circle of r_n, e.g., the vehicle needs to visit p^0, p^1, p^2, p^3, p^4 before picking up $r_n.s$ in Figure 4.4c. Hence, all these stops should be covered by the waiting circle of r_n. Similar to the insert-append case, we only check the ending stop.

Lemma 4.11. *A vehicle c_i may be matched with r_n in the append-append case only if the ending stop of its trip schedule, p^m, is covered by the waiting circle of r_n, i.e., $p^m \in r_n.wc$.*

We omit the proof of Lemma 4, Lemma 6, Lemma 7, and Lemma 8 due to the space limitation. In our implementation, we use a set of Minimum Bounding Rectangle (MBR) to represent ellipses and circles as they are easier to operate on and tightly bound the ellipses and circles.

4.3.4 Applying the Pruning Rules

When a new request r_n arrives, we first compute the waiting circle and the detour ellipse of r_n. Then, we compute a set of vehicle candidates that may match r_n based on Lemmas 4.9, 4.10, 4.11.

To facilitate the pruning, we compute sets of vehicles that:

(1) have trip schedule segments with detour ellipses that cover $r_n.s$ (for the insert-insert and insert-append cases);

(2) have trip schedule segments with detour ellipses that cover $r_n.e$ (for the insert-insert case);

(3) have the ending stop of the trip schedule covered by $r_n.wc$ (for the append-append case);

(4) have the ending stop of the trip schedule covered by $r_n.rd$ (for the insert-append case).

To find vehicles that satisfy a pruning rule, we just need to join the relevant sets of vehicles computed above. For example, vehicles that may satisfy the insert-insert case are those in both the first and the second sets above.

R-tree based pruning. We build two R-trees [72] to accelerate the computation process, although other spatial indices may also be applied. One R-tree store the detour ellipses of all segments for all vehicle trip schedules, denoted by T_{seg}; the other R-tree stores the location of the ending stops of all non-empty vehicles, denoted as T_{end}. We run four queries:

(1) $Q_1 = T_{seg}.pointQuery(r_n.s)$ is a point query that returns all segments whose detour ellipses cover $r_n.s$; each segment returned may be used to insert $r_n.s$.

(2) $Q_2 = T_{seg}.pointQuery(r_n.e)$ is a point query that returns all segments whose detour ellipses cover $r_n.e$; each segment returned may be used to insert $r_n.e$.

(3) $Q_3 = T_{end}.rangeQuery(r_n.wc)$ is a range query that returns all ending stops covered by $r_n.wc$; each ending stop returned may be used to append $r_n.s$ and $r_n.e$.

(4) $Q_4 = T_{end}.rangeQuery(r_n.rd)$ is a range query that returns all ending stops covered by $r_n.rd$; each ending stop returned may be used to append $r_n.e$.

The returned segments and ending stops are further pruned based on their time and capacity constraints. For each segment (p^{k-1}, p^k) returned for inserting $r_n.s$ ($r_n.e$), we check whether the insertion violates the latest arrival time of p^k and $r_n.s$ ($r_n.e$). The schedule between (p^{k-1}, p^k) becomes $(p^{k-1}, r_n.s(r_n.e), p^k)$ after the insertion. For the new schedule, the arrival time of $r_n.s$ ($r_n.e$) and p^k is estimated based on the arrival time of p^{k-1} plus the travel time between them. If the earliest estimated arrival time of $r_n.s(r_n.e)$ or p^k exceeds their latest arrival time, the segment is discarded. For each ending stop (p^m) returned for appending $r_n.s$ ($r_n.e$), we estimate the arrival time of $r_n.s$ ($r_n.e$) with the appended schedule by summing up the end stop arrival time and the travel time from the end stop to $r_n.s$ ($r_n.e$). If the estimated time exceeds the latest

arrival time of $r_n.s$ ($r_n.e$), we also discard the ending stop. We also check the capacity constraint for segments to insert $r_n.s$. If a segment (p^{k-1}, p^k) is returned for inserting $r_n.s$, we sum up the number of passengers carried in (p^{k-1}, p^k) and that of r_n and discard the segment if the sum exceeds the capacity.

Let the sets of vehicles corresponding to the segments and ending stops returned by the four queries above (after filtering) be O_1, O_2, O_3 and O_4, respectively. The sets of vehicles satisfying the three pruning cases are: $F_1 = O_1 \cap O_2$ (insert-insert); $F_2 = O_1 \cap O_4$ (insert-append); $F_3 = O_3$ (append-append). The union of these three sets, $F = F_1 \cup F_2 \cup F_3$, is returned as the candidate vehicles.

Processing empty vehicles. Empty vehicles do not have designated trip schedules yet. We only need to check whether they are in the waiting circle of the new request by a range query over all empty vehicles using the waiting circle as the query range.

Since our goal is to minimize the system-wide travel time, the optimal empty vehicle is the nearest one. We thus take a step further and directly compute the optimal empty vehicle with a network nearest neighbor algorithm named *IER* [73] that has been shown to be highly efficient [78] (other algorithms may also apply [174]).

4.4 The GeoPrune Algorithm

Next, we describe our *pruning*, *match update*, and *move update* algorithms based on the pruning rules above.

Pruning. Algorithm 1 summarizes the pruning process. For a new request r_n, we compute its waiting circle and detour ellipse (line 1). We run four queries to compute Q_1, Q_2, Q_3, and Q_4 as described in Section 4.3.4 (lines 2 to 5). Each returned segment and ending stop is checked against the capacity and time constraints as described in Section 4.3.4 (lines 6 to 8). The vehicles of the remaining segments and ending stops are our candidates (lines 10 to 15).

Match update. If a new trip request r_n is matched with a vehicle c_i, we update the data structures as summarized in Algorithm 2. If c_i is an empty vehicle, the vehicle now becomes occupied. We remove the vehicle from an R-tree denoted by T_{ev} that stores the empty vehicles for fast nearest empty vehicle computation (lines 1 and 2). Otherwise, we

Algorithm 1: Prune non-empty vehicles

Input: A new trip request r_n
Output: a set of possible vehicles to serve r_n
`// Pruning stage`
1 $r_n.wc \leftarrow$ waiting circle of r_n; $r_n.rd \leftarrow$ detour ellipse of r_n
2 $Q_1 \leftarrow T_{seg}.pointQuery(r_n.s)$
3 $Q_2 \leftarrow T_{seg}.pointQuery(r_n.e)$
4 $Q_3 \leftarrow T_{end}.rangeQuery(r_n.wc)$
5 $Q_4 \leftarrow T_{end}.rangeQuery(r_n.rd)$
6 **for** *an element in* Q_1, Q_2, Q_3, *and* Q_4 **do**
7 **if** *the time or capacity constraint is violated* **then**
8 remove the element
9 Record the corresponding vehicles of the elements in Q_1, Q_2, Q_3, Q_4 in O_1, O_2, O_3, O_4.
10 $F, F_1, F_2, F_3 \leftarrow \emptyset$
11 $F_1 \leftarrow O_1 \cap O_2$ `// insert-insert case`
12 $F_2 \leftarrow O_1 \cap O_4$ `// insert-append case`
13 $F_3 \leftarrow O_3$ `// append-append case`
14 $F \leftarrow F_1 \cup F_2 \cup F_3$
15 **return** F

Algorithm 2: Update index - match

Input: A new trip request r_n and the matched vehicle c_i
1 **if** c_i *empty* **then**
2 $T_{ev}.remove(c_i)$
3 **else**
4 **for** *a segment in the trip schedule of* c_i **do**
5 remove the ellipse of the *segment* from T_{seg}
6 T_{end}.remove(ending stop of c_i)
7 add $r_n.s$ and $r_n.e$ to the trip schedule of c_i
8 **for** *a segment in the trip schedule of* c_i **do**
9 compute the detour ellipse of the *segment*
10 insert the ellipse of the *segment* into T_{seg}
11 T_{end}.insert(the end stop of c_i)

first remove the segments and the ending stop of c_i from the two R-trees T_{seg} and T_{end} (lines 4 to 6). Then, we add the new trip request to the trip schedule of the matched vehicle c_i (line 7). Based on the updated vehicle schedule, we recompute the detour ellipses and insert them into T_{seg} (lines 8 to 10). The new ending stop is also inserted into T_{end} (line 11).

Move update. We also update the data structures when the vehicles move, as summarized in Algorithm 3. At every time point, we check if a vehicle has reached a stop

Algorithm 3: Update index - move

Input: A moving vehicle c_i

1 $P \leftarrow$ obsolete segments of c_i
2 **for** $p \in P$ **do**
3 $\quad T_{seg}.remove(p)$
4 **if** c_i *reaches the ending stop* **then**
5 $\quad T_{end}.remove$(ending stop of c_i)
6 $\quad T_{ev}.insert(c_i)$

in its trip schedule. If yes, the segments before the reached stop become obsolete and their detour ellipses are removed from T_{seq} (lines 1 to 3). When the vehicle reaches its ending stop, the vehicle becomes empty. We remove it from T_{end} and insert it into T_{ev} (lines 4 to 6).

4.4.1 Algorithm Complexity

We measure the complexity of our algorithm by two parameters $|S|$ and C that are key to our algorithm: $|S|$ is the maximum number of stops of the vehicle schedules (a small constant constrained by the vehicle capacity) and $|C|$ is the number of vehicles. We note that instead of using $|S|$, we may also use the number of requests $|R|$ because there is a linear relationship: $|S||C| \propto |R|$. The state-of-the-art pruning algorithms [28, 29] lack complexity analysis.

Pruning. It takes $O(1)$ time to compute the waiting circle and the detour ellipse of a new request. There are at most $|S||C|$ MBRs in T_{seg} and $|C|$ entries in T_{end}. The point query on T_{seg} returns at most $|S||C|$ results and hence the complexity is $O(\sqrt{|S||C|} + |S||C|)$ [175]. At most $|C|$ results will be returned from the range query on T_{end} and the complexity is $O(\sqrt{|C|} + |C|)$. The time complexity of the queries on R-trees is thus $O(\sqrt{|S||C|} + |S||C|)$. Checking the time and capacity constraints takes $O(|S||C| + |C|)$ time.

It takes $O(|S||C| + |C|)$ time to retrieve the corresponding vehicles and at most $|C|$ vehicles will be returned in each set after sorting ($O(|S||C| \log(|S||C|))$ time). The set intersection hence takes O($|C|$) time [176]. The overall time complexity of GeoPrune is thus $O(\sqrt{|S||C|} + |S||C| \log(|S||C|))$.

Update. When a new trip request is assigned to a vehicle c_i, it takes $O(\log |C|)$ time to delete c_i from T_{ev} if c_i was empty, $O(|S| \log$ $(|S||C|))$ time to remove invalid segments from T_{seg}, and $O(\log |C|)$ time to remove the obsolete record in T_{end} [175]. For the new schedule of c_i, there are at most $|S|$ new segments. It thus takes $O(|S|)$ time to compute the new detour ellipses for the new segments and $O(|S| \log^2(|S||C|))$ time to insert the ellipses to T_{seg} [175]. The overall update time for a new request is $O(|S| \log^2(|S||C|))$.

When a vehicle moves, the number of obsolete scheduled stops is at most $|S|$. Therefore, the time to remove obsolete vehicle ellipses from T_{seg} is $O(|S| \log(|S||C|))$. At most $|C|$ vehicles change their status while moving, hence the time to update T_{end} and T_{ev} is at most $O(|C| \log^2 |C|)$. Therefore, the overall update time for moving all vehicles in a time slot is $O(|S| \log(|S||C|) + |C| \log^2 |C|)$.

4.5 Experiments

In this section, we study the empirical performance of GeoPrune and compare it against the state-of-the-art. All algorithms are implemented in C++ r on a 64-bit virtual node with a 1.8 GHz CPU and 128 GB memory from an academic computing cloud (Nectar [177]) running on OpenStack. The travel distance between points is computed by a shortest path algorithm on road networks [24].

4.5.1 Experimental Setup

Dataset. We described the datesets of the road network and the trip requests in Chapter 3.

Implementation. We run simulations following the settings of previous studies [25, 26]. The vehicle initial positions are randomly selected from the road network vertices. Non-empty vehicles move on the road network following their trip schedules (shortest paths) while empty vehicles stay at their last drop-off location until they are committed to new requests. Similar to previous studies [25, 27], we use a constant travel speed for all edges (48km/h). For the selection step, we apply the state-of-the-art insertion algorithm [26]

TABLE 4.1: Experiment parameters

Parameters	**Values**	**Default**
Number of vehicles	2^{10} to 2^{17}	2^{13}
Waiting time (min)	2, 4, 6, 8, 10	4
Detour ratio	0.2, 0.4, 0.6, 0.8	0.2
Number of requests	20k to 100k	60k
Frequency of requests (# requests/second)	1 to 10	refer to table 3.2
Transforming speed (km/h)	20 to 140	48

to minimize the total travel distance for all methods. If no satisfying vehicle is found for a new trip request, the trip request is ignored. We use in-memory R-trees for indexing.

By default (Table 4.1), we simulate ride-sharing on 2^{13} vehicles with a capacity of 4 and 60,000 trip requests, and the maximum waiting time and the detour ratio are 4 min and 0.2.

Baselines. We compare with the following state-of-the-art pruning algorithms using their originally reported parameter values.

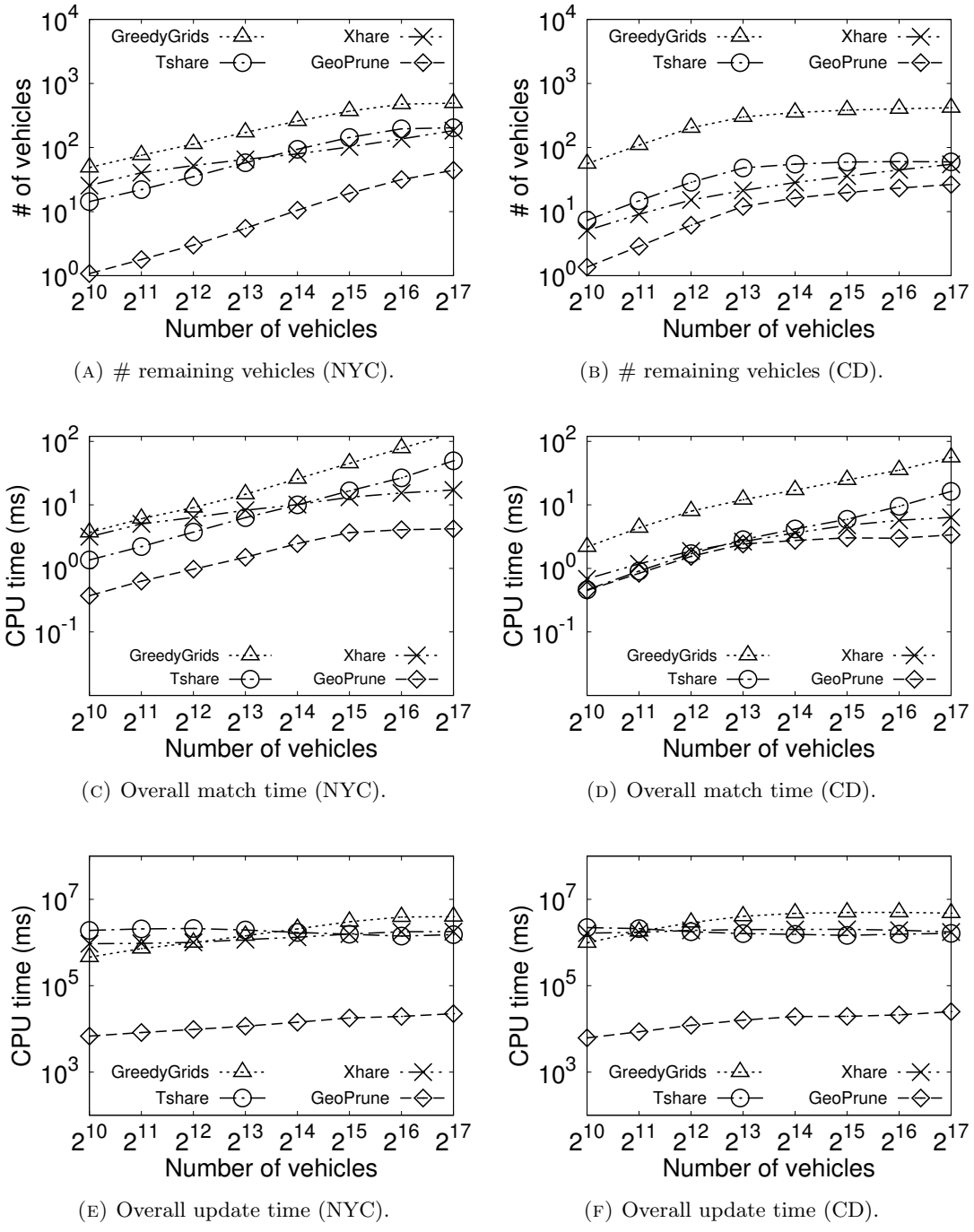
(1) **GreedyGrids [26].** This algorithm retrieves all vehicles that are currently in the nearby grid cells.

(2) **Tshare [28].** This is the single-side search algorithm of Tshare. (their dual-side search algorithm terminates when a feasible vehicle is found while we aim to find all feasible vehicles). The grid cell lengths of both GreedyGrids and Tshare are set to 1 km [26].

(3) **Xhare [29].** This algorithm only checks non-empty vehicles. For a fair comparison, we prune empty vehicles in Xhare using the same algorithm applied in our method (see Section 4.3.4). We optimize the update process by precomputing the pair-wise distance between clusters. The landmark size is set to 16,000 for NYC and 23,000 for Chengdu, and the grid length is set to 10 m. The maximum distance between landmarks in a cluster is set to 1 km.

Metrics. We measure and report the following metrics:

(1) *Number of remaining vehicles* – the number of remaining candidate vehicles after the pruning. Note that GeoPrune prunes empty vehicles and non-empty vehicles separately



(A) # remaining vehicles (NYC).

(B) # remaining vehicles (CD).

(C) Overall match time (NYC).

(D) Overall match time (CD).

(E) Overall update time (NYC).

(F) Overall update time (CD).

FIGURE 4.6: Effect of the number of vehicles.

with different criteria, and such a scheme is applied on Xhare to make it applicable. GreedyGrids and Tshare, however, process the two types of vehicles together and return both types after pruning. For consistency, we only compare the number of remaining non-empty vehicles.

(2) *Match time* – the total running time of the matching process, including both pruning and selection time.

(3) *Overall update time* – the overall match update and move update time.

(4) *Memory consumption* – the memory cost of the data structures of an algorithm.

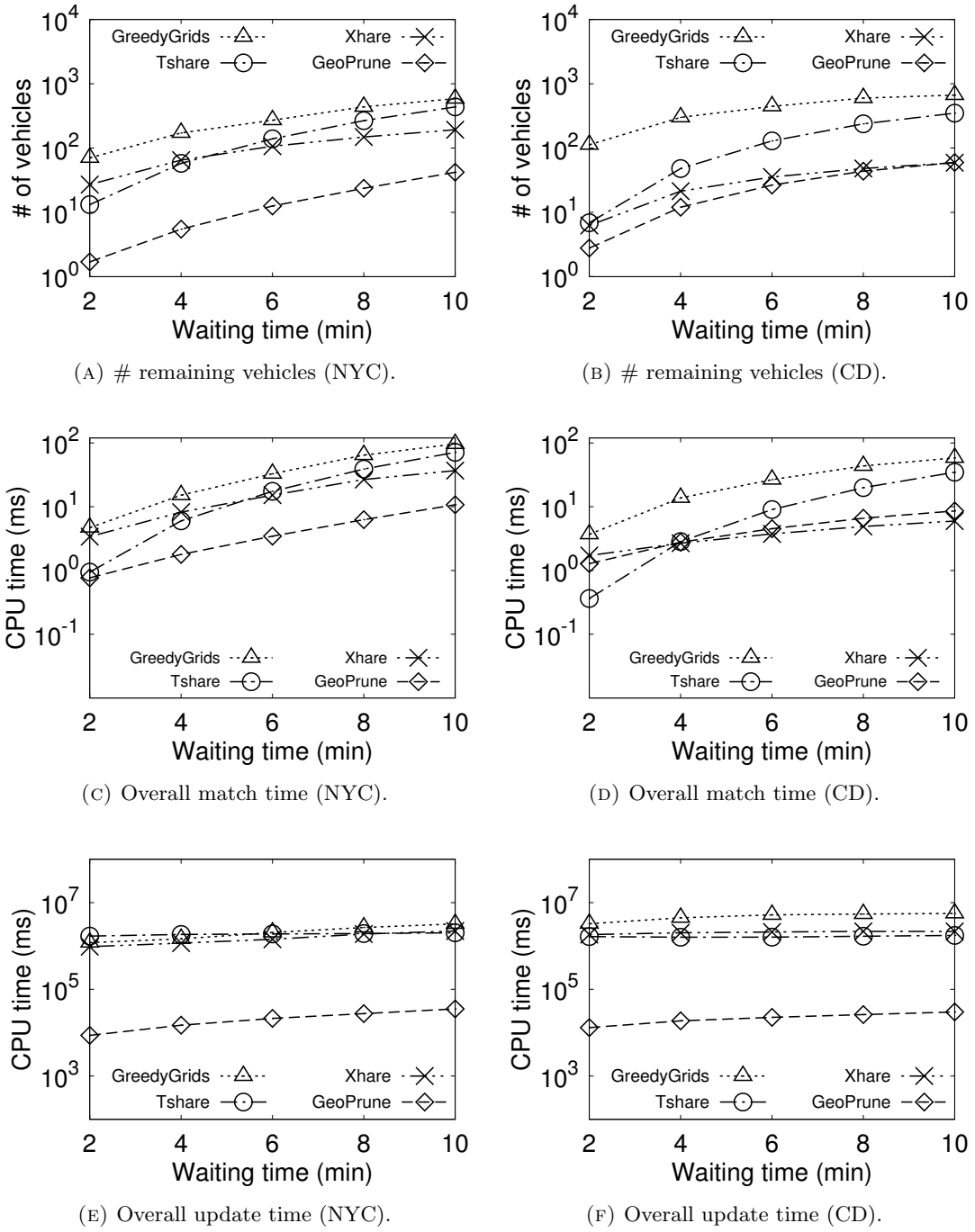
As we use the state-of-the-art selection algorithm [26], we obtain the same matching quality as GreedyGrids [26], including total increased travel distance (our optimization goal) and matching ratio. Tshare and Xshare are approximate algorithms and the false negatives may randomly impact the matching quality. We omit detailed matching quality results as we focus on pruning. We also omit the results on varying the capacity due to the space limit and the stable behavior of all algorithms (as observed in [26, 121]). GeoPrune consistently outperforms others in all capacity settings.

4.5.2 Experimental Results

Effect of the Number of Vehicles

Figure 4.6 shows the results on varying the number of vehicles. GeoPrune substantially reduces the number of remaining candidate vehicles. When there are 2^{13} vehicles, the average number of candidates of GeoPrune is only 5 on the NYC dataset, while the other algorithms return $57 \sim 172$ candidates per request. GreedyGrids returns the largest set of candidates as it retrieves all vehicles in nearby grid cells, among which only a few are feasible. Tshare and Xhare find fewer candidates than GreedyGrids but may result in false negatives due to approximation. Tshare and Xhare perform better on Chengdu than on NYC. The reason might be that requests of NYC have a higher frequency than those of Chengdu, while Tshare and Xhare are more sensitive to the frequency of trip requests (consistent with our results in Figure 4.10).

The number of remaining vehicles largely affects the running time of the selection stage and the overall match time. As shown in Figure 4.6c and Figure 4.6d, GeoPrune reduces the overall match time by 71% to 90% on the NYC dataset and up to 80% on the Chengdu dataset. Consistent with experiments shown in the previous studies [26, 37, 121], all algorithms exhibit longer pruning time with more vehicles as the number of vehicle candidates increases. The match time of Tshare and Xhare is comparable with GeoPrune



(A) # remaining vehicles (NYC).

(B) # remaining vehicles (CD).

(C) Overall match time (NYC).

(D) Overall match time (CD).

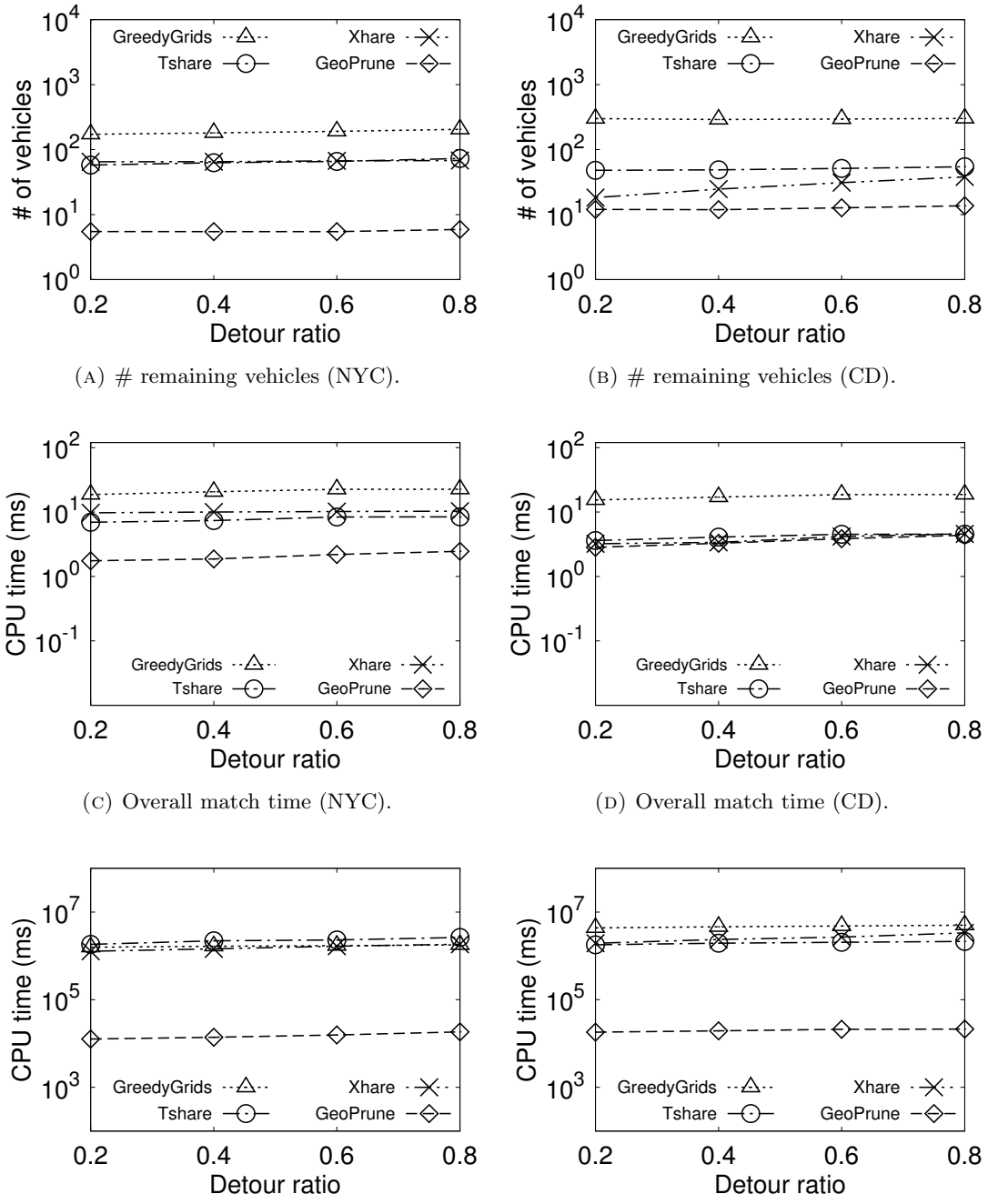
(E) Overall update time (NYC).

(F) Overall update time (CD).

FIGURE 4.7: Effect of the waiting time.

on the Chengdu dataset when the number of vehicles is small but continuously increases with more vehicles, showing that GeoPrune scales better with the increase in the number of vehicles.

As for the update cost, GeoPrune is two to three orders of magnitude faster since it only relies on circles and ellipses for pruning while others need real-time maintenance of



(A) # remaining vehicles (NYC).

(B) # remaining vehicles (CD).

(C) Overall match time (NYC).

(D) Overall match time (CD).

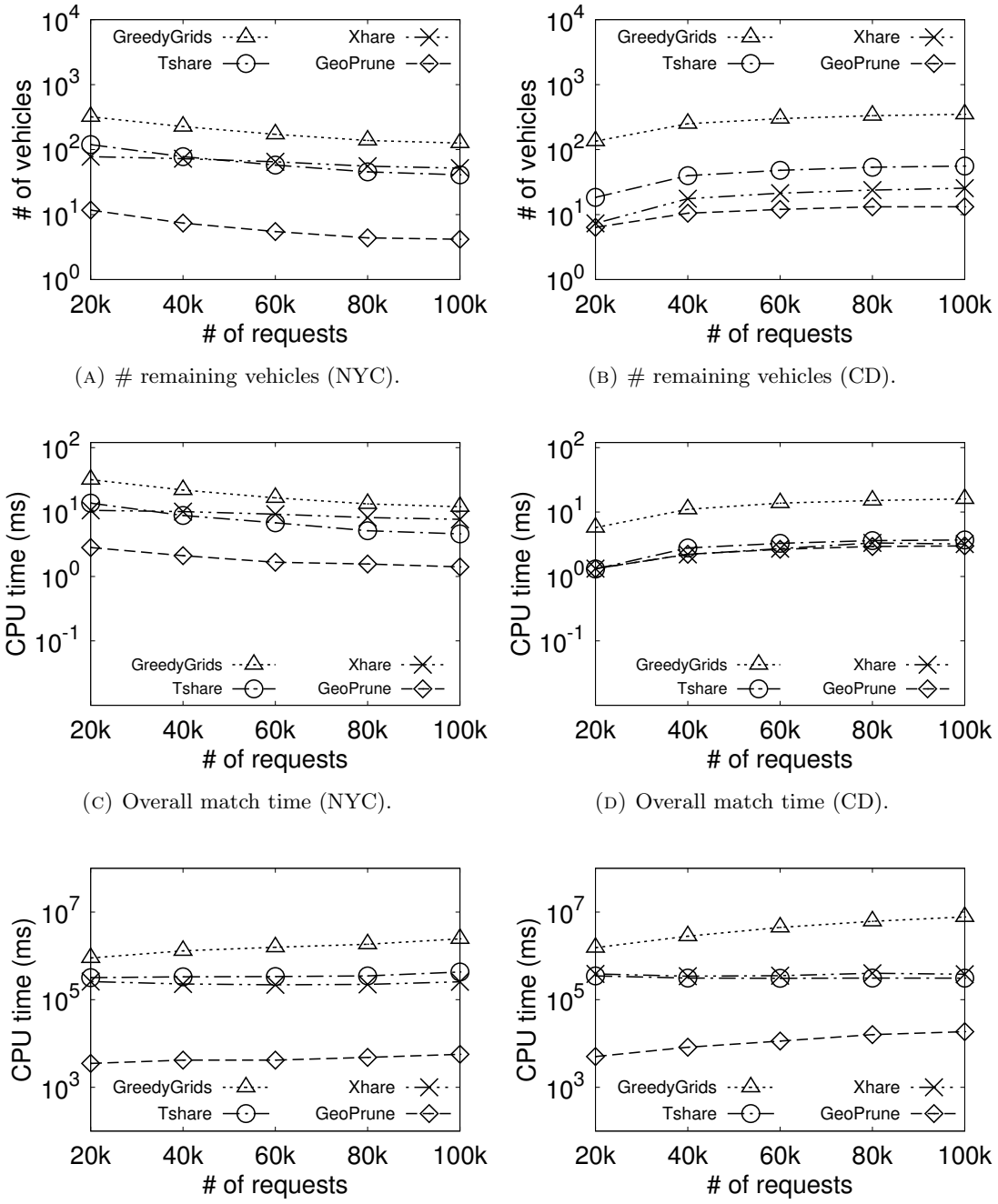
(E) Overall update time (NYC).

(F) Overall update time (CD).

FIGURE 4.8: Effect of the detour ratio.

indices on the networks.

Effect of the Maximum Waiting Time Figure 4.7 shows the experimental results when varying the maximum waiting time. All algorithms exhibit longer times with the increasing waiting time because of larger shareability between requests and more



(A) # remaining vehicles (NYC).

(B) # remaining vehicles (CD).

(C) Overall match time (NYC).

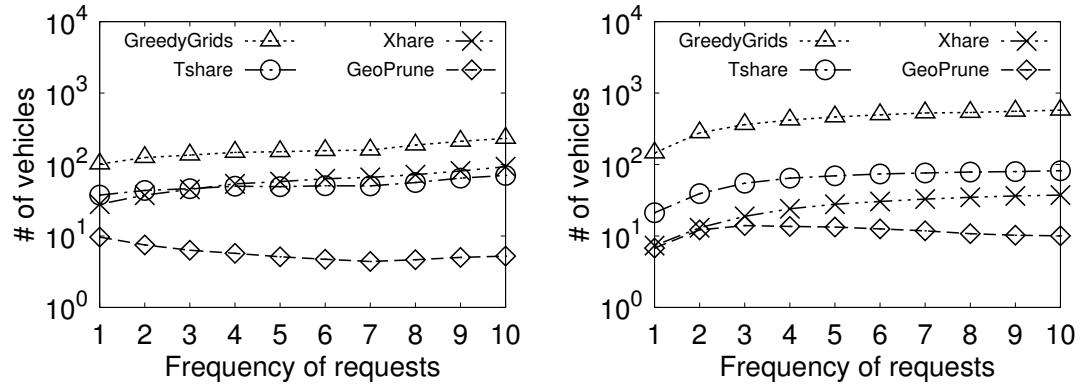
(D) Overall match time (CD).

(E) Overall update time (NYC).

(F) Overall update time (CD).

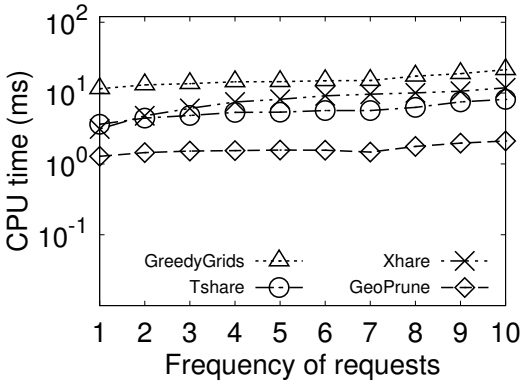
FIGURE 4.9: Effect of the number of requests.

returned vehicle candidates. Increasing the waiting time of requests leads to more possibilities for requests to share with each other and thus results in more candidates. GeoPrune again shows the best pruning performance in almost all cases. Tshare requires less match time than GeoPrune when the waiting time is 2 min on the Chengdu dataset. However, longer waiting time requires Tshare to check more grid cells and continuously increase their match time, which becomes five times slower than GeoPrune when the

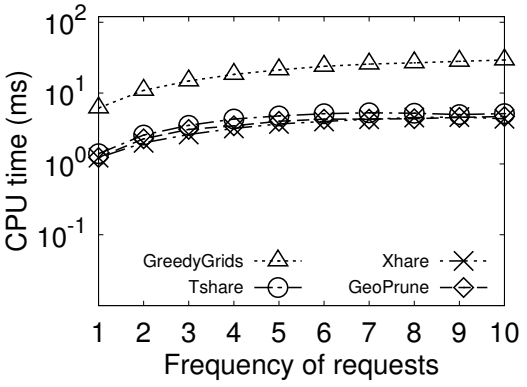


(A) # remaining vehicles (NYC).

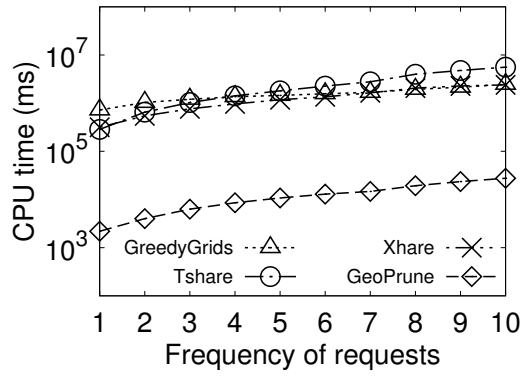
(B) # remaining vehicles (CD).



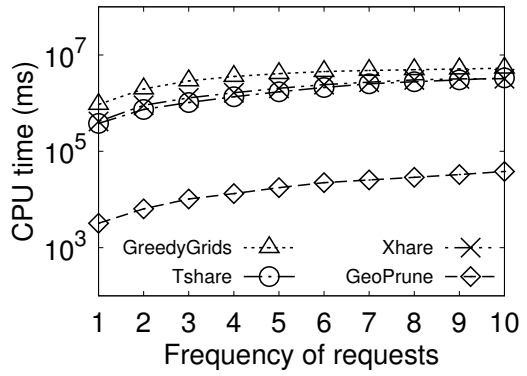
(C) Overall match time (NYC).



(D) Overall match time (CD).



(E) Overall update time (NYC).



(F) Overall update time (CD).

FIGURE 4.10: Effect of the frequency of requests.

waiting time is 10 min. Xhare finds fewer vehicles and requires less match time than GeoPrune when the waiting time is longer than 6 min on the Chengdu dataset. This is because Xhare assumes vehicles travel on predefined routes, and new requests can only be served on the way of these routes. A long waiting time brings more feasible vehicles with append-append case and Xhare may miss these vehicles.

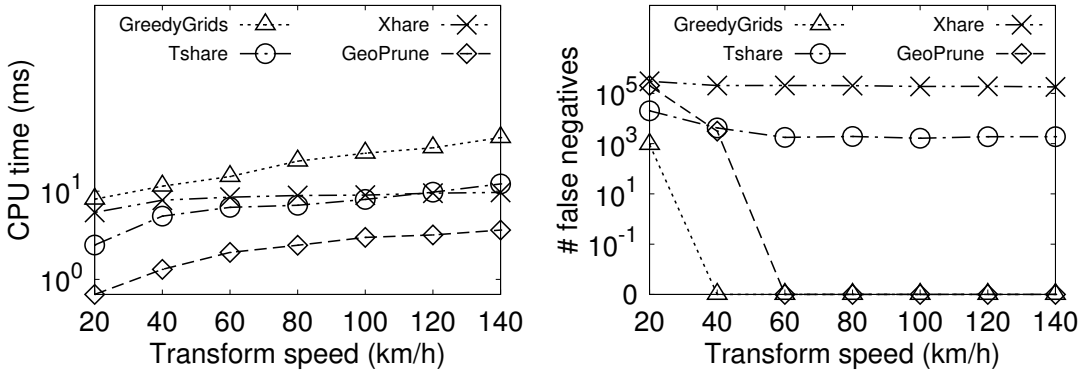
Figure 4.7e and Figure 4.7f show the update cost, which increases for all algorithms with the larger waiting time as the vehicle schedules become longer and more requests can be shared. Still, GeoPrune is two to three orders of magnitude faster on update than others.

Effect of the Detour Ratio Figure 4.8 shows the sensitivity over the detour ratio. Again, GeoPrune prunes more infeasible vehicles and its match time is three to ten times faster than others on the NYC dataset and comparable with Tshare and Xhare on the Chengdu dataset. The number of remaining vehicles of all algorithms keeps almost stable due to the limited shareability. The update cost of all algorithms remains stable (and three orders of magnitude smaller for GeoPrune) as the length of vehicle schedules is barely affected.

Effect of the Number of Trip Requests Figure 4.9 shows the experiments when the number of requests varies. Interestingly, algorithms show different behavior on the two datasets. When the number of requests changes from 20 k to 100 k, the candidates returned by GeoPrune for each request decreases from 11 to 4 on the NYC dataset but increases from 6 to 13 on the Chengdu dataset, meaning that the shareability between requests decreases on the NYC dataset while increases on the Chengdu dataset (may be caused by the different geographical distribution of requests and vehicles). The trend of the match time is consistent with that of the number of remaining vehicles, which again confirms that the match time is largely affected by the pruning effectiveness.

More trip requests correspond to longer simulation time and increase the total update cost (with GeoPrune still being two to three orders of magnitude cheaper in terms of update cost).

Effect of the Trip Request Frequency Figure 4.10 shows the scalability of algorithms with the frequency of trip requests varying from 1 to 10 requests per second over 3 hours. Note that the frequencies of the original NYC and Chengdu datasets are 5.19 and 3 requests per second respectively. To generate trip requests less frequent than the original datasets, we uniformly sample trip requests from the original datasets. As for more frequent trip requests, we extract a certain number of trip requests according to the frequency, e.g., 10,800×7=75,600 trip requests when the frequency is 7. We then uniformly distribute the request issue time over 3 hours.



(A) Overall match time.

(B) # of false negatives.

FIGURE 4.11: Effect of the transforming speed (NYC).

TABLE 4.2: Memory consumption (MB) (# vehicles = 2^{13}).

	GreedyGrids	**Tshare**	**Xhare**	**GeoPrune**
NYC	0.38	100.34	1546.40	6.56
Chengdu	1.67	9965.37	21282.46	6.43

All algorithms return more vehicle candidates with higher frequency while GeoPrune keeps almost stable, showing that GeoPrune provides tighter pruning and is more scalable to dynamic scenarios. The match time of GeoPrune consistently outperforms others on NYC dataset and is comparable with Tshare and Xhare on Chengdu dataset. The update cost of all algorithms grows with higher frequency due to more frequent updates while GeoPrune again outperforms others by two to three orders of magnitude.

Effect of the Transforming Speed: All algorithms need a speed value to transform the time constraint to distance constraint so that pruning based on geographical locations can be applied. Figure 4.11 shows the effect of the transforming speed. A higher speed enlarges the search space and thus all algorithms show longer match time. However, GeoPrune consistently performs efficient pruning with all speed values. Figure 4.11b shows the total number of false negatives wrongly pruned for 60,000 requests. Same as GreedyGrids, GeoPrune ensures no false negatives when the speed is greater than vehicle speed (48km/h), whereas Xhare and Tshare still have false negatives even with high transforming speed. This verifies the robustness of GeoPrune on real-time traffic conditions, where the transforming speed is set as the maximum speed so that the pruning is still correct and the processing time only increases slightly.

Memory Consumption Table 4.2 illustrates the maximum memory usage of the algorithms under the default setting. All state-of-the-arts consume more memory on the Chengdu dataset as it has a large road network, while GeoPrune keeps stable. For example, the grid size of Tshare in NYC is 46×46 but increases to 174×174 in Chengdu. GeoPrune, however, only maintains several R-trees and thus is less affected by the road network size. GreedyGrids has the smallest memory footprint as it only records a list of in-cell vehicles for each grid cell, which is consistent with the observation in [26]. Tshare and Xhare consume much more memory than GeoPrune due to the large road network index.

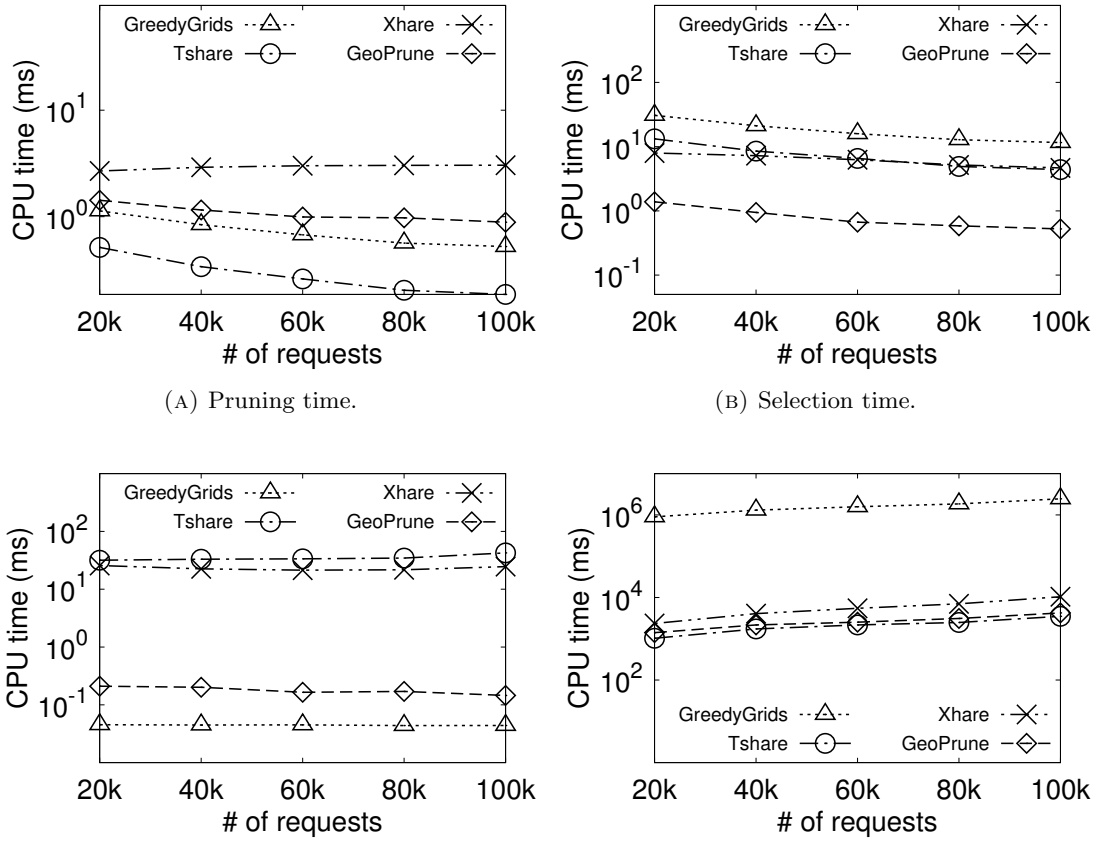
Cost Breakdown of Algorithm Steps Figure 4.12 compares the cost of different phases in the match process and update process when varying the number of requests on the NYC dataset. Figure 4.12a shows the cost of the pruning algorithms while Figure 4.12b shows the selection cost based on their pruning results. GeoPrune requires a slightly longer time for pruning than Tshare and Xhare but can reduce the selection time by more than 88% due to the fewer remaining vehicles. The selection time of algorithms (Figure 4.12b) is consistent with the number of remaining vehicles (Figure 4.9a), which again demonstrates that the selection step is largely affected by the number of remaining vehicles.

Figure 4.12c shows the update cost when a request is newly assigned. GeoPrune is slightly slower than GreedyGrids to update the R-trees. Xhare and Tshare, however, need much more time than GeoPrune to update the reachable areas of the matched vehicle while GeoPrune quickly bounds the areas by ellipses.

Figure 4.12d compares the update cost when vehicles are moving. GreedyGrids is two orders of magnitude slower than the other three algorithms because it needs to track the located grid cells of continuously moving vehicles.

4.6 Summary

We studied the dynamic ride-sharing matching problem and proposed an efficient algorithm named GeoPrune to prune infeasible vehicles to serve trip requests. Our algorithm applies circles and ellipses to bound the areas that vehicles can visit without violating the service constraints of passengers. The circles and ellipses are simple to compute



(A) Pruning time.

(B) Selection time.

(C) Match update time.

(D) Move update time.

FIGURE 4.12: The cost breakdown of algorithm steps.

and further indexed using efficient data structures, which make GeoPrune highly efficient and scalable. Our experiments on real-world datasets confirm the advantages of GeoPrune in pruning effectiveness and matching efficiency.

Chapter 5

Efficient Multi-hop Ride-sharing Matching Algorithm

In Chapter 4, we studied efficient dispatching algorithms on single-hop ride-sharing. Single-hop ride-sharing only allows passengers to be served by one vehicle. However, passengers may find it hard to find direct trips especially when the supply of vehicles is in shortage. As illustrated in Chapter 2, previous studies showed that multi-hop ride-sharing that allows passengers to transfer between vehicles brings various benefits, e.g., providing more flexible service of passengers, increasing the number of served passengers, and decreasing the travel distance of vehicles. However, it is challenging to implement multi-hop ride-sharing in the real-world as the transfers largely increase the number of possible matches. Existing algorithms need to check all possible matches exhaustively, hence are only applicable to small instances but cannot be applied to the large-scale real-world scenarios.

This chapter proposes efficient and scalable multi-hop ride-sharing algorithms. In Chapter 4, we proposed the algorithm GeoPrune for efficient and effective pruning in single-hop ride-sharing. This chapter explains how to extend GeoPrune to the multi-hop ride-sharing. We show that it is imperative to improve its efficiency since the combinations of vehicles and transfer points may be large in multi-hop ride-sharing. We thus propose algorithms to achieve higher efficiency when more flexible transfer points are

Part of the contents of Chapter 5 has been published in the following paper:
Yixin Xu, Lars Kulik, Renata Borovica-Gajic, Abdullah Aldwyish, Jianzhong Qi. *Highly Efficient and Scalable Multi-hop Ride-sharing*, accepted by the International Conference on Advances in Geographic Information Systems (**SIGSPATIAL**) 2020.

available. We further speed-up the response time of the proposed algorithms by integrating techniques such as deep learning and approximating the transfer points. The acceleration strategies help to achieve faster matching time while providing comparable matching quality to the exact algorithms.

5.1 Overview

Chapter 4 proposed an efficient and scalable matching algorithm in ride-sharing. Many other dispatching (i.e., request matching) algorithms have been proposed to improve the efficiency and effectiveness of ride-sharing services in the literature [8, 28, 34, 36, 37, 121, 178]. However, most of them only consider direct trips but do not investigate the possibility of multi-hop trips. A multi-hop trip dispatches more than one vehicle to serve a request and the passengers will transfer between the dispatched vehicles. The benefits of enabling multi-hop trips have been documented in many studies [158–160, 163]. The proposed approaches provide more flexible trips, leading to an increased matching ratio, lower travel costs of vehicles, and reduced congestion.

Figure 5.1 shows an example road network that denotes the travel times (in minutes) at the edges. A passenger requests a trip from A to G and needs to arrive within 25 minutes at their destination. Two vehicles c_a and c_b offer shared trips and their scheduled routes are highlighted in red and blue, respectively. None of the vehicles can serve the request on time if they can only accept small detours (e.g., 5 minutes of additional trip time for other passengers). Nevertheless, the passenger can arrive on time through a multi-hop trip: vehicle c_a carries the passenger from A to D and vehicle c_b carries the passenger from D to G.

Despite its advantages, multi-hop ride-sharing has only be applied to small networks for a limited number of vehicles and passengers [158–160, 163] as they exhaustively enumerate all possibilities. Most of the existing multi-hop ride-sharing algorithms find multi-hop trips by searching for overlapping routes between passengers and vehic,les and enumerate all possible routes of vehicles and passengers, which leads to an exponential time complexity as even a slight detour generates a different route. Thus, previous methods are restricted to small road networks (at most 10000 nodes). Efficient solution

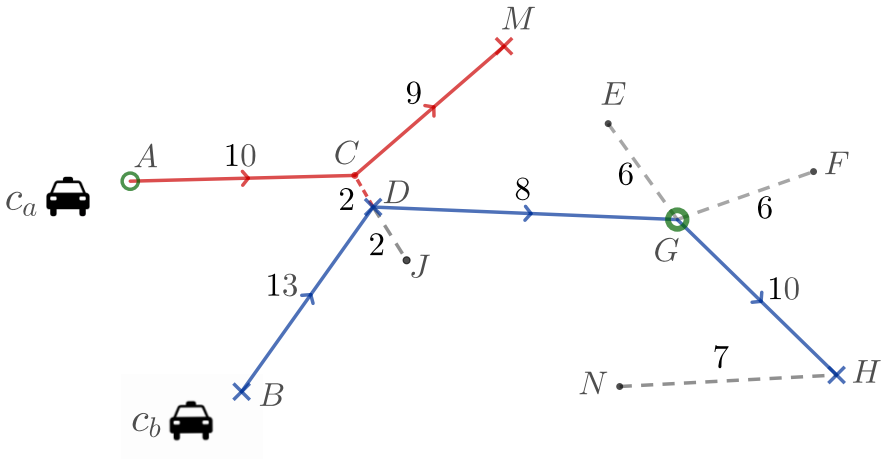


FIGURE 5.1: A multi-hop ride-sharing example.

for multi-hop ride-sharing in real-world scenarios for large city road networks remains challenging.

To fully exploit the benefits of multi-hop ride-sharing, we propose scalable real-time multi-hop dispatching algorithms for large scale real-world scenarios. We propose pruning strategies to reduce the search space using the time constraints of requests, thus achieving real-time responses. We propose two algorithms that cater for different scenarios: *Station-first* and *Vehicle-first*.

The Station-first algorithm first computes possible transfer points and then finds feasible vehicles. Its efficiency depends on the number of transfer points and is preferable when the transfer points are sparse. However, it might be less desired when more transfer points with a higher density are available. The Vehicle-first algorithm first prunes candidate vehicles that can serve a request (stage one) and then computes an optimal transfer point for each candidate trip (stage two). Based on the key observation that the time constraints of the committed requests limit the area (called the *reachable area*) that a vehicle can reach, we prune vehicles by computing whether or not their reachable areas cover the new request. We then solve the problem of finding an optimal transfer point by reducing it into a variation of a group nearest neighbor query problem [93]. The Vehicle-first algorithm avoids checking on all possible transfer points and achieves higher efficiency with dense transfer points.

Since in real applications the provision of real-time responses to requests may be more important than finding the optimal trip, e.g., when it rains and a passenger quickly

needs to find a trip. We propose two strategies to further accelerate the two stages of the Vehicle-first algorithm. The first strategy reduces the number of paired vehicles returned in stage one. In the exact Vehicle-first algorithm, we represent reachable areas of vehicles through ellipses (denoted as *bounding ellipses*) and pair vehicles if their bounding ellipses overlap. Since the actual reachable areas are often smaller than the bounding ellipses due to the road network constraints, we infer the actual reachable areas instead of using the ellipses, which leads to fewer overlapping vehicles and candidate trips to check. Recent studies [48, 51] show that deep learning has great potential in computing road network distances. Hence, we predict the actual reachable areas using deep learning techniques. We achieve accurate predictions by integrating the bounding ellipses. The second approximation strategy is performed in the second stage of the Vehicle-first algorithm, i.e., computating the optimal transfer points. Instead of exhaustively checking all potential transfer points, we only check a few estimated transfer point that seems optimal, which substantially reduces the search time.

The contributions of this chapter are summarized as follows:

- We propose scalable real-time multi-hop ride-sharing dispatching algorithms. To the best of our knowledge, this is the first work applicable to real-world ride-sharing scenarios with large sets of transfer points.
- We propose two algorithms to cater for different application scenarios. Both algorithms apply efficient and effective pruning strategies to reduce the search space and achieve high overall efficiency.
- We propose approximation strategies to reduce response time by utilizing deep learning and efficient indices.
- We experimentally verify the benefits of multi-hop ride-sharing and demonstrate the efficacy and efficiency of the proposed algorithms over real-world datasets. Our algorithms are two to three orders of magnitude faster than the state-of-the-art, and can be improved by another order of magnitude quicker response times if approximation strategies are applied.

5.2 Preliminaries

5.2.1 Problem Definition

Vehicle schedule. We denote the trip schedule of vehicle c_i as a sequence of locations $c_i.S = \langle p^0, p^1, p^2, ..., p^m \rangle$, where p^i is a node in the road network representing a *stop* that is a source, destination or transfer point of an assigned request.

As we explained in Chapter 3, after a stop p^k, the vehicle can only visit a restricted area to satisfy the maximum allowed travel time of the segment (p^k, p^{k+1}). We denote such an area as the *reachable area* after p^k, i.e., $reach(p^k)$. The slack time and maximum allowed travel time after the last stop p^m are infinite, and the reachable area after the last stop p^m covers the whole space.

We focus on direct trips and trips with one transfer point (i.e., 2-hop trips), since previous studies [158, 159, 161, 179] have shown that allowing more than one transfer in a trip brings marginal benefits.

For a direct trip, a vehicle will be dispatched to pick up the request and deliver the passenger(s) to their destination directly. As for a multi-hop (i.e., 2-hop) trip, two vehicles will be dispatched to the request with a transfer point assigned. The first vehicle carries the request from the source to the transfer point where the request transfers to the second vehicle to continue the remaining trip until the destination is reached. We denote the trip from the source to the transfer point as the **first itinerary** and the trip from the transfer point to the destination as the **second itinerary**.

A **match** of a new request r_n, denoted by $r_n.m = \langle c_1, c_2, \phi, \Gamma \rangle$, consists of four elements: the vehicle carrying the first itinerary c_1, the vehicle carrying the second itinerary c_2, the transfer point ϕ, and the insertion positions $\Gamma = \{s, \phi_1, \phi_2, e\}$ including four values:

1. $\Gamma(s)$: insertion position of the source to the first vehicle's schedule.
2. $\Gamma(\phi_1)$: insertion position of the transfer point to the first vehicle's schedule.
3. $\Gamma(\phi_2)$: insertion position of the transfer point to the second vehicle's schedule.
4. $\Gamma(e)$: insertion position of the destination to the second vehicle's schedule.

We use ϕ_1 and ϕ_2 to distinguish the transfer points in the two vehicle schedules if necessary. For direct matches, c_2 equals to c_1, and the insertions related to transfers become null, i.e., $\Gamma(\phi_1)$, $\Gamma(\phi_2)$.

Example 5.1. *In Figure 5.1, a multi-hop trip* $r_n.m = \langle c_a, c_b, D, \Gamma = \{c_a.1, c_a.1, c_b.2, c_b.2\}\rangle$ *of* r_n *indicates that* r_n *will be first carried by* c_a *and then transfer to* c_b *at the location* D*. Both source (A) and the transfer point (D) will be inserted after the first stop of* c_a*, updating the trip schedule from* $A \rightarrow M$ *to* $A \rightarrow \boldsymbol{A} \rightarrow \boldsymbol{D} \rightarrow M$*. Similarly, both the transfer point (D) and the destination (G) will be inserted after the second stop of the vehicle, updating the trip schedule from* $B \rightarrow D \rightarrow H$ *to* $B \rightarrow D \rightarrow \boldsymbol{D} \rightarrow \boldsymbol{G} \rightarrow H$*. Note that we record stops as opposed to vertices in the schedule, thus the schedule of* c_a *is* $A \rightarrow M$ *as opposed to* $A \rightarrow C \rightarrow M$*.*

For a multi-hop trip, the assigned two vehicles may reach the transfer point at different times. If the first vehicle arrives earlier, the passengers have to stay at the transfer point to meet the second vehicle. If the second vehicle arrives earlier, the second vehicle needs to wait at the transfer point until the first vehicle comes.

A feasible multi-hop match. A feasible multi-hop match should satisfy the time constraints of the new request and all existing requests. We consider the following service constraints:

- r_n must be picked up and dropped off on time, i.e., $arr[\Gamma(s)]+t(p^{\Gamma(s)}, r_n.s) \leq r_n.\tau_s$; $arr[\Gamma(e)]+det(\phi_2)+t(p^{\Gamma(e)}, r_n.e) \leq r_n.\tau_e$, where $det(x)$ denotes the additional trip time to visit the location x. $det(\phi_2)$ includes waiting time of the second vehicle at ϕ if necessary.
- The detour time of the first vehicle cannot exceed its maximum allowed detour time to ensure the latest arrival times of all committed requests, i.e., $det(r_n.s) \leq slk[\Gamma(s)+1]$; $det(r_n.s) + det(\phi_1) < slk[\Gamma(\phi_1)+1]$.
- The detour time of the second vehicle cannot exceed its maximum allowed detour time, i.e., $det(\phi_2) \leq slk[\Gamma(\phi_2)+1]$; $det(\phi_2) + det(r_n.e) \leq slk[\Gamma(e)+1]$.

Matching objective. Although our proposed algorithms are applicable to other optimization goals, we simplify the discussion by studying a popular optimization objective – minimizing the travel distance of vehicles [26, 28, 121]. Assume that the overall scheduled travel distance of all vehicles was T before dispatching requests, and the distance

becomes T' after dispatching all requests $R = \{r_1, r_2, ..., r_n\}$, the goal is to minimize the additional distance $T' - T$ to serve all requests.

As the optimization problem is NP-hard [26, 28] we apply a popular strategy [26, 28, 121] that sequentially matches passengers ordered by their issue times. For every new request, we dispatch an optimal trip that minimizes the additional travel distance $T' - T$ to serve it. We compute the optimal direct trip using the state-of-the-art single-hop dispatching algorithm proposed in Chapter 4 and the optimal multi-hop trip using the proposed multi-hop dispatching algorithm. An optimal one among them is assigned to the request.

We keep dispatches unchanged once allocated and assume vehicles to follow the scheduled routes. We leave the discussion of handling incidents such as cancellation of requests for future work.

Algorithm complexity of the state-of-the-art [159]. We next analyze the time complexity of the state-of-the-art algorithm [159]and show that this algorithm takes $O(k|V||S|(|E| + |V|\log|V|) + k^2|V|^2|T|^4|S||C| + k|V|^2\log|V|)$ time to run. Here, k : the number of alternative paths; $|V|$: the number of nodes in the graph; $|T|$: the size of divided time slots; $|S|$: the maximum number of stops scheduled for a vehicle; $|C|$: the number of vehicles.

The existing algorithm [159] computes k shortest paths for passengers and stops of vehicles and then searches for their overlaps. The state-of-the-art k shortest path algorithm takes $O(k|V|(|E| + |V|\log|V|))$ time [65], where $|E|$ denotes the number of edges. A vehicle is scheduled to visit at most $|S|$ stops, requiring $O(k|V||S|(|E| + |V|\log|V|))$ to compute the k shortest paths $(|S| - 1)$ times.

The algorithm then constructs TEG for the passenger and vehicles. A TEG node (l_i, t_i) is created if the passenger/vehicle can visit location l_i at time period t_i. A TEG node may connect with all other TEG nodes with the same location, resulting in $O(|T|)$ waiting edges. Meanwhile, it may connect with all TEG nodes representing the next scheduled location on the route, leading to $O(|T|)$ transfer edges. There are at most $|V|$ locations scheduled on a route. Thus, a route creates at most $|V||T|$ TEG nodes and $O(|V||T|^2)$ edges.

The TEG of the passenger contains $O(k|V||T|^2)$ edges to represent the k shortest paths. The TEG of a vehicle contains at most $(O(k|V||T|^2|S|))$ edges as (|S|-1) routes are

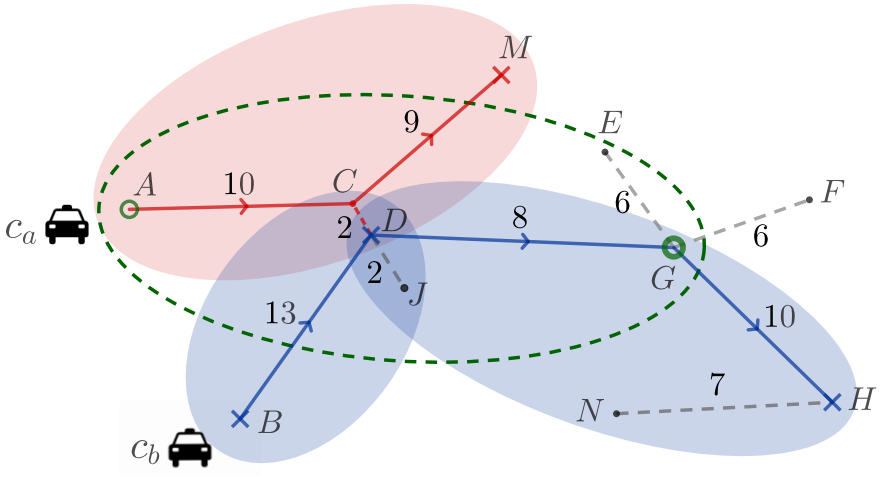


FIGURE 5.2: Detour ellipses of Figure 5.1.

required to connect the $|S|$ scheduled stops. For every TEG edge of the passenger, the algorithm searches for overlap TEG edge considering all $|C|$ vehicles, thus requiring $O(k^2|V|^2|T|^4|S||C|)$ time.

5.3 Station-first Algorithm

Next, we present our Station-first algorithm that identifies potential transfer points and then searches for possible vehicles by adopting the state-of-the-art single-hop algorithm GeoPrune proposed in Chapter 4 that is proposed in Chapter 4.

We next detail our multi-hop Station-first algorithm. The basic idea of the algorithm is to split a request trip by a transfer point and then apply the GeoPrune algorithm on the two generated itineraries.

Stage 1: identify possible transfer points. We only examine transfer points within the detour ellipse, i.e., ellipse($r_n.s$, $r_n.e$). Visiting any points outside of the request detour ellipse will violate the latest arrival time of the request [159] and thus points outside of the ellipse cannot be a transfer point. We build an R-tree on all possible transfer points on the road network (T_{tsf}) and run a range query using ellipse($r_n.s$, $r_n.e$) to retrieve the possible transfer points.

Stage 2: check each possible transfer point. We then check the feasibility of every possible transfer point. A transfer point ϕ splits the request trip into two itineraries

$(r_n.s, \phi)$ and $(\phi, r_n.e)$. We derive their time constraints so as to apply the single-hop algorithm.

For the source $r_n.s$, the earliest arrival time ($ear()$) is the request issue time, and the latest arrival time ($lat()$) is the latest pickup time, i.e., $ear(r_n.s) = r_n.t$, $lat(r_n.s) = r_n.\tau_s$. For the destination $r_n.e$, the earliest arrival time is the request issue time plus the shortest trip time of the request, i.e., $ear(r_n.e) = r_n.t + t(r_n.s, r_n.e)$. The latest arrival time is the latest drop-off time, i.e., $lat(r_n.e) = r_n.\tau_e$.

The time constraints of the transfer point ϕ depend on its location and are different in the two itineraries. The earliest arrival time of ϕ in the first itinerary (ϕ_1) is the issue time of the request $r_n.t$ plus the direct trip time from $r_n.s$ to ϕ, i.e., $ear(\phi_1) = r_n.t + t(r_n.s, \phi)$. The system must reserve a time longer than the direct trip from ϕ to $r_n.e$ to ensure the latest arrival time of the request. Thus, the latest arrival time at ϕ is $lat(\phi_1) = r_n.\tau_e$ - $t(\phi, r_n.e)$.

We apply GeoPrune to compute all feasible matches to serve the first itinerary before setting time constraints for the second itinerary. The selection stage needs to enumerate all insertion positions of source and destination for every vehicle candidate and find all feasible insertion positions. The first itinerary may find multiple matches with different estimated arrival times at the transfer point ($est(\phi_1)$) and thus define different second itineraries.

The transfer point is regarded as a source in the second itinerary (ϕ_2). Its earliest arrival time is its estimated arrival time in the first itinerary, i.e., $arr(\phi_2) = est(\phi_1)$. Its latest arrival time is the same as that of the first itinerary, i.e., $lat(\phi_2) = lat(\phi_1) = r_n.\tau_e$ - $t(\phi, r_n.e)$.

We again apply GeoPrune on each generated second itinerary to compute all feasible matches and insertion positions. The combination of a feasible match of the first itinerary and that of the corresponding second itinerary forms a feasible multi-hop match. Among all feasible multi-hop matches and direct matches, we select the optimal one and return it to the request.

Algorithm 4 summarizes the Station-first algorithm. In Stage 1 (line 1 to line 3), we first initialize the optimization target value as zero (line 1) and then using the request's detour ellipse to obtain all transfer points within the detour ellipse (line 2, line 3). The

matching feasibility and optimization value of each remaining transfer point are checked in Stage 2 (line 4 to line 15). For a transfer point ϕ, we generate the first itinerary ($r_n.s$, ϕ) (line 5) and derive the earliest arrival time and latest arrival time constraint of $r_n.s$ and ϕ (line 6). We then apply GeoPrune to find all feasible matches of the first itinerary (line 7). Every feasible match of the first itinerary generates a second itinerary (line 9) with different time constraints derived (line 10). We then apply GeoPrune to find feasible matches of each generated second itinerary (line 10). If GeoPrune finds a feasible match for the second itinerary (line 12), a feasible multi-hop match is formed as the combination of the match to the first itinerary and that to the second itinerary. We check the additional distance of the new multi-hop and update the optimal multi-hop trip if a smaller additional distance is obtained (line 13 to line 15).

Algorithm 4: Station first algorithm (SF)

Input: A new trip request $r_n = \langle t, s, e, \tau_s, \tau_e, \eta \rangle$
Output: An optimal match of r_n: $r_n.m$

```
1  best_add_dist ← 0
   /* Stage 1:  identify possible transfer points                    */
2  ellipse(r_n.s, r_n.e) ← detour ellipse of r_n
3  Φ ← T_tsf.rangeQuery(ellipse(r_n.s, r_n.e)
   /* Stage 2:  Check each possible transfer point                   */
4  for φ in Φ do
5      itin_1 ← (r_n.s, φ)
6      derive the time constraint of itin_1
7      M_1 ← all feasible matches of itin_1 (GeoPrune(itin_1))
8      for m_1 ∈ M_1 do
9          itin_2 ← (φ, r_n.e)
10         derive the time constraint of itin_2
11         M_2 ← all feasible matches of itin_2 (GeoPrune(itin_2))
12         for m_2 ∈ M_2 do
13             if add_dist(m_1) + add_dist(m_2) > best_add_dist then
14                 best_add_dist ← add_dist(m_1) + add_dist(m_2)
15                 r_n.m ← multi-hop(m_1,m_2)
16 return r_n.m
```

Algorithm complexity. Assume there are $|P|$ transfer points, $|C|$ vehicles and the maximum number of stops of the vehicle schedule is $|S|$. For each transfer point, GeoPrune is applied once on the first itinerary. The maximum number of feasible matches of the first itinerary is $|C||S|^2$ as there are $|C|$ vehicles and each vehicle has $|S|$ positions to insert the source and $|S|$ positions to insert the destination. Each feasible match of the first itinerary defines a second itinerary and each generated second

itinerary conducts GeoPrune once (at most $|C||S|^2$). Every second itinerary may also has at most $|C||S|^2$ insertion positions. Thus, the GeoPrune will be conducted at most $(|C||S|^2 + 1)$ times for each transfer point and at most $|C|^2|S|^4$ insertion positions will be checked with $O(1)$ checking time. Combining with the GeoPrune complexity $O(\sqrt{|C||S|} + |C||S| \log(|C||S|))$, the overall complexity of the Station-first algorithm is $O(|P|(|C|^2|S|^4 + |C|^2|S|^3 \log(|C||S|)))$.

Example 5.2. *The algorithm first identifies points within the request ellipse as possible transfer points, i.e., A, C, D, E, G, J. A transfer point splits the trip into two itineraries and the algorithm conducts GeoPrune on these two itineraries. For example, D splits the trip into two itineraries:* $A - D$ *and* $D - G$*. A and G represent the source and the destination and their time constraints refer to the service constraints of the request, i.e.,* $A.ear = r_n.t$*,* $A.lat = r_n.\tau_s$*,* $G.ear = r_n.t + t(r_n.s + r_n.e)$*,* $G.lat = r_n.\tau_e$*. In the first itinerary* $A - D$*, the earliest arrival time of D is* $D.ear = A.ear + t(A, D)$ *(when the vehicle directly travels from A to D). The arrival at D must before* $D.lat = G.lat - t(D, G)$ *to ensure the latest arrival time of the request destination G G.lat, considering that travelling from D to G takes at least* $t(D, G)$*. The algorithm then applies GeoPrune to find feasible matches of the first itinerary. Assume* c_a *is the only vehicle candidate and that A and D can only be inserted after the first stop of* c_a *(new schedule* $c_a = A - (\boldsymbol{A}) - \boldsymbol{D} - M$*), then the possible insertion position of the source A is* $\Gamma(s) = c_a.1$ *and that of the transfer point D is* $\Gamma(\phi_1) = c_a.1$*. Each feasible match of the first itinerary defines a second itinerary with different time constraints. In the second itinerary* $D - G$*, the earliest arrival time of D is the estimated arrival time at D in the first itinerary. Its latest arrival time is the same as in the first itinerary, i..e,* $D.lat = G.lat - t(D, G)$*. The algorithm then applies GeoPrune on the second itinerary. Assume* c_b *can carry the passenger from D to G with the new trip schedule* $B - D - (\boldsymbol{D}) - \boldsymbol{G} - H$*. The new stop D can be inserted before the existing stop D or after it, corresponding to two insertion positions: after the first stop or after the second stop, i,e,*$\Gamma(\phi_2) = c_b.1$ *or* $\Gamma(\phi_2) = c_b.2$*. The insertion position of the destination G is after the second stop, i.e.,* $\Gamma.e = c_b.2$*. The algorithm then returns the combination of the feasible matches of both itineraries as the multi-hop trip:* $\langle c_a, c_b, \phi, \Gamma \rangle$ *to respond to the passenger, where* Γ *can be:* $\Gamma = (c_a.1, c_a.1, c_b.1, c_b.2)$*,* $\Gamma = (c_a.1, c_a.1, c_b.2, c_b.2)$*.*

5.4 Vehicle-first Algorithm

The Station-first algorithm examines all possible transfer points, which may become expensive when there are many such points. Next, we propose an algorithm called Vehicle-first that examines fewer transfer points while preserving exact solutions.

The intuition is that two vehicles can be paired up in a trip only if their reachable areas (ellipses) overlap and the transfer points must be within the detour ellipses of both vehicles and the request. By using efficient data structures, we can quickly locate overlapping ellipses and identify possible vehicle pairs and insertion positions.

A vehicle pair and the specified insertion positions form a multi-hop trip candidate. Our further analysis shows that the optimal transfer point of each trip candidate only depends on three or four stops of vehicles, which enables us to reduce the problem into a variation of group nearest neighbor query (GNN) in road networks. Repeatedly running GNN queries on each trip candidate lacks efficiency when there are many possible vehicle pairs. We hence propose a novel algorithm to accelerate this process. The algorithm can be easily extended to other goals by customizing the algorithm of finding the optimal transfer points.

5.4.1 Stage 1: Find Possible Insertion Positions

We first explain how to identify possible vehicle pairs and the insertion positions of source, destination and the transfer point. We first compute *circle($r_n.s$)* and *ellipse($r_n.s$, $r_n.e$)* for a given request r_n and then run a two-phase refinement process to find the possible trip candidates. The first phase locates the possible insertions of the source $r_n.s$ and destination $r_n.e$ to a vehicle schedule. The second phase detects the overlap areas between vehicles and identify possible insertions of the transfer point.

Phase 1: identify possible insertions of source and destination. There are two types of insertion positions to add a new stop to the schedule of a vehicle: *insert-between* and *insert-after*. An insert-between position indicates an insertion between two existing stops of the vehicle schedule while an insert-after position indicates appending the new stop after the last stop.

The same as GeoPrune (proposed in Chapter 4), we construct two R-trees to index the detour ellipses and the last stops of vehicles respectively: T_{seg} and T_{end}. We run four queries to identify possible insertions of the source and destination (and thus the possible vehicles).

1. insert-between for source $r_n.s$: querying segments with the detour ellipse covering $r_n.s$, $T_{seg}.pointQuery(r_n.s)$.
2. insert-after for source $r_n.s$: querying vehicles with the last stop covered by circle($r_n.s$), $T_{end}.rangeQuery(circle(r_n.s))$.
3. insert-between for destination $r_n.e$: querying segments with the detour ellipse covering $r_n.e$, $T_{seg}.pointQuery(r_n.e)$.
4. insert-after for destination $r_n.e$: querying vehicles with the last stop covered by circle($r_n.e$), $T_{end}.rangeQuery(circle(r_n.e))$.

We discard an insertion position if it violates the time constraints of any stop. Assume that the previous stop in the new schedule is p^k and the new stop is p, if the estimated arrival time of p (computed by summing up the estimated arrival time of p^k and the travel time from p^k to p) is later than its allowed latest arrival time, we discard the insertion position. Besides, if the insertion position refers to a segment (p^k, p^{k+1}), we discard the insertion position if the caused detour time is larger than the slack time of the segment. The range query $T_{end}.range(circle(r_n.e))$ may return many vehicles due to the possibly large querying circle. However, most of the vehicles may have late arrival times at their last stops and the last stops may be far from the destination. Checking the time constraints helps to safely discard these infeasible vehicles.

Phase 2: identify overlap reachable areas. After running the four queries, we obtain vehicles that can be scheduled to visit the source or the destination of the request. In Phase 2, we pair up such vehicles by detecting the overlap reachable areas of these vehicles based on the following constraints: 1) The transfer point must be within the detour ellipse of the request. 2) In the schedule of the first vehicle, the insertion position of the transfer point must be **no earlier than** that of the source. 3) In the schedule of the second vehicle, the insertion position of the transfer point must be **no later than** that of the destination.

For each pair of source insertion and destination insertion belonging to two different vehicles, we check if there is a reachable area *after* the source insertion overlaps with

a reachable area *before* the destination insertion. If we detect an overlap within the request ellipse, we create a multi-hop trip candidate with the two vehicles and insertion positions specified. The transfer window must be within the overlapped reachable areas and the request ellipse.

Algorithm 5 summarizes the procedure of the Vehicle-first algorithm. We first conduct Stage 1 to find all possible multi-hop trips with specified insertion positions (line 1), which is described in Algorithm 6. We then compute its GNN related information for each multi-hop trip candidate (line 2 to line 5). The *prev_dist* sums the length of the existing trip schedules of the two matching vehicles (line 3). The *gnn_stop* records the GNN stops of the match (line 4). The *non_gnn_new_dist* records the total travel distance connecting the non-GNN stops in both vehicles (line 5). The additional distance of the a match m thus equals to $add\_dist(m) = m.gnn\_dist + m.non\_gnn\_new\_dist - m.prev\_dist$, where only $m.gnn\_dist$ is variable and depends on the optimal transfer point to be computed in the Stage 2 (line 6 to line 9). We apply the collaborative-IER (Algorithm 7) to calculate the optimal transfer point of each multi-hop trip for exact solutions, and approximate the transfer point (Algorithm 8) when real-time response time is critical.

Algorithm 5: Vehicle-first algorithm (VF)

Input: A new trip request $r_n = \langle t, s, e, \tau_s, \tau_e, \eta \rangle$
Output: An optimal match of r_n, $r_n.m$

`/* Stage 1: Find possible insertion positions */`
1 $M \leftarrow FindInsPositions(r_n)$
2 **for** $m \in M$ **do**
3 $\quad m.pre\_dist \leftarrow len(m.c_1.S) + len(m.c_2.S)$
4 $\quad m.gnn\_stop \leftarrow$ the GNN stops of m
5 $\quad m.non\_gnn\_new\_dist \leftarrow$ overall travel distance connecting non-GNN stops

`/* Stage 2: Find the optimal multi-hop trip */`
6 **if** *findExactSolution* **then**
7 $\quad r_n.m \leftarrow$ CollaborativeIER(M)
8 **else**
9 $\quad r_n.m \leftarrow$ ApxGNN(M)
10 **return** $r_n.m$

Algorithm 6 summarizes the steps to compute the possible insertion positions and multi-hop trips. We first create M to record all possible multi-hop trips (line 1). Then we compute two circles centering at $r_n.s$ and $r_n.e$ respectively, and one detour ellipse with $r_n.s$ and $r_n.e$ as focal points (line 2 to line 4). The possible insertion positions of $r_n.s$ and

Algorithm 6: FindInsPositions

Input: A new trip request $r_n = \langle t, s, e, \tau_s, \tau_e, \eta \rangle$

Output: All possible matches of r_n, M

```
1  M ← ∅
2  circle(r_n.s) ← waiting circle of r_n.s
3  circle(r_n.e) ← waiting circle of r_n.e
4  ellipse(r_n.s, r_n.e) ← detour ellipse of r_n
5  I(s) ← T_seg.pointQuery(r_n.s) ∪ T_end.rangeQuery(circle(r_n.s))
6  I(e) ← T_seg.pointQuery(r_n.e) ∪ T_end.rangeQuery(circle(r_n.e))
7  for i_s ∈ I(s) do
8      for i_e ∈ I(e) do
9          if A reachable area before i_s (i_φ1) overlaps with a reachable area after i_e
             (i_φ2) within ellipse(r_n.s, r_n.e) then
10             m.c_1 ← vehicle id of i_s
11             m.c_2 ← vehicle id of i_e
12             m.Γ ← (i_s, i_φ1, i_φ2, i_e)
13             m.transfer_window ← reach(i_φ1) ∩ reach(i_φ2) ∩ ellipse(r_n.s, r_n.e)
14             M.insert(m)
15 return M
```

$r_n.e$ are then obtained by conducting point queries and range queries and recorded in I_s and I_e, respectively (line 5 to line 6). For every pair of source insertion and destination insertion (i_s, i_e), if we detect a reachable area before i_s overlaping with one after i_e within the detour ellipse $ellipse(r_n.s, r_n.e)$ (line 9), we create a multi-hop trip candidate (line 7 to line 14).

Example 5.3. *In Figure 5.2, we first search for vehicles that can reach source A and destination G, respectively. For source A, only the first ellipse of c_a (the red ellipse) covers A. Assuming the waiting circle of $r_n.s$ covers the last stop of neither vehicles, A can only be inserted after the first stop of c_a's schedule. As for destination G, the second ellipse of c_b (the blue ellipse on the right) covers G. Assuming the waiting circle of G covers both last stops of c_a and c_b, G has three possible insertion positions: after the second stop of c_b, after the last (third) stop of c_b, and after the last (second) stop of c_a. Next, we check the time constraints of these insertion positions. We assume that inserting G to the last stop of c_b violates the latest arrival time of G and discard this insertion possibility. We then pair up the possible insertion positions of A and G. The insertion pair $(\Gamma(s), \Gamma(e)) = (c_a.1, c_a.2)$ is infeasible because the source and destination must be served by two different vehicles. Thus, only one insertion pair is remained: $(\Gamma(s), \Gamma(e)) = (c_a.1, c_b.2)$.*

We then check the transfer possibility for the remaining insertion pair $(\Gamma(s), \Gamma(e)) = (c_a.1, c_b.2)$. *We check if there is any reachable area after the first stop of* c_a *overlap with that before the second stop of* c_b. *Since the first ellipse* c_a *overlaps with the first and second ellipses of* c_b, *the possible insertions of the transfer point are: 1) after the first stop of* c_a *and after the first stop of* c_b, *2) after the first stop of* c_a *and after the second stop of* c_b, *yielding two multi-hop trip candidates: 1)* $\langle c_a, c_b, \phi, \Gamma = (c_a.1, c_a.1, c_b.1, c_b.2)\rangle$; *2)* $\langle c_a, c_b, \phi, \Gamma = (c_a.1, c_a.1, c_b.2, c_b.2)\rangle$. *Their optimal transfer points are not decided yet and bounded by the overlap areas.*

5.4.2 Stage 2: Find the Optimal Transfer Point

The remaining challenge is how to quickly return an optimal match with optimal transfer point. Given a multi-hop trip candidate, a naive solution of finding the optimal transfer point is to check all transfer points within the *transfer window*. Such a solution is inefficient when there are many points in the transfer window. Next, we reduce the problem into a variation of the group nearest neighbor query to enable a more efficient solution.

Problem reduction.

Given a multi-hop candidate, the *additional distance* equals to the travel distance of the new schedules of the two vehicles minus the travel distance of their existing schedules. As the travel distance of existing schedules is a constant, the problem is reduced to minimizing the distance of the new schedules.

If we append both source and transfer point to the first vehicle's trip schedule, the transfer point will be the last stop and only the source stop will be connected to the transfer stop in the new schedule. If we insert both source and the transfer point in the middle of the first vehicle's schedule, there will be two stops connecting the transfer point, one at the front and one afterward. On the other hand, in the schedule of the second vehicle, the destination stop must be placed after the transfer stop. Hence, the transfer stop cannot be the last stop and there must be two stops connecting the transfer point in the second vehicle's schedule.

We denote the stops connecting the transfer point in the new schedule as **GNN stops** and the sum of distances from a point to the GNN stops as the **GNN distance**. The

additional distance of a multi-hop match can be computed as the sum of two parts: a constant part that equals to the travel distance connecting non-GNN stops in the new schedules *minus* the travel distance of the existing trip schedules, and a variable part representing the GNN distance. Finding the optimal transfer point is thus reduced to finding a transfer point with minimum sum of distances to three or four fixed (GNN) stops (a group nearest neighbor query (GNN) in a road network) while satisfying the time constrains.

Example 5.4. *Consider a multi-hop option* $\langle c_a, c_b, \phi, \Gamma = (c_a.1, c_a.1, c_b.2, c_b.2)\rangle$. *The schedule of* c_a *changes from* $A-M$ *to* $A-A-\phi-M$. *The travel distance of the existing schedule* $A-M$ *is a constant* $d(A,M)$. *The travel distance of the new schedule equals to* $d(A,\phi)+d(\phi,M)$ *(*$d(A,A)=0$*), which is changing with the choice of* ϕ. *Similarly, the schedule of* c_b *changes from* $B-D-H$ *to* $B-D-\phi-G-H$. *The travel distance of the existing schedule is a constant* $d(B,D)+d(D,H)$. *The travel distance of the new schedule is* $d(B,D)+d(D,\phi)+d(\phi,G), d(G,H)$. $d(B,D)$ *and* $d(G,H)$ *are fixed while* $d(D,\phi), d(\phi,G)$ *may vary based on* ϕ. *Therefore, the problem of minimizing the additional distance is reduced to minimizing* $d(A,\phi)+d(\phi,M)+d(D,\phi)+d(\phi,G)$, *which is further reduced to finding a feasible transfer point to minimize the sum distance to* A, M, D, G.

Collaborative IER

Applying an existing GNN algorithm to each multi-hop trip lacks efficiency when many multi-hop candidates remains. Observing that many multi-hop trips may share overlap transfer windows that will be searched multiple times and cause redundant computation, we propose to collaboratively process all multi-hop candidates while only explore the space once.

IER. We first describe an existing GNN algorithm *Incremental Euclidean Restriction* (IER) [93]. IER traverses the R-tree indexing all transfer points from top to bottom. Given an R-tree node *Rnode*, the GNN distance of any point indexed under *Rnode* must be larger than the sum Euclidean distance from the GNN query points (GNN stops in our problem) to the minimum bounding box of *Rnode*. To exploit this property, the algorithm maintains a priority queue to sort R-tree nodes by their sum Euclidean distance to the GNN query points. Initializing the queue as the root of T_{tsf}, IER iteratively extracts minimum elements from the queue and inserts its children nodes to

the queue. Hence, points with smaller Euclidean GNN distances will be visited first. If the extracted element refers to a transfer point, we check its feasibility (Section 5.2.1) and update the optimal GNN distance *best_dist*. If the key of the next element in the queue is larger than *best_dist*, IER terminates as the sum network distance of any unchecked nodes must be larger than the current best result.

Collaborative IER. The basic idea is that when reaching an R-tree node, instead of computing the GNN lower bound of a single multi-hop candidate, we consider the lower bounds of **all** multi-hop candidates. The algorithm is guided to first visit the transfer points with smaller lower bounds among all multi-hop candidates, which enables earlier termination and reduced search space.

Every R-tree node maintains three additional variables during the traversal: active multi-hop candidates (*active_cand*), lower bound of these active candidates (*active_LB*), and the minimum lower bound of these active candidates(*min_LB*). *active_cand* stores all multi-hop candidates with their transfer window intersecting the MBR of the R-tree node, while all other multi-hop trips are infeasible to transfer within the indexed area. For each candidate in *active_cand*, the lower bound of minimum additional distance considering any transfer points within the indexed area is recorded in *active_LB*. Note that the recorded lower bound of a multi-hop candidate is the lower bound of the minimum additional distance that considers both the GNN distance and the constant part. The minimum *active_LB* of all active multi-hop candidates is indicated by *min_LB*.

The same as IER, we maintain a priority queue that sorts R-tree nodes by their minimum lower bound *min_LB*. We initialize the queue with the R-tree root and iteratively dequeue top elements. After each dequeue, we process the children nodes of the dequeued node. We remove those multi-hop candidates that are no longer active in the children nodes if their transfer windows become non-overlapping with the indexed. We also update the lower bounds of all active candidates for each child node. We insert a child node to the queue if its active multi-hop is not empty and set the key as the updated minimum lower bound. If the extracted node is a leaf node, we check the feasibility of the corresponding transfer point considering all its active multi-hop candidates. We update the optimal additional distance *best_add_dist* if a smaller distance is achieved. When the key of the top element in the queue exceeds the recorded optimal distance

best_add_dist, we terminate the search and return the optimal multi-hop match and optimal transfer point.

Algorithm 7 summarizes the algorithm to compute the optimal transfer point of every multi-hop trip with one R-tree traversal. First, we initialize a priority queue H (line 2). For the R-tree root indexing all transfer points, we mark all multi-hop trips returned from Algorithm6 as active with infinite additional distance lower bounds (line 3 to line 6). We enqueue the R-tree root with the key labeled as infinity (line 7). We then gradually extract R-tree nodes from the heap and enqueue its child nodes until the key of the next entry from H is larger than the found optimal additional distance (line 8 to line 26). If the retrieved R-tree node refers to a transfer point, we check the feasibility of every active multi-hop trip recorded by the node and update the optimal road network additional distance with the transfer point assigned (line 12 to line 17). Otherwise, we enqueue its child nodes and compute the active candidates and lower bounds of each child node (line 18 to line 26).

Algorithm complexity. Querying the possible insertion positions of source and destination takes $O(\sqrt{|S||C|} + |S||C|)$ time. The source has $|C||S|$ possible insertion positions, each of which has at most $|S|$ positions to insert the corresponding transfer position. Therefore, the insertion position is $|C||S|^2$ for the first vehicle and $|C||S|^2$ for the second vehicle. The overall number of routes is thus at most $|C|^2|S|^4$. In the worst case, the algorithm visits all nodes during the R-tree traversal ($O(|V| + \log|V|)$) and each node processes all routes ($O(|V||C|^2|S|^4)$). The overall complexity is therefore $O(\log|V| + |V||C|^2|S|^4)$.

5.4.3 Performance Enhancement through Deep Learning and Approximation

The above algorithms guarantee the finding of the optimal multi-hop trip. We propose two approximation strategies to accelerate the two stages of the Vehicle-first algorithm while achieving near-optimal matches. The first strategy employs deep learning to shrink the representation of reachable areas such that fewer vehicles will overlap and be paired up. The second strategy approximates the optimal transfer point such that fewer points need to be checked.

Algorithm 7: Collaborative-IER

Input: Set of all possible matches of r_n, M
Output: The optimal match of r_n, $r_n.m$

```
1  best_add_dist ← 0
2  initialize a priority queue H
3  for m ∈ M do
4      Root(T_tsf).active_cand.insert(m)
5      Root(T_tsf).active_LB[m] ← ∞
6  enqueue(H, Root(T_tsf), ∞)
7  while H is not empty do
8      e ← dequeue(H)
9      if e.min_LB > best_add_dist then
10         stop
11     if e is a transfer point then
12         for m ∈ e.active_cand do
13             add_dist ← sumNetworkDist(e, m.gnnStops) + m.non_gnn_new_dist -
                m.prev_dist
14             if add_dist < best_add_dist then
15                 m.φ ← e
16                 r_n.m ← m
17     else
18         for each child node e' of e do
19             for m ∈ e'.active_cand do
20                 if m.transfer_window ∩ e'.MBR ≠ ∅ then
21                     e'.active_cand.insert(m)
22                     e'.active_LB[m] ← sumEuclideanDist(e', m.gnnStops) +
                        m.non_gnn_new_dist - m.prev_dist
23                     if e'.active_LB[m] < e'.min_LB then
24                         e'.min_LB ← e'.active_LB[m]
25                     enqueue(H, e', e'.min_LB)
26 return r_n.m
```

5.4.3.1 Learning the Reachable Area

The exact algorithms use ellipses to bound the reachable areas and guarantee the pruning correctness. However, it may bring extra computation as the actual reachable areas considering the road network distance are usually smaller than the ellipses. If we use actual reachable areas instead of the bounding ellipses in the Vehicle-first algorithm, fewer multi-hop candidates may survive the pruning thus the query efficiency can be improved. Computing the actual reachable areas online is costly due to the expensive graph traversal. We predict the reachable areas using deep learning considering the high

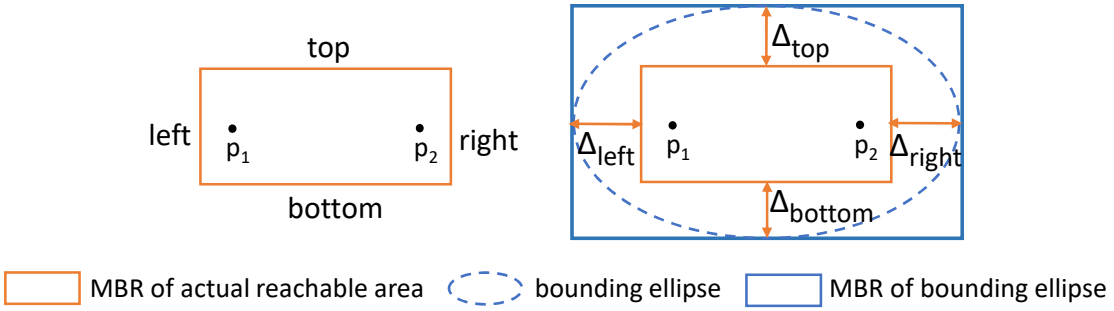


FIGURE 5.3: Reachable area prediction (left: Boundary Prediction; right: Gap & GapCustom Prediction).

accuracy and low prediction cost [48, 51].

The model is a multi-layer feed-forward network with 12 hidden layers. The architecture follows encoder-decoder architecture style where number of neurons in each layer is (64, 64, 128, 128, 256, 256, 256, 256, 128, 128, 64, 64). We use ReLU as the activation function for the hidden layers and linear for the output layer.

Given a road network, a reachable area is determined by three factors: source, destination, and time budget. Each location has two coordinates (longitude and latitude) and thus an input to our neural network is comprised of five elements: *source_lon*, *source_lat*, *dest_lon*, *dest_lat*, *time_budget*. For fast retrieval and updates, we still represent reachable areas as rectangles. The prediction goal is thus to obtain the four boundaries of a rectangle, i.e., *left*, *bottom*, *right*, *top*. We consider the following three prediction strategies:

Boundary prediction. The strategy predicts the four boundaries of the actual reachable rectangle directly, as shown in the left of Figure 5.3. The training uses the Mean Square Error (MSE) as the loss function to minimize the differences between the predicted boundaries and the actual boundaries.

Gap prediction. This strategy uses the bounding information provided by the MBRs of ellipses, which is inspired by the key observation that the actual reachable area is always a sub-area of the bounding ellipse (rectangle). Instead of predicting the boundaries directly, for each boundary, we predict the gap between the MBR of the bounding ellipses and the MBR of the actual reachable area: Δ_{left}, Δ_{bottom}, Δ_{right} and Δ_{top} (right of Figure 5.3). We use the MSE as the loss function to minimize the difference between the predicted gaps and the actual gaps when training. Let the boundaries

of the bounding ellipse be $bound\_left$, $bound\_right$, $bound\_top$, and $bound\_bottom$, the predicted rectangle can be inferred as:

1. $pred\_left = bound\_left + \Delta_{left}$;
2. $pred\_right = bound\_right - \Delta_{right}$;
3. $pred\_bottom = bound\_bottom + \Delta_{bottom}$;
4. $pred\_top = bound\_top - \Delta_{top}$.

GapCustom prediction. If the predicted gap is larger than the actual gap, the predicted area will be smaller than the actual reachable area. Thus, some feasible vehicle pairs (with overlapped actual reachable areas) may be missed if the predicted areas no longer overlap. To avoid such false negative pairs, we propose the GapCustom prediction that applies a customized loss function to the Gap prediction and penalizes predictions larger than the actual gaps:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} \varphi(y\_true - y\_pred))^2,$$

where φ is the penalty factor: $\varphi = 1$ if $y\_pred \leq y\_true$ whereas $\varphi > 1$ if $y\_pred > y\_true$. In our experiments, we choose the penalty factor $\phi = 10$ for larger prediction.

The predicted boundaries can not produce a closed rectangle if $pred\_left > pred\_right$ or $pred\_bottom > pred\_top$. For such cases, we still represent the reachable area using the bounding ellipse. Besides, as the applied gap value must be non-negative, we treat a predicted negative gap as zero in the Gap prediction and GapCustom prediction, indicating no shrink on the boundary.

5.4.3.2 Quickly Locating the Optimal Transfer Point

The Vehicle-first algorithm checks the feasibility of multiple transfer points before returning an optimal feasible one. In an extreme case, no transfer point is feasible and the algorithm needs to check all transfer points within the transfer window. To accelerate the process, we propose an approximation strategy to check only a few transfer points for each vehicle pair. The basic idea is to estimate the optimal transfer point in the road network with the optimal Euclidean GNN transfer points. Specifically, for each candidate trip, we only check k transfer points ($k = 5$ in our experiments) with the top-k

minimum sum Euclidean distance to the GNN stops, which can be quickly obtained by using data structures such as R-trees [93, 96].

Algorithm 8: ApxGNN

Input: Set of all possible matches of r_n, M
Output: An optimal match of r_n, $r_n.m$

```
1  for m ∈ M do
2      i ← 0
3      while i < k do
4          m.φ ← nextEuclideanGNN(m.gnnStops)
5          if m is feasible then
6              add_dist(m) ← sumNetworkDist(m.φ, m.gnnStop) +
                m.non_gnn_new_dist - m.prev_dist
7              if add_dist(m) < best_add_dist then
8                  best_add_dist ← add_dist(m)
9                  r_n.m ← m
10         i ← i + 1
11 return r_n.m
```

Algorithm 8 describes the method of estimating the optimal transfer point. For each multi-hop candidate m, we iteratively compute top-k transfer points with the minimum sum of Euclidean distances to all query stops (line 4). We fix the transfer point of m at the retrieved point. If m is feasible, we compute the additional distance *add_dist* (line 5 to line 6) and update the optimal match $r_n.m$ if m obtains a smaller *add_dist* (line 7 to line 9). We terminate the examination if k transfer points are already checked (line 3).

5.5 Experiments

In this section, we study the empirical performance of the proposed algorithms. We run all experiments on Linux OS with 2.7 GHz CPU and 32 GB memory. We train the deep learning models using tensorflow in Python and implement all other algorithms in C++. We first investigate the benefits of allowing transfers in ride-sharing under different parameter settings and then compare the efficiency and effectiveness of the proposed algorithms.

5.5.1 Experimental Setup

Dataset. The datasets of the road network and the trip requests are explained in Chapter 3.

We iteratively sample the transfer points by applying the k-medoids clustering on the network nodes, i.e., the smaller sets of transfer points are generated by clustering large sets of transfer points.

We generate the training and testing datasets by sampling the source, destination, and time budget and normalize them before training the model. We generate 50,000 samples and randomly select 40,000 of them as the training dataset and the rest as the test dataset. We train our model using Tensorflow with the maximum number of training epochs specified as 5000. The learning rate and the batch size are selected as 0.001 and 100, respectively. We split 20% data from the training dataset as the evaluation dataset, which is used to evaluate the model at the end of each training epoch. We monitor the loss value on the validation dataset and terminate the training if the validation loss stays unimproved for more than 50 epochs.

Matching strategies. We consider both single-hop and multi-hop settings. *No-multiHop:* a ride-sharing system where no transfer is allowed during the matching. *MultiHop:* a ride-sharing system where both single-hop and two-hop trips may be returned.

Comparing algorithms. We compare the following algorithms:

SF. the Station-first algorithm described in Section 5.3.

VF. the Vehicle-first algorithm described in Section 5.4.

SF-pred. SF plus reachable area prediction (Section 5.4.3.1).

VF-pred. VF plus reachable area prediction (Section 5.4.3.1).

VF-apxgnn. VF plus GNN approximation (Section 5.4.3.2).

VF-apxgnn-pred. VF plus both reachable area prediction and GNN approximation.

Metrics. We measure the following metrics: *# unmatched requests* – number of requests that cannot find feasible matches; *average trip distance* – average trip distance per request; *matching time* – average time required to respond a request.

Default setting. By default, we run experiments on a scenario when the vehicles are insufficient to serve all requests directly. Specifically, we simulate on 4,096 vehicles with 1,000 transfer points and set the detour ratio and maximum waiting time of trip requests as 0.4 and $4min$, respectively.

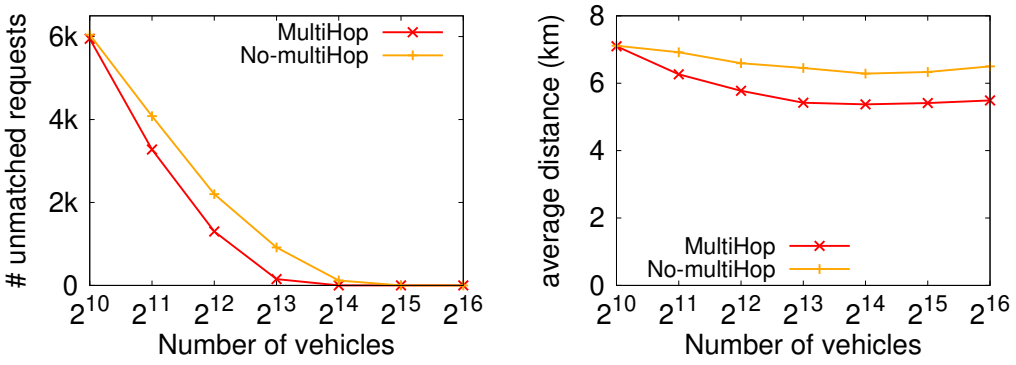
The state-of-the-art multi-hop matching algorithm [159] requires more than one week to match requests under the default setting. To compare the performance, we extract a small area from the central network of Chengdu that contains 910 nodes and 1610 edges. We randomly generate 200 vehicles and 100 requests while considering all nodes as possible transfer points. Due to the hardness of enumerating all possible paths, [159] only computes k' shortest paths between two locations to construct the TEGs. Our experiments show that when $k' = 50$, [159] costs more than 42s while our algorithms can respond in 0.04s. Besides, its travel distance is almost twice as long as ours because of the k shortest path simplification.

5.5.2 Benefits of Multi-hop Trips

We first study the benefits of multi-hop trips with the effect of number of vehicles, detour ratio, and number of transfer points.

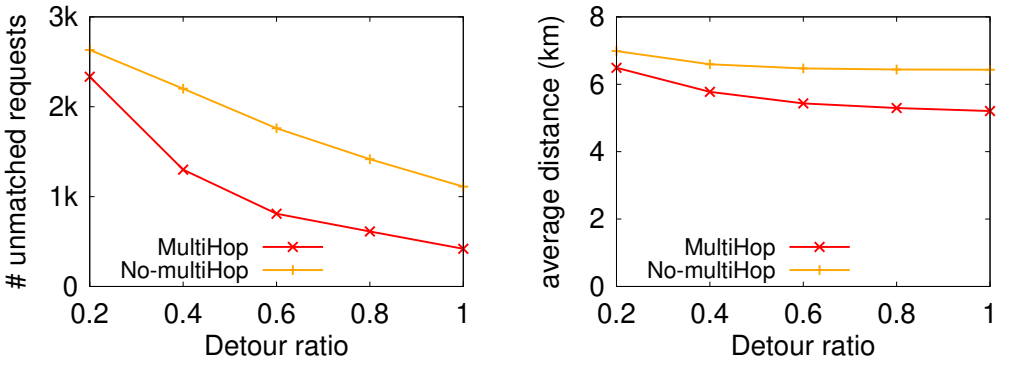
Effect of the number of vehicles. Figure 5.4 shows the effect of allowing multi-hop trips as the number of vehicles varies. Overall, enabling multi-hop creates more trip matches and reduces the average trip distance, which confirms that multi-hop is an effective strategy to improve system performance. Interestingly, the matching quality diminishes for multi-hop with only a few vehicles (less than 2^{10}). A possible reason is that a small number of vehicles can be easily fully occupied even by direct trips, and the multi-hop trips lead to extra visits to transfer points.

Effect of the detour ratio. Figure 5.5 illustrates the effect of multi-hop with different detour willingness of passengers. The matching quality for multi-hop is observed to outperform that without multi-hop in all cases. The advantage of multi-hop becomes more noticeable with higher detour ratios since passengers are more likely to share routes



(A) # unmatched requests. (B) Average trip distance.

FIGURE 5.4: Effect of the number of vehicles.



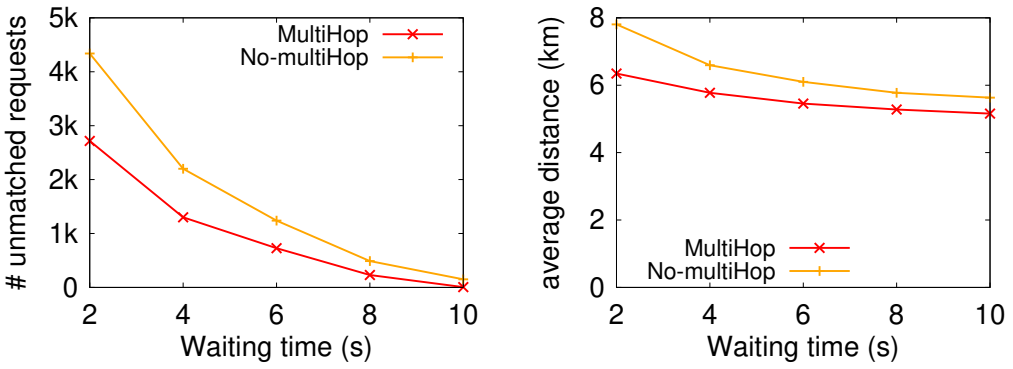
(A) # unmatched requests. (B) Average trip distance.

FIGURE 5.5: Effect of the detour ratio.

if they relax their time constraints. A reasonably high detour ratio may be observed in real-world applications for long-distance travelers who seek to save the travel cost by accepting longer arrival times.

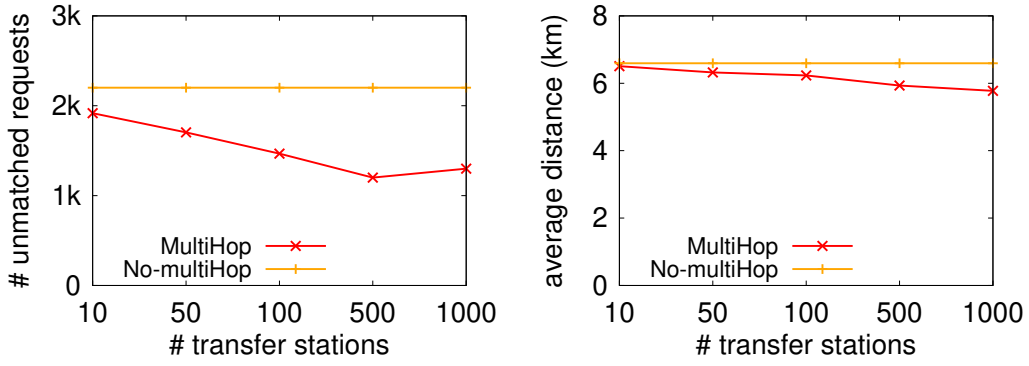
Effect of the number of transfer points. Figure 5.7 shows the system performance when varying the number of transfer points. Overall, providing more transfer points matches more requests and reduces the trip distance, despite slight fluctuations in the number of unmatched requests impacted by the greedy dispatching strategy.

Figure 5.6 illustrates the benefits of enabling multi-hop trips when varying the waiting time of requests. Multi-hop ride-sharing matches more requests with less travel distance in all cases compared to the no multi-hop system. The benefits are more noticeable when the waiting time is short in both the matching ratio and travel distance. Passengers may find it hard to be served by direct trips if they only accept short waiting times.



(A) # unmatched requests. (B) Average trip distance.

FIGURE 5.6: Effect of the waiting time.



(A) # unmatched requests. (B) Average trip distance.

FIGURE 5.7: Effect of the number of transfer points.

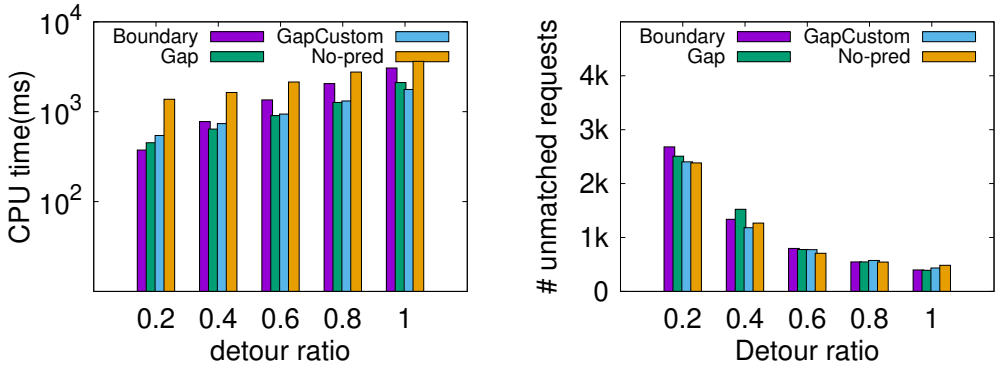
Prediction	Boundary	Gap	GapCustom
IoT	94.62%	93.25%	97.68%

TABLE 5.1: Prediction quality.

Nevertheless, they can largely increase their matching possibilities by looking for multi-hop trips.

5.5.3 Prediction Quality

We next investigate the prediction quality of the models proposed in Section 5.4.3.1. We use the metric *Intersection over True (IoT)* that is calculated by dividing the



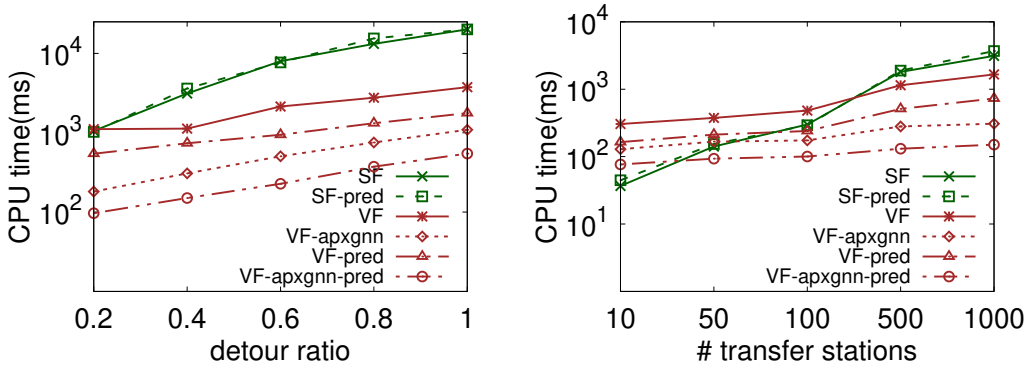
(A) Matching time (ms). (B) # unmatched requests.

FIGURE 5.8: Matching quality of prediction strategies.

intersection between the actual area and the predicted area by the actual area. IoT indicates the fraction of the actual reachable area that is correctly predicted. A higher IoT implies more correctly predicted areas.

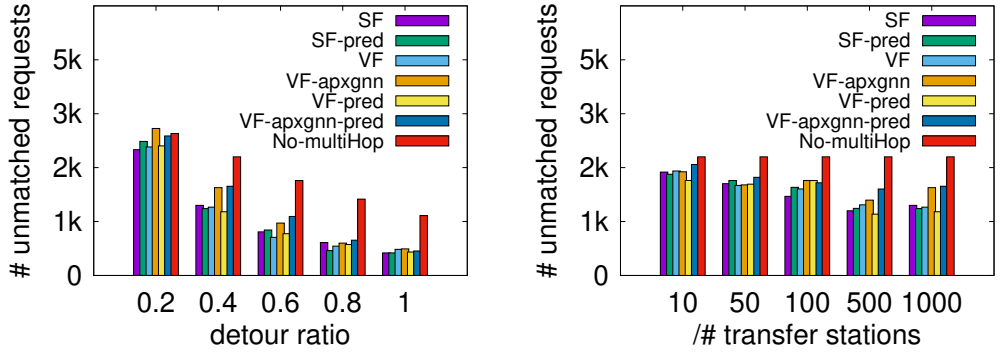
As shown in Table 5.1, All prediction strategies can successfully predict more than 90% of the actual reachable areas (as indicated by IoT), which confirms the effectiveness of applying deep learning to estimate the reachable area. The GapCustom prediction yields the highest IoT because the customized loss function produces larger predicted areas within the bounding ellipses, while the Boundary prediction may predict many areas outside of the actual reachable areas and the Gap prediction may cause excessive shrinkage.

Figure 5.8 compares the effect of prediction strategies on the ride-sharing dispatching process and No-pred still represents the reachable areas as ellipses. All prediction strategies achieves comparable travel distance. We omit the result showing the travel distance due to the space limit. All prediction strategies are observed to reduce the matching time by more than 50% while achieving comparable matching quality. The GapCustom prediction yields the best balance between the matching time and the matching quality as it applies the customized loss function that improves the matching ratio of Gap prediction with a slightly longer matching time. The Gap prediction offers faster matching time than the GapCustom prediction in almost all cases but results in fewer matched requests and longer travel distance. The Boundary prediction needs longer matching time in almost all cases except when the number of transfer points is 10, in which case the obtained number of matched requests is substantially smaller than other strategies.



(A) Effect of the detour ratio. (B) Effect of the # transfer points.

FIGURE 5.9: Matching time.



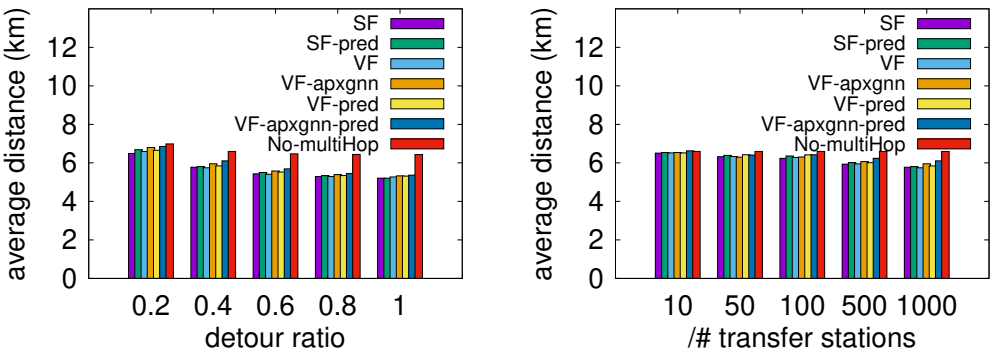
(A) Effect of the detour ratio. (B) Effect of the # transfer points.

FIGURE 5.10: # unmatched requests.

5.5.4 Algorithm Performance

Figure 5.9 illustrates the performance of proposed algorithms. Figure 5.9a shows the effect of the detour ratio. All algorithms become more costly when increasing the detour ratio due to the more examined trip candidates. The Station-first algorithm is most sensitive to the detour ratio. The reason is that its complexity largely depends on the number of examining transfer points. A larger detour ratio results in a larger reachable area covering more transfer points.

Figure 5.9b shows the effect of the number of transfer points on the matching time. The Station-first strategy is faster than the Vehicle-first strategy when the number of transfer points is small. However, its matching time becomes remarkably more expensive when increasing the number of transfer points, while the Vehicle-first strategy is barely



(A) Effect of the detour ratio.

(B) Effect of the # transfer points.

FIGURE 5.11: Average trip distance.

affected. Therefore, the Station-first algorithm is shown to perform better for scenarios with only a few transfer points. When more transfer points are desired, the Vehicle-first algorithm becomes more efficient and favorable.

We evaluate the effectiveness of the reachable area prediction on both Station-first and Vehicle-first algorithms. Interestingly, the reachable area prediction largely reduces the matching time of the Vehicle-first algorithm but is unable to accelerate the Station-first algorithm. The reason is that large parts of the checking candidates are generated from the waiting circle of the second itinerary in the Station-first algorithm. Shrinking the reachable area cannot help to reduce the number of candidates.

As shown in Figure 5.9, both approximation strategies accelerate the Vehicle-first algorithm and they show the largest improvement when applied together. The Vehicle-first algorithm achieves one more order of magnitude faster response time when the two proposed approximation strategies are applied together. Surprisingly, predicting the reachable area improves the number of matched requests for both Vehicle-first and Station-first algorithms slightly (as shown in Figure 5.10). The reason might be that the smaller reachable areas reduce the matching possibility of requests visiting remote areas, leading to more vehicles moving around the central areas where more trips are requested and served. Approximating the transfer point saves more matching time than the reachable area prediction. It largely reduces the cost of finding optimal transfer points that requires expensive shortest path computations. Compared to only checking one optimal transfer point in Euclidean space, checking top-k points reduces missing

trips and improves approximation quality. Applying approximation strategies achieves comparable travel distance of both algorithms, as shown in Figure 5.11.

5.6 Summary

We studied a real-time and scalable multi-hop ride-sharing system that allows transfers between vehicles. Our experiments show that offering multi-hop trips can increase service request matching ratio by up to 10% while reducing the average trip distance by 12%. Such an improvement enables more than ten thousand requests to be matched that were previously discarded in big cities such as Chengdu and NYC everyday. We propose exact algorithms to compute multi-hop trips, which achieves more than two orders of magnitude faster response time than the state-of-the-arts while improving the matching quality significantly. We further accelerate the matching efficiency by another order of magnitude using deep learning and other approximation strategies.

Chapter 6

Efficient All Nearest Neighbor Algorithm in Road Networks

Chapter 4 enables real-time matching of passengers and Chapter 5 enhances the flexibility of ride-sharing by allowing transfers between vehicles. In this Chapter, we further improve the system performance by assigning pick-up and drop-off points to passengers. Enabling pick-up/drop-off points helps to reduce the detour costs of vehicles, shorten the travel time of passengers, and facilitate safer getting on/off. We study how to efficiently assign every passenger with a nearest pick-up/drop-off point. We modal the problem as an all nearest neighbor in road networks, which is a fundamental query type in spatial database and has various application in location-based services. The brute-force algorithm needs to check the shortest path distance from passengers to every potential pick-up/drop-off point. Such a method needs to run the shortest path query multiple times and thus lacks efficiency. Another strategy is to apply the nearest neighbor query for every passenger separately, which also suffers from low efficiency as it redundantly visits the same area multiple times. We propose an algorithm called VIVET for efficient finding of nearest pick-up/drop-off points for passengers. VIVET only traverses the graph once during the preprocessing phase and answers a nearest neighbor query in constant time.

Part of the contents of Chapter 6 has been published in the following paper:
Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, Lars Kulik. *Finding All Nearest Neighbors with a Single Graph Traversal*, International Conference on Database Systems for Advanced Applications (**DASFAA**) 2018.

6.1 Overview

Finding the nearest neighbor is an important query in spatial databases. Its variation includes reverse nearest neighbor search [180], continuous nearest neighbor search [181], all nearest neighbor search [87] and so forth. An important variant of the nearest neighbor query, the *all nearest neighbor query*, returns the nearest neighbor of each query object over a road network. Despite its importance, this query has not been addressed in the research literature on road networks.

ANN queries have many applications. We briefly discuss two of them: (i) ridesharing and (ii) carparks. For ridesharing, the average number of daily trips using Uber reached 20.7 million globally and 556,000 in New York City in 2019 [182], which shows the importance of highly scalable and efficient algorithms to plan trips for passengers instantly. A key problem in ride-sharing is to quickly determine the pick-up and drop-off locations of passengers. To achieve this, the service provider chooses a set of locations as potential pick-up/drop-off points and needs to quickly find the nearest pick-up (drop-off) point of the source (destination) for every passenger after receiving their requests. (ii) According to a study on parking spaces of 27 districts in the United States [183], the average oversupply ratio of parking spaces to cars requiring parking is 45% among districts that have identified parking shortages. The large oversupply ratio implies that building more parking spaces is not an effective solution to the perceived lack of parking spaces. Instead, this study shows that there is an increasing need for real-time parking management, which is able to quickly report the locations of the nearest parking spaces for all drivers. This real-time parking management requires finding nearest neighbors (carparks) for all drivers in a road network. Both applications are examples of ANN queries in road networks. Figure 6.1 shows an example of an ANN query. Given two data objects o_1, o_2 and four query objects q_1, q_2, q_3, q_4, an ANN query is to compute the nearest data object for each query object, e.g., o_1 for q_1 and q_2, and o_2 for q_3 and q_4.

Existing studies on ANN mainly focus on the Euclidean space [84–88, 90, 92], where the distance between two points is determined by their Euclidean distance. In the real world, movements of objects are usually restricted by the underlying road network. The traveling cost between two points is not only determined by their relative positions but also affected by the route between them. Take v_7 and v_{13} in Fig. 6.1 as an example.

The travel distance between them is much larger than their Euclidean distance because the route must make a long detour to avoid the lake. In road networks, the distance between two points is measured by the length of their shortest path. Data structures and heuristics used by ANN algorithms in the Euclidean space, e.g., R-tree [72] and grid-partitioning [84], are not applicable to road networks due to the different distance concepts. Our study fills the need for an efficient ANN algorithm in road networks. To the best of our knowledge, this is the first study on ANN queries in road networks.

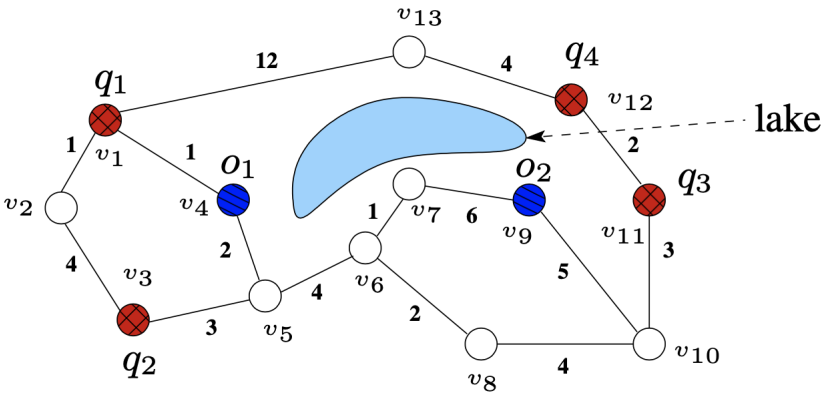


FIGURE 6.1: An example of all nearest neighbor query.

A straightforward solution to find ANNs in road networks is to apply a state-of-the-art road network *nearest neighbor* (NN) algorithm for each query object individually. However, this solution is inefficient for large numbers of query objects. Besides, it does not scale to large networks due to high memory cost.

Applying NN algorithms straightforwardly is inefficient due to the overlap of the search regions of some query objects. For example, in Fig. 6.1, both the search regions of q_3 and q_4 cover the two edges between v_{10} and v_{11} and between v_9 and v_{10}, as these two edges are both on the shortest paths to the nearest neighbor of q_3 and q_4 (i.e., o_2). When the number of query objects is large, a large part of the network may be visited multiple times, thereby severely impacting query performance. Thus, an efficient ANN algorithm needs a careful design to avoid unnecessary visits.

The reason that the straightforward solution is not scalable is due to the index structures used by road network NN algorithms. Most of the recent road network NN algorithms improve their query performance by building indices during a precomputation phase. However, these indices are memory-intensive and thus do not scale to large networks. Table 6.1 depicts the average memory consumption of two state-of-the-art road network

NN algorithms, G-tree [77] and IER-PHL [24], over five real-world road networks. The two algorithms consume rapidly increasing memory with the growing network size, which renders them inapplicable to large networks. To illustrate, IER-PHL requires over 64 GB memory to index networks with roughly 14 million vertices.

TABLE 6.1: Memory consumption of G-tree, IER-PHL, and VIVET over five road networks.

Road network	**number of vertices**	**Memory consumption**		
		G-tree	**IER-PHL**	**VIVET**
Northwest US	1 million	88.4 MB	845.1 MB	9.2 MB
East US	3.6 million	339.8 MB	6.4 GB	27.5 MB
Western US	6.3 million	543.3 MB	10.4 GB	47.8 MB
Central US	14.1 million	1.5 GB	>64 GB	107.4 MB
Full US	23.9 million	2.4 GB	>64 GB	182.7 MB

We propose **Virtual vertex traversal (VIVET)**, a road network ANN algorithm that overcomes the above limitations. In the precomputation phase, VIVET runs Dijkstra's algorithm starting with a virtual vertex. The virtual vertex is created by connecting it to every data object with an edge of weight *zero* (as shown in Fig. 6.2). After the traversal, the shortest path from the virtual vertex to each vertex in the network is obtained. For each vertex v_i, we observe that there is always *one* data object o_j on the shortest path from the virtual vertex to v_i, and o_j is the nearest neighbor of v_i. We store the nearest neighbors of all vertices in an array N. For query processing, VIVET reports the nearest neighbor of every query object by a simple lookup to N.

VIVET significantly outperforms solutions adapted from state-of-the-art nearest neighbor algorithms described above in terms of precomputation and query cost. The precomputation of VIVET is efficient and easy to implement compared with other nearest neighbor indices because it only requires a *single traversal* over the network. Furthermore, the memory consumption of the VIVET index (the array N) is linear to the number of vertices in the network, which makes it scalable to large networks. Taking Table 6.1 as an example, the memory consumption of VIVET is more than an order of magnitude lower compared with the index of G-tree and two orders of magnitude lower compared with the index of IER-PHL. In query processing, VIVET refers to the array N directly to report the query results and thus outperforms the state-of-the-art NN algorithms by almost two orders of magnitude. For example, VIVET needs less than

0.02 seconds to answer 500,000 query objects while existing NN algorithms require more than 6 seconds under the same setting.

To summarize, our contributions are as follows:

- To the best of our knowledge, this is the first study on all nearest neighbor queries in road networks.
- We propose a simple and efficient algorithm called VIVET for ANN queries in road networks. VIVET is applicable to both undirected and directed networks.
- Our theoretical analysis proves the advantage of VIVET. The precomputation of VIVET requires $O((|E|+|V|+n)\log|V|)$ time and $O(|V|)$ space, where n represents the number of data objects and $|E|$, $|V|$ represent the number of edges and vertices, respectively. The overall query complexity is linear to the number of query objects m, i.e., $O(m)$.
- We conduct experiments on both real-world and synthetic data, showing that VIVET outperforms the state-of-the-art algorithms by one to two orders of magnitude in terms of query times and precomputation costs.

6.2 Preliminaries

We start with a few basic concepts, based on which we define the all nearest neighbor query in road networks.

We consider a set of n data objects $\mathcal{O} = \{o_1, o_2, ..., o_n\}$ and a set of m query objects $\mathcal{Q} = \{q_1, q_2, .., q_m\}$. Both the data objects and the query objects are represented by points on a road network.

We consider both directed and undirected graphs. For ease of presentation, we assume an undirected graph by default, and will discuss how our techniques and algorithms can be adapted to directed graphs in Section 6.3.3.

For simplicity, we assume that the data objects and the query objects are located at the graph vertices. This assumption can be easily met by adding vertices that represent the data objects or query objects to the graph.

Nearest neighbor query in a road network. Given a query object q and a set of data objects $\mathcal{O}$ in a road network G, a *nearest neighbor* query finds the nearest data object

$o_i \in \mathcal{O}$ with the smallest shortest path distance to q, denoted by $NN(q)$:

$$NN(q) = \{o_i \in \mathcal{O} | \forall o_j \in \mathcal{O} : d_n(o_i, q) \leq d_n(o_j, q)\}$$

All nearest neighbor query in a road network. Given a set of query objects $\mathcal{Q}$ and a set of data objects $\mathcal{O}$ in a road network G, an *all nearest neighbor* query finds the nearest data object $o_j \in \mathcal{O}$ with the smallest shortest path distance to every query object $q_i \in \mathcal{Q}$. The query answer is a set of tuples of a query object and its nearest data object, denoted by $ANN(\mathcal{Q}, \mathcal{O})$. Formally,

$$ANN(\mathcal{Q}, \mathcal{O}) = \{\langle q_i, o_j \rangle | q_i \in \mathcal{Q}, o_j \in \mathcal{O}, o_j = NN(q_i)\}$$

In Fig. 6.1, $ANN(Q, O) = \{\langle q_1, o_1 \rangle \langle q_2, o_1 \rangle, \langle q_3, o_2 \rangle, \langle q_4, o_2 \rangle\}$.

6.3 VIVET

In this section, we present our VIVET algorithm for ANN queries in road networks. The VIVET algorithm precomputes and stores the nearest data object of every vertex in the network. When an ANN query is issued, we simply lookup for the vertices where the query objects lie on and return the corresponding nearest data object. Next, we detail the precomputation process of VIVET, which computes the nearest neighbors for all the vertices in a road network with a single traversal over the network.

6.3.1 Precomputation

To compute the nearest neighbors for all the vertices, a straightforward method is to run a graph shortest path search algorithm such as Dijkstra's algorithm [10] starting from every vertex in the network. However, this algorithm may traverse the network too many times and access the same vertices and edges repetitively.

To avoid such repetitive computation and overlapping network traversals, we propose to traverse the network starting from a virtual vertex which connects to all data objects. The traversal will go through every vertex in the network. When the traversal reaches

a vertex, the corresponding path reaching the vertex must pass a data object and this data object will be recorded as the nearest neighbor of the vertex.

We first augment the graph G with a virtual vertex v^* and connect it to every data object $o_i \in \mathcal{O}$ with a directed edge $\overrightarrow{e(v^*, v_i)}$ of weight 0. As we assume that the data objects are all on the vertices, this process effectively connects the virtual vertex to every vertex v_i in V on which a data object lies. We denote the resulting graph as G^*, $G^* = \langle V^*, E^* \rangle$, where $V^* = V \cup \{v^*\}$ and $E^* = E \cup \{\overrightarrow{e(v^*, v_i)} | \ \overrightarrow{e(v^*, v_i)}$ connects v^* to $o_i \in \mathcal{O}\}$.

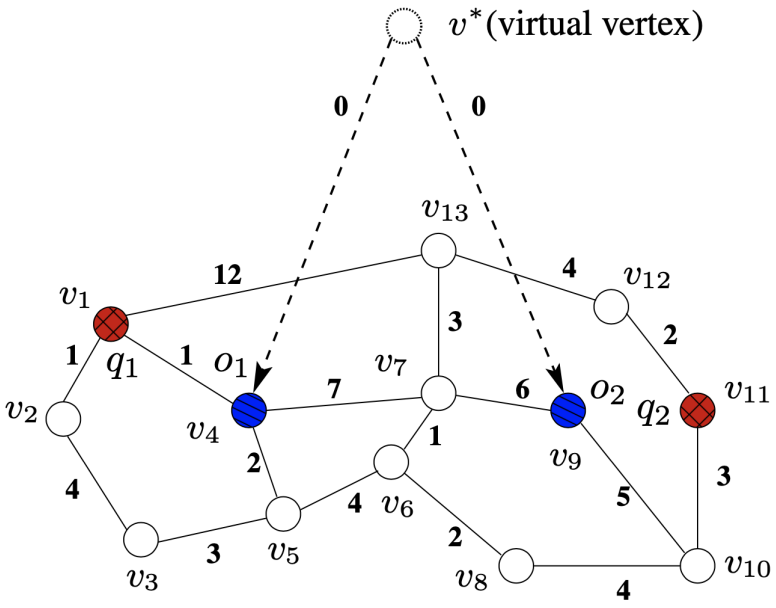


FIGURE 6.2: An example of VIVET.

We call G^* the *augmented graph*. Figure 6.2 illustrates such a graph. The virtual vertex v^* is connected to vertices v_4, v_9 where there are data objects o_1 and o_2. Note that even though the original graph G is undirected, the augmented graph G^* contains directed edges that only allows traveling from v^* to the vertices (data objects) in V. The directed edges here are used to ensure that the graph traversal starting from v^* will not go back to v^* (note the zero weight for the edges connecting v^* to the data objects), so as to guarantee the validity of our precomputation algorithm.

Once the augmented graph G^* is computed, we run a single-source graph shortest path algorithm (e.g., Dijkstra's algorithm) starting from the virtual vertex v^* to find the shortest path to every vertex in V. We record the data object that a path goes through. When the traversal reaches a vertex v_i, the data object on the path to v_i is recorded.

We show that there is always one and only one data object on the shortest path from v^* to v_i and this data object is the nearest data object of v_i with the following two lemmas.

Lemma 6.1. *Given a connected graph $G = \langle V, E\rangle$ and an augmented graph $G^* = \langle V^*, E^*\rangle$ created from G, for every vertex $v_i \in V$, there must be one and only one data object on the shortest path from v^* to v_i.*

Proof. First, we prove that there must be at least one data object on the shortest path to v_i. Since G is connected, there must be a path that connects v^* to v_i by the design of the augmented graph G^*. Since v^* is only connected to the data objects, any path including the shortest path from v^* to v_i must go through at least one data object.

Next, we prove by contradiction that there is at most one data object on the shortest path to v_i. Let $\langle P = v^*, o_j, ..., v_i\rangle$ be the shortest path from v^* to v_i. The second vertex on the path must be a vertex on which a data object o_j lies by design of G^*. Suppose that there is another data object o_k in the path, i.e., $P = \langle v^*, o_j, ...o_k, ..., v_i\rangle$. Then, the distance between o_j and v_i must be larger than that between o_k and v_i, i.e., $d_n(o_j, v_i) > d_n(o_k, v_i)$. Since there is an edge that connects from v^* to every data object, there must be another path $P' = \langle v^*, o_k, ..., v_i\rangle$. The edge between v^* to every data object has a zero weight. Thus, the path length $l(P) = d_n(o_j, v_i) > d_n(o_k, v_i) = l(P')$, which contradicts that P is the shortest path between v^* and v_i. □

For example, in Fig 2, there is only one data object o_2 on the shortest path between v^* and v_{11}, which goes through v^*, o_2, v_{10}, v_{11}.

Lemma 6.2. *Given a connected graph $G = \langle V, E\rangle$ and an augmented graph $G^* = \langle V^*, E^*\rangle$ created from G, the data object on the shortest path from v^* to every vertex $v_i \in V$ is the nearest data object of v_i.*

Proof. The proof is similar to the second half of Lemma 1's proof and omitted. □

Lemmas 1 and 2 guarantee the correctness of using a single-source shortest path algorithm to compute the nearest neighbor for every vertex. Any single-source graph shortest path algorithms can be used. We use Dijkstra's algorithm for its simplicity and efficiency [10].

Once the nearest neighbors of the vertices are computed, we store them as an array of vertex-NN pairs (together with the shortest path distance) for fast retrieval at query processing. We call this array the *NN array*. Table 6.2 illustrates the NN array built for the example shown in Fig. 2.

TABLE 6.2: VIVET index of Fig. 6.2.

	$\boldsymbol{v_1}$	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	$\boldsymbol{v_{11}}$	v_{12}	v_{13}
NN	$\boldsymbol{o_1}$	o_1	o_1	o_1	o_1	o_1	o_2	o_1	o_2	o_2	$\boldsymbol{o_2}$	o_2	o_2
distance	**1**	2	5	0	2	6	6	8	0	5	**8**	10	9

Algorithm 9: Precomputation

Input : $G = \langle V, E \rangle$, object set $\mathcal{O}$

Output: NN array indexing the nearest object of every vertex $v_i \in V$.

1 create a virtual vertex v^*;
2 $E^* \leftarrow E, V^* \leftarrow V \cup \{v^*\}$;
3 **for** $o_i \in \mathcal{O}$ **do**
4 create a virtual edge $\overrightarrow{e_{*,o_i.vid}}$; `// ` $o_i.vid$ `is the vertex ID of` o_i
5 $w(\overrightarrow{e_{*,o_i.vid}}) \leftarrow 0$;
6 $E^* \leftarrow E^* \cup \{\overrightarrow{e_{*,o_i.vid}}\}$;
7 initialize an array N with size $|V|$;
8 **for** $v_i \in V$ **do**
9 $N[v_i].nndistance \leftarrow \infty$;
10 **for** $o_i \in \mathcal{O}$ **do**
11 $N[o_i.vid].nndistance \leftarrow 0$;
12 $N[o_i.vid].nnid \leftarrow o_i.oid$; `// ` $o_i.oid$ `is the object ID of` o_i
13 initialize a priority queue PQ;
14 PQ.insert (v^*);
15 **while** $PQ \neq \emptyset$ **do**
16 $v_i \leftarrow$ the first element in PQ;
17 **if** v_i *has not been visited before* **then**
18 **for** *each adjacent vertex* v_j *of* v_i *that have not been visited before* **do**
19 **if** $N[j].nndistance > N[i].nndistance + w(e(v_i, v_j))$ **then**
20 $N[j].nndistance \leftarrow N[i].nndistance + w(e(v_i, v_j))$;
21 $N[j].nnid \leftarrow N[i].nnid$;
22 **if** $N[j].nndistance > N[i].nndistance + w(e(v_i, v_j))$ *or* $v_i = v^*$ **then**
23 PQ.insert(v_j);
24 mark v_i as visited;
25 **return** N;

Algorithm 9 summarizes the precomputation procedure of VIVET. The algorithm starts with creating the augmented graph G^* based on the road network G (Lines 1 to 6). Then, it initializes an array N of size $|V|$ to store the NN pairs (Line 7). The data objects located at vertices are the nearest data objects of those vertices, which yield a nearest

neighbor distance of 0 (Lines 8 to 10). Next, the graph traversal starts. We use a priority queue PQ to facility the traversal (Line 11). Each element in the queue is a vertex in G^* to be visited, which is prioritized by its distance to the nearest data object computed so far in the NN array. The virtual vertex v^* is inserted into PQ to initialize the traversal (Line 12). A loop is run to keep popping out vertices from PQ (Lines 13 to 21). The vertex with the smallest distance to the nearest data object in PQ is popped out first (Line 14). When a vertex v_i is popped out and visited for the first time, vertices connected to it that have not been visited before are inserted into PQ (Lines 16 to 20). For each such vertex v_j, if the path through v_i is shorter than the existing shortest path to v_j, we update the distance to nearest data object of $v_j(N[j].nndistance)$ to be the nearest neighbor distance of v_i plus the weight of the edge between v_i and v_j (Line 18), and the nearest data object of $v_j(N[j].nnid)$ is updated to be that of v_i (Line 19). When PQ becomes empty, all vertices will have been visited and their nearest data objects are computed and stored in the NN array N. The array N is returned and the algorithm terminates (Line 22).

6.3.2 Query Processing

Once the NN array is computed, an ANN query can be processed by first locating the vertex v_j on which every query object q_i lies and then retrieving the nearest data object of v_j from the NN array, which is returned as the nearest data object of q_i. If a query object is lying on an edge, we locate both vertices of the edge and retrieve their nearest data objects. We compare the distances of the two retrieved data objects to q_i and return the closer one as the nearest data object of q_i. We omit the pseudocode of the query processing procedure for conciseness.

Continuing with the example shown in Fig. 2, where there are two query objects q_1, q_2 represented by the two red circles, q_1 is at v_1 and q_2 is at v_{11}. The nearest neighbor of v_1 is o_1 and the nearest neighbor of v_{11} is o_2 as shown in the NN array listed in Table 6.2. VIVET reports o_1 and o_2 as the nearest neighbors of q_1 and q_2, respectively.

6.3.3 Generalizing the Algorithm

VIVET can be generalized to directed networks and to process ANN queries without precomputation with small changes.

When applied to directed networks, we need to update the traversal for single-source graph shortest path computation as follows. When a vertex is visited (Line 15 of Algorithm 1), we retrieve its inbound edges and add the vertices connected by these edges to the priority queue PQ to be visited next, i.e., we update Line 16 of Algorithm 1 to be "for every vertex v_j that has an edge pointing to v_i". We need to reverse the direction of the edges of the virtual vertex v^* such that they point from the data object vertices to v^* instead of from v^* to the data objects. By doing so, we find the shortest "reverse" paths from the data objects to the vertices in the network, which are the shortest paths from the vertices to the data objects. This approach is correct because we still use Dijkstra's algorithm graph expansion procedure, but restricting the direction of the edges to ensure that the paths found are going from the vertices to the data objects. Our experiments show similar behavior of VIVET for both undirected and directed graphs.

When processing an ANN query without the precomputed NN array, we run the single-source graph shortest path computation online. We find the shortest paths from the virtual vertex v^* to all query objects instead of all network vertices. When the shortest paths are found, the data object o_j on the shortest path to query object q_i is returned as the nearest data object of q_i. The correctness of doing so is guaranteed by Lemmas 1 and 2 above straightforwardly. Our experiments verify the efficiency of VIVET in dynamic scenarios, especially when the number of query objects is large. For example, when the network has over one million vertices and 2^{11} data objects, dynamic VIVET requires 0.5 seconds to answer an ANN query with 2^{16} query objects while the other state-of-the-art algorithms requires at least 0.8 seconds.

A *multi-source Dijkstra's algorithm* has been proposed in the literature [184] that starts by adding multiple source vertices into the priority queue PQ. The focus of [184] is to run the multi-source Dijkstra's algorithm to test the reachability of different points and find out the most time-consuming shortest path in the graph for emergency services. Another study [185] shares a similar idea and uses a multi-source shortest path approach for location privacy. To the best of our knowledge, we are the first to apply this technique for finding ANNs in road networks.

6.3.4 Algorithm Complexity

Next, we analyze the complexity of VIVET. We denote the number of data objects as n and the number of query objects as m. We also denote the numbers of vertices and edges in G^* as $|V^*|$ and $|E^*|$, which equals to $|V|+1$ and $|E|+n$, respectively.

Precomputation. Creating the augmented graph G^* takes $O(|V|+1+|E|+n)$ time. The time for traversing G^* to compute the nearest data objects is determined by the time of the single-source shortest path algorithm. We use Dijkstra's algorithm, which has a time complexity of $O((|E^*|+|V^*|)\log|V^*|)$ in the worst case by using a binary heap for the priority queue PQ, which is equivalent to $O((|E|+|V|+n)\log|V|)$. Overall, the time complexity of the precomputation of VIVET is $O((|E|+|V|+n)\log|V|)$. The size of the NN array is linear to the number of vertices, i.e., $O(|V|)$.

Query processing. The query time complexity of VIVET is linear to the number of query objects. For each query object, the nearest neighbor is computed in constant time from the NN array. Therefore, the query time complexity of VIVET is $O(m)$.

6.4 Experiments

We experimentally compare the performance of VIVET against the state-of-the-art NN algorithms, IER-PHL [78], G-tree [77] and INE [73]. All algorithms are implemented in C++ and run on a 64-bit virtual node with a 1.8 GHz CPU and 64 GB memory from an academic computing cloud (Nectar [177]) running on OpenStack.

6.4.1 Experimental Setup

The road network dataset is described in Chapter 3. As the experimental results on the travel distance dataset are consistent with that on the travel time dataset in most cases, we focus on showing experiments on the travel time dataset. We use two methods to create data object sets: mapping real-world POIs and synthetically sampling. We use eight types of real-world POIs extracted from OpenStreetMap by Abeywickrama et al. [78]. We also synthetically sample vertices of the networks to be the data objects and query objects following two distributions, uniform and clustered. The uniform distribution simulates scenarios where areas with more vertices tend to have more objects, while

the clustered distribution simulates scenarios where objects may be clustered in some areas. The maximum number of vertices is 50 in every cluster.

TABLE 6.3: Experiment settings.

Parameters	**Values**	**Default**
Road Networks	Refer to table 3.3	*NW*
number of data objects	2^7 to 2^{16}	2^{11}
number of query objects	2^7 to 2^{16}	2^{10}
Real-world POIs	Refer to Table 2 in [78]	*Parking*
Synthetic data objects distributions	Uniform, Clustered	*Uniform*
Synthetic query objects distributions	Uniform, Clustered	*Uniform*

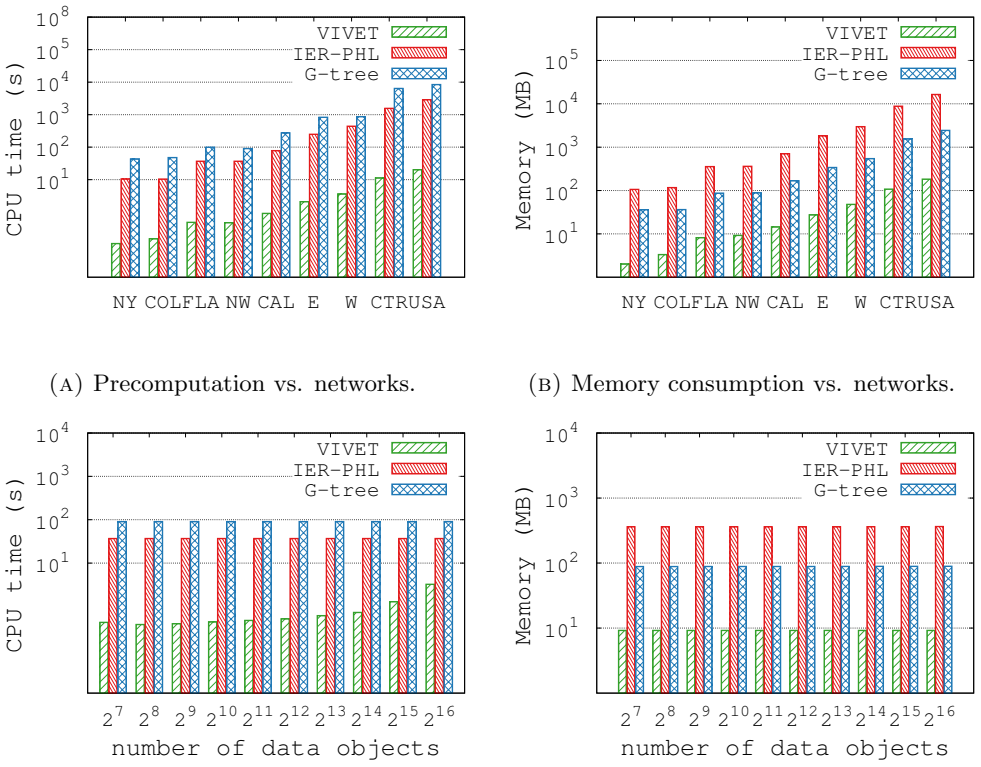
Table 6.3 shows the range of variables we use in our experiments. In a default setting, we run queries on 2^{11} uniformly distributed data objects and 2^{10} uniformly distributed query objects over the network *NW*. *Park* is the default POI type in experiments using real-world POIs as data objects. We first show the algorithm performance on undirected graphs and then compare algorithms on directed graphs.

6.4.2 Precomputation Costs

We compare the precomputation costs of VIVET with the index-based NN algorithms IER-PHL and G-tree by measuring their time and memory consumption.

Effect of the network size. Figures 6.3a and 6.3b show the precomputation costs over different networks. All algorithms require longer time and larger memory to build indices when the network has more nodes and edges. Compared with IER-PHL and G-tree, VIVET reduces the precomputation time by two orders of magnitude and saves the memory consumption by one order of magnitude due to a single traversal over the network. Compared to the precomputation costs on the travel distance dataset as shown in Table 6.1, both G-tree and VIVET require consistent precomputation costs on the two datasets. IER-PHL, however, requires less memory on the travel time dataset by taking advantage of the travel speed (geometrical length divided by travel time) to improve the effectiveness of the highway decomposition, thereby reducing the index size.

Effect of the number of data objects. Figures 6.3c and 6.3d show the effect of the number of data objects on the precomputation costs. Varying the number of data objects has little effect on the precomputation costs of both IER-PHL and G-tree because their



(A) Precomputation vs. networks.

(B) Memory consumption vs. networks.

(C) Precomputation time vs. # data objects.

(D) Memory consumption vs. # data objects.

FIGURE 6.3: Precomputation costs.

precomputation costs are dominated by the process of building network indices. As for VIVET, its precomputation time increases with the growing number of data objects, which is caused by the increasing number of virtual edges added in the augmented network G^*. Even though the precomputation costs of VIVET are impacted by the number of data objects, the number of data objects in real world scenarios usually lies within a reasonable range. For example, the number of parking spaces in NW is 5098 [78], which lies in the range between 2^{12} and 2^{13} as shown in Fig. 6.3c. The precomputation time of VIVET in this range is approximately 1.5% of that of IER-PHL and 0.5% of that of G-tree. In terms of memory consumption VIVET has a constant index size when the number of data objects changes as its index size is determined only by the number of vertices. Its index size is at least an order of magnitude smaller than those of IER-PHL and G-tree.

6.4.3 Query Costs

We further analyze the query performance of IER-PHL, G-tree, INE, and VIVET by comparing their query times.

Effect of the network size. Figure 6.4 shows the query times of the four algorithms on different networks. The query times of G-tree and INE increase rapidly with the growing network size due to their larger search region, while those of IER-PHL and VIVET are much less impacted by the network size. However, IER-PHL consumes large size of memory for large networks as shown in Fig.6.3b. VIVET outperforms the other three algorithms by more than two orders of magnitude over all networks, which shows the efficiency and scalability of VIVET in large networks.

Effect of the number of data objects. Figure 6.6 shows the query times of uniform and clustered data objects when varying number of data objects. VIVET again outperforms the state-of-the-art by more than two orders of magnitude. Furthermore, the query time of VIVET is unaffected by the size and distribution of data objects as it only performs a simple lookup to answer an ANN query.

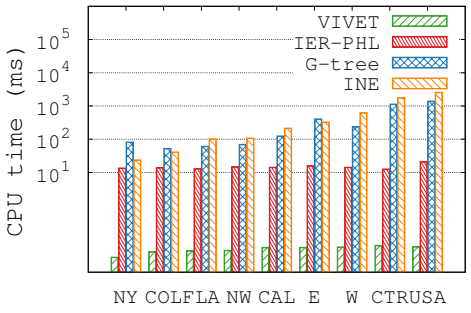


FIGURE 6.4: Query time vs. network size.

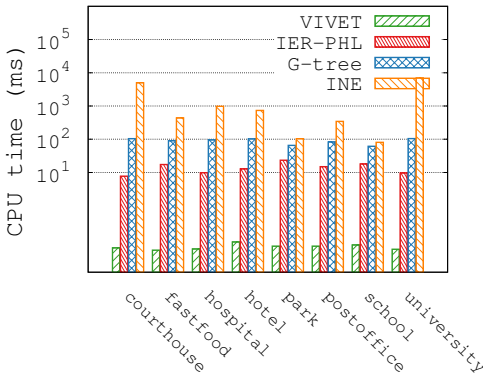
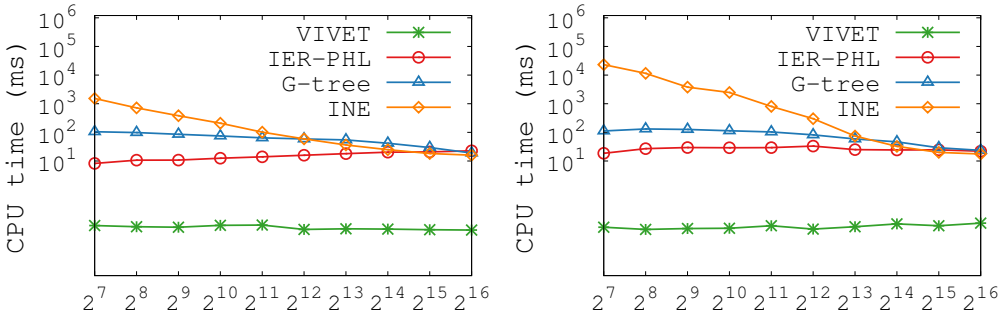


FIGURE 6.5: Query time vs. real data objects.

Effect of the number of query objects. Figure 6.7 shows the effect of the number of query objects on the query performance. Query objects in Fig. 6.7a are generated following the uniform distribution while those in Fig. 6.7b are generated following a clustered distribution. As expected, the query times of all algorithms grow with the increasing number of query objects. VIVET is two orders of magnitude faster than the most efficient baseline IER-PHL. Furthermore, our scalability experiments show that VIVET can answer an ANN query with 10 million query objects within 0.3 seconds,

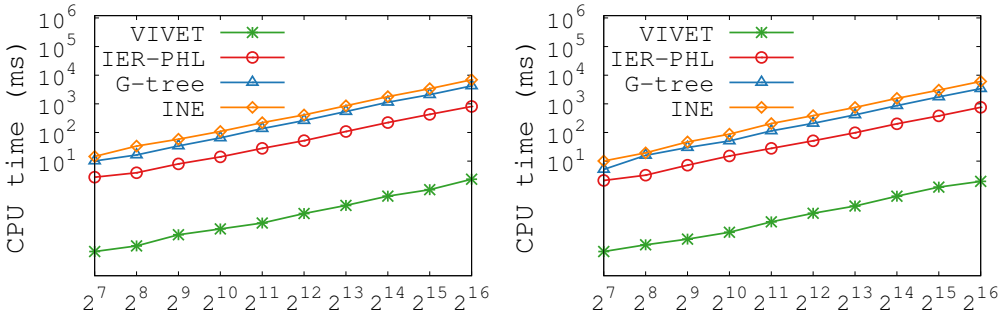
while the other state-of-the-art algorithms require more than 2 seconds to answer such large number of query objects.

Real-world object sets. Figure 6.5 shows the query times of different algorithms when data objects are generated based on real-world POIs. VIVET outperforms other algorithms by more than two orders of magnitude on all types of POIs examined.



(A) uniform data objects (B) clustered data objects

FIGURE 6.6: Effect of the number of data objects on query time.



(A) uniform query objects (B) clustered query objects

FIGURE 6.7: Effect of the number of query objects on query time.

6.4.4 Experiments on Directed Graphs

Given the fact that G-tree requires undirected graphs for its graph partitioning phase, and that IER-PHL assumes undirected graphs for indexing, we only compare the performance of VIVET against INE on the directed network *Europe*.

Precomputation cost. Figure 6.8 shows the precomputation time of VIVET when the number of data objects varies. The precomputation time required by VIVET increases slightly with increasing number of data objects, which is consistent with the experiments

in undirected graphs. The memory consumption of VIVET on *Europe* is 137 MB, which remains linear to the number of vertices in the network.

Query cost. The query times of VIVET and INE on directed network are compared in Fig. 6.9. VIVET outperforms INE by up to four orders of magnitude in this set of experiments. When the number of data objects increases, the query performance of INE improves due to the smaller size of the search region. However, even for dense data objects, VIVET still outperforms INE by three orders of magnitude.

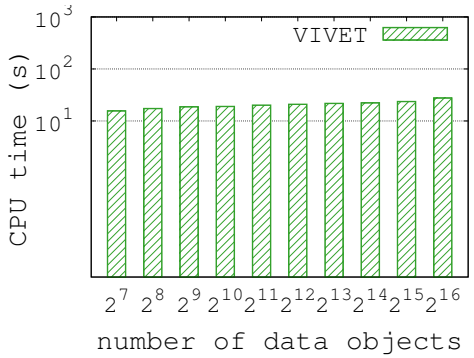


FIGURE 6.8: Precomputation time (directed graph).

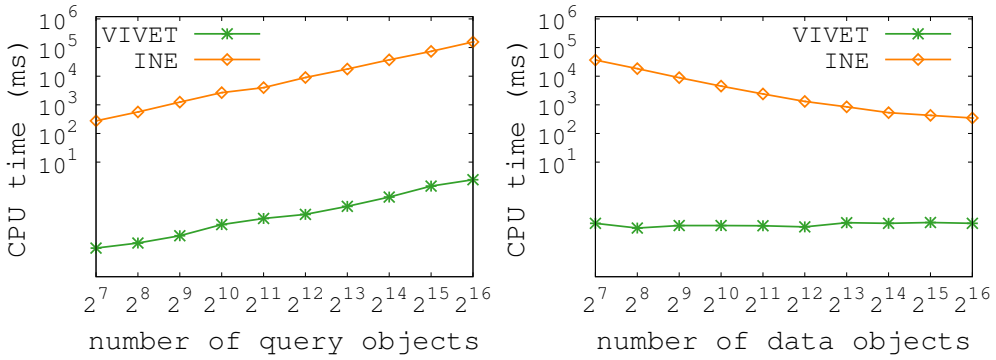


FIGURE 6.9: Query time (directed graph).

6.5 Summary

We studied all nearest neighbor queries in road networks and proposed a scalable and efficient algorithm named VIVET. Compared with the methods adapted from state-of-the-art nearest neighbor algorithms, VIVET reduces the precomputation and query costs by one to two orders of magnitude. The improvements are achieved via a shared computation technique that computes the nearest neighbors for all query objects at the same time with a single graph traversal. Extensive experiments using real road networks

confirm the advantages of VIVET in terms of precomputation time, storage space, and query time compared to the state-of-the-art network NN algorithms.

Looking ahead, considering the fact that existing index structures for nearest neighbor queries suffer due to large memory consumption, while VIVET effectively overcomes this limitation, it is worth exploring the applicability of the VIVET index structure on other variants of the nearest neighbor problems in road networks. Our preliminary results already confirmed the advantage of VIVET on NN queries compared to the state-of-the-art in terms of precomputation costs and query performance.

Chapter 7

Conclusion and Future Work

In this chapter, we summarize our contributions and discuss how the proposed techniques can be leveraged for research problems to be explored in the future.

This thesis proposed efficient and scalable algorithms for ride-sharing. We first solved a basic problem in ride-sharing: efficiently finding vehicles to serve new requests. Although various speed-up techniques have been proposed in the literature, they either cannot consider all constraints or suffer from high updating costs. We overcome the challenges of state-of-the-art by representing the time constraints of requests and vehicles using geometric objects. The geometric objects are easy to compute and index, enabling fast assessing on potential vehicles and low updating cost for highly dynamic scenarios. The proposed algorithm is generic and provides effective pruning for various optimization goals. In light of the achieved superior performance, the proposed algorithm has great potential to apply on and improve the efficiency of many other ride-sharing problems.

We then investigated multi-hop ride-sharing. Multi-hop ride-sharing creates key opportunities for increasing the matching ratio, reducing the travel distance of vehicles, and improving the system performance of ride-sharing. However, it was mainly researched in the transportation area and existing algorithms were only evaluated on small instances due to the computational complexity. We proposed the first efficient multi-hop ride-sharing matching algorithm that is scalable to large networks and highly dynamic scenarios. Our study supports the deployment of multi-hop ride-sharing in real-world and provides opportunities for many interesting future works, e.g., further enhance the system performance and consider more factors in multi-hop ride-sharing.

We further proposed the first efficient and scalable all nearest neighbor algorithm in road networks, enabling efficient assignment of nearest pick-up/drop-off locations for passengers. All nearest neighbor query is a typical query in spatial database and has various applications in location-based services. However, it has only been investigated on other metrics but has not been studied in road networks. Our work is the first study of all nearest neighbor in road networks. We achieved constant query time for a nearest neighbor query with the assist of a lightweight index that is built by only one traversal on the road network. Besides accelerating the ride-sharing matching process, the proposed ANN algorithm enables real-time responding time for many other location-based services.

7.1 Conclusion

The contribution of each chapter is detailed as follows.

In Chapter 2, we reviewed existing shortest path algorithms that compute the distance between locations in road networks. We summarized nearest neighbor algorithms that efficiently find the nearby objects for querying locations in road networks. We further surveyed the state-of-the-art ride-sharing algorithms and attempt to improve the flexibility of ride-sharing.

In Chapter 3, we defined the ride-sharing dispatching problem and the related concepts.

In Chapter 4, we improved the matching time by proposing an efficient pruning algorithm. Matching trip requests to vehicles efficiently is critical for the service quality of ride-sharing. To match trip requests with vehicles, a prune-and-select scheme is commonly used. The pruning stage identifies feasible vehicles that can satisfy the trip constraints (e.g., trip time). The selection stage selects the optimal one(s) from the feasible vehicles. The pruning stage is crucial to lowering the complexity of the selection stage and to achieving efficient matching. The performance of the pruning algorithms largely depends on how the ride-sharing data is retrieved and updated through a ride-sharing index. Existing pruning algorithms need to store and update a large amount of information, thus suffering from the expensive maintenance cost for the highly dynamic scenarios. Besides, they may wrongly prune feasible matches due to the approximation of the road network distance. We proposed an effective and efficient pruning algorithm

called GeoPrune. GeoPrune represents the time constraints of trip requests using circles and ellipses. The index thus only needs to maintain a set of geometric objects, which can be computed and updated efficiently. Meanwhile, GeoPrune guarantees to find all possible matches. Experiments on real-world datasets showed that GeoPrune reduces the number of vehicle candidates by an order of magnitude and the update cost by two to three orders of magnitude compared to the state-of-the-art.

In Chapter 5, we studied the multi-hop ride-sharing to bring more flexibility to the system and enhance the matching quality. Multi-hop ride-sharing allows passengers to transfer between vehicles within a single trip, which significantly extends the benefits of ride-sharing and offers ride opportunities that have not been possible possible otherwise. Despite its advantages, offering real-time multi-hop ride-sharing services at large scale is a challenging computational task due to the large combination of vehicles and passenger transfer points. To address these challenges, we proposed efficient and scalable algorithms that find potential multi-hop trips quickly in large metropolitan areas. Our experiments on real-world datasets showed the benefits of multi-hop ride-sharing services and demonstrated that our proposed algorithms are more than two orders of magnitude faster than the state-of-the-art. Our approximation algorithms offer a comparable trip quality to our exact algorithm, while improving the ride-sharing request matching time by another order of magnitude.

In Chapter 6, we proposed VIVET, an index-based algorithm to efficiently process ANN queries. ANN is a fundamental query and has various applications in location-based services including ride-sharing. Existing ride-sharing systems usually select a set of locations as potential pick-up/drop-off stations. The passengers then need to walk to (from) the nearest stations of their source (destination) to get on (off) the assigned vehicles. Finding the nearest stations for every request is an application of all nearest neighbor query (ANN) in road networks. Existing algorithms for finding the nearest station require expensive shortest path distance computation that delays the response times. Unlike other index-based algorithms that require expensive pre-processing time and large index size, our proposed algorithm VIVET performs a single traversal on a road network to precompute the nearest data object for every vertex in the network, further enabling us to answer an ANN query through a simple lookup on the precomputed nearest neighbors. We analyzed the cost of the proposed algorithm both theoretically and empirically. Our results showed that the algorithm is highly efficient and scalable.

It outperforms adapted state-of-the-art nearest neighbor algorithms in both precomputation and query processing costs by more than one order of magnitude.

7.2 Future Work

The algorithms proposed in this thesis substantially reduce the search space and improve the dispatching efficiency. The provided techniques not only tackle the proposed research problems but also provide key opportunities for problems that were difficult to solve. Next, we envision future development for advanced ride-sharing systems and discuss how the proposed techniques could potentially solve new problems.

Traffic-aware ride-sharing. Existing ride-sharing algorithms assume a maximum speed of all roads but overlook the impact of real-world traffic. The real-time traffic condition affects the travel times between locations and delays the estimated arrival time of requests, hence a vehicle may become infeasible to serve the new request if the traffic is considered. One possible solution is traffic-aware pruning. In Chapter 3, we defined a concept called reachable area and developed criteria to prune potential vehicles by detecting the coverage of reachable areas. Representing reachable areas using MBRs enables fast retrieval on possible vehicles with low update cost required, which largely improves the dispatching efficiency. The calculation of the reachable areas is critical to the matching efficiency. Smaller reachable areas help to prune more vehicles and are expected to improve the selection cost and the matching time. In Chapter 3, we proved that the reachable areas are bounded by ellipses. In Chapter 4, we applied deep learning to shrink the representation of reachable areas considering the road network structure. However, the applied model assumes a maximum travel speed on all roads and unable to model real-time travel speeds. It is worth exploring how to consider the real-time traffic conditions in the model to further shrink the representation of the reachable areas and achieve more effective pruning. The reachable area depends on the source, destination, and the maximum allowed travel time between source and destination, The traffic condition is affected by the temporal features such as time of the day and other external features such as weather conditions. The model should jointly learn these features for the reachable areas prediction.

In our proposed algorithms and other existing algorithms, the vehicles are assumed to follow the scheduled route to sequentially serve the committed requests. The schedules may be interrupted by incidents such as the delay arrival times and substantially impact the satisfaction of passengers. One possible solution is to propose a metric and measure the robustness of every vehicle such that the vehicle with the highest robustness can be selected and dispatched to the new request. The metric should consider the possibility and extent of the delay, the chance of interrupted incidents and other factors. Computing these factors should consider the future traffic conditions. A possible formulation is to linearly combine these factors.

Consideration of future impacts. Routing and dispatching vehicles in ride-sharing is a sequential decision-making process. The dispatching results of preceding requests affect the subsequent requests and the overall system performance. A current optimal decision may lead to an undesirable future distribution of vehicles and inferior system performance. Therefore, advanced ride-sharing systems should consider the long-term effects when dispatching or routing vehicles. Although several existing studies consider future demands while dispatching/routing, they can only model the effect on the next time slot but unable to optimize the long-term impacts [33, 132, 142–144]. Deep learning methods such as reinforcement learning naturally model the sequential decision-making process, which has great potential to model the long-term impacts and achieve better global optimization. However, it is challenging to consider the enormous involved participants (i.e., drivers and passengers) and the complicated mutual impact between these participants when designing the model. Several studies have verified the benefits of applying reinforcement learning to improve dispatching quality [186–190]. However, they only consider the non-sharing taxi service when every vehicle only serves one request at a time but unable to explore the sharing possibility between passengers. The main bottleneck of employing reinforcement learning on ride-sharing is the large action space caused by the massive vehicle-request pairs. Specifically, they usually assign a reward value for every vehicle-request pair considering the current and future gains and dispatch pairs to maximize the overall reward value. It is expensive to compute the reward value for every pair and group pairs for global optimization. The highly effective pruning achieved by GeoPrune may help to substantially reduce the solution space, thus enabling the employment of deep learning and reducing the training cost.

Dynamic pricing strategies. Setting appropriate pricing for trips is crucial in ride-sharing. On one hand, the supply and demand situation affects the pricing, e.g., trips originated from the under-supplied area should be higher-priced. On the other hand, the pricing of trips conversely affects the future supply and demand, e.g., areas with more expensive trips will attract more drivers. Besides, an unreasonably high price is unattractive to users, while a low price undermines the benefits for both drivers and the service provider. The pricing is critical for all participants, including passengers, drivers, and the service provider. Despite the importance, existing studies cannot fully exploit the benefits of the dynamic pricing in ride-sharing. A key overlooked factor is the sharing possibility along the route. Trips with higher possibilities to meet and share future passengers have greater potential to reduce the operating cost, thus should be priced lower. It is challenging to determine the sharing possibility of a route because sharing trips should not only originate around the route but also need to head similar directions. Nevertheless, the concept of the reachable area may provide an effective way to detect the sharing possibility. Specifically, the pruning criteria proposed in Chapter 4 have great potential to measure the number of possible sharing trips after predicting future requests. One can compute the reachable areas and distributions of all future requests and compute the sharing possibility by detecting the coverage of these reachable areas.

Collaborative routing. Traffic congestion is a severe problem faced by modern cities. Vehicles may be congested for a long time if they are routing individually and unaware of the other vehicles. Nevertheless, ride-sharing applications and other navigation platforms provide key opportunities for sharing the real-time locations and routing schedules between vehicles. A platform can split the traffic flows by collaboratively recommending routes to vehicles, hence relieving traffic congestion. One key problem in collaborative routing is to find alternative paths for vehicles, which usually requires expensive graph traversals. The concept of reachable area applied in Chapter 4 and Chapter 5 bounds the area of alternative paths, which helps terminate the traversal in an early stage and reduce the search space. The reachable area may also help to quickly determine whether it is necessary to recommend other alternative paths to a vehicle. Vehicles with their reachable areas overlapping a few other reachable areas indicate low flows along their routes and thus it is unnecessary to switch the paths for these vehicles. On the other hand, vehicles with their reachable areas that overlap with a large amount of other

reachable areas have high risks of congestion and it is better to plan alternative routes for them. Another possible solution is to employ spatial indices and shortest path algorithms to determine the crowdness. For example, we can partition the space into sub-graphs and count the number of pass-through vehicles at different time slots for each sub-graph. Only vehicles that are scheduled to visit crowded areas need to switch to other alternative paths.

Highly flexible ride-sharing. The algorithms proposed in Chapter 5 enhance the flexibility of ride-sharing by allowing transfers between vehicles. Despite the efforts, there remain multiple avenues to further improve the flexibility of ride-sharing system.

Most of the existing works plan one trip for every passenger. However, passengers may enjoy higher flexibility if offered multiple trip options. These trip options should consider various factors such as payment, detour cost, waiting time. Several previous algorithms attempt to find skyline results to passengers [27, 134], i.e., trips beat others regarding at least one factor. The response time is a major concern for returning skyline results. Our GeoPrune algorithm proposed in Chapter 4 has great potential to improve the efficiency considering the observed effective pruning performance.

Another direction for highly flexible ride-sharing is enabling multi-modal trip planning. Planning a multi-modal trip helps to combine the benefits of different transportation modes and improve urban mobility. For example, vehicles have higher flexibility to reach remote areas while the metro can avoid traffic congestion. For passengers traveling from a remote area to the central area in rush hours, their best travel plan may be to share a ride first and then transit to a train/subway. However, existing studies are unable to find ride-sharing trips that integrate with other transportation modes such as public transportation. Similar to the multi-hop ride-sharing, the main bottleneck of multi-modal trip planning is a large number of trip plans and transfers points. In Chapter 4, we proposed to first identify possible trip plans and then compute the optimal transfer point. This idea helps to largely reduce the search space and overcome the efficiency bottleneck. Planning multi-modal trips can apply similar ideas to it. One could first identify which lines to transfer and then select the optimal transfer points. Another possible idea to explore is the hierarchical ranking that was widely applied in existing shortest path algorithms. Public transportation means such as subways usually provide faster and cheaper trips and are preferred by commuters. The algorithm can hierarchically rank

the trip roads and give higher priority to these lines such that the optimal trip can be quickly located.

Efficient and scalable k shortest path algorithms. Finding k shortest paths is an important and fundamental problem in the road networks. For example, drivers prefer to get recommendations on multiple paths to gain higher flexibility and select their preferred routes. However, finding k shortest paths in the road network efficiently is still challenging. Although many indices are proposed to speed up the shortest path query, these indices can only return one shortest path but fail to find k shortest paths. Although [66] extends the PHL algorithm on the k shortest path problem, the algorithm requires a large memory consumption on large networks and thus suffers from poor scalability. Proposing an efficient and scalable k shortest path algorithm is an important research gap to be filled. One possible solution is to extend the hierarchical network partition such as G-tree [77] and accelerate the query time using the pre-computed and stored information between subgraphs in the tree. However, different from one shortest path problem, k shortest path query needs to retrieve distance information for k paths and the value of k is undetermined. It is challenging to design the techniques to pre-compute and store distance information for every graph partition, and to retrieve the distance information so as to reduce the search space in the query phase.

Bibliography

[1] 2020. URL https://www.uber.com/global/en/cities/.

[2] 2016. URL https://www.marsdd.com/news/ride-sharing-the-rise-of-innovative-transportation-services/.

[3] 2020. URL https://www.businessofapps.com/data/lyft-statistics/.

[4] 2020. URL https://www.businessofapps.com/data/uber-statistics/.

[5] 2020. URL https://www.mordorintelligence.com/industry-reports/ridesharing-market.

[6] 2016. URL https://techcrunch.com/2016/05/10/uber-says-that-20-of-its-rides-globally-are-now-on-uber-pool.

[7] Hua Cai, Xi Wang, Peter Adriaens, and Ming Xu. Environmental benefits of taxi ride sharing in beijing. *Energy*, 174:503–508, 2019.

[8] Javier Alonso-Mora, Samitha Samaranayake, Alex Wallar, Emilio Frazzoli, and Daniela Rus. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proceedings of the National Academy of Sciences (PNAS)*, 114(3):462–467, 2017.

[9] 2020. URL https://toddwschneider.com/dashboards/nyc-taxi-ridehailing-uber-lyft-data/.

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

[11] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. *9th DIMACS Implementation Challenge — Shortest Path*, pages 41–72, 2006.

[12] Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal-directed shortest-path queries using precomputed cluster distances. *J3ea*, 14(3):2, 2009.

[13] Sungwon Jung and S. Pramanik. Hiti graph model of topographical road maps in navigation systems. In *International Conference on Data Engineering (ICDE)*, pages 76–84, 1996.

[14] Sungwon Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(5):1029–1046, 2002.

[15] H. Bast, Stefan Funke, and Domagoj Matijevic. TRANSIT: Ultrafast shortest-path queries with linear-time preprocessing. *9th DIMACS Implementation Challenge — Shortest Path*, pages 175–192, 2006.

[16] Julian Arz, Dennis Luxen, and Peter Sanders. Transit node routing reconsidered. In *2013 Symposium on Experimental and Efficient Algorithms*, pages 55–66, 2013.

[17] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. *International Conference on Experimental Algorithms*, pages 319–333, 2008.

[18] Ronald J. Gutman. Reach-based routing: a new approach to shortest path algorithms optimized for road networks. *Algorithm Engineering and Experiments and Analytic Algorithmics and Combinatorics (ALENEX/ANALCO)*, pages 100–111, 2004.

[19] Adi Botea. Ultra-fast optimal pathfinding without runtime search. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 122–127, 2011.

[20] Adi Botea and Daniel Harabor. Path planning with compressed all-pairs shortest paths data. In *International Conference on International Conference on Automated Planning and Scheduling (ICAPS)*, pages 293–297, 2013.

[21] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Conference on Experimental Algorithms (SEA)*, pages 230–241, 2011.

[22] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical hub labelings for shortest paths. In *Annual European conference on Algorithms (ESA)*, pages 24–35, 2012.

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

[24] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Algorithm Engineering and Experiments(ALENEX)*, pages 147–154, 2014.

[25] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. Large scale real-time ridesharing with service guarantee on road networks. *Proceedings of the VLDB Endowment (PVLDB)*, 7(14):2017–2028, 2014.

[26] Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, Jieping Ye, and Ke Xu. A unified approach to route planning for shared mobility. *Proceedings of the VLDB Endowment (PVLDB)*, 11(11):1633–1646, 2018.

[27] Lu Chen, Qilu Zhong, Xiaokui Xiao, Yunjun Gao, Pengfei Jin, and Christian S Jensen. Price-and-time-aware dynamic ridesharing. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1061–1072, 2018.

[28] Shuo Ma, Yu Zheng, and Ouri Wolfson. T-share: A large-scale dynamic taxi ridesharing service. In *IEEE International Conference on Data Engineering (ICDE)*, pages 410–421, 2013.

[29] Raja Subramaniam Thangaraj, Koyel Mukherjee, Gurulingesh Raravi, Asmita Metrewar, Narendra Annamaneni, and Koushik Chattopadhyay. Xhare-a-Ride: A search optimized dynamic ride sharing system with approximation guarantee. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1117–1128, 2017.

[30] Hui Luo, Zhifeng Bao, Farhana Murtaza Choudhury, and J. Shane Culpepper. Dynamic ridesharing in peak travel periods. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, pages 1–1, 2020.

[31] Na Ta, Guoliang Li, Tianyu Zhao, Jianhua Feng, Hanchao Ma, and Zhiguo Gong. An efficient ride-sharing framework for maximizing shared route. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 30(2):219–233, 2017.

[32] Xiaoyi Duan, Cheqing Jin, Xiaoling Wang, Aoying Zhou, and Kun Yue. Real-time personalized taxi-sharing. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 451–465, 2016.

[33] Zhidan Liu, Zengyang Gong, Jiangzhou Li, and Kaishun Wu. Mobility-aware dynamic taxi ridesharing. In *IEEE International Conference on Data Engineering (ICDE)*, 2020.

[34] Yongxin Tong, Libin Wang, Zimu Zhou, Lei Chen, Bowen Du, and Jieping Ye. Dynamic pricing in spatial crowdsourcing: A matching-based approach. In *ACM International Conference on Management of Data (SIGMOD)*, pages 773–788, 2018.

[35] Mohammad Asghari, Dingxiong Deng, Cyrus Shahabi, Ugur Demiryurek, and Yaguang Li. Price-aware real-time ride-sharing at scale: an auction-based approach. In *ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, page 3, 2016.

[36] Libin Zheng, Lei Chen, and Jieping Ye. Order dispatch in price-aware ridesharing. *Proceedings of the VLDB Endowment (PVLDB)*, 11(8):853–865, 2018.

[37] James J Pan, Guoliang Li, and Juntao Hu. Ridesharing: simulator, benchmark, and evaluation. *Proceedings of the VLDB Endowment (PVLDB)*, 12(10):1085–1098, 2019.

[38] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16 (1):87–90, 1958.

[39] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5 (6):345, 1962.

[40] George Bernard Dantzig. *Linear programming and extensions*. Springer, 1963.

[41] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. *Algorithm Engineering*, pages 19–80, 2016.

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

[43] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 156–165, 2005.

[44] Alexandros Efentakis and Dieter Pfoser. Optimizing landmark-based routing and preprocessing. In *ACM SIGSPATIAL International Workshop on Computational Transportation Science*, pages 25–30, 2013.

[45] Andrew V. Goldberg and Renato Fonseca F. Werneck. Computing point-to-point shortest paths from external memory. In *Algorithm Engineering and Experiments and Analytic Algorithmics and Combinatorics (ALENEX/ANALCO)*, pages 26–40, 2005.

[46] Yikai Zhang and Jeffrey Xu Yu. Hub labeling for shortest path counting. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1813–1828, 2020.

[47] Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In *Proceedings of the International Conference on Experimental algorithms (WEA)*, volume 4525, pages 52–65, 2007.

[48] Ishan Jindal, Tony, Qin, Xuewen Chen, Matthew Nokleby, and Jieping Ye. A unified neural network approach for estimating travel time and distance for a taxi trip. *arXiv preprint arXiv:1710.04350*, 2017.

[49] Fatemeh Salehi Rizi, Joerg Schloetterer, and Michael Granitzer. Shortest path distance approximation using deep learning techniques. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 1007–1014, 2018.

[50] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, volume 2016, pages 855–864, 2016.

[51] Jianzhong Qi, Wei Wang, Rui Zhang, and Zhuowei Zhao. A learning based approach to predict shortest-path distances. In *International Conference on Extending Database Technology (EDBT)*, pages 367–370, 2020.

[52] Bin Yang, Chenjuan Guo, and Christian S. Jensen. Travel cost inference from sparse, spatio temporally correlated time series using markov models. *Proceedings of the VLDB Endowment (PVLDB)*, 6(9):769–780, 2013.

[53] Jingyuan Wang, Qian Gu, Junjie Wu, Guannan Liu, and Zhang Xiong. Traffic speed prediction and congestion source exploration: a deep learning method. In *International Conference on Data Mining (ICDM)*, pages 499–508, 2016.

[54] Bei Pan, Ugur Demiryurek, and Cyrus Shahabi. Utilizing real-world transportation data for accurate traffic prediction. In *International Conference on Data Mining (ICDM)*, pages 595–604, 2012.

[55] Hongjian Wang, Yu-Hsuan Kuo, Daniel Kifer, and Zhenhui Li. A simple baseline for travel time estimation using large-scale trip data. In *ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, page 61, 2016.

[56] Dong Wang, Wei Cao, and Junbo Zhang. When will you arrive? estimating travel time based on deep neural networks. In *AAAI Conference on Artificial Intelligence*, pages 2500–2507, 2018.

[57] Zheng Wang, Kun Fu, and Jieping Ye. Learning to estimate the travel time. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 858–866, 2018.

[58] Wuwei Lan, Yanyan Xu, and Bin Zhao. Travel time estimation without road networks: an urban morphological layout representation approach. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1772–1778, 2019.

[59] Tao yang Fu and Wang-Chien Lee. DeepIST: Deep image-based spatio-temporal network for travel time estimation. In *Conference on Information and Knowledge Management (CIKM)*, pages 69–78, 2019.

[60] Haitao Yuan, Guoliang Li, Zhifeng Bao, and Ling Feng. Effective travel time estimation: When historical trajectories over road networks matter. In *ACM International Conference on Management of Data (SIGMOD)*, 2020.

[61] David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28 (2):652–673, 1999.

[62] Víctor M. Jiménez and Andrés Marzal. Computing the k shortest paths: A new algorithm and an experimental comparison. In *Proceedings of the International Workshop on Algorithm Engineering (WAE)*, pages 15–29, 1999.

[63] Víctor M. Jiménez and Andrés Marzal. A lazy version of eppstein's k shortest paths algorithm. In *Proceedings of the International Conference on Experimental and efficient algorithms (WEA)*, pages 179–191, 2003.

[64] Husain Aljazzar and Stefan Leue. K*: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence*, 175(18):2129–2154, 2011.

[65] Jin Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[66] Takuya Akiba, Takanori Hayashi, Nozomi Nori, Yoichi Iwata, and Yuichi Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 2–8, 2015.

[67] Hossain Mahmud, Ashfaq Mahmood Amin, Mohammed Eunus Ali, Tanzima Hashem, and Sarana Nutanong. A group based approach for path queries in road networks. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 367–385, 2013.

[68] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. Batch processing of shortest path queries in road networks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11393, pages 3–16, 2019.

[69] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. Efficient batch processing of shortest path queries in road networks. In *IEEE International Conference on Mobile Data Management (MDM)*, pages 100–105, 2019.

[70] Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. Fast query decomposition for batch shortest path processing in road networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1189–1200, 2020.

[71] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. Progressive top-k nearest neighbors search in large road networks. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1781–1795, 2020.

[72] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM International Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[73] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 802–813, 2003.

[74] Tenindra Nadeeshan Abeywickrama and Muhammad Aamir Cheema. Efficient landmark-based candidate generation for knn queries on road networks. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 425–440, 2017.

[75] Mohammad Kolahdouzan and Cyrus Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 840–851, 2004.

[76] Ken CK Lee, Wang-Chien Lee, Baihua Zheng, and Yuan Tian. ROAD: A new spatial object search framework for road networks. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(3):547–560, 2012.

[77] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. G-tree: An efficient index for knn search on road networks. In *International Conference on Information and Knowledge Management (CIKM)*, pages 39–48, 2013.

[78] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *Proceedings of the VLDB Endowment (PVLDB)*, 9(6):492–503, 2016.

[79] Hanan Samet, Jagan Sankaranarayanan, and Houman Alborzi. Scalable network distance browsing in spatial databases. In *ACM International Conference on Management of Data (SIGMOD)*, pages 43–54. ACM, 2008.

[80] Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, and Yudian Zheng. TOAIN: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks. *Proceedings of the VLDB Endowment (PVLDB)*, 11 (5):594–606, 2018.

[81] Dan He, Sibo Wang, Xiaofang Zhou, and Reynold Cheng. An efficient framework for correctness-aware knn queries on road networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1298–1309, 2019.

[82] Kenneth L Clarkson. Fast algorithms for the all nearest neighbors problem. In *24th Annual Symposium on Foundations of Computer Science*, pages 226–232, 1983.

[83] Harold N Gabow, Jon Louis Bentley, and Robert E Tarjan. Scaling and related techniques for geometry problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

[84] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jing Hu. Gorder: an efficient method for knn join processing. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 756–767, 2004.

[85] Jun Zhang, Nikos Mamoulis, Dimitris Papadias, and Yufei Tao. All-nearest-neighbors queries in spatial databases. In *Scientific and Statistical Database Management (SSDBM)*, pages 297–306, 2004.

[86] Hue-Ling Chen and Ye-In Chang. All-nearest-neighbors finding based on the hilbert curve. *Expert Systems with Applications*, 38(6):7462–7475, 2011.

[87] Christian Böhm and Florian Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowledge and Information Systems*, 6(6):728–749, 2004.

[88] Yun Chen and Jignesh M Patel. Efficient evaluation of all-nearest-neighbor queries. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1056–1065, 2007.

[89] Jagan Sankaranarayanan, Hanan Samet, and Amitabh Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers & Graphics*, 31(2):157–174, 2007.

[90] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. Efficient index-based knn join processing for high-dimensional data. *Information and Software Technology*, 49 (4):332–344, 2007.

[91] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and HV Jagadish. Indexing the distance: An efficient method to knn processing. In *Proceedings of the VLDB Endowment (PVLDB)*, volume 1, pages 421–430, 2001.

[92] Tobias Emrich, Franz Graf, Hans-Peter Kriegel, Matthias Schubert, and Marisa Thoma. Optimizing all-nearest-neighbor queries with trigonometric pruning. In *Scientific and Statistical Database Management (SSDBM)*, pages 501–518. Springer, 2010.

[93] Man Lung Yiu, Nikos Mamoulis, and Dimitris Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(6):820–833, 2005.

[94] Liang Zhu, Yinan Jing, Weiwei Sun, Dingding Mao, and Peng Liu. Voronoi-based aggregate nearest neighbor query processing in road networks. In *ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 518–521, 2010.

[95] Da Yan, Zhou Zhao, and Wilfred Siu Hung Ng. Efficient algorithms for finding optimal meeting point on road networks. *Very Large Data Bases*, 4(11):968–979, 2011.

[96] Bin Yao, Zhongpu Chen, Xiaofeng Gao, Shuo Shang, Shuai Ma, and Minyi Guo. Flexible aggregate nearest neighbor queries in road networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 761–772, 2018.

[97] Tenindra Abeywickrama, Muhammad Aamir Cheema, and Sabine Storandt. Hierarchical graph traversal for aggregate k nearest neighbors search in road networks. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 2–10, 2020.

[98] Fangda Guo, Ye Yuan, Guoren Wang, Lei Chen, Xiang Lian, and Zimeng Wang. Cohesive group nearest neighbor queries over road-social networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 434–445, 2019.

[99] Ke Deng, Xiaofang Zhou, and Heng Tao Shen. Multi-source skyline query processing in road networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 796–805, 2007.

[100] Lei Zou, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. Dynamic skyline queries in large graphs. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 62–78, 2010.

[101] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert. Route skyline queries: A multi-preference path planning approach. In *IEEE 26th International Conference on Data Engineering (ICDE)*, pages 261–272, 2010.

[102] Xuegang Huang and Christian S. Jensen. In-route skyline querying for location-based services. In *Proceedings of the International Conference on Web and Wireless Geographical Information Systems (W2GIS)*, pages 120–135, 2004.

[103] Su Min Jang and Jae Soo Yoo. Processing continuous skyline queries in road networks. In *International Symposium on Computer Science and its Applications*, pages 353–356, 2008.

[104] Yuan-Ko Huang, Chia-Heng Chang, and Chiang Lee. Continuous distance-based skyline queries in road networks. *Information Systems*, 37(7):611–633, 2012.

[105] Xiaoyi Fu, Xiaoye Miao, Jianliang Xu, and Yunjun Gao. Continuous range-based skyline queries in road networks. *World Wide Web (WWW)*, 20(6):1443–1467, 2017.

[106] Ramez A. Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. 1989.

[107] Xiaoye Miao, Yunjun Gao, Su Guo, and Gang Chen. On efficiently answering why-not range-based skyline queries in road networks. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 30(9):1697–1711, 2018.

[108] Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem: models and algorithms. *Annals of Operations Research*, 153(1):29–46, 2007.

[109] Ulrike Ritzinger, Jakob Puchinger, and Richard F. Hartl. A survey on dynamic and stochastic vehicle routing problems. *International Journal of Production Research*, 54(1):215–231, 2016.

[110] Sin C. Ho, W.Y. Szeto, Yong-Hong Kuo, Janny M.Y. Leung, Matthew Petering, and Terence W.H. Tou. A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B-methodological*, 111:395–421, 2018.

[111] Yves Molenbruch, Kris Braekers, and An Caris. Typology and literature review for dial-a-ride problems. *Annals of Operations Research*, 259(1):295–325, 2017.

[112] Jean-Franois Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3):573–586, 2006.

[113] Marco Diana and Maged M. Dessouky. A new regret insertion heuristic for solving large-scale dial-a-ride problems with time windows. *Transportation Research Part B-methodological*, 38(6):539–557, 2004.

[114] Jang-Jei Jaw, Amedeo R. Odoni, Harilaos N. Psaraftis, and Nigel H.M. Wilson. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation Research Part B-methodological*, 20(3):243–257, 1986.

[115] Paolo Toth and Daniele Vigo. Heuristic algorithms for the handicapped persons transportation problem. *Transportation Science*, 31(1):60–71, 1997.

[116] Ka-Io Wong and Michael G.H. Bell. Solution of the dial-a-ride problem with multi-dimensional capacity constraints. *International Transactions in Operational Research*, 13(3):195–208, 2006.

[117] Line Blander Reinhardt, Tommy Clausen, and David Pisinger. Synchronized dial-a-ride transportation of disabled passengers at airports. *European Journal of Operational Research*, 225(1):106–117, 2013.

[118] Ying Luo and Paul Schonfeld. A rejected-reinsertion heuristic for the static dial-a-ride problem. *Transportation Research Part B-methodological*, 41(7):736–755, 2007.

[119] Irina Ioachim, Jacques Desrosiers, Yvan Dumas, Marius M. Solomon, and Daniel Villeneuve. A request clustering algorithm for door-to-door handicapped transportation. *Transportation Science*, 29(1):63–78, 1995.

[120] Suleyman Karabuk. A nested decomposition approach for solving the paratransit vehicle scheduling problem. *Transportation Research Part B-methodological*, 43 (4):448–465, 2009.

[121] Yi Xu, Yongxin Tong, Yexuan Shi, Qian Tao, Ke Xu, and Wei Li. An efficient insertion operator in dynamic ridesharing services. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1022–1033, 2019.

[122] Jean-François Cordeau and Gilbert Laporte. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B-methodological*, 37(6):579–594, 2003.

[123] Julie Paquette, Jean-François Cordeau, Gilbert Laporte, and Marta M.B. Pascoal. Combining multicriteria analysis and tabu search for dial-a-ride problems. *Transportation Research Part B-methodological*, 52:1–16, 2013.

[124] Dominik Kirchler and Roberto Wolfler Calvo. A granular tabu search algorithm for the dial-a-ride problem. *Transportation Research Part B-methodological*, 56: 120–135, 2013.

[125] Roberto Wolfler Calvo and Nora Touati-Moungla. A matheuristic for the dial-a-ride problem. *International Conference on Network Optimization (INOC)*, pages 450–463, 2011.

[126] Paolo Detti, Francesco Papalini, and Garazi Zabalo Manrique de Lara. A multi-depot dial-a-ride problem with heterogeneous vehicles and compatibility constraints in healthcare. *Omega-international Journal of Management Science*, 70: 1–14, 2017.

[127] Santiago Muelas, Antonio LaTorre, and José-María Peña. A variable neighborhood search algorithm for the optimization of a dial-a-ride problem in a large city. *Expert Systems With Applications*, 40(14):5516–5531, 2013.

[128] Santiago Muelas, Antonio LaTorre, and José-María Peña. A distributed vns algorithm for optimizing dial-a-ride problems in large-scale scenarios. *Transportation Research Part C-emerging Technologies*, 54:110–130, 2015.

[129] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

[130] Kris Braekers, An Caris, and Gerrit K. Janssens. Exact and meta-heuristic approach for a general heterogeneous dial-a-ride problem with multiple depots. *Transportation Research Part B-methodological*, 67:166–186, 2014.

[131] Ulrike Ritzinger, Jakob Puchinger, and Richard F. Hartl. Dynamic programming based metaheuristics for the dial-a-ride problem. *Annals of Operations Research*, 236(2):341–358, 2016.

[132] Jiachuan Wang, Peng Cheng, Libin Zheng, Chao Feng, Lei Chen, Xuemin Lin, and Zheng Wang. Demand-aware route planning for shared mobility services. *Proceedings of the VLDB Endowment (PVLDB)*, 13(7):979–991, 2020.

[133] Peng Cheng, Hao Xin, and Lei Chen. Utility-aware ridesharing on road networks. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1197–1210, 2017.

[134] Bin Cao, Louai Alarabi, Mohamed F Mokbel, and Anas Basalamah. Sharek: A scalable dynamic ride sharing system. In *IEEE International Conference on Mobile Data Management (MDM)*, volume 1, pages 4–13, 2015.

[135] Hui Zhao, Mingjun Xiao, Jie Wu, An Liu, and Baoyi An. Reverse-auction-based competitive order assignment for mobile taxi-hailing systems. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 660–677, 2019.

[136] Libin Zheng, Peng Cheng, and Lei Chen. Auction-based order dispatch and pricing in ridesharing. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1034–1045, 2019.

[137] Siddhartha Banerjee, Ramesh Johari, and Carlos Riquelme. Pricing in ride-sharing platforms: A queueing-theoretic approach. In *ACM Conference on Economics and Computation*, pages 639–639, 2015.

[138] M. Keith Chen. Dynamic pricing in a labor market: surge pricing and flexible work on the uber patform. In *ACM Conference on Economics and Computation*, pages 455–455, 2016.

[139] Juan Camilo Castillo, Dan Knoepfle, and Glen Weyl. Surge pricing solves the wild goose chase. In *ACM Conference on Economics and Computation*, pages 241–242, 2017.

[140] Kostas Bimpikis, Ozan Candogan, and Daniela Saban. Spatial pricing in ride-sharing networks. *Operations Research*, 67(3):744–769, 2019.

[141] Mohammad Asghari and Cyrus Shahabi. ADAPT-pricing: a dynamic and predictive technique for pricing to maximize revenue in ridesharing platforms. In *ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 189–198, 2018.

[142] Qiulin Lin, Lei Dengt, Jingzhou Sun, and Minghua Chen. Optimal demand-aware ride-sharing routing. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 2699–2707, 2018.

[143] Qiulin Lin, Wenjie Xu, Minghua Chen, and Xiaojun Lin. A probabilistic approach for demand-aware ride-sharing optimization. In *ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 141–150, 2019.

[144] Chak Fai Yuen, Abhishek Pratap Singh, Sagar Goyal, Sayan Ranu, and Amitabha Bagchi. Beyond shortest paths: route recommendations for ride-sharing. In *The World Wide Web Conference (WWW)*, pages 2258–2269, 2019.

[145] Junbo Zhang, Yu Zheng, and Dekang Qi. Deep spatio-temporal residual networks for citywide crowd flows prediction. In *AAAI Conference on Artificial Intelligence*, pages 1655–1661, 2016.

[146] Kamel Aissat and Ammar Oulamara. Dynamic ridesharing with intermediate locations. In *2014 IEEE Symposium on Computational Intelligence in Vehicles and Transportation Systems (CIVTS)*, pages 36–42, 2014.

[147] Kamel Aissat and Ammar Oulamara. A priori approach of real-time ridesharing problem with intermediate meeting locations. *Journal of Artificial Intelligence and Soft Computing Research*, 4(4):287–299, 2014.

[148] Mitja Stiglic, Niels Agatz, Martin Savelsbergh, and Mirko Gradisar. The benefits of meeting points in ride-sharing systems. *ERIM Report Series Research in Management Erasmus Research Institute of Management*, 2015.

[149] Allan F. Balardino and André G. dos Santos. Heuristic and exact approach for the close enough ridematching problem. In *International Conference on Hybrid Intelligent Systems*, pages 281–293, 2016.

[150] Xin Li, Sangen Hu, Wenbo Fan, and Kai Deng. Modeling an enhanced ridesharing system with meet points and time windows. *PLOS ONE*, 13(5), 2018.

[151] Wenyi Chen, Martijn R. K. Mes, Johannes M. J. Schutten, and J. Quint. A ride-sharing problem with meeting points and return restrictions. *Transportation Science*, 2017.

[152] Meng Zhao, Jiateng Yin, Shi An, Jian Wang, and Dejian Feng. Ridesharing problem with flexible pickup and delivery locations for app-based transportation service: mathematical modeling and decomposition methods. *Journal of Advanced Transportation*, 2018(PT.4):6430950.1–6430950.21, 2018.

[153] Yan Lyu, Victor C. S. Lee, Joseph Kee-Yin Ng, Brian Y. Lim, Kai Liu, and Chao Chen. Flexi-Sharing: A flexible and personalized taxi-sharing system. *IEEE Transactions on Vehicular Technology*, 68(10):9399–9413, 2019.

[154] Paul Czioska, Ronny J. Kutadinata, Aleksandar Trifunovic, Stephan Winter, Monika Sester, and Bernhard Friedrich. Real-world meeting points for shared demand-responsive transportation systems. *Public Transport*, 11(2):341–377, 2019.

[155] Paul Czioska, Dirk C. Mattfeld, and Monika Sester. Gis-based identification and assessment of suitable meeting point locations for ride-sharing. *Transportation Research Procedia*, 22:314–324, 2017.

[156] Paul Czioska, Aleksandar Trifunović, Sophie Dennisen, and Monika Sester. Location- and time-dependent meeting point recommendations for shared interurban rides. *Journal of Location Based Services*, 11:181–203, 2017.

[157] Preeti Goel, Lars Kulik, and Kotagiri Ramamohanarao. Optimal pick up point selection for effective ride sharing. *IEEE Transactions on Big Data*, 3(2):154–168, 2017.

[158] Florian Drews and Dennis Luxen. Multi-hop ride sharing. In *Annual Symposium on Combinatorial Search (SoCS)*, 2013.

[159] Neda Masoud and R Jayakrishnan. A real-time algorithm to solve the peer-to-peer ride-matching problem in a flexible ridesharing system. *Transportation Research Part B: Methodological*, 106:218–236, 2017.

[160] Wesam Herbawi and Michael Weber. Evolutionary multiobjective route planning in dynamic multi-hop ridesharing. In *European Conference on Evolutionary Computation in Combinatorial Optimisation (EvoCOP)*, pages 84–95, 2011.

[161] Yunfei Hou, Xu Li, and Chunming Qiao. TicTac: From transfer-incapable carpooling to transfer-allowed carpooling. In *IEEE Global Communications Conference (GLOBECOM)*, pages 268–273, 2012.

[162] Brian Coltin and Manuela Veloso. Ridesharing with passenger transfers. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3278–3283, 2014.

[163] Wesam Herbawi and Michael Weber. Modeling the multihop ridematching problem with time windows and solving it using genetic algorithms. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, volume 1, pages 89–96, 2012.

[164] San Yeung, Evan Miller, and Sanjay Madria. A flexible real-time ridesharing system considering current road conditions. In *IEEE International Conference on Mobile Data Management (MDM)*, volume 1, pages 186–191, 2016.

[165] 2019. URL https://www.uber.com/us/en/ride/uberpool/.

[166] Zhiming Ding and Ralf Hartmut Guting. Managing moving objects on dynamic transportation networks. In *Scientific and Statistical Database Management (SSDBM)*, pages 287–296. IEEE, 2004.

[167] Yuxiang Zeng, Yongxin Tong, and Lei Chen. Last-mile delivery made practical: An efficient route planning framework with theoretical guarantees. *Proceedings of the VLDB Endowment (PVLDB)*, 13(3):320–333, 2019.

[168] Openstreetmap, 2020. URL https://www.openstreetmap.org/.

[169] Tlc trip record data, 2017. URL https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[170] Gaia open dataset, 2016. URL https://outreach.didichuxing.com/appEn-vue/DataList.

[171] 2006. URL http://www.dis.uniroma1.it/challenge9/.

[172] Ming Zhu, Xiao-Yang Liu, and Xiaodong Wang. An online ride-sharing path-planning strategy for public vehicle systems. *IEEE Transactions on Intelligent Transportation Systems (TITS)*, 20(2):616–627, 2018.

[173] Preeti Goel, Lars Kulik, and Kotagiri Ramamohanarao. Privacy-aware dynamic ride sharing. *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, 2 (1):4, 2016.

[174] Kevin Shaw, Elias Ioup, John Sample, Mahdi Abdelguerfi, and Olivier Tabone. Efficient approximation of spatial network queries using the m-tree with road network embedding. In *Scientific and Statistical Database Management (SSDBM)*, pages 11–11, 2007.

[175] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N Papadopoulos, and Yannis Theodoridis. *R-trees: theory and applications*. Springer Science & Business Media, 2010.

[176] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *Proceedings of the VLDB Endowment (PVLDB)*, 4(4):255–266, 2011.

[177] Nectar cloud, 2020. URL http://nectar.org.au/research-cloud/.

[178] Yan Zhao, Kai Zheng, Yue Cui, Han Su, Feida Zhu, and Xiaofang Zhou. Predictive task assignment in spatial crowdsourcing: a data-driven approach. In *IEEE International Conference on Data Engineering (ICDE)*, 2020.

[179] Ashutosh Singh, Abubakr O Alabbasi, and Vaneet Aggarwal. A distributed model-free algorithm for multi-hop ride-sharing using deep reinforcement learning. *arXiv: Learning*, 2019.

[180] Maytham Safar, Dariush Ibrahimi, and David Taniar. Voronoi-based reverse nearest neighbor query processing on spatial networks. *Multimedia Systems*, 15(5): 295–308, 2009.

[181] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias, and Nikos Mamoulis. Continuous nearest neighbor monitoring in road networks. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 43–54, 2006.

[182] 2019. URL https://www.businessofapps.com/data/uber-statistics/.

[183] Rachel R Weinberger and Joshua Karlin-Resnick. Parking in mixed-use US districts: oversupplied no matter how you slice the pie. *Transportation Research Record: Journal of the Transportation Research Board*, 2537(2537):177–184, 2015.

[184] Peter W Eklund, Steve Kirkby, and Simon Pollitt. A dynamic multi-source dijkstra's algorithm for vehicle routing. In *Australian and New Zealand Conference on Intelligent Information Systems*, pages 329–333, 1996.

[185] Matt Duckham and Lars Kulik. A formal model of obfuscation and negotiation for location privacy. In *International conference on pervasive computing*, pages 152–170. Springer, 2005.

[186] Zhe Xu, Zhixin Li, Qingwen Guan, Dingshui Zhang, Qiang Li, Junxiao Nan, Chunyang Liu, Wei Bian, and Jieping Ye. Large-scale order dispatch in on-demand ride-hailing platforms: A learning and planning approach. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 905–913, 2018.

[187] Xiaohui Bei and Shengyu Zhang. Algorithms for trip-vehicle assignment in ride-sharing. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 3–9, 2018.

[188] Jiarui Jin, Ming Zhou, Weinan Zhang, Minne Li, Zilong Guo, Zhiwei Qin, Yan Jiao, Xiaocheng Tang, Chenxi Wang, Jun Wang, Guobin Wu, and Jieping Ye. CoRide: Joint order dispatching and fleet management for multi-scale ride-hailing platforms. In *Conference on Information and Knowledge Management (CIKM)*, pages 1983–1992, 2019.

[189] Mengjing Chen, Weiran Shen, Pingzhong Tang, and Song Zuo. Dispatching through pricing: modeling ride-sharing and designing dynamic prices. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 165–171, 2019.

[190] Vedant Nanda, Pan Xu, Karthik Abinav Sankararaman, John Dickerson, and Aravind Srinivasan. Balancing the tradeoff between profit and fairness in rideshare platforms during high-demand hours. In *AAAI Conference on Artificial Intelligence*, pages 2210–2217, 2020.