Algorithms for Scalable Dynamic Ride-Pooling

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

von der KIT-Fakultät für Informatik des

Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Moritz Carl Laupichler

aus Neuwied

Tag der mündlichen Prüfung: 16. Juli 2026

1. Referent: Prof. Dr. Peter Sanders
2. Referent: Prof. Dr. Sabine Storandt

To Melissa.
Forever and Always.

Abstract

To reach sustainability goals, urban transportation systems need to shift from polluting, inefficient cars to high-capacity shared public transit. However, transit cannot effectively cover all travel demand, so a transition from cars to transit risks reducing the quality of transportation in some areas. *Ride-pooling* is a flexible alternative to traditional fixed transit that could ease these concerns. In dynamic ride-pooling, travelers make a request for immediate travel to a central dispatcher, stating their origin and destination locations. The dispatcher assigns a taxi-like road vehicle to each request that picks the traveler up at their origin and brings them to their destination. In contrast to traditional taxis, the dispatcher may "pool" independent travelers in the same vehicle if they have matching itineraries. Ride-pooling is both shared and on-demand, which promises better efficiency than cars while maintaining greater flexibility than fixed public transit. As such, it could become a staple of future transportation systems, especially due to the advent of shared autonomous vehicles.

Operating a ride-pooling vehicle fleet requires the dispatcher to effectively assign vehicles to travelers with the goal of optimizing trip quality for riders and minimizing the amount of resources used by the vehicles, e.g. fuel, emissions, or driver costs. In a real-world ride-pooling system, an assignment needs to be computed quickly to allow for interactive routing applications. Recent approaches for the dynamic ride-pooling problem focus on the quality of assignments, but do not scale to the expected future demand for ride-pooling in metropolitan areas. Thus, there is a need for faster, more scalable routing solutions.

In this dissertation, we make several key contributions to advancing routing for dynamic ride-pooling. We base our approach on previous work that showed that efficient shortest-path algorithms for road networks should be at the core of dynamic ride-pooling. We engineer these shortest-path algorithms for different variants of the problem.

Our dispatcher KaRRi is designed for fast dispatching in dynamic ride-pooling with meeting points, where riders walk to a pickup location to reduce the vehicle detour required for the pickup. As the dispatcher can choose the meeting point, this routing problem requires us to solve many-to-many shortest-path problems between all possible meeting points and all existing vehicle stops. KaRRi solves these problems by exploiting the locality of meeting points in the road network for cache-efficient bundled shortest-path queries with instruction-level parallelism and improved pruning. KaRRi is the first dispatcher for ride-pooling with meeting points that has a running time in the order of milliseconds per request, which makes it as fast as previous solutions without meeting points. At the same time, meeting points improve the total vehicle operation time and average rider trip time by about 10% compared to a solution without meeting points. In effect, KaRRi leverages its efficient routing to obtain improved solution quality at the same computational cost.

Furthermore, we propose Mt-KaRRi, a multi-threaded version of KaRRi, with the goal of analyzing ride-pooling on the largest available input data for urban transportation systems. We show that a simple synchronization scheme based on iterative re-assignments allows us to concurrently process requests that come in at a similar time. Experimental results show that this approach loses only about 1% solution quality compared to the sequential solution, but leads to good speedups of up to 53 for 96 threads. With this, we can find ride-pooling

assignments for all 25 million trips in a Los Angeles rush hour in real time. This level of throughput exceeds previous approaches by two orders of magnitude.

We use KaRRi to conduct a study of ride-pooling as an integrated part of metropolitan scale transportation systems. Using a model for traveler mode choice, we evaluate ride-pooling in competition with walking, cars, and public transit. As an improvement over previous studies, the scalability of KaRRi allows us to analyze the effect of huge vehicle fleets and an adoption of ride-pooling by the whole population.

We also explore complex extended scenarios that could profit from the efficiency of KaRRi. As a first step, we consider journeys that allow a single transfer between ride-pooling vehicles. As the main source of added complexity, we now need to find the best transfer locations for pairs of vehicles. We first propose an exact variant of our algorithm KaRRiT that exhaustively searches for the best transfer location by combining engineered one-to-all shortest-path queries and pruning based on problem constraints. As the exact variant is too slow for real-world systems, we also propose a heuristic variant of KaRRiT. This heuristic variant is about two orders of magnitude slower than the no-transfer approach, but its transfer journeys generate tangible benefits for the effectiveness of ride-pooling.

Finally, we consider the integration of ride-pooling and public transit, which is required to make optimal use of the flexibility of ride-pooling and the efficiency of public transit. For this, we combine KaRRi with a framework for public transit journey planning to find multimodal journeys that use both ride-pooling and transit. We identify key differences to traditional multimodal journeys with walking and transit, and show that our solution needs to optimize journeys in a multi-criteria sense by considering ride-pooling cost and rider trip time. We adapt KaRRi to efficiently solve one-to-many ride-pooling queries between a traveler's origin and all public transit stations. A multi-criteria public transit query then uses these ride-pooling access trips to find complete multimodal journeys. Our solution PaRRot computes multimodal journeys with an overhead of less than 2.5ms over a ride-pooling-only query. We show that combined journeys reduce system-wide rider trip times, and that they can improve the total time driven by ride-pooling vehicles. Combined journeys prove particularly important for travelers that previously had no alternative to using a ride-pooling-only trip.

Deutsche Zusammenfassung

Um Nachhaltigkeitsziele zu erreichen, müssen städtische Transportsysteme einen Wandel vom ineffizienten und umweltschädlichen Auto hin zu öffentlichem Nahverkehr mit höherer Kapazität und besserer Effizienz vollziehen. Allerdings kann öffentlicher Nahverkehr nicht den gesamten Reisebedarf abdecken, womit dieser Wandel das Risiko birgt, die Qualität des Personentransports in Teilgebieten einer Stadt zu verschlechtern. *Ride-Pooling* ist eine flexible Alternative zu traditionellem Nahverkehr, welche dieses Risiko minimieren könnte. In dynamischem Ride-Pooling stellen Reisende eine Anfrage mit Startort und Zielort an eine zentralen Verwaltung. Die Verwaltung weist jeder solchen Anfrage ein Taxi-artiges Straßenfahrzeug zu, das den Reisenden an seinem Startort abholt und zum Zielort bringt. Anders als bei traditionellen Taxis, dürfen unabhängige Reisende mit passenden Reiserouten demselben Fahrzeug zugewiesen werden, sodass sich die Reisenden das Fahrzeug teilen ("Pooling"). Ride-Pooling ist ein geteilter Transportmodus, der auf Abruf funktioniert. Damit verspricht es bessere Effizienz als motorisierter Individualverkehr und größere Flexibilität als öffentlicher Nahverkehr. Ride-Pooling könnte ein wichtiger Bestandteil zukünftiger Transportsysteme werden, insbesondere durch die Einführung von autonomen Fahrzeugen.

Um eine Ride-Pooling Fahrzeugflotte effizient zu verwenden, muss der Verwalter Zuweisungen von Reisenden zu Fahrzeugen finden, welche gute Reisequalität für die Reisenden aufweisen und gleichzeitig den Verbrauch von Fahrzeugressourcen wie Treibstoff, Emissionen, oder Fahrergehalt minimieren. In einer praktischen Umsetzung von Ride-Pooling ist es wichtig, diese Zuweisungen schnell zu berechnen, um interaktive Routing-Applikationen zu ermöglichen. Existierende Zuweisungsprozeduren konzentrieren sich üblicherweise auf eine möglichst hohe Qualität der Zuweisungen, aber sind nicht skalierbar genug, um das erwartete zukünftige Niveau an Nachfrage für Ride-Pooling abzudecken. Daher besteht die Notwendigkeit, schnellere Lösungen für das dynamische Ride-Pooling-Problem zu finden.

In dieser Dissertation machen wir wichtige Beiträge zu effizientem Routing von dynamischem Ride-Pooling. Frühere Werke haben erkannt, dass schnelle Kürzeste-Wege-Algorithmen für Straßennetzwerke ein integraler Bestandteil von Routing für dynamisches Ride-Pooling sein können. Basierend auf dieser Einsicht passen wir moderne Kürzeste-Wege-Algorithmen für unterschiedliche Varianten des Ride-Pooling Problems an.

Unser Algorithmus KaRRi löst das dynamische Ride-Pooling Problem mit Meeting Points. In diesem Szenario gehen Reisende zu einem Abholort, der den Umweg für das Ride-Pooling-Fahrzeug reduziert. Da der Algorithmus den Abholort wählen kann, ist es nun nötig, Kürzeste Wege zwischen allen Abholorten und allen existierenden Fahrzeugstopps zu berechnen. KaRRi löst diese Probleme, indem es die Lokalität der Abholorte im Straßennetzwerk für Cache-effiziente, gebündelte Kürzeste-Wege-Suchen ausnutzt, welche Parallelismus auf dem Instruktions-Level sowie besseres Pruning erlauben. KaRRi ist der erste Algorithmus für Ride-Pooling mit Meeting Points, der eine Laufzeit im Bereich von Millisekunden pro Anfrage erreicht. Damit ist KaRRi so schnell wie vorherige Ansätze ohne Meeting Points. Gleichzeitig führen Meeting Points zu einer Reduktion in Reisezeiten und Fahrzeugkosten von 10%.

Wir entwickeln Mt-KaRRi, eine parallele Version von KaRRi auf Basis von Multi-Threading, mit dem Ziel, Ride-Pooling auf den größten verfügbaren Eingabedaten für

städtische Verkehrssysteme zu evaluieren. Wir zeigen, dass ein einfacher Ansatz zur Synchronisation auf Basis von iterativen Neuzuweisungen ausreicht, um mehrere Anfragen parallel zu bearbeiten. Experimentelle Ergebnisse zeigen, dass Mt-KaRRi im Vergleich mit KaRRi nur etwa 1% Qualität verliert, aber eine Beschleunigung von bis zu 53 mit 96 Threads erreicht. Damit können wir in Echtzeit Ride-Pooling-Zuweisungen für alle 25 Millionen Reisenden im Berufsverkehr in Los Angeles finden. Dieses Level an Durchsatz ist zwei Größenordnungen größer als bei bisherigen Ansätzen.

Wir verwende KaRRi, um Ride-Pooling als integrierten Bestandteil des Transportsystems einer Metropole zu analysieren. Mithilfe eines Modells für die Modus-Wahl der Reisenden können wir Ride-Pooling im Wettbewerb mit Gehen, Auto, und öffentlichem Nahverkehr studieren. Die Skalierbarkeit von KaRRi erlaubt uns, den Einfluss von riesigen Fahrzeugflotten oder die Annahme von Ride-Pooling in der gesamten Bevölkerung zu analysieren.

Im zweiten Teil der Dissertation analysieren wir komplexe erweiterte Szenarien, die von der Skalierbarkeit von KaRRi profitieren können. Im ersten Schritt betrachten wir Reisen mit exakt einem Umstieg zwischen Ride-Pooling-Fahrzeugen. Die Wahl eines optimalen Umstiegspunktes für ein Paar von Fahrzeugen stellt die größte Quelle zusätzlicher Komplexität dar. Die exakte Variante unseres Algorithmus KaRRiT löst dieses Problem, indem er optimierte one-to-all Kürzeste-Wege-Suchen mit problemspezifischen Einschränkungen kombiniert. Da diese exakte Variante zu langsam für Produktionssysteme ist, schlagen wir außerdem eine heuristische Variante von KaRRiT vor. Die Berechnungszeit dieser heuristischen Variante ist etwa eine Größenordnung größer als die Zeit zur Berechnung einer Reise ohne Umstiege. Jedoch führen Reisen mit Umstiegen zu merklichen Verbesserungen der Systemeffizienz von Ride-Pooling.

Am Ende dieser Arbeit betrachten wir die Integration von Ride-Pooling und öffentlichem Nahverkehr, welche dazu dient, die guten Qualitäten beider Modi optimal auszunutzen. Dazu kombinieren wir KaRRi mit einem Routing-Algorithmus für Nahverkehr, um multimodale Reisen zu finden, welche sowohl Ride-Pooling als auch Nahverkehr verwenden. Wir identifizieren entscheidende Unterschiede zum traditionellen multimodalen Routing mit Gehen und Nahverkehr, und zeigen, dass unsere Lösung sowohl die Ride-Pooling-Kosten als auch die Ankunftszeit des Reisenden in einem multikriteriellen Sinne optimieren muss. Wir passen KaRRi an, um effizient one-to-many Ride-Pooling Reisen vom Startort eines Reisenden zu allen Stationen des Nahverkehrs zu berechnen. Eine multikriterielle Suche im Nahverkehrsnetz kann diese Zugangsreisen dann verwenden, um vollständige multimodale Reisen zu finden. Unser Algorithmus PaRRot berechnet multimodale Routen mit weniger als 2.5ms zusätzlichem Zeitaufwand gegenüber der Berechnung einer unimodalen Ride-Pooling-Reise. Wir zeigen das kombinierte Reisen die systemweiten Reisezeiten senken, und dass sie die Gesamtfahrtzeit der Ride-Pooling Fahrzeugflotte reduzieren können. Multimodale Reisen mit Ride-Pooling und Nahverkehr sind besonders nützlich für Reisende, welche vorher keine gute Alternative zu einer unimodalen Ride-Pooling-Reise hatten.

Acknowledgements

I thank my advisor Peter Sanders for his guidance over the years and for giving me the opportunity to work on algorithms that might contribute to a more sustainable future. I also thank Sabine Storandt for agreeing to serve as co-reviewer in my dissertation committee.

I am grateful for my current and former coworkers Jannick Borowitz, Adrian Feilhauer, Daniel Funke, Lars Gottesbüren, Ruben Götz, Ragnar Groot Koerkamp, Stefan Hermann, Demian Hespe, Tobias Heuer, Lukas Hübner, Ashlin Iser, Florian Kurpicz, Sebastian Lamm, Hans-Peter Lehmann, Nikolai Maas, Tobias Maier, Niccolò Rigi-Luperti, Jonas Sauer, Matthias Schimek, Dominik Schreiber, Daniel Seemaier, Tim Niklas Uhl, Stefan Walzer, Marvin Williams, Sascha Witt, and Michael Zündorf who have taught me a lot and have made my time as a doctoral student extremely enjoyable. I specifically want to thank Lars Gottesbüren and Tobias Heuer for getting me into algorithm engineering, and Ruben Götz for sharing his sourdough starter with me. I also want to thank Anja Blancani for her patience in dealing with a whole gaggle of doctoral students.

I thank Robin Andre, Jelle Kübler, Kim Kandler, and Peter Vortisch at the Institute for Transportation Studies for our fruitful cooperation and for bearing with an algorithm engineer's naïve view of the real world.

Thanks to my co-authors Robin Andre, Thomas Bläsius, Johannes Breitling, Muhammad Farhan, Adrian Feilhauer, Markus Jung, Kim Kandler, Henning Koehler, Peter Sanders, Peter Vortisch, Jiawen Wang, Qing Wang, and Michael Zündorf for their collaboration on many interesting routing problems.

I thank all of the bright students that I had the pleasure to work with: Johannes Breitling, Luca Buchholz, Zhuojun Cai, Henrik Csöre, Kevin Frisen, Tim Luis Gladbach, Markus Jung, Jordan Körte, Arne Kuchenbecker, Lisann Jill Morlok, Ha Linh Nguyen, Patrick Steil, Louis Tiede, and Max Willich. I specifically thank Johannes Breitling, Ha Linh Nguyen, and Patrick Steil who brought many ideas to the table during their time as student researchers.

Finally, thanks to Jannick Borowitz, Nikolai Maas, Jonas Sauer, Matthias Schimek and Daniel Seemaier for proofreading parts of this dissertation.

Table of Contents

1. Introduction **1**
1.1. Contributions . . . 2
1.2. Outline . . . 4

2. Related Work **5**
2.1. Introduction . . . 5
2.2. Ride-Pooling in the Context of Vehicle Routing . . . 5
2.2.1. Ride-Pooling . . . 5
2.2.2. Vehicle Routing Problem . . . 6
2.2.3. Related Problems . . . 7
2.3. Routing and Scheduling Algorithms for Ride-Pooling . . . 8
2.3.1. Algorithms for the Static Ride-Pooling Routing Problem . . . 9
2.3.2. Algorithms for the Dynamic Ride-Pooling Routing Problem . . . 10
2.3.3. Algorithms for Ride-Matching . . . 13

3. Preliminaries **14**
3.1. Shortest Paths in Road Networks . . . 14
3.1.1. Dijkstra's Shortest Path Algorithm . . . 14
3.1.2. Contraction Hierarchies . . . 15
3.1.3. Bucket Contraction Hierarchy Searches . . . 16
3.1.4. PHAST . . . 16
3.1.5. RPHAST . . . 17
3.2. Dynamic Ride-Pooling Model . . . 17

I. Efficient Dynamic Ride-Pooling **19**

4. Dynamic Ride-Pooling with Meeting Points **20**
4.1. Introduction . . . 20
4.1.1. Related Work . . . 21
4.2. Problem Statement . . . 24
4.3. Preliminaries . . . 25
4.3.1. Many-to-Many Techniques for BCH . . . 25
4.3.2. LOUD . . . 26
4.4. Algorithm Overview . . . 27
4.5. Conceptual Changes for Multiple Pickup and Dropoff Locations . . . 29
4.5.1. Walking Time and Walking to the Destination . . . 29
4.5.2. Vehicles Waiting for Riders . . . 29
4.5.3. Added Vehicle Operation Time and Rider Trip Times . . . 30
4.5.4. Generalization to Other Insertion Types . . . 31
4.6. Ordinary, Ordinary Paired, and Pickup Before Next Stop Insertions . . . 32
4.6.1. Elliptic BCH Searches . . . 32
4.6.2. PD-Distance Searches . . . 33
4.6.3. Enumerating Ordinary, Ordinary Paired, and Pickup Before Next Stop Insertions . . . 33
4.7. Pickup After Last Stop Insertions . . . 34
4.7.1. Last Stop BCH Searches for PALS . . . 34

4.7.2. Collective Last Stop Searches for PALS . . . 36
4.8. Dropoff After Last Stop Insertions . . . 40
4.9. Experiments . . . 41
4.9.1. Bundled Searches . . . 43
4.9.2. Sorted Buckets . . . 45
4.9.3. Collective BCH Searches . . . 46
4.9.4. Comparison with Baseline Dispatcher . . . 47
4.10. Conclusion . . . 49

5. Parallel Batched Dynamic Ride-Pooling 52
5.1. Introduction . . . 52
5.1.1. Related Work . . . 53
5.2. Parallelization of KaRRi . . . 54
5.3. Experiments . . . 55
5.3.1. Setup . . . 55
5.3.2. Effect of t_{batch} on Solution Quality . . . 56
5.3.3. Impact of Batch Interval t_{batch} on the Running Time . . . 57
5.3.4. Weak Scaling . . . 57
5.3.5. Strong Scaling . . . 60
5.4. Conclusion . . . 60

6. Analysis of Ride-Pooling in Large-Scale Transportation Systems with Multiple Modes of Transportation 61
6.1. Introduction . . . 61
6.1.1. Related Work . . . 62
6.2. Mode Choice . . . 63
6.2.1. Choice Model . . . 64
6.2.2. Application of Discrete Choice Model . . . 64
6.3. Input Data and Methodology . . . 64
6.3.1. Ride-Pooling Demand Data . . . 64
6.3.2. Definition of Urban Core and Periphery . . . 67
6.3.3. Performance and Quality Metrics . . . 67
6.4. Experiments . . . 68
6.4.1. Setup . . . 68
6.4.2. Impact of Customer Base . . . 68
6.4.3. Impact of Meeting Points . . . 70
6.4.4. Impact of Fleet . . . 72
6.4.5. Comparison between Urban Core and Periphery . . . 73
6.5. Conclusion . . . 77

II. Transfers and Multimodal Journeys 78

7. Dynamic Ride-Pooling with Transfers 79
7.1. Introduction . . . 79
7.1.1. Related Work . . . 80
7.2. Ride-Pooling Model with Transfers . . . 80
7.2.1. Cost Computation of Single-Transfer Insertions . . . 81

7.3. Algorithm Overview . . . 82
7.3.1. Detour Ellipses for Transfers . . . 82
7.3.2. Types of Transfers . . . 82
7.3.3. Structure of Algorithm . . . 83
7.4. Exact Transfer Points . . . 84
7.4.1. Computing Detour Ellipses On-the-Fly . . . 84
7.4.2. Optimal Ordinary Transfers . . . 85
7.4.3. Optimal After-Last-Stop Transfers . . . 85
7.4.4. Speeding up Enumeration of Insertions . . . 86
7.5. Heuristic Transfer Points . . . 87
7.6. Experimental Evaluation . . . 88
7.6.1. Experimental Setup and Benchmark Instances . . . 88
7.6.2. Analysis of Optimizations for the Exact Algorithm . . . 88
7.6.3. Effect of Heuristic on Running Time . . . 90
7.6.4. Impact of Transfers on Solution Quality . . . 91
7.7. Conclusion . . . 92

8. Multimodal Routing for Ride-Pooling and Public Transportation 93
8.1. Introduction . . . 93
8.1.1. Related Work . . . 94
8.2. Problem Statement . . . 98
8.2.1. Public-Transportation Model . . . 98
8.2.2. Ride-Pooling Model . . . 100
8.2.3. Multimodal Model . . . 100
8.3. Preliminaries . . . 101
8.3.1. RAPTOR . . . 101
8.3.2. Unlimited Transfers . . . 102
8.4. Algorithm Overview . . . 103
8.4.1. Difference to Traditional Feeder Modes . . . 103
8.4.2. Combined Journeys Require More-Criteria Routing . . . 104
8.4.3. Structure of Our Algorithm . . . 106
8.5. Finding Access RP Trips Efficiently . . . 109
8.5.1. Direct Distances from Pickups to Stations . . . 109
8.5.2. Ordinary Access Insertions . . . 110
8.5.3. After-Last-Stop Access Insertions . . . 111
8.6. Experiments . . . 112
8.6.1. Setup and Input Instances . . . 112
8.6.2. Parameter Tuning . . . 113
8.6.3. Fleet Size . . . 115
8.6.4. Meeting Points . . . 118
8.6.5. Impact of Combined Journeys . . . 119
8.6.6. Qualitative Comparison with Lorente et al. [Lor+23] . . . 123
8.7. Conclusion . . . 124

9. Conclusion 125
9.1. Summary of Results . . . 125
9.2. Future Work . . . 126

Appendix **129**

10. Technical Appendix **131**
10.1. Departure Times 131
10.2. Generating Road Networks 131
10.2.1. Generating Vehicle and Pedestrian Networks 131
10.2.2. Mapping Rider Requests onto Generated Networks 133
10.3. Tuning of KaRRi Model Parameters 133
10.3.1. Detour Cost Parameter 133
10.3.2. Trip Time Constraint Parameters 134
10.3.3. Maximum Waiting Time Parameter 135
10.3.4. Walking Cost Parameter 135

Acronyms **139**

List of Algorithms **140**

List of Figures **141**

List of Tables **143**

Publications and Supervised Theses **144**

Bibliography **147**

1 Introduction

Transportation systems in industrialized and developing nations are largely dominated by private cars and other forms of motorized individual transportation [Bel+25; Eur18]. This reliance on cars is in opposition to the UN Sustainable Development Goal 11 ("Make cities and human settlements inclusive, safe, resilient and sustainable") [Uni15], as cars come with numerous negative effects like air pollution, the emission of greenhouse gases, the use of public land for streets and parking, noise pollution, and traffic accidents [Van+23].

In well designed cities, short trips can be made by walking or cycling, but the limited range, and often a lack of good infrastructure, restrict these modes compared to the car. For longer trips, public transit with fixed lines and schedules, such as buses, trams, or trains, are often the only alternative to the car. However, transit can only be operated efficiently where enough people use it. The introduction of cars to previously transit-oriented cities can start a vicious cycle where fewer transit riders lead to cost cutting in transit, which reduces service quality and further decreases the number of transit riders [BMB13; Moh72]. Thus, car dependency has grown, particularly in peripheral areas of cities [NKK16]. In the long term, transit-oriented development can alleviate these patterns, but reaching the required demand in existing cities comes with many financial, political, and logistical obstacles [Ibr+20].

Novel modes of shared transportation that dynamically respond to demand in time and space could fill this gap. *Ride-pooling* is an on-demand mode of transport based on shared taxi-like vehicles. Travelers indicate travel demand in the form of an origin location, destination location, and earliest departure time (usually equal to the time of issuing the request) to a central dispatcher. The dispatcher manages a fleet of vehicles and assigns each request to a vehicle that is tasked with picking the traveler up and bringing them to their destination. In contrast to traditional taxis, the dispatcher can assign independent travelers with matching itineraries to the same vehicle at the same time, so their rides are "pooled".

Proponents of ride-pooling claim that it can provide a convenient and reliable alternative to cars, and that effective pooling can reduce the total vehicle distance traveled on roads [Alo+17; Ota+15; San+14; SC19]. The advent of autonomous vehicles is expected to make on-demand mobility a natural way of travel. In fact, if autonomous vehicles become widely available, ride-pooling is *necessary* to avoid an increase in road traffic volume that could be caused by vehicles driving empty and by an induced demand for car trips [Ful+17; Gar+25; GYB21; MvAvW17]. As a risk, the convenience of ride-pooling may instead lead to a decrease in ridership for the more efficient traditional public transit [Car+23; Rod+18]. A well-balanced transportation system has to combine the advantages of ride-pooling and public transit.

Ride-pooling requires effective short-term coordination of trips, which is a new challenge not faced by traditional modes of transport. The ride-pooling routing problem formalizes this as an algorithmic problem: Given a set of travel requests and vehicles in a road network, the problem asks which requests should be assigned to which vehicles and in what order the requests should be visited to optimize a global cost function, usually some combination of operational costs and rider trip quality. Existing algorithms for the ride-pooling routing problem (see our overview in Section 2.3) are often designed for the study or simulation of ride-pooling and do not match the requirements of future large-scale production systems. For

instance, many approaches fail to address the full extent of dynamic properties of ride-pooling by assuming that traveler requests are known in advance or that travel times in a road network are fixed and implicitly known. Furthermore, there is little work on important multimodal integrations with local non-motorized transport and with public transit. Critically, the majority of existing dispatchers do not consider the scalability of the system and the need for fast response times to allow for interactive applications in a production system.

1.1. Contributions

This dissertation develops new algorithmic solutions for the dynamic ride-pooling problem. Our algorithms use engineered shortest-path speedup techniques to achieve improved throughput and scalability compared to existing approaches. Based on these advancements, we propose efficient algorithms for important variations of the dynamic ride-pooling problem that have previously received little attention due to their computational complexity. To the best of our knowledge, we improve the state-of-the-art in terms of computation time and scalability in all of the problem variants while offering good solution quality.

This thesis is based on two conference papers [BL25; LS24] and one report, due to be submitted for journal publication [Lau+26].

The Most Scalable Dispatcher for Dynamic Ride-Pooling. We develop the efficient ride-pooling dispatcher KaRRi. Based on an earlier approach by Buchhold et al. [BSW21], we identify shortest-path problems that arise in dynamic ride-pooling and rigorously apply a shortest-path speedup technique called bucket contraction hierarchies (BCH). Some of these problems represent a bottleneck in the work by Buchhold et al., which we remove with new BCH pruning techniques. Additionally, we develop a multi-threaded solution called Mt-KaRRi. Experiments show that sequential KaRRi is up to an order of magnitude faster than previous approaches. Furthermore, we experimentally demonstrate that Mt-KaRRi achieves unprecedented scalability without a significant loss in quality. Mt-KaRRi can solve ride-pooling for metropolitan-scale input instances with millions of requests per hour in real time, which makes it one to two orders of magnitude faster than the previous state of the art.

Fast Dynamic Ride-Pooling with Meeting Points. Meeting points describe a fundamental multimodal integration of ride-pooling and local non-motorized transport (e.g. walking). The dispatcher chooses a location in the proximity of the traveler at which the traveler meets the ride-pooling vehicle to be picked up. Although travelers now have to walk a short distance, meeting points have been shown to reduce overheads for door-to-door service and improve both operational costs and the quality of rides [FBA21; GK22; Mou+21]. When processing a request, the choice of the optimal meeting point for every possible vehicle adds an additional degree of freedom to the complexity of the problem. Existing dispatchers for dynamic ride-pooling with meeting points choose the meeting points heuristically and report substantially higher response times compared to a solution without meeting points.

Our dispatcher KaRRi is designed to support meeting points. We propose algorithmic solutions for the choice of meeting points that exploit the fact that meeting points are highly localized in the road network and that the required shortest-path computations for different meeting points share similar search spaces. Thus, we can bundle shortest-path queries, which improves memory locality and allows us to use instruction-level parallelism on modern processor hardware. Additionally, we find a technique to prune the number of eligible rider assignments with meeting points by testing whether one meeting point may be dominated

by another during the shortest-path queries. In an experimental study, we observe that meeting points can lead to savings in both operational cost and rider trip times of about 10%. KaRRi with meeting points is only about three to four times slower than the variant without meeting points. In effect, KaRRi with meeting points achieves improved quality and solves ride-pooling with meeting points in milliseconds per request, which, as far as we know, makes it the only publicly available dispatcher to do so.

Large-Scale Simulation Study. Most experimental studies on ride-pooling consider input instances of limited size due to large simulation times caused by inefficient dispatching algorithms. The largest commonly used input instance is based on records of taxi trips in New York City. Some studies use a large road network of the entire city and consider taxi requests over a long period of time. However, the rate of taxi requests over time is usually much smaller than the suggested rate of ride-pooling requests in an envisioned future transportation system. Thus, existing work does not address ride-pooling in scenarios with consistently high levels of demand.

The scalability of our dispatcher KaRRi allows the study of ride-pooling in these scenarios for the first time. In cooperation with transportation experts, we generate new ride-pooling input instances, the largest of which represents 25 million journeys made in the Los Angeles metro area in a morning rush hour. We run a detailed simulation study that evaluates ride-pooling in metropolitan scale transportation networks. We employ a simple mode choice model that simulates a traveler's choice between taking their own car, walking, using fixed public transportation, or using ride-pooling. We study the effects of the size of the ride-pooling customer base and the number of ride-pooling vehicles in operation. Additionally, we show that ride-pooling is well suited for trips in peripheral areas of cities where it can reduce the dominance of private cars. Our simulation study shows that KaRRi is well suited for studies of ride-pooling at scale and as a part of future large-scale production systems.

Dynamic Ride-Pooling with Transfers. Allowing travelers to transfer between ride-pooling vehicles promises to optimize pooling and provide better trips, especially in situations of high demand where vehicle routes get saturated with riders. However, transfers fundamentally increase the complexity of the ride-pooling problem as there are more possible journeys and transfers can be made at any location in the road network. Existing dispatchers allow transfers only at fixed locations to reduce the number of potential solutions.

We explore KaRRiT, a variant of KaRRi that allows ride-pooling trips with a single transfer. In an exact variant, KaRRiT allows transfers at arbitrary locations in the network. KaRRiT finds an optimal pair of pickup and dropoff vehicles with the optimal transfer location for the request at hand. We employ the (R)PHAST shortest-path algorithm to efficiently compute the distances between existing vehicle stops and all transfer locations. We devise pruning techniques to reduce the number of journeys that need to be considered and apply multi-threading for additional speedups. In addition to the exact variant, we propose a heuristic form of KaRRiT that restricts the location of transfers to heuristically identified "important" vertices in the road network. Experiments show that our engineering is effective, but the exact variant of KaRRiT is likely too slow for production systems with running times of hundreds of milliseconds per request. The heuristic variant is ten times faster, and is a promising candidate for further study of ride-pooling with transfers. Preliminary quality experiments indicate that transfers improve the average vehicle occupancy.

Multimodal Journey Planning for Ride-Pooling and Public Transportation. As a shared mode of transport, ride-pooling can be more efficient and sustainable than private cars. However, fixed public transportation is still much more efficient than ride-pooling in terms of space usage, energy, and cost. Thus, to reduce road traffic volumes and generate opportunities for improved urban development, ride-pooling needs to augment and complement public transportation. We focus on the case of using ride-pooling to take people from their origin locations to transit stations and from transit stations to their destination. We need to be able to efficiently determine the best multimodal journey from origin to destination which requires combined routing of ride-pooling and public transit. This problem is more complex than routing for ride-pooling only, as every journey can begin with a ride-pooling trip to any transit station, so we face a one-to-many ride-pooling problem. Most existing approaches do not plan full journeys or make simplifying assumptions to reduce the problem complexity.

We combine our ride-pooling dispatcher KaRRi and the ULTRA framework for multimodal journey planning. First, we identify key difficulties when using ride-pooling in multimodal journeys that do not arise with other modes like walking. We show that KaRRi can solve these difficulties by adapting the algorithm to the one-to-many ride-pooling problem with pruning based on the best ride-pooling-only journey. We propose a multi-criteria public-transit routing strategy to find optimal combined journeys with respect to a cost function. In an experimental evaluation, we show that our planner PaRRot can find a multimodal journey on a metropolitan size network in 10ms, which is only 2.3 times slower than finding a ride-pooling-only journey and 6.8 times slower than finding a public-transportation-only journey. To the best of our knowledge, PaRRot is the first multimodal journey planner for ride-pooling and public transportation with response times in the order of milliseconds.

1.2. Outline

This dissertation is structured as follows. Chapter 2 reviews the literature on ride-pooling and related problems. In Chapter 3, we explain the shortest-path speedup techniques that are at the core of our algorithms and define the basic model for dynamic ride-pooling used throughout this work. The main body of this thesis is split into two main parts. In Part I, we describe our fundamental dispatcher KaRRi for ride-pooling with meeting points (Chapter 4) and the multi-threaded version Mt-KaRRi (Chapter 5). Additionally, we use Mt-KaRRi to conduct a large-scale analysis of ride-pooling in metropolitan areas (Chapter 6). In Part II, we describe the extension of KaRRi to transfers (Chapter 7) and the integration with public transportation (Chapter 8). Chapter 9 gives concluding remarks and outlines directions for future research. The technical appendix in Chapter 10 contains details on generating input instances and tuning KaRRi's model parameters.

2 Related Work

Summary: Ride-pooling promises to improve shared transportation in current car-dependent transportation systems, and is expected to become a vital mode of transport with increasing access to autonomous vehicles. We discuss the idea of ride-pooling, its proposed merits, and current commercial operations. We explain other vehicle routing problems that are related to ride-pooling. Finally, we survey existing routing algorithms for static and dynamic ride-pooling.

Attribution: The majority of this chapter is original work written by the author of this dissertation for this thesis. Parts of Sections 2.2.1 and 2.3 have overlap with the related work sections of the papers "Fast Many-to-Many Routing for Dynamic Taxi Sharing with Meeting Points" [LS23; LS24], co-authored with Peter Sanders, and "Advancing Dynamic Ride-Pooling Simulation – A Highly Scalable Dispatcher" [Lau+26], co-authored with Robin Andre, Kim Kandler, Peter Sanders, and Peter Vortisch. The related work section of the latter was originally written by Kim Kandler.

2.1. Introduction

This dissertation deals with algorithmic approaches for the dynamic ride-pooling problem and extensions of this problem. Ride-pooling is part of a wide field of research on vehicle routing problems, which contains many similar and related problems. In this chapter, we describe the place of ride-pooling in this larger context and give an overview on existing routing algorithms for the ride-pooling problem.

2.2. Ride-Pooling in the Context of Vehicle Routing

We begin by discussing the idea of ride-pooling, potential benefits for transportation systems, and existing commercial ride-pooling services. Then, we summarize how ride-pooling fits into the broad concept of vehicle routing problems. Finally, we explain related terms that are sometimes conflated with ride-pooling to better delimit our understanding of ride-pooling.

2.2.1. Ride-Pooling

Traditional taxis represent a form of non-shared on-demand transportation. A taxi must serve requests one at a time, i.e., they must always bring the current rider to their destination before picking up the next rider. Therefore, taxis have a low average rider occupancy and are inefficient in terms of operational costs like fuel, vehicle wear, or emissions per rider.

Ride-pooling (RP) (also called *taxi (ride) sharing*, *cab sharing*, *shared taxis*, or *ride-splitting*) is a novel mode of transportation that introduces shared rides into the traditional non-shared model of taxis or ride-hailing services. In ride-pooling, the goal is to find independent travelers with similar itineraries and *pool* their rides in the same vehicle. Pooling may limit the total distance traveled by each vehicle, which can reduce operational costs. Each rider may experience detours for picking up or dropping off other riders, but riders

may be incentivized to use a pooled ride with reduced fares which can be financed by the savings in operational costs.

Ride-pooling has garnered a lot of attention in transportation science, operations research, and economics due to expected future demand caused by the advent of autonomous driverless vehicles (AVs). AVs could reduce the operational costs of ride-pooling and increase acceptance of on-demand mobility services as an alternative to privately owned cars [DR18; FK14; GYB21; MvAvW17]. In fact, ride-pooling may be a required feature of transportation systems with AVs, since non-shared AV travel could significantly increase the amount of traffic and congestion [Ful+17; Gar+25; GYB21; MvAvW17; Rod+18]. The proposed benefits of ride-pooling systems with regard to the quality and efficiency of transportation systems have been studied in numerous field tests [Gar+15; Gil+20; JSM19; KFK21; TW08; Wec+18; Yu+17; Zhu21] and simulation studies [Aga+11; BMN17; FK14; Kue+23; Wil+21; Zwi+22].

Commercial Ride-Pooling Services. The number of commercial operators of ride-pooling services is limited. Some ride-hailing companies allow users to specifically opt-in to pooled rides for trips with a reduced fare. The American ride-hailing company Uber Technologies Inc. offers pooled rides in a program called "UberX Share" [Ube26]. UberX Share allows riders to be pooled with up to two other riders and aims to keep the detour compared to a direct ride to at most eight minutes. The system uses meeting points (i.e., walking to a pickup location or from a dropoff location), and has variants for pooled rides on busy corridors ("route share") pre-booked pooled rides ("scheduled share"). Uber claims a fare reduction of up to 20% compared to a non-pooled ride. Lyft Inc., a different US-based ride-hailing company, used to offer a similar service called "Lyft Shared", but appears to have discontinued this service in 2023 [Wal23]. The Singaporean ride-hailing operator Grab Holdings Inc. offers pooled rides with at most one other rider under the name "GrabShare" [Gra16]. Grab claims that shared rides reduce the fare by up to 30%. The Chinese ride-hailing operator Didi Chuxing Technology Company offers ride-pooling in a feature called "Express Pool".

MOIA GmbH runs a commercial ride-pooling service in the German city of Hamburg [MOI26b]. The service was also available in Hannover, Germany, until 2025 [Spi25]. As opposed to the aforementioned services by ride-hailing operators, there is no option for an exclusive ride and every MOIA ride can be pooled. Like UberX Share, MOIA uses meeting points by means of "virtual transit stops". The company reports operating 280 vehicles in regular service in Hamburg [MOI26a]. MOIA cooperates with local transport authorities and claims that their ride-pooling service complements the public transit system. Kostorz-Weiss et al. [Kos+24] support this claim with an accompanying study and an agent-based simulation based on MOIA ridership data. MOIA is also involved in developing ride-pooling with autonomous vehicles [MOI26c] and reports test runs in Austin, Texas [MOI26a].

2.2.2. Vehicle Routing Problem

Ride-pooling is a specific case of the fundamental *vehicle routing problem (VRP)*. According to Irnich et al. [ITV14], the most general definition of the VRP is the following: "Given a set of transportation requests and a fleet of vehicles", the problem is to "determine a set of vehicle routes to perform all (or some) transportation requests with the given vehicle fleet at minimum cost [...]". The vehicle routing problem is NP-complete as it is a generalization of the traveling salesperson problem (TSP) [GJ02; Lap92].

Due to its broad definition, the VRP can be seen as an umbrella that contains many more specialized problems that naturally arise in the transportation and logistics of people and goods, including ride-pooling. Many surveys categorize these problems and establish a

taxonomy of VRPs [BRN16; EVR09; KP12; TV14]. The ride-pooling problem is considered a *capacitated pickup-and-delivery VRP with time windows* [DS14]. We briefly explain these terms and how they relate to ride-pooling. We base our description on Irnich et al. [ITV14]. More details can be found in the according chapters of Toth and Vigo [TV14].

In the *pickup-and-delivery problem (PDP)*, each transportation request r has a pickup location p_r and a dropoff location d_r in the road network. In the PDP, a request r must be served by a vehicle first traveling to p_r and then to d_r. In the most general form, a vehicle can transport an arbitrary number of requests at the same time and may visit pickup and dropoff locations in any order. Ride-pooling is a PDP because travelers usually have individual origin and destination locations that differ for every request.

In the *capacitated vehicle routing problem (CVRP)*, each transportation request is associated with a weight and each vehicle has a fixed capacity. The CVRP adds the constraint that the total weight of requests simultaneously served by a vehicle cannot exceed the capacity of this vehicle. Traditionally, all vehicles are considered to have the same capacity. Ride-pooling is a CVRP, as every vehicle can only transport a maximum number of people at the same time, with seating capacities of ride-pooling vehicles usually ranging from two to six passengers.

In the *vehicle routing problem with time windows (VRPTW)*, a time window is associated with every transportation request. The VRPTW demands that a request can only be served if a vehicle reaches the location of the request within the time window. When combined with the PDP, each request may define a time window for the pickup and delivery. In the case of ride-pooling, these time windows represent the desired departure time and maximum waiting time at the pickup location as well as a latest arrival time at the dropoff location.

In a *dynamic vehicle routing problem (dynamic VRP)*, the requests are revealed one at a time, e.g. at certain fixed points in time. Thus, plans have to be made with incomplete information about the future and potentially have to be updated when new requests come in. Hybrid problems allow part of the requests to be pre-booked (i.e., known at the start of the operation period) while other requests are revealed dynamically in an online fashion. The dynamic VRP is relevant for ride-pooling as it models spontaneous travel demand.

2.2.3. Related Problems

In this section, we give an overview on vehicle routing problems that are related to ride-pooling to point out similarities and differences in assumptions, models, and solution approaches. Mourad et al. [MPC19] and Shaheen and Cohen [SC19] summarize more related problems.

Ride-Hailing. *Ride-hailing* describes services that operate like traditional taxis, but use technology like mobile internet and positioning services to match travelers and vehicles. Large ride-hailing operators like Uber, Lyft, Grab, or Didi let drivers use their personal cars to fulfill travel requests as part of a growing digital gig economy [JM25]. Waymo and Tesla Robotaxi offer ride-hailing using autonomous driver-less vehicles in some US cities. Though some of these operators offer features for shared rides, they focus on non-shared rides. An increase in ride-hailing use without allowing pooling could lead to higher traffic volumes and more congestion in cities [Gar+25].

Dial-a-Ride. The operation of a ride-pooling system can be considered largely equivalent to the well-studied *Dial-a-Ride problem (DARP)* [CL07; DS14; Ho+18]. DARP describes a capacitated pickup-and-delivery VRP with time windows with the specific connotation of transporting people and not goods. Therefore, as opposed to general vehicle routing problems, solutions of the DARP should guarantee a certain trip quality to travelers. Doerner

and Salazar-González [DS14] name the example that vehicles should not wait for a long time while carrying a passenger. Ride-pooling is understood to use modern technology to facilitate DARP-like operations. The relationship between ride-pooling and DARP is thus similar to the relationship between ride-hailing and taxis. The DARP problem is known to be NP-complete [MZW13; Sav85].

Ride-Matching. In ride-pooling, a vehicles' only purpose is to service riders. In contrast to this, the closely related *ride-matching* or *ride sharing* problem assumes that the driver of each vehicle is a private entity with their own origin, destination and time constraints [Aga+12; Fur+13; HW12]. Thus, the goal of ride-matching is to match non-driving travelers with drivers that have a similar itinerary. Travelers pay a fare to the driver to compensate them for the detour made. Companies like BlaBlaCar, Hitch, and Hovr offer a central platform to match travelers and riders and manage the financial transaction.

Car-Pooling and Van-Pooling. *Car-pooling* can be considered a special case of ride-pooling where a group of requests all share the same destination [SCB24]. This is a typical use case for commuting to a common workplace, but can also represent airport shuttles or a demand-responsive connector (DRC) service that takes travelers to a fixed transit station [Kof04; LQ10]. Routing for car-pooling problems is a much simpler problem than general ride-pooling due to the shared destination location. *Van-pooling* may refer to standard car-pooling using vehicles with at least six passenger seats [SC19], or to a variant where groups of travelers additionally agree to meet at a given location [KO13]. In the latter case, the routing problem is concerned with identifying the correct groups of people and their optimal meeting point.

Other Related Terms. The term *car-sharing* usually refers to flexible, low-cost car rental services in urban areas. The optimization of car-sharing systems is concerned with challenges like fleet sizing and vehicle allocation that differ from the issues of ride-pooling.

Demand-responsive transit (DRT) is a broad term for (parts of) transit systems that do not operate on fixed schedules and lines, but instead respond to actual, observed demand [Dau+24; Häm13]. Though the term DRT is mostly used in the context of augmenting public transit networks, ride-pooling can be considered a kind of DRT.

2.3. Routing and Scheduling Algorithms for Ride-Pooling

In the *ride-pooling routing problem (RPRP)*, a dispatcher manages a fleet of ride-pooling vehicles and assigns traveler requests to vehicles. Each vehicle has a fixed capacity and service time window. Each traveler request specifies an origin location, a destination location, and a desired departure time. Additionally, each request defines constraints like a maximum wait time at the origin location or a latest permissible arrival time at the destination. The dispatcher tries to find assignments of travelers to vehicles that are feasible with respect to the constraints of vehicle capacity, vehicle service time, traveler departure time, and traveler arrival time. The goal is to pick feasible assignments that optimize quality criteria like the total vehicle distance traveled, the number of travelers served, the traveler trip times, etc.

In the static variant of the RPRP, all requests are known in advance and the dispatcher can create a schedule based on complete information. In the dynamic variant of the RPRP, requests are revealed as they arrive at certain points in time. Here, the dispatcher needs to make ad-hoc decisions and incrementally update plans for new requests due to incomplete information about future requests. Hybrid variants assume that some requests are pre-booked

and known in advance, but additional requests are allowed to be made in an online fashion like in the dynamic variant.

In the following, we review previous work on solution approaches for the static and dynamic variants of the RPRP. If appropriate, we also consider work that refers to the DARP (see Section 2.2.3) and not explicitly to the RPRP. For an additional overview of models and solution approaches for the RPRP and related problems we refer to Mourad et al. [MPC19].

2.3.1. Algorithms for the Static Ride-Pooling Routing Problem

The static RPRP has been studied more extensively than the dynamic variant. Although the problem is NP-complete [MZW13; Sav85], working with full information on all rides allows for various approaches that trade off solution quality and computation time.

Exact Solutions. We start by reviewing exact solutions that compute an optimal result at the cost of large running times.

The seminal work by Psaraftis [Psa80] proposes an exact solution of the static DARP with a single vehicle based on dynamic programming. The approach optimizes a linear combination of the total vehicle travel time, the sum of rider waiting times, and the sum of rider in-vehicle times. Trying to optimize a combination of operational costs and traveler trips constitutes an important direction for later work on the problem.

Most exact solutions for the static RPRP formulate a mixed integer program (MIP) that can then be solved by black-box solver software. Cordeau [Cor06] and Hosni et al. [HNA14] define MIPs for the static DARP and add optimizations based on branch-and-cut and Lagrangian decomposition, respectively, to compute solutions faster. These approaches find optimal solutions for instances with up to 10 vehicles or up to 48 requests with running times in the order of hours. Ropke et al. [RCL07] refine the approach by Cordeau [Cor06] to increase its scalability and solve an instance with 96 requests and 8 vehicles in about 70 minutes. Garaix et al. [Gar+11] provide an improved MIP based on a column generation approach. This MIP can be solved in seconds or minutes for instances of hundreds of requests. Alonso-Mora et al. [Alo+17] identify that the aforementioned MIP formulations suffer from bad scalability because the MIPs explicitly model the underlying road network, which ends up comprising most of the MIP variables. The authors suggest an approach that avoids this problem. The algorithm first identifies possible groupings of riders and vehicles on the road network, and then forms a MIP that models only these relationships between (groups of) riders and vehicles. A MIP solver can iteratively improve the best known solution based on this MIP. In an experimental study, the authors evaluate a variant of the algorithm that solves the dynamic RPRP by continuously constructing MIPs for batches of new requests in a rolling-horizon approach. We describe this variant in more detail in Section 2.3.2. Although there is no evaluation of the approach for the static problem, results suggest that it can find an optimal solution for a batch of thousands of requests within minutes.

Metaheuristics. Due to the complexity of vehicle routing problems, including the RPRP, metaheuristics are a popular way to generate good solutions quickly [LRV14]. Cordeau and Laporte [CL03] describe an approach for the DARP based on tabu search. They report running times in the order of hours for instances with hundreds of requests. Parragh and Schmid [PS13] combine a MIP based on column generation with a large neighborhood search heuristic. They are able to solve the instances from Ropke et al. [RCL07] in seconds as opposed to tens of minutes for the original exact solution. In many cases, the approach finds an optimal solution. Lin et al. [Lin+12] propose a solution based on simulated annealing. In

an experimental study on a small input instance (29 requests), they report improvements over traditional taxis.

2.3.2. Algorithms for the Dynamic Ride-Pooling Routing Problem

The dynamic ride-pooling problem cannot be solved optimally like the static variant, since unknown future requests can cause any past decision to become non-optimal in the global sense and assignment decisions cannot be reverted once the ride is over. Therefore, the problem is solved incrementally with the available information at any point in time. Two distinct ways to approach the dynamic RPRP have emerged [BRT14]. First, algorithms can use greedy heuristics that make a feasible online decision for every incoming request, and potentially re-optimize later. Second, exact approaches or metaheuristics for the static RPRP can be adapted to periodically incorporate batches of new requests into the current solution (*rolling horizon*). In this section, we give an overview on the most popular type of greedy heuristic and on approaches that use the rolling-horizon strategy. At the end, we summarize work that compares both ideas. We do not discuss approaches that incorporate stochastic information on future demand and refer to Bektaş et al. [BRT14] for more information.

Insertion Heuristics. *Insertion heuristics* are some of the simplest heuristics for the dynamic RPRP. In an insertion heuristic, the dispatcher evaluates the cost (by some metric) of inserting the pickup and dropoff of an incoming request as new stops in an existing vehicle route without changing any existing stops [Psa80]. The dispatcher greedily chooses the insertion with the smallest possible cost and updates the route of the respective vehicle accordingly before processing the next request. Usually, requests for which no feasible insertion can be found are denied service. Jung et al. [JJP16] name a restricted case of an insertion heuristic where only the vehicle closest to the request's origin is evaluated for an insertion.

Jaw et al. [Jaw+86] propose one of the first insertion heuristics, originally intended as a fast solution for the static DARP. Madsen et al. [MRR95] extend this solution with the express purpose of handling the dynamic DARP. The authors report generating an initial solution for an instance of 300 requests with 24 vehicles in about 10 seconds. Dynamically arriving requests can be handled in less than one second. Hunsaker and Savelsbergh [HS02] contribute a proof that the feasibility of an insertion with respect to rider time window constraints can be determined in time linear to the route length.

More recent works are concerned with the scalability of insertion-based approaches. Huang et al. [Hua+14] develop a dynamic tree data structure to query candidate vehicles for insertions more efficiently. Ma et al. [MZW13; MZW15] propose an insertion heuristic that finds candidate vehicles based on a spatio-temporal index and addresses the issue of finding shortest paths needed for cost estimation and feasibility checks. They report average running times of about 10ms per request. Ota et al. [Ota+15; Ota+17] aim to scale an insertion heuristic to tens of thousands of vehicles. For this, they devise a heuristic for finding the best insertion of a request into a vehicle's route that has a linear running time in the number of stops of the route. Additionally, they explore parallelism to speed up the enumeration of all possible assignments for a request. With this, they achieve running times of less than 1ms per request. Buchhold et al. [BSW21] utilize the constraints of existing passengers to prune advanced shortest-path algorithms. Thus, the algorithm is able to compute the required shortest paths and generate a small set of candidate vehicles that is guaranteed to contain the optimal insertion. Even without parallelization, this approach finds the optimal insertion in less than 1ms per request on average. The approaches of Ma et al. [MZW13; MZW15],

Ota et al. [Ota+15; Ota+17], and Buchhold et al. [BSW21] were evaluated on road networks of similar size with about 10 000 vehicles.

Psaraftis [Psa80] points out that insertion heuristics make local decisions and may miss beneficial (or even feasible) solutions without allowing for re-optimization of previously made decisions. Thus, Horn [Hor02] proposes an insertion heuristic with intermittent local-search style re-optimizations. Results of a simulation study show improved solution quality due to these re-optimizations. However, the dynamic dispatching process has to be paused for up to almost a minute to allow the re-optimization procedure to make changes to vehicle schedules. Attanasio et al. [Att+04] run a parallelized tabu search as a re-optimization procedure immediately after inserting a request. Similarly, Häll et al. [HHL12] allow for generic ruin-and-recreate re-optimizations in a framework for insertion-based dynamic DARP. The re-optimization procedure can be started after inserting a request to continuously update the current solution until the next request arrives. At that point, the algorithm can compute an insertion for the new request based on the best solution seen since the last insertion.

Rolling-Horizon Approaches. The rolling-horizon strategy for the dynamic RPRP calls for periodic re-optimization of a feasible solution, usually using active (already assigned but not finished) requests and new requests that have come in since the last re-optimization procedure [BRT14]. Typically re-optimization is applied every 30 or 60 seconds. Requests for which no assignment can be found in their first round of re-optimization can be postponed to the next round until a timeout is reached based on some fixed maximum waiting time.

We first consider the approach by Alonso-Mora et al. [Alo+17]. As mentioned in Section 2.3.1, this algorithm constructs a mixed integer program (MIP) that models possible groupings of active and new travelers and their assignments to vehicles that were identified in a pre-processing step on the road network. The authors propose using this process as an optimal re-optimization procedure in a rolling-horizon approach. Here, optimal can be understood to mean that the procedure would converge to an optimal solution for the active requests if no more new requests were to come in. In an experimental evaluation, a time window of 30 seconds is used and it is shown that the re-optimization is fast enough for real-time processing, i.e., all requests in a batch (up to 2000 active requests) can be processed in less than 30 seconds. Engelhardt et al. [EDB20] argue that the approach by [Alo+17] does not scale well to larger input instances. They report that the re-optimization procedure is no longer capable of real-time dispatching for an input instance with 3000 vehicles and batches with 5000 active requests. Thus, they propose heuristics to limit the number of groupings of travelers and vehicles that are part of the MIP. In experiments, these heuristics speed up the construction of the MIP by a factor of eight, which makes the whole re-optimization procedure 2.5 times faster and restores the real-time property. A quality comparison shows almost no quality loss compared to the exact solution.

More commonly, metaheuristics are used in the re-optimization step, as they iteratively improve a solution. This allows the dispatcher to terminate the re-optimization at any point (for example once the next time period ends) and proceed with the best solution found so far. Jung et al. [JJP16] evaluate an approach based on hybrid simulated annealing. In their experiments, they use a time window of 60 seconds, which leads to an average of 76 new requests per batch for the largest input instance. Results indicate improved quality over greedy heuristics and running times for the re-optimization procedure of less than 15 seconds per batch (with 600 vehicles). Santos and Xavier [SX15] apply a greedy randomized adaptive search procedure (GRASP) with path relinking to break out of local minima. First, a local search procedure finds a local optimum. Then, in the path relinking step, the algorithm

iteratively resolves differences between the local optimum and a randomly generated solution. The best feasible solution traversed by this process becomes the initial solution for another local search. The authors experimentally show that path relinking leads to a higher number of matched travelers and lower fares per traveler than a previous solution based on GRASP without path relinking [SX13]. Zhan et al. [Zha+21] apply path relinking in the artificial bee colony metaheuristic. Experiments show that this approach significantly outperforms the GRASP approach in terms of number of shared rides and rider fares.

Comparison of Insertion Heuristics and Rolling Horizon. Insertion heuristics usually aim to achieve the best possible running times with acceptable solution quality. In contrast to this, rolling-horizon approaches try to achieve the best possible solution quality with acceptable running times. Research on the modern understanding of dynamic ride-pooling is a rather recent and highly interdisciplinary field of research. As such, no standard framework for the comparison of algorithms has emerged. Most works use their own objective function, input instances, and experimental setups fit for their specific goals. We are aware of only a small number of comparisons between insertion heuristics and rolling-horizon approaches.

Engelhardt et al. [EDB20] compare the solution quality of the rolling-horizon approach by Alonso-Mora et al. [Alo+17], their own heuristic variant of this algorithm, and an insertion heuristic on an input instance with up to 3000 vehicles and 180 000 requests in a day. They analyze the total number of requests served, which includes all requests for which a ride with a fixed maximum wait time and a maximum detour compared to a direct trip was available. Additionally, they evaluate the total distance traveled by RP vehicles relative to the distance that RP riders would travel by private car. Results indicate that the exact rolling-horizon approach and their heuristic rolling-horizon variant serve a similar number of requests. The rolling-horizon approaches are able to serve at least 96% of requests in the largest demand scenario, while the insertion heuristic only serves 91% of requests. In terms of saved travel distance, the exact rolling-horizon approach performs up to 30% better than the heuristic rolling-horizon approach. The insertion heuristic saves 20-30% less travel distance than the heuristic rolling-horizon strategy. The heuristic rolling-horizon approach is reported to be about 2.5 times faster than the exact variant. Unfortunately, the paper does not include the running times of the insertion heuristic.

Ruch et al. [Ruc+21] similarly compare the rolling-horizon strategy by Alonso-Mora et al. [Alo+17] to a set of three insertion heuristics [BM16; FK18; MZW13]. As an additional baseline, the study considers an approach for non-pooled ride-hailing only. All approaches are evaluated on a medium-size instance based on taxi data with 16 439 total requests in one day. The study varies the number of vehicles and compares the total number of vehicle kilometers traveled (VKT) by the RP fleet and the percentage of rides that are pooled with at least one other rider for at least part of their ride. The rolling-horizon approach has the highest ratio of pooled rides across all fleet sizes. Given a sufficient number of vehicles, one of the insertion heuristics starts to behave like ride-hailing with no pooled rides, while the other two heuristics exhibit a minimum share of pooled rides for all fleet sizes due to a stricter preference for pooled rides. However, these two heuristics with a higher preference for pooled rides have the highest VKT, even higher than the no-pooling baseline. In contrast, the more lenient heuristic achieves savings in VKT compared to the baseline in some cases (up to 10% saved). Thus, forced pooling appears to be ineffective. The rolling-horizon approach has consistently lower VKT than the no-pooling baseline, saving up to about 20% of VKT. The authors conclude that ride-pooling may not offer sufficient savings over non-pooled ride-hailing to make up for potential disadvantages like longer rides. However, the small size

of the input instance, and especially the low rate of requests over time in this study leave a research gap in comparing the algorithms on instances with high demand. The study does not compare the running times of the different approaches.

2.3.3. Algorithms for Ride-Matching

Algorithms for the ride-matching problem (see Section 2.2.3) are largely faced with the same challenges as ride-pooling, as they also need to explore all possible assignments of riders to vehicles. However, ride-matching problems can often be solved more efficiently since the fixed itinerary of the driver adds additional constraints that limit the space of feasible solutions.

Solutions for the static ride-matching problem can be found with integer programming [Aga+11; Sti+15] and branch-and-bound algorithms [BFR15], or approximated with metaheuristics such as evolutionary algorithms [HW12]. Approaches for the dynamic variant include locality-constrained greedy matching algorithms [GKR16b; Pel+15] and the application of static solutions in a rolling-horizon strategy [Aga+11; HW12]. As with ride-pooling [BSW21], advanced shortest-path algorithms can exploit problem constraints to efficiently compute vehicle detours and find potential matches [Gei+10]. For overviews on ride-matching, we refer to [Fur+13] and [Aga+12].

3 Preliminaries

Summary: We give an overview on speedup techniques for the computation of shortest paths in road networks, with a focus on hierarchical techniques used in our algorithms. Additionally, we establish our basic model for dynamic ride-pooling.

Attribution: The descriptions of shortest path algorithms in Section 3.1 are based on several papers [BL25; Lau+26; LS24]. The ride-pooling model in Section 3.2 is based on the paper "Fast, Exact and Scalable Dynamic Ridesharing" [BSW21] by Valentin Buchhold, Peter Sanders, and Dorothea Wagner, as well as its adaptation for the paper "Fast Many-to-Many Routing for Dynamic Taxi Sharing with Meeting Points" [LS24] by Peter Sanders and the author of this dissertation. The text was written by the author of this dissertation.

3.1. Shortest Paths in Road Networks

Given a directed graph $G = (V, E)$ and vertices $u, v \in V$, we call a sequence of vertices $p = \langle u = v_0, \dots, v_k = w \rangle$ with $(v_i, v_{i+1}) \in E$ a *path* from u to w. Given an edge weight metric $\ell : E \to \mathbb{R}$, the *length* of the path is $\ell(p) := \sum_{i=0}^{k-1} \ell(v_i, v_{i+1})$. As a special case, $\ell(\langle u \rangle) = 0$. The *shortest path (SP)* between two vertices $u, w \in V$ is a path from u to w of minimal length. For a shortest path p from u to w, we call $\delta(u, w) := \ell(p)$ the *shortest-path distance* (or just *distance*) from u to w.

The *point-to-point* SP problem asks to find the shortest path between two given vertices $s, t \in V$. In the *one-to-all* SP problem, the goal is to find the shortest paths from one source $s \in V$ to every other vertex $v \in V$. In the *one-to-many* SP problem, we aim to find the shortest paths from one source $s \in V$ to every $t \in T$ for a set of targets $T \subseteq V$. In the *many-to-many* SP problem, we are given a set of sources $S \subseteq V$ and a set of targets $T \subseteq V$. The goal is to compute the shortest path from every $s \in S$ to every $t \in T$.

We focus on finding shortest paths in road networks. Road networks have favorable properties like small separators and an inherent hierarchy that are exploited by various speedup techniques. According to [Bas+16], shortest-path algorithms for road networks can be categorized into goal-oriented techniques (e.g. A* [HNR68], Arc-Flags [Hil+09]), separator-based techniques (e.g. Customizable Route Planning [Del+17]), hierarchical techniques (e.g. Reach [Gut04], Contraction Hierarchies [Gei+12]), and bounded-hop techniques (e.g. Hub Labeling [Coh+03], Transit-Node Routing [Bas+07]). We mainly employ hierarchical techniques as they are fast and allow flexible preprocessing and queries.

In the following, we explain some of the shortest-path algorithms used in this work. We focus on finding the shortest-path distances $\delta(u, v)$ between vertices $u, v \in V$ instead of the paths themselves. All algorithms mentioned here can also compute the actual path, though.

3.1.1. Dijkstra's Shortest Path Algorithm

Dijkstra's shortest path algorithm [Dij59] computes the shortest path from a source $s \in V$ to all other vertices in a weighted graph $G = (V, E, \ell)$. We call the execution of Dijkstra's algorithm a *Dijkstra search*.

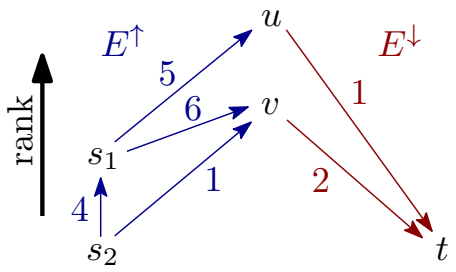


Figure 3.1 Example CH. Edges are annotated with weights. Vertical order of vertices indicates rank. Upward edges are blue and downward edges are red.

The algorithm stores a distance label $\tilde{\delta}(s,v)$ for every $v \in V$. An addressable priority queue Q with $\text{key}(v) = \tilde{\delta}(s,v)$ contains active vertices. Initially, $Q := \{s\}$, $\tilde{\delta}(s,s) := 0$ and $\tilde{\delta}(s,v) := \infty$ for $v \neq s$. The algorithm repeatedly extracts the active vertex with the smallest distance label from Q and *settles* it. To settle $u \in V$, each outgoing edge $(u,v) \in E$ is *relaxed* by trying to improve the distance label $\tilde{\delta}(s,v)$ with $\tilde{\delta}(s,u) + \ell(e)$. If the distance is improved, v is inserted into Q. The algorithm stops when Q becomes empty.

Dijkstra's algorithm settles each vertex exactly once. As soon as a vertex v is settled, $\delta(s,v) = \tilde{\delta}(s,v)$ (label setting). Thus, when solving the point-to-point or one-to-many problems, the algorithm can stop as soon as all target vertices have been settled.

A bidirectional variant of Dijkstra's algorithm can be used to solve the point-to-point problem faster. The bidirectional algorithm runs a Dijkstra search rooted at s in G (forward search) and another Dijkstra search rooted at t in the reverse graph $G_{\text{rev}} = (V, E_{\text{rev}} = \{(v,u) \mid (u,v) \in E\})$ (reverse search). The algorithm interleaves the searches, i.e., it initializes both searches and then proceeds iteratively by settling the next vertex in the forward or reverse search (in alternating fashion or by choosing the smaller distance out of both priority queues). The algorithm can stop as soon as a *meeting vertex* $v \in V$ has been settled by both sides. Then, the shortest path from s to t is via v, i.e., $\delta(s,t) = \tilde{\delta}(s,v) + \tilde{\delta}(v,t)$. The bidirectional variant is faster than the base algorithm for the point-to-point problem, as it reduces the number of vertices settled (size of the *search space*).

3.1.2. Contraction Hierarchies

Contraction Hierarchies (CHs) [Gei+12] are a speed-up technique for shortest path computations that exploits the hierarchical nature of road networks. A CH is constructed in a pre-processing phase. Then, shortest path queries can be computed on the CH using restricted Dijkstra searches.

To construct a CH, all vertices in a road network $G = (V,E)$ are *contracted* in an order that heuristically represents increasing importance of vertices [Gei+12]. The contraction of $v \in V$ temporarily removes v from the graph. To preserve shortest paths, a *shortcut edge* (u,w) is created if $(u,v,w) \in E^2$ is the only shortest path between u and w. Determining an order of contractions that minimizes the number of shortcuts is NP-hard [Bau+10], but heuristics perform well in practice. Let $\text{rk}(v)$ be the rank of $v \in V$ in the order of contractions.

Let E^+ contain all original edges E and all shortcut edges. The graph $G^+ = (V, E^+)$ constitutes the CH. The length $\ell^+(e)$ of a shortcut e is the sum of the lengths of replaced edges and δ^+ is the according distance function. For the query phase, we partition E^+ into *up-edges* $E^\uparrow = \{(u,v) \in E^+ \mid \text{rk}(u) < \text{rk}(v)\}$ and *down-edges* $E^\downarrow = \{(u,v) \in E^+ \mid \text{rk}(u) > \text{rk}(v)\}$. We define an *upwards search graph* $G^\uparrow := (V, E^\uparrow)$ and a *downwards search graph* $G^\downarrow := (V, E^\downarrow)$. The distances $\delta^\uparrow$ and $\delta^\downarrow$ represent δ^+ constrained to $G^\uparrow$ and $G^\downarrow$.

For any two vertices $s, t \in V$, it can be shown that there is a shortest path from s to t that is an *up-down path* in the CH, i.e., consists of only up-edges followed by only down-edges [Gei+12]. A *CH query* from a source $s \in V$ to a target $t \in V$ runs a forward Dijkstra search from s in $G^{\uparrow}$ and a reverse Dijkstra search from t in $G^{\downarrow}$. Whenever the searches meet, they find an up-down-path from s to t, eventually finding a shortest path. The query can stop once the radius of either Dijkstra search exceeds the best previously found distance from s to t. For any vertex $v \in V$, let $G_v^{\uparrow}$ denote the sub-graph of $G^{\uparrow}$ that contains all vertices $V_v^{\uparrow}$ and edges $E_v^{\uparrow}$ that are reachable from v. Similarly, let $G_v^{\downarrow}$ denote the sub-graph of $G^{\downarrow}$ that contains all vertices $V_v^{\downarrow}$ and edges $E_v^{\downarrow}$ from which v can be reached. We call $G_v^{\uparrow}$ and $G_v^{\downarrow}$ the *forward* and *reverse CH search space* of v, respectively.

3.1.3. Bucket Contraction Hierarchy Searches

Bucket Contraction Hierarchy (BCH) searches [Gei+12; Kno+07] use CHs to solve the many-to-one shortest path problem, i.e., they find all shortest path distances from a set of sources $S \subseteq V$ to a target $t \in V$ in a road network $G = (V, E)$.

The algorithm has two phases. In the *selection phase*, the algorithm constructs a *(source) bucket* $B^{\uparrow}(v)$ at each vertex $v \in V$. Conceptually, $B^{\uparrow}(v)$ is a list of entries, each one of which stores the upwards distance from one of the sources to v. For each source $s \in S$, a forward search in $G^{\uparrow}$ is run that adds an entry $(s, \delta^{\uparrow}(s, v))$ to $B^{\uparrow}(v)$ for every settled $v \in V$. In the *query phase*, a reverse search from t in $G^{\downarrow}$ can compute tentative shortest path distances as $\delta^{\uparrow}(s, v) + \delta^{\downarrow}(v, t)$ for every bucket entry $(s, \delta^{\uparrow}(s, v)) \in B^{\uparrow}(v)$ at every settled vertex v.

Consider the example CH depicted in Figure 3.1. A BCH for $S = \{s_1, s_2\}$ would have the buckets $B^{\uparrow}(u) = \langle (s_1, 5), (s_2, 9) \rangle$ and $B^{\uparrow}(v) = \langle (s_1, 6), (s_2, 1)) \rangle$. A reverse search from t traverses $G^{\downarrow}$ and finds up-down paths between s_1, s_2 and t by scanning $B^{\uparrow}(u)$ and $B^{\uparrow}(v)$.

BCH searches can analogously compute the distances from a single source to a set of targets (one-to-many). In that case, we speak of *target buckets* $B^{\downarrow}(v)$ for every $v \in V$. In the case of a many-to-many shortest-path problem with a set of sources $S \subseteq V$ and a set of targets $T \subseteq V$, we can compute either source buckets or target buckets and run a single query from every target or every source, respectively.

The advantage of BCH searches over point-to-point CH queries is that the search space of each source and each target is only traversed once, either to compute bucket entries or to scan bucket entries. However, BCH searches require more memory to store the buckets.

3.1.4. PHAST

PHAST [Del+13] is a speedup technique for the one-to-all shortest-path problem in road networks. PHAST uses a CH as well as a specific memory layout to linearize the process of settling vertices and relaxing edges in memory. The algorithm proceeds in two phases.

First, given a source vertex $s \in V$, PHAST runs a forward search in $G^{\uparrow}$ rooted at s, exploring the entire search space and initializing a distance $d[v] := \delta^{\uparrow}(s, v)$ at every settled vertex v. Every vertex not settled gets $d[v] := \infty$. Second, PHAST settles every $v \in V$ in decreasing order of CH rank, propagating the distances from the forward search through $G^{\downarrow}$ by relaxing incoming edges in $E^{\downarrow}$. This finds shortest up-down paths to every $v \in V$. Scanning vertices in top-down order ensures that $d[u]$ is finished before $d[v]$ for $(u, v) \in E^{\downarrow}$.

PHAST reorders the vertices in $G^{\downarrow}$ according to the order in which vertices are scanned in the top-down sweep. Thus, the write operations to $d[v]$ during the sweep are in sequential order and reads of $d[u]$ for relaxed edges $(u, v) \in E^{\downarrow}$ are likely to hit the cache.

PHAST leverages both instruction parallelism and multi-threading for additional speedups. Instruction parallelism is applicable if there are $k > 1$ sources (a *many-to-all* problem). Then, $d[v]$ is a distance vector of width k where the i-th entry refers to the distance from the i-th source to v. Edge relaxations can use vector instructions to update the distance for all k sources simultaneously. To utilize multi-threading, PHAST groups vertices into CH levels such that vertices within the same level can be settled in parallel (for details, see [Del+13]).

3.1.5. RPHAST

RPHAST [DGW11] is a variant of PHAST for the one-to-many problem. Like BCH, the algorithm has a selection phase and a query phase. In the selection phase, RPHAST computes a subgraph $H^{\downarrow}$ of $G^{\downarrow}$ that contains all vertices from which any $t \in T$ can be reached using only edges in $E^{\downarrow}$. The query phase is akin to a PHAST query rooted at s, but the downward sweep can be restricted to $H^{\downarrow}$ and still find the shortest path to each $t \in T$. For a many-to-many problem, $H^{\downarrow}$ is constructed only once and then used for a query from every source.

RPHAST enjoys most of the benefits of PHAST like an optimized memory layout and straight-forward efficient parallelization. RPHAST outperforms BCHs for one-to-many and many-to-many queries on large road networks [DGW11]. However, in the context of ride-pooling, BCHs are often better suited than RPHAST because ride-pooling regularly defines upper bounds for relevant distances. We can utilize these upper bounds to prune BCH queries, but not RPHAST queries. We quickly explain the intuitive reason for this: The core of a BCH query are a Dijkstra search and the scanning of buckets. Dijkstra searches scan vertices in increasing order of distance and we can often order the entries of every bucket so that the distances during a bucket scan also increase monotonically (see Section 4.3.1). This allows a BCH query to stop processing early if the current distance exceeds the upper bound from the context, potentially saving a lot of work for vertices and bucket entries that cannot be relevant due to the upper bound. In an RPHAST query, vertices and edges in the downward subgraph $H^{\downarrow}$ are not processed in increasing order of distance, but the order is instead optimized for fast memory accesses. Thus, an RPHAST query cannot be pruned using an upper bound on relevant distances like a BCH query.

We mostly use BCH queries for one-to-many and many-to-many shortest-path computations in the context of ride-pooling but we employ RPHAST in cases where there are no good upper bounds on the distances (see Chapter 7).

3.2. Dynamic Ride-Pooling Model

In this section, we define the basic terms of our ride-pooling model. We base this model on [BSW21]. Every subsequent chapter uses these terms and extends or modifies the model as needed for the respective problem variant.

Road Network. We consider a *road network* to be a directed graph $G = (V, E)$ where edges represent road segments and vertices represent intersections. Every edge $e = (v, w) \in E$ has a travel time $\ell(e) = \ell(v, w)$. We denote the *shortest-path distance* (i.e., travel time) from a vertex v to a vertex w by $\delta(v, w)$.

Vehicle, Stop. We manage a fleet F of *vehicles*. Each vehicle $\nu \in F$ has an initial location $\mathrm{loc}_{\mathrm{init}}(\nu)$, a service time interval $[t^{\min}_{\mathrm{serv}}(\nu), t^{\max}_{\mathrm{serv}}(\nu)]$, and a seating capacity $\mathrm{cap}(\nu)$. The current *route* $R(\nu) = \langle s_0(\nu), \ldots, s_{k(\nu)}(\nu) \rangle$ of a vehicle $\nu \in F$ is a sequence of *stops* scheduled for the vehicle. The vehicle's current location is always somewhere between its previous (or

current) stop $s_0(\nu)$ and its next stop $s_1(\nu)$. We update the routes accordingly as vehicles reach stops or are assigned new stops. Thus, $k(\nu) = |R(\nu)| - 1$ is the number of stops that the vehicle yet has to visit. Each stop s is mapped to a vertex $\text{loc}(s) \in V$ in the graph. Given a sufficiently clear context, we write s_i instead of $s_i(\nu)$ and only s_i instead of $\text{loc}(s_i)$.

Request. In our scenario, the dispatcher receives ride requests and immediately assigns them to vehicles. A request $r = (\text{orig}, \text{dest}, t_{\text{req}})$ has an origin location $\text{orig} \in V$, a destination location $\text{dest} \in V$ and a time t_{req} at which the request is issued. In the base model, we do not allow pre-booking, i.e., the request time is also the earliest possible departure time[1].

Insertion. Our dispatchers implement an insertion heuristic (see Section 2.3.2). Given a request $r = (\text{orig}, \text{dest}, t_{\text{req}})$, the dispatcher picks the best insertion according to a cost function and the current state of vehicle routes. We update the route state for the best insertion before moving on to the next request. We formalize an insertion as a tuple (r, ν, i, j) indicating that vehicle ν picks up request r at orig immediately after stop s_i and drops off r at dest immediately after stop s_j with $0 \leq i \leq j \leq k(\nu)$.

Our dispatchers use insertion cost functions that are linear combinations of quality metrics like the added vehicle operation time, rider trip time, and walking time. We define the specific metrics and cost function in each chapter.

Insertion Constraints. We consider a number of constraints on insertions that were originally put forth in [BSW21]. These constraints ensure feasibility and provide a guaranteed minimum service quality to riders. Consider an insertion $\mathcal{I} = (r, \nu, i, j)$. First, $\mathcal{I}$ must not cause a vehicle's occupancy to exceed its seating capacity at any point along its route. If the occupancy of vehicle ν is already equal to its capacity $\text{cap}(\nu)$ on any leg between $s_i(\nu)$ and $s_j(\nu)$, the insertion $\mathcal{I}$ is not feasible. Second, $\mathcal{I}$ must not cause the vehicle to be active past the end of its service time interval $t^{\max}_{\text{serv}}(\nu)$. Third, a rider should not wait for too long to be picked up. Assume that a different rider r' of ν previously accepted a ride with a tentative pickup time T_p. If $\mathcal{I}$ delays the pickup of r' to later than $T_p + t^{\max}_{\text{wait}}$ (with model parameter $t^{\max}_{\text{wait}}$), then $\mathcal{I}$ is considered infeasible. Fourth, the trip times of riders should not become arbitrarily long. Again, consider an existing rider r' who accepted a ride with a tentative trip time T_t. If $\mathcal{I}$ increases the trip time of r' to more than $\alpha \cdot T_t + \beta$ (with model parameters α and β), then $\mathcal{I}$ is infeasible. Thus, every rider is guaranteed a latest possible arrival time relative to the trip time they originally accepted.

Note that these constraints differ slightly from [BSW21] and [LS24]. The original variants use wait time and trip time constraints based on the request time and direct shortest-path travel time from origin to destination. At the same time, travelers are offered rides that break these constraints if no other ride is available. With this, individual riders can restrict vehicles to not allow any detours, making pooling impossible and paralyzing fleets. Thus, we use constraints relative to the pickup time and trip time of the accepted ride, which ensures that some detours are always possible. Furthermore, the original cost function adds penalties to the cost of an insertion if it exceeds a threshold on the wait time or trip time of the new rider. This is used to prefer insertions that do not break the hard constraints of the request itself. Since our hard constraints are relative to the accepted ride instead of the request, we do not require these penalties.

[1] Chapters 7 and 8 use limited pre-booking that is necessary to facilitate transfers between different ride-pooling vehicles or between ride-pooling vehicles and public transit, respectively.

Part I.

Efficient Dynamic Ride-Pooling

4 Dynamic Ride-Pooling with Meeting Points

Summary: In ride-pooling with meeting points, travelers are picked up and dropped off in the proximity of their origin and destination locations, respectively, instead of offering door-to-door service. This can reduce operational costs and rider trip times, but choosing optimal meeting points adds an additional degree of freedom to the complex ride-pooling routing problem.
We develop a ride-pooling dispatcher that leverages algorithmic speedup techniques to allow efficient ride-pooling with meeting points. We analyze emerging many-to-many shortest-path problems and engineer solutions based on bucket contraction hierarchies (BCH). We exploit the locality of meeting points and introduce new pruning techniques that allow us to use BCH to solve a bottleneck of previous solutions.
Without meeting points, our algorithm is up to an order of magnitude faster than the state of the art. With meeting points, our algorithm takes 1-3ms *per request, which is only three to four times slower than the solution without meeting points. At the same time, we observe savings in vehicle operation times and rider trip times by about* 10% *Thus, we show that meeting points can bring substantial quality gain with a limited increase in computation time.*

Attribution: This chapter is based on the conference paper "Fast Many-to-Many Routing for Dynamic Taxi Sharing with Meeting Points" [LS24], published together with Peter Sanders, and its extended version [LS23]. The text of the paper was mainly written by the author of this dissertation, with editing by Peter Sanders. Some terminology and notation is borrowed from Buchhold et al. [BSW21]. The algorithmic ideas were developed by the author of this dissertation and Peter Sanders. The author of this dissertation implemented the code and performed the experimental evaluation.

4.1. Introduction

Current transportation systems are largely based on a combination of individual transport (often with heavy, polluting cars that consume a lot of energy and space) and public transportation that is often slow, inconvenient, and underdeveloped. Recently, *ride-pooling* systems that intelligently control fleets of shared taxi-like vehicles have garnered attention as a promising means of interpolating between the economical and ecological benefits of public transportation and the convenience and flexibility of individually used cars. The traffic engineering community has extensively studied the possible advantages of such systems in a large number of simulation studies [Aga+11; BMN17; FK14; Kue+23; Wil+21; Zwi+22] and real-world field tests [Gar+15; Gil+20; JSM19; KFK21; TW08; Wec+18; Yu+17; Zhu21]. A widespread adaptation of ride-pooling is expected to coincide with an increased demand for sustainable personal transportation [Son+21; Yu+17] and the availability of autonomously piloted vehicles [Bad+21; DR18; FK14; FK18; MvAvW17].

A main issue of current such systems is that the potential for shared rides is usually limited as each additional stop made to pick up or drop off a rider causes delays for other riders. This makes ride-pooling less attractive and makes larger capacity vehicles infeasible.

We focus on the question of how riders can use local transportation (e.g., walking, bicycles or scooters) to reach a pickup or dropoff location (*meeting point*) that causes less delay for a vehicle [AO14; Sti+15], may be shared with other customers [KO13; Sti+15], and may alleviate concerns of privacy for riders [GKR16a]. Meeting points act as a first step towards a hierarchy of personal transportation consisting of local transportation, ride-pooling, and public transit, promising economical and ecological benefits compared to current transportation systems.

However, choosing the right meeting point adds an additional degree of freedom to the dispatching process and increases its complexity. To determine the best assignment of a traveler to a ride-pooling vehicle, a number of many-to-many routing problems between vehicle locations and *all* possible meeting points needs to be solved. Existing work on ride-pooling with meeting points focuses mostly on the potential benefits for travelers and ride-pooling operators, but often fails to address this issue of scalability.

Our starting point is the dynamic ride-pooling dispatcher by Buchhold et al. [BSW21]. It uses one-to-many routing based on bucket contraction hierarchies (BCHs) [Gei+12; Kno+07] to efficiently compute the best feasible assignments of riders to vehicles. We introduce the KaRRi (Karlsruhe Rapid Ridepooling) algorithm that extends the baseline dispatcher with the possibility of performing the pickup and dropoff at meeting points which can be any location close to the rider's origin and destination. The algorithm computes assignments of riders to vehicles and chooses the best meeting points such that the cost of the assignment with respect to a global objective function is minimized. We adapt the baseline dispatcher's objective function to the new scenario with meeting points by incorporating rider trip times and overheads for local travel to and from meeting points. To tackle the issue of scalability with meeting points, we propose novel speedup techniques both for general BCH queries and for the specific case of localized sources or targets. Many of these techniques are also applicable for faster routing in a scenario without meeting points.

Our experimental evaluation uses realistic data sets to evaluate the efficiency of these measures. In a scenario without meeting points, our implementation is several times faster than the state-of-the-art dispatcher [BSW21]. For multiple meeting points, our routing techniques are up to three orders of magnitude faster than a naïve extension of previous techniques. We also give first indications that meeting points can reduce the operating costs of a vehicle fleet and improve rider wait times and trip times at the same time. A closer investigation of possible effects on the transport system can be found in Chapter 6.

4.1.1. Related Work

We summarize related work on ride-pooling dispatchers that compute shortest-path distances on-the-fly and/or consider meeting points. For a general overview on ride-pooling and similar problems, refer to Chapter 2.

Fast On-the-Fly Distance Computation in Ride-Pooling. A lot of work on dynamic ride-pooling focuses on enumerating assignments and assessing their feasibility with respect to the riders' time constraints [HS02; Jaw+86; MRR95]. For this, the dispatcher needs to know the extent of the vehicle detours made to service the new rider. Oftentimes, these detours are assumed to be known a priori [HHL12; Hor02; HS02; Jaw+86; Lot+22; MRR95; Psa80]. To satisfy this assumption one could simply pre-compute shortest-path distances between

every pair of vertices in the road network [Mou+21; Ota+15; Ota+17; San+14]. However, this takes a lot of pre-processing time and a considerable amount of memory. As travel times in a road network frequently change, such an index of distances becomes stale and needs to be re-computed repeatedly. Thus, dispatchers should be able to efficiently compute the required shortest-path distances on-the-fly.

Some recent dispatchers with on-the-fly distance computation employ filtering heuristics (e.g. based on geodesic distances [BMN17; HNA16] or spatial indices [Hua+14; MZW13; MZW15]) to find a small set of candidate assignments to which shortest-path queries are limited. These heuristic dispatchers use varying shortest-path algorithms as a black box, ranging in efficiency from Dijkstra's algorithm [Dij59] to hub labeling [Abr+11].

Buchhold et al. [BSW21] employ a more involved approach by using the time constraints of already assigned riders to prune bucket contraction hierarchy searches [Gei+12; Kno+07], a state-of-the-art one-to-many shortest-path algorithm. This allows the shortest-path algorithm itself to act as a filter of feasible assignments, efficiently computing both a set of candidate vehicles that is guaranteed to contain the best assignment and the required shortest paths. The algorithm is also equipped to work with bucket based searches in customizable contraction hierarchies [DSW16] which allow for fast readjustment of travel times in the road network caused by changing traffic conditions.

Meeting Points in Ride-Pooling. *Meeting points* allow riders to be picked up and dropped off at locations close to their origin and destination, respectively. This requires the rider to walk a small distance but potentially reduces the cost of an assignment. Ride-pooling with meeting points has started garnering attention only recently. Most works that we are aware of focus on the positive effects of meeting points on the operation costs and service quality of ride-pooling systems.

We first summarize a number of results that do not consider the increased complexity that comes with meeting points, but motivate the use of meeting points with observed quality improvements. Gurumurthy and Kockelman [GK22] analyze ride-pooling when using fixed meeting points in the road network. They find that the average vehicle occupancy increases when using meeting points. Vehicle miles traveled decrease by up to about 20% when using meeting points that are spaced 0.5 miles apart. Lotze et al. [Lot+22] explore *stop pooling*, a restricted form of meeting points, for dynamic ride-pooling. Though the authors' experiments use a simple model with uniformly distributed requests and Euclidean distances, they find compelling evidence that stop pooling can help to break the traditional trade-off between vehicle operation times and rider trip times. Li et al. [Li+26] use meeting points in a batch-processing approach for dynamic ride-pooling in which all requests that are issued in a short period are processed together as one batch. The authors propose that the dispatcher should be allowed to bump a request to the next batch which may give the traveler enough time to walk to a good meeting point and give vehicles more time to make their way to a meeting point. A reinforcement learning approach is used to learn in which cases to bump a traveler based on the accessibility of the traveler's origin location, the current routes of vehicles, and the traveler's willingness to walk. Experiments suggest that the combined regiment of batch processing with bumping and meeting points improves the quality of rider trips and reduces operational costs.

For the rest of this section, we focus on the limited amount of work that gives insights into the added computational complexity of ride-pooling with meeting points.

Fielbaum et al. [FBA21] and Mounesan et al. [Mou+21] independently extend a rolling-horizon approach by Alonso-Mora et al. [Alo+17] with meeting points. In rolling-horizon

approaches, requests that are issued within a fixed time window are processed as a batch by constructing and solving a mixed integer program (MIP) (cf. Section 2.3.2). Both works find that meeting points increase the complexity of this MIP substantially. Both works evaluate their approach on a road network of Manhattan. On this small network, Fielbaum et al. [FBA21] report running times of ten hours per "iteration", which we assume to mean iteration of the rolling-horizon approach. With a time window of one minute, this would mean their approach is 600 times slower than real time, which does not work for a production system. Mounesan et al. [Mou+21] use heuristics during the construction of the MIP and set time limits for the MIP solver to avoid exceedingly long running times. With this, they report average running times of one second per request. However, due to the rolling-horizon approach, every traveler has to wait until the optimization phase for the next batch is done, so the response time is much higher (at least 23 seconds with the reported time limits). In conclusion, existing rolling-horizon approaches are too slow for real-time dispatching on large input instances when considering meeting points.

We are aware of only one work that considers the scalability of ride-pooling with meeting points on realistic metropolitan-scale road networks. For this purpose, Mounesan et al. [Mou+21] develop the insertion heuristic STaRS+ next to their aforementioned rolling-horizon solution. STaRS+ processes each request as soon as it is issued and greedily chooses the best vehicle and meeting points according to the current vehicle routes. A distance cache based on a partition of the road network is designed to facilitate pre-computing and storing all-pairs shortest-path distances within reasonable memory limits for larger road networks (4.8GiB for New York City). Using the distance cache, STaRS+ is shown to be able to answer requests on the road network of all five boroughs of New York City in about 10ms with a fleet of 10 000 vehicles, which represents an increase by a factor of about six compared to using no meeting points. The authors find a reduction in the trip time and total vehicle miles traveled compared to a scenario without meeting points. The experiments do not evaluate the query times for the new distance cache, inhibiting a comparison to related shortest-path techniques like transit node routing [Bas+07] or customizable route planning [Del+17].

Though STaRS+ addresses the same problem as our work, there are some important differences in both the model details and the solution approach. Firstly, STaRS+ assumes a fixed hard limit to each rider's wait time and trip time. Since many vehicles can then be excluded from consideration for most requests, this speeds up the dispatching process. However, the hard time limits may cause requests to be rejected entirely. Meanwhile, our approach tries to service every request which necessitates a more careful consideration of a larger set of candidate vehicles. Secondly, STaRS+ chooses the best meeting points for a given request heuristically while we explicitly develop methods that consider all combinations of potential pickup and dropoff locations and find the best one. Thirdly, STaRS+ uses a pre-computed static distance cache, whereas a focus of this work is the computation of shortest paths on-the-fly during the dispatching process. This has the advantage of being more dynamic: As travel times in road networks frequently change, e.g. due to congestion, the underlying data structures for on-the-fly shortest-path queries (e.g. a contraction hierarchy) can be updated in seconds while a distance cache has to be reconstructed from scratch. The authors of STaRS+ report a running time in the order of hours for this, which is too slow for updates in a production system.

Meeting Points in Ride-Matching. Beyond this limited amount of work on meeting points in ride-pooling, several publications have studied meeting points in the closely related ride-matching problem, in which the driver of each vehicle has has their own origin, destination,

and time constraints (see Section 2.2.3). As ride-matching is computationally simpler due to the added constraints for vehicles, we only summarize the positive effects of meeting points that could generalize to the ride-pooling problem.

Goel et al. [GKR16a; GKR16b] and Aïvodji et al. [Aïv+16] show that meeting points allow privacy-aware ride-matching where the driver is not able to obtain e.g. the home address of a rider. Aissat and Oulamara [AO14], Stiglic et al. [Sti+15], and Baudru et al. [BSB26] show that meeting points can improve the chance of finding a match and reduce the driver detours.

4.2. Problem Statement

This section describes the formal foundations for the dynamic ride-pooling problem with meeting points. We extend the base model described in Section 3.2 to include meeting points.

Meeting Points. We assume that riders can reach meeting points in their vicinity using local transportation such as walking or cycling. We represent the paths accessible to this mode of transportation in a road network $G_{\text{psg}} = (V_{\text{psg}}, E_{\text{psg}})$. Let $\delta_{\text{psg}}(u, v)$ denote the *rider shortest-path distance* between vertices u and v in G_{psg}.

Consider a request $r = (\text{orig}, \text{dest}, t_{\text{req}})$. We use a set of pickup locations (or *pickups*) $P_\rho(r)$ that contains all eligible vertices that the rider can reach in G_{psg} from orig within a time radius ρ, i.e.,

$$P_\rho(r) := \{p \in V_{\text{psg}} \cap V \mid \delta_{\text{psg}}(\text{orig}, p) \leq \rho\}.$$

Similarly, our set of dropoff locations (or *dropoffs*) is defined as

$$D_\rho(r) := \{p \in V_{\text{psg}} \cap V \mid \delta_{\text{psg}}(d, \text{dest} \leq \rho\}.$$

We collectively refer to the pickups and dropoffs of r as the *meeting points* of r. Let $N^p_\rho(r) := |P_\rho(r)|$ and $N^d_\rho(r) := |D_\rho(r)|$. We call a pair of pickup and dropoff a *PD-pair* and the distance between a pickup and a dropoff a *PD-distance*. The radius ρ is a model parameter. For the sake of simplicity, in this chapter, we use the same ρ for every request, but the model also permits varying ρ with each request.

If the context allows it, we omit r in the notation of the terms defined above. Further, for $p \in P_\rho$ and $d \in D_\rho$, we use $\delta_{\text{psg}}(p)$ and $\delta_{\text{psg}}(d)$ as shorthand for $\delta_{\text{psg}}(\text{orig}, p)$ and $\delta_{\text{psg}}(d, \text{dest})$. For ease of notation in the rest of this chapter, we use the term "walking" to mean any mode of local transportation for riders.

Insertion. In dynamic ride-pooling with meeting points, our dispatcher needs to choose the best combination of vehicle and meeting points for each traveler. Therefore, we extend the concept of an insertion to include the chosen pickup and dropoff location. Insertions with meeting points have the form (r, p, d, ν, i, j), indicating that vehicle ν picks up request r at pickup location $p \in P_\rho$ immediately after stop s_i and drops off r at dropoff location $d \in D_\rho$ immediately after stop s_j with $0 \leq i \leq j \leq k(\nu)$.

Cost Function. The cost of an insertion is a linear combination of added trip time for new and existing travelers, added vehicle operation time, and walking time. It takes the form

$$c(\mathcal{I}) = t_{\text{trip}}(\mathcal{I}) + t^+_{\text{trip}}(\mathcal{I}) + w_{\text{detour}} \cdot t_{\text{detour}}(\mathcal{I}) + w_{\text{walk}} \cdot t_{\text{walk}}(\mathcal{I}).$$

In the following, we explain the intuition behind each of the terms in the cost function. The exact definitions can be found in Section 4.5. The trip time $t_{\text{trip}}(\mathcal{I})$ is the total trip time for the new rider according to $\mathcal{I}$ under the assumption that there will be no future detours of ν that increase this time. To account for the increased trip times of other riders that may be caused by the detours for $\mathcal{I}$, we add the term $t^{+}_{\text{trip}}(\mathcal{I})$, which is the sum of increases in trip time for all affected riders. The added vehicle operation time $t_{\text{detour}}(\mathcal{I})$ represents the additional time that vehicle ν spends driving or waiting to facilitate the insertion. The model parameter w_{detour} determines the importance of the detour time in comparison to the rider trip time. To account for the discomfort of walking, we introduce the cost term $w_{\text{walk}} \cdot t_{\text{walk}}(\mathcal{I})$, which considers the time $t_{\text{walk}}(\mathcal{I})$ spent walking to the pickup location or from the dropoff location to the destination. The model parameter w_{walk} defines the impact of walking discomfort. The form of the cost function differs slightly from [LS24] as we use a weight parameter for the detour term instead of the trip terms. Our cost function can also easily be replaced with a more intricate function that takes into account other metrics such as monetary cost or emissions.

If an insertion breaks any constraint described in Section 3.2, we set $c(\mathcal{I}) = \infty$.

4.3. Preliminaries

In this section, we introduce techniques that speed up many-to-many shortest-path queries and outline the dispatcher that our solution is based on.

4.3.1. Many-to-Many Techniques for BCH

Our dispatcher solves one-to-many and many-to-many shortest-path problems in road networks that arise from ride-pooling with meeting points. We employ bucket contraction hierarchies (BCH) (see Section 3.1.3) to compute shortest-path distances. Here, we describe additional augmentations of BCH that are useful in our dispatcher.

Sorted Buckets. Assume that we run a BCH query from a source $s \in v$ to a set of targets $T \subseteq V$ in a graph $G = (V, E)$. Further, assume that the context of the BCH query defines an upper bound $\delta_{\max}$ on relevant distances, so target $t \in T$ is not relevant for the context if $\delta(s, t) > \delta_{\max}$. Then, the number of bucket entries that need to be scanned for the BCH query can be reduced by sorting the entries of each bucket by distance. Consider a sorted bucket $B(v) = \langle (t_1, \delta^{\downarrow}(v, t_1)), \ldots, (t_k, \delta^{\downarrow}(v, t_k)) \rangle$ of vertex $v \in V$ with $t_1, \ldots, t_k \in T$ and $\delta^{\downarrow}(v, t_i) \leq \delta^{\downarrow}(v, t_{i+1})$ for $i = 1, \ldots, k-1$. When the BCH query settles v with a distance of $\delta^{\uparrow}(s, v)$, it scans the entries in $B(v)$ in order. The query can stop scanning the bucket as soon as an entry $(t_i, \delta^{\downarrow}(v, t_i))$ is found for which $\delta^{\uparrow}(s, v) + \delta^{\downarrow}(v, t_i) > \delta_{\max}$. Since the bucket is sorted, all subsequent entries $(t_j, \delta^{\downarrow}(v, t_j))$ for $j = i+1, \ldots, k$ also lead to distances greater than $\delta_{\max}$ that cannot be relevant for the context of the query. Thus, the query does not need to scan them.

The idea of sorted BCH buckets is used in ULTRA [Bau+23], a framework for journey planning in public transportation with unlimited walking. Here, BCHs are used to compute the walking distances from the origin location orig of a traveler to all public transit stations. In this context, it is never optimal to walk to a station that is further away from orig than the traveler's destination dest. Thus, the distance $\delta(\text{orig}, \text{dest})$ acts as the required upper bound $\delta_{\max}$, and sorted buckets can be used for speedups.

Bundled Searches. Dijkstra-based shortest path algorithms for multiple sources can be *bundled* s.t. the searches for k sources are advanced simultaneously. A bundled search maintains k tentative distance labels at each vertex. When the search relaxes an edge $(u, v) \in E$, it tries to update all k distance labels at v.

A bundled relaxation can be more cache efficient than k individual relaxations as all k distances are stored in consecutive memory. However, the relaxation of $(u, v) \in E$ may perform unproductive work if not all k searches have reached u yet. Thus, bundling is effective if all k searches relax largely the same edges. The value of k is a tuning parameter.

Bundled searches were first introduced for Dijkstra searches used for the computation of arc-flags under the name *centralized searches* [Hil+09]. Since then, bundled searches have been used in a number of Dijkstra-based many-to-many shortest path algorithms [BD10; Del+13; Del+17; DGW11; Yan10], including point-to-point queries in CHs [BSW19].

SIMD Parallelism in Bundled Searches. Bundled searches can be sped up using single-instruction multiple-data (SIMD) parallelism [BSW19]. Modern CPUs provide special vector registers and instructions that can store and manipulate multiple data items simultaneously. We can vectorize the computations needed during edge relaxations s.t. k computations are performed at the same time using a single vector instruction. SIMD instructions can substantially speed up bundled searches [BSW19].

4.3.2. LOUD

Our algorithm is based on the dynamic ride-pooling dispatching algorithm *LOUD* [BSW21].

Given a fleet of vehicles and routes, the online algorithm matches incoming ride-pooling requests to vehicles. For each request, a feasible insertion of the request's origin orig and destination dest into a vehicle's route is found s.t. the detour of the vehicle is minimized.

Elliptic Pruning. To compute the costs of possible insertions, the algorithm requires the distances between existing vehicle stops and orig and dest. LOUD computes these distances using BCHs (see Section 3.1.3) with bucket entries for each vehicle stop and queries run from orig and dest. We refer to these BCH searches as *elliptic BCH searches* as they utilize a pruning technique for the buckets called *elliptic pruning*, which we explain in the following.

Each insertion is subject to the constraints that we describe in Section 3.2. The wait time and trip time constraints of riders already assigned to a vehicle $\nu \in F$ define latest permissible arrival times $t^{\max}_{\text{arr}}(s_i)$ for every stop $s_i \in R(\nu)$. If ν arrives at s_i later than $t^{\max}_{\text{arr}}(s_i)$, then a constraint of some existing rider will be broken.

LOUD defines a *(detour) leeway* $\lambda(s_i, s_{i+1}) = t^{\max}_{\text{arr}}(s_{i+1}) - t^{\min}_{\text{dep}}(s_i)$ between each pair of consecutive stops $(s_i, s_{i+1}) \in R(\nu)$. An insertion is infeasible if it requires a detour between s_i and s_{i+1} that exceeds $\lambda(s_i, s_{i+1})$, since it is guaranteed to break a constraint. Thus, the leeway $\lambda(s_i, s_{i+1})$ determines a set of vertices $\mathcal{E}(s_i) = \{v \in V \mid \delta(s_i, v) + \delta(v, s_{i+1}) \leq \lambda(s_i, s_{i+1}))\}$ at which a pickup or dropoff may be made between s_i and s_{i+1} without breaking a hard constraint. Since the inequality in the definition of $\mathcal{E}(s_i)$ defines an ellipse in euclidean geometry, $\mathcal{E}(s_i)$ is called the *(detour) ellipse* of s_i.

LOUD uses ellipses to prune BCH buckets. The authors show that it suffices to generate bucket entries for s_i and s_{i+1} only at a small subset of vertices within $\mathcal{E}(s_i)$ to find the relevant distances for all feasible insertions. Elliptic pruning comes with two advantages. First, there are vastly fewer bucket entries. Since the algorithm spends most of its time on BCH searches, which in turn spend most of their time scanning bucket entries, this significantly reduces the overall running time of the algorithm. Second, elliptic pruning acts

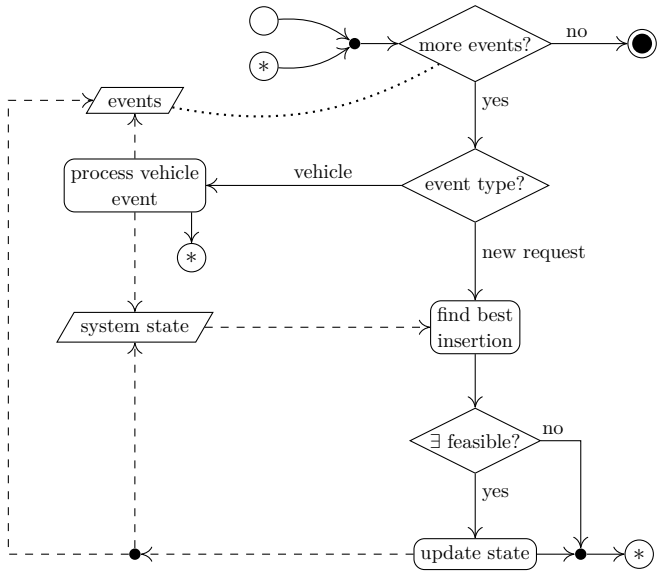


Figure 4.1 Solid arrows indicate control flow. Dashed arrows indicate data reads and writes. Dotted lines indicate information for decisions. The node ⊛ signifies a loop to the next event.

as a filter for candidate insertions. Since no bucket entries are generated for vehicles that cannot lead to feasible insertions, the BCH searches will often not find any distances for these vehicles and their stops. When testing possible insertions, the algorithm needs to only consider the stops and vehicles for which valid distances have been found, which minimizes the overhead for testing infeasible insertions. These advantages of elliptic pruning became a guiding idea for the design of additional pruning techniques of our dispatcher KaRRi.

Last Stop Distances. LOUD also allows the insertion of the origin and/or destination after the last stop $s_{k(\nu)}$ of a vehicle's route. Here, elliptic pruning is not applicable since there is no next stop $s_{k(\nu)+1}$ and the leeway is unbounded after the last stop. Instead, LOUD uses reverse Dijkstra queries in the road network rooted at orig or dest to find the distances from last stops to orig or dest. These Dijkstra queries, particularly for the destination of a request, constitute a significant part (at least 60% and up to more than 90%) of the total runtime of LOUD.

4.4. Algorithm Overview

We introduce the KaRRi (Karlsruhe Rapid Ride-Pooling) algorithm that efficiently answers ride-pooling requests with multiple meeting points using fast many-to-many routing.

Event Simulation. The KaRRi algorithm is embedded into a simple event simulation. Every event can update the system state, which consists of vehicle routes and the data structures for shortest-path queries. We illustrate the workflow in Figure 4.1.

For vehicles, there are three types of events: startup, reached stop, and shutdown. Initially, all vehicles are inactive. For every vehicle $\nu \in F$, the simulation contains a startup event at the beginning of the vehicle's service time $t^{\min}_{\text{serv}}(\nu)$ and a shutdown event at the end of the service time $t^{\max}_{\text{serv}}(\nu)$. When a startup event occurs, the vehicle becomes active and a single

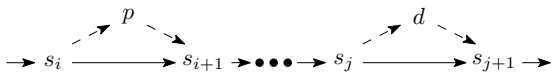
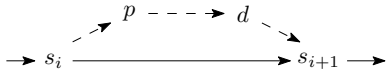
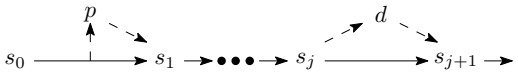
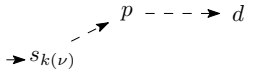
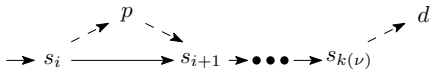
Ordinary	$0 < i < j < k(\nu)$	
Ordinary Paired (OP)	$0 < i = j < k(\nu)$	
Pickup Before Next Stop (PBNS)	$0 = i \leq j < k(\nu)$	
Pickup After Last Stop (PALS)	$0 \leq i = j = k(\nu)$	
Dropoff After Last Stop (DALS)	$0 \leq i < j = k(\nu)$	

Figure 4.2 Insertion types. Shows characterization of each type based on the pickup and dropoff insertion points i and j of an insertion $\mathcal{I} = (r, p, d, \nu, i, j)$. Illustrations depict the current route of ν (solid arrows) with stops $s \in R(\nu)$ as well as the detours to and from p and d (dashed lines).

stop is inserted into the vehicle's route at its initial location. The algorithm inserts bucket entries for the new stop into the BCH data structures to allow shortest path queries for this stop. As long as the vehicle has exactly one stop, it idles there until it is assigned a rider, prompting it to move to new stops. When a shutdown event occurs, the vehicle becomes inactive, and its route becomes empty. Due to the service time constraint for insertions, the vehicle is always idle when the shutdown event is reached.

For every vehicle ν with a non-empty route, the event simulation contains a "reached stop" event at the time when $s_1(\nu)$ will be reached. Triggering this event means that the vehicle has finished its current leg, which causes the stop $s_0(\nu)$ to become obsolete. The algorithm removes $s_0(\nu)$ from the route $R(\nu)$ and deletes its bucket entries. If more than one stop remains in $R(\nu)$, a new "reached stop" event is added for the new $s_1(\nu)$ according to the current vehicle schedule.

For every request r, the simulation contains a "new request" event at the request's time of issue t_{req}. When this event is reached, the algorithm computes the best insertion based on the current vehicle routes. We explain this process in more detail at the end of this section. If at least one feasible insertion is found, KaRRi performs the insertion by adding stops to the vehicle's route, either in between existing stops for a detour or appended to the end of the route. For any new stops, the algorithm generates bucket entries in the BCH data structures. If the affected vehicle ν was previously idle or the next stop $s_1(\nu)$ has changed, the vehicle immediately starts moving from its current location to the new $s_1(\nu)$ on the shortest path. Additionally, a new "reached stop" event is added for the new $s_1(\nu)$.

Finding the Best Insertion. To compute the best insertion for a new request r, the algorithm first finds the possible meeting points in a walking radius ρ around the origin and destination using bounded Dijkstra searches. Then, the algorithm evaluates all insertions in the order of types illustrated in Figure 4.2. For each insertion, KaRRi computes the cost according to the cost function (see Section 4.2). The insertion with the smallest cost $\mathcal{I}^*$ is repeatedly updated and eventually returned.

Since we consider sets of possible meeting points, the number of potential insertions

becomes the main challenge of the algorithm. In particular, we face the issue of computing the shortest paths between existing vehicle stops and each meeting point. For this, we do not employ inexact filtering heuristics to reduce the number of necessary shortest-path computations. Instead, we find the insertion with optimal cost by applying many-to-many routing techniques to all combinations of vehicle stops and meeting points. We engineer these routing techniques to act as filters of feasible insertions themselves by discarding sub-optimal candidates as early as possible during our searches using a variety of pruning methods.

In the next section, we describe conceptual changes that come with meeting points and detail how the cost of an insertion is computed. Afterwards, we detail our routing methods for each insertion type and associated shortest-path query.

4.5. Conceptual Changes for Multiple Pickup and Dropoff Locations

We observe that introducing meeting points requires a careful consideration of their effects on vehicle detours and rider trips. Here, we describe these effects in detail and establish the formal foundation of our cost function (see Section 4.2).

We focus on the case of ordinary insertions that insert the pickup and dropoff along an existing vehicle route (see Figure 4.2) to avoid bloated definitions. We explain how to generalize the definitions to other types of insertions at the end of this section.

4.5.1. Walking Time and Walking to the Destination

The walking time of an insertion $\mathcal{I} = (r, p, d, \nu, i, j)$ is $t_{\text{walk}}(\mathcal{I}) := \delta_{\text{psg}}(p) + \delta_{\text{psg}}(d)$.

We also allow a rider r to walk from their origin to their destination without ever boarding a vehicle. This requires a manner of pseudo-insertion $\mathcal{I}_{\text{psg}}$ where the rider is matched to no vehicle at all. Then, the cost of $\mathcal{I}_{\text{psg}}$ depends only on the walking distance $t_{\text{walk}}(\mathcal{I}_{\text{psg}}) = t_{\text{trip}}(\mathcal{I}_{\text{psg}}) = \delta_{\text{psg}}(\text{orig}, \text{dest})$. The pseudo-insertion affects no vehicle operation times or trip times of other riders. In effect, the total cost is $c(\mathcal{I}_{\text{psg}}) = (1 + w_{\text{walk}}) \cdot \delta_{\text{psg}}(\text{orig}, \text{dest})$. We explicitly allow pseudo-insertions for any distance $\delta_{\text{psg}}(\text{orig}, \text{dest})$, i.e., the distance does not have to be found within the radius ρ around orig or dest. Instead, we use a CH query in the rider road network to find $\delta_{\text{psg}}(\text{orig}, \text{dest})$. The cost $c(\mathcal{I}_{\text{psg}})$ can serve as a first upper bound $\hat{c}$ on the cost of any insertion.

4.5.2. Vehicles Waiting for Riders

In dynamic ride-pooling without meeting points, a rider $r = (\text{orig}, \text{dest}, t_{\text{req}})$ always waits to be picked up by the vehicle at orig. The request is issued at time t_{req} and the vehicle ν matched to the request can start making its way to the pickup location at the earliest at t_{req}. This means that only the rider can wait for the vehicle, not the other way around. Upon arriving at orig, the vehicle can immediately depart (ignoring the time to pick up the rider). Thus, a vehicle's schedule is fully characterized by the departure time $t^{\min}_{\text{dep}}(s_i)$ at each stop (which is equal to the arrival time $t^{\min}_{\text{arr}}(s_i)$). The shortest-path distances between stops can then be inferred as $\delta(s_i, s_{i+1}) = t^{\min}_{\text{dep}}(s_{i+1}) - t^{\min}_{\text{dep}}(s_i)$.

In the presence of meeting points, vehicle schedules become more complex. Consider an insertion $\mathcal{I} = (r, p, d, \nu, i, j)$ with $p \neq \text{orig}$. Both the vehicle and the rider have to travel to p starting at $t^{\min}_{\text{dep}}(s_i)$ and t_{req}, respectively. Consequently, the vehicle can arrive at p earlier than the rider, precisely if $t^{\min}_{\text{dep}}(s_i) + \delta(s_i, p) < t_{\text{req}} + \delta_{\text{psg}}(p)$. In that case, the vehicle needs to wait for the rider at p for a time $w(s_a)$. Thus, the actual departure time at a pickup is

the maximum of the earliest possible departure time of the vehicle and that of the rider:

$$t_{\text{dep}}(\mathcal{I}) = \max\{t^{\min}_{\text{dep}}(s_i) + \delta(s_i, p), t_{\text{req}} + \delta_{\text{psg}}(p)\}.$$

We later use $t_{\text{dep}}(\mathcal{I})$ in the definitions of the vehicle detour, rider wait time and rider trip time needed for the cost function (see Section 4.2). In particular, the wait times of a vehicle or of a rider contribute to the vehicle operation times and rider trip times.

Whereas in the traditional model $t^{\min}_{\text{arr}}(s) = t^{\min}_{\text{dep}}(s)$ for every stop s, we have to now explicitly store $t^{\min}_{\text{arr}}(s)$ and $t^{\min}_{\text{dep}}(s)$. We can then derive the vehicle wait times as $w(s_i) = t^{\min}_{\text{dep}}(s_i) - t^{\min}_{\text{arr}}(s_i)$ and the distance between a pair of consecutive stops as $\delta(s_i, s_{i+1}) = t^{\min}_{\text{arr}}(s_{i+1}) - t^{\min}_{\text{dep}}(s_i)$.

4.5.3. Added Vehicle Operation Time and Rider Trip Times

In the following, we define the added vehicle operation time $t_{\text{detour}}(\mathcal{I})$, the trip time for the new rider $t_{\text{trip}}(\mathcal{I})$, as well as the sum of added trip times for existing riders $t^{+}_{\text{trip}}(\mathcal{I})$ for an insertion $\mathcal{I} = (r, p, d, \nu, i, j)$.

Detours. We can express all changes to the vehicle operation time and rider trip times in terms of the detours made by ν to perform an additional pickup at p and dropoff at d.

▶ **Definition 1.** *The* full pickup (dropoff) detour *for an insertion $\mathcal{I} = (r, p, d, \nu, i, j)$ is the detour that results from the vehicle ν first driving to p (d) after stop s_i (s_j) instead of driving to s_{i+1} (s_{j+1}) directly.*
Formally, we define the full pickup detour $\blacktriangle_p(\mathcal{I})$ and the full dropoff detour $\blacktriangle_d(\mathcal{I})$ as

$$\blacktriangle_p(\mathcal{I}) := t_{dep}(\mathcal{I}) - t^{min}_{dep}(s_i) + \delta(p, s_{i+1}) - \delta(s_i, s_{i+1})$$
$$\blacktriangle_d(\mathcal{I}) := \delta(s_j, d) + \delta(d, s_{j+1}) - \delta(s_j, s_{j+1})$$

Intuitively, $\mathcal{I}$ causes the vehicle operation time to increase by the sum of these full detours. However, existing vehicle wait times at later stops of ν (see Section 4.5.2) can act as buffers that reduce the added vehicle operation time. Assume that ν has to wait for a rider at a stop s_a with $i < a \leq j$. Then, the insertion $\mathcal{I}$ will cause ν to arrive at s_a with a delay of $\blacktriangle_p(\mathcal{I})$. However, ν would have spent a time $w(s_a)$ waiting at s_a anyways. This time is now spent making (part of) the detour $\blacktriangle_p(\mathcal{I})$. Thus, the arrival at s_{a+1} is only delayed by $\blacktriangle_p(\mathcal{I}) - w(s_a)$. We call this a *residual detour*.

▶ **Definition 2.** *The* residual detour $\triangle_a(\mathcal{I})$ *for an insertion $\mathcal{I} = (r, p, d, \nu, i, j)$ at stop $s_a \in R(\nu)$ describes how much later the vehicle ν will arrive at stop s_a after the insertion is performed. We define it inductively as*

$$\triangle_{a+1}(\mathcal{I}) := \begin{cases} 0 & \text{if } a < i \\ \blacktriangle_p(\mathcal{I}) & \text{if } a = i \\ \max\{\triangle_j(\mathcal{I}) - w(s_j), 0\} + \blacktriangle_d(\mathcal{I}) & \text{if } a = j \\ \max\{\triangle_a(\mathcal{I}) - w(s_a), 0\} & \text{otherw.} \end{cases}$$

Added Vehicle Operation Time, Trip Times. Residual detours allow us to define the added vehicle operation times as well as the trip times of both the new rider and existing riders.

The added operation time of ν is the delay of the arrival of ν at its last scheduled stop. This is simply the residual detour at the last stop.

▶ **Definition 3.** *The added vehicle operation time $t_{detour}(\mathcal{I})$ caused by an insertion $\mathcal{I} = (r, p, d, \nu, i, j)$ is defined as*

$$t_{detour}(\mathcal{I}) := \triangle_{k(\nu)}(\mathcal{I})$$

Further, residual detours define the new arrival times $t^{\min\prime}_{\text{arr}}$ and departure times $t^{\min\prime}_{\text{dep}}$ after performing $\mathcal{I}$ as

$$t^{\min\prime}_{\text{arr}}(s_a, \mathcal{I}) = t^{\min}_{\text{arr}}(s_a) + \triangle_a(\mathcal{I}) \text{ and}$$
$$t^{\min\prime}_{\text{dep}}(s_a, \mathcal{I}) = \max\{t^{\min\prime}_{\text{arr}}(s_a), t^{\min}_{\text{dep}}(s_a)\}.$$

With this, we can calculate the new rider's total trip time.

▶ **Definition 4.** *The trip time $t_{trip}(\mathcal{I})$ for an insertion $\mathcal{I} = (r, p, d, \nu, i, j)$ is defined as*

$$t_{trip}(\mathcal{I}) := t^{min\prime}_{dep}(s_j, \mathcal{I}) + \delta(s_j, d) + \delta_{psg}(d) - t_{req}(r)$$

Each existing rider experiences an added trip time depending on where they are dropped off. The delay of each rider's arrival at their dropoff stop is the residual detour at that stop.

▶ **Definition 5.** *Let $N^d_\rho(s)$ be the number of dropoffs currently scheduled to be performed at stop $s \in R(\nu)$ for a vehicle $\nu \in F$. The combined added trip time for existing riders $t^+_{trip}(\mathcal{I})$ caused by an insertion $\mathcal{I} = (r, p, d, \nu, i, j)$ is defined as*

$$t^+_{trip}(\mathcal{I}) := \sum_{a=i+1}^{k(\nu)} N^d_\rho(s_a) \cdot \triangle_a(\mathcal{I})$$

4.5.4. Generalization to Other Insertion Types

The definitions given above can be generalized to insertions that are not ordinary (see Figure 4.2) as follows. We only state which definitions need to be changed and all definitions not mentioned here still apply as specified above.

Ordinary Paired. In an *ordinary paired* insertion ($0 < i = j < k(\nu)$), the vehicle performs the pickup and the dropoff after stop s_i before moving on to stop s_j. In this case, there is one full detour covering the pickup and the dropoff, which can be defined as:

$$\blacktriangle_{pd}(\mathcal{I}) := t_{\text{dep}}(\mathcal{I}) - t^{\min}_{\text{dep}}(s_i) + \delta(p, d) + \delta(d, s_{i+1}) - \delta(s_i, s_{i+1}).$$

Then, the inductive definition of the residual detour changes to

$$\triangle_{a+1}(\mathcal{I}) := \begin{cases} 0 & \text{if } a < i \\ \blacktriangle_{pd}(\mathcal{I}) & \text{if } a = i \\ \max\{\triangle_a(\mathcal{I}) - w(s_a), 0\} & \text{otherw.} \end{cases}$$

Pickup Before Next Stop. If the new rider is picked up before the vehicle reaches its next stop according to the current schedule ($0 = i \leq j < k(\nu)$), the vehicle makes a detour from its current location $\text{loc}(\nu)$ to p before moving on to s_1 (or to d if $i = j$). The departure time at p depends on $\text{loc}(\nu)$ with

$$t_{\text{dep}}(\mathcal{I}) = \max\{t_{\text{req}} + \delta(\text{loc}(\nu), p), t_{\text{req}} + \delta_{\text{psg}}(p)\}.$$

In the case of $0 = i = j$, the full detour and residual detour are defined like for ordinary paired insertions.

Pickup After Last Stop. If both the pickup and dropoff are performed after the current last stop ($0 \leq i = j = k(\nu)$), there are no residual detours and there is no added trip time for existing riders. The added vehicle operation time is $t_{\text{detour}}(\mathcal{I}) := t_{\text{dep}}(\mathcal{I}) - t^{\min}_{\text{dep}}(s_{k(\nu)}) + \delta(p, d)$.

Dropoff After Last Stop. If the pickup is performed before the current last stop and the dropoff is performed after the current last stop ($0 \leq i < j = k(\nu)$), then the definitions of full pickup detour and residual detours remain the same (including the changes for pickup before next stop if $i = 0$). However, the added vehicle operation time needs to account for the leg from the last stop to the dropoff with $t_{\text{detour}}(\mathcal{I}) := \triangle_{k(\nu)}(\mathcal{I}) + \delta(s_{k(\nu)}, d)$.

4.6. Ordinary, Ordinary Paired, and Pickup Before Next Stop Insertions

This section is concerned with *ordinary*, *ordinary paired*, and *pickup-before-next-stop* insertions (see Figure 4.2). In all three insertion types mentioned, both the pickup p and the dropoff d are inserted between two existing stops of the route $R(\nu)$. To compute the cost of any insertion of one of these types, we need to know the distances between existing vehicle stops and the meeting points. BCH searches with elliptic pruning (see Section 4.3.2) have been shown to efficiently compute these distances [BSW21]. We call these *elliptic BCH searches*. Additionally, cost calculations for paired insertions require the PD-distance $\delta(p, d)$. In this section, we explain how we extend the required distance queries for multiple meeting points.

4.6.1. Elliptic BCH Searches

We can extend elliptic BCH queries to multiple meeting points by simply repeating the queries for each meeting point. Even with elliptic pruning, this can lead to impractical running times for large numbers of meeting points, though. Therefore, we supplement elliptic BCH searches with two techniques for better scalability to larger numbers of meeting points. We describe these techniques only for pickups but they work analogously for dropoffs.

Elliptic BCH Searches with Sorted Buckets. First, we propose using *sorted buckets* to reduce the number of bucket entries scanned by BCH queries. We explain the principle only for source buckets but it can be analogously applied for target buckets.

Recall that the constraints for existing riders of a vehicle ν define a leeway $\lambda(s_i, s_{i+1})$ for the detour between any two consecutive vehicle stops s_i and s_{i+1} of ν (see Section 4.3.2). When an elliptic BCH search to a pickup p scans a source bucket entry $(s, \delta^{\uparrow}(s_i, v)) \in B^{\uparrow}(v)$, the tentative distance $\delta^{\uparrow}(s_i, v) + \delta^{\downarrow}(v, p)$ can only lead to an insertion that holds all hard constraints if $\delta^{\uparrow}(s_i, v) + \delta^{\downarrow}(v, p) \leq \lambda(s_i, s_{i+1})$. This means the entry is only relevant for p if $\delta^{\downarrow}(v, p) \leq \lambda(s_i, s_{i+1}) - \delta^{\uparrow}(s_i, v)$. We call $\lambda_{\text{res}}(s_i, v) := \lambda(s_i, s_{i+1}) - \delta^{\uparrow}(s_i, v)$ the *remaining leeway* of a source bucket entry $(s_i, \delta^{\uparrow}(s_i, v))$.

We sort the entries of each source bucket at each vertex v by their remaining leeway in decreasing order. Then, an elliptic BCH search to a pickup p can stop scanning the entries at v once an entry $(s, \delta^{\uparrow}(s, v))$ is scanned for which $\delta^{\downarrow}(v, p) > \lambda_{\text{res}}(s, v)$. In this case, we have $\delta^{\downarrow}(v, p) > \lambda_{\text{res}}(s, v) \geq \lambda_{\text{res}}(s', v)$ for any subsequent entry $(s', \delta^{\uparrow}(s', v))$, so the remaining entries cannot lead to any insertions that adhere to the hard constraints. Maintaining the order of each bucket comprises a time overhead when inserting bucket entries. However, since bucket sizes are small, this overhead is limited. Note that sorted buckets can also be applied in the case of only a single pickup.

Bundled Elliptic BCH Searches. Second, we employ *bundled elliptic BCH searches* that exploit the locality of pickups.

Like any bundled search, a bundled elliptic BCH search is rooted at k pickups and updates k distances with each edge relaxation (see Section 4.3.1). Additionally, we can bundle bucket entry scans. Whenever a bucket entry for a stop s is scanned, the bundled search tries to improve upon each of the k tentative distances between s and any of the k pickups. We can effectively bundle the edge relaxations and bucket entry scans of elliptic BCH searches because the localized pickups share similar CH search spaces. Moreover, we can use vectorized instructions to parallelize both edge relaxations and bucket entry scans. At the same time, elliptic pruning and sorted buckets can still be applied. To our knowledge, our algorithm is the first to explicitly use bundled BCH searches. The idea follows from the bundled CH searches used in [BSW19].

4.6.2. PD-Distance Searches

Computing the PD-distances, i.e., the distances between pickups and dropoffs, is a many-to-many shortest-path problem where the set of sources and the set of targets are localized.

Our algorithm uses a BCH approach to address this problem. We generate bucket entries for all dropoffs in their reverse CH search spaces. Then, we run queries in the upward CH search graph rooted at each pickup to find the PD-distances using the dropoff bucket entries. We propose two methods to improve these BCH searches.

Firstly, let $\delta_{\text{PD}}^{\max}$ be an an upper bound on all PD-distances. Then, we only have to generate and scan bucket entries in a radius of $\delta_{\text{PD}}^{\max}$. We use

$$\delta_{\text{PD}}^{\max} := \max_{p \in P_\rho} \delta(p, \text{orig}) + \delta(\text{orig}, \text{dest}) + \max_{d \in D_\rho} \delta(\text{dest}, d).$$

We can compute $\delta(p, \text{orig})$ for all $p \in P_\rho$ and $\delta(\text{dest}, d)$ for all $d \in D_\rho$ using two local Dijkstra searches rooted at orig and dest, respectively. We obtain $\delta(\text{orig}, \text{dest})$ with a single preliminary CH query.

Secondly, we can once again use bundled BCH searches. More specifically, we can generate bucket entries for batches of k dropoffs and then run queries for batches of k pickups where k is a configuration parameter. Again, bundled PD-distance searches utilize the locality of pickups and dropoffs and allow us to employ SIMD parallelism.

4.6.3. Enumerating Ordinary, Ordinary Paired, and Pickup Before Next Stop Insertions

After running our elliptic BCH queries and PD-distance searches, we know all distances that are required for *ordinary* and *ordinary paired* insertions. We enumerate the insertions $\mathcal{I} = (r, p, d, \nu, i, j)$ with $0 < i \leq j < k(\nu)$ for a set of candidate vehicles found by the elliptic BCH queries [BSW21]. We compute the cost $c(\mathcal{I})$ for each insertion and update $\mathcal{I}^*$ to $\mathcal{I}$ if $c(\mathcal{I}) < c(\mathcal{I}^*)$.

In a *pickup-before-next-stop (PBNS)* insertion $\mathcal{I} = (r, p, d, \nu, 0, j)$, the pickup p is inserted between stops s_0 and s_1 of the vehicle. This requires the vehicle to be redirected at its current location $\text{loc}(\nu)$ to drive to p next instead of s_1 (cf. Figure 4.2). Thus, to compute the cost of a PBNS insertion, we need to know the distance $\delta(\text{loc}(\nu), p)$.

In order to avoid finding $\delta(\text{loc}(\nu), p)$ for every $\nu \in F$ and $p \in P_\rho$, we employ a filtering technique proposed by Buchhold et al. [BSW21]. The technique exploits that $\delta(s_0, p)$ is a lower bound on $\delta(s_0, \text{loc}(\nu)) + \delta(\text{loc}(\nu), p)$. The distance $\delta(s_0, p)$ can be computed by the

elliptic BCH searches which means we can use it to compute a lower bound on the pickup detour as well as the cost of $\mathcal{I}$. If the lower bound cost of $\mathcal{I}$ exceeds the best known cost, we can discard $\mathcal{I}$.

Most of the time, this filter leaves us with a very small number of pairs of vehicle ν and pickup p for which we actually need to compute $\delta(\text{loc}(\nu), p)$. KaRRi uses a bucket based approach for the remaining necessary queries. We generate source bucket entries for the current location of every affected vehicle and run bundled queries from the pickups. The average number of such queries per request is less than 0.5.

4.7. Pickup After Last Stop Insertions

In this section, we consider *pickup-after-last-stop* (PALS) insertions. The main challenge of PALS insertions is the computation of the distances from last stops to pickups. The authors of LOUD find that elliptic pruning is not applicable for the computation of these distances [BSW21]. Instead, LOUD uses a reverse Dijkstra search rooted at orig that is stopped early when the search can no longer find an insertion better than the best known one. For multiple pickups, we can compute the required distances by analogously running reverse Dijkstra searches for each pickup. These Dijkstra searches may also be bundled to exploit the locality of the pickups.

However, even with a single pickup, this Dijkstra search takes up a significant part of the running time of the LOUD algorithm. Thus, for a large number of pickups, we expect infeasible running times. In this section, we introduce two new BCH based approaches for the computation of last stop distances. For the rest of this section, let $\hat{c}$ denote an upper bound on the best known insertion cost (initially $\hat{c} := c(\mathcal{I}^*)$).

Reformulation of Cost Function for PALS Insertions. Note that the cost of any PALS insertion $\mathcal{I} = (r, p, d, \nu, k(\nu), k(\nu))$ is fully characterized by the pickup p, the PD-distance $\delta(p, d)$, the walking distance $\delta_{\text{psg}}(d)$, the departure time $t^{\min}_{\text{dep}}(s_{k(\nu)})$ of ν at $s_{k(\nu)}$, and the last stop distance $\delta(s_{k(\nu)}, p)$. Thus, we can write the cost of $\mathcal{I}$ as

$$c(\mathcal{I}) = c'(r, p, \delta(p, d), \delta_{\text{psg}}(d), t^{\min}_{\text{dep}}(s_{k(\nu)}), \delta(s_{k(\nu)}, p)).$$

Importantly, the value of c' monotonously increases with its last four arguments. Furthermore, it will be helpful to define $\delta^{\min}_{\text{pd}} := \min_{p \in P_\rho, d \in D_\rho} \delta(p, d)$ as a lower bound on any PD-distance.

4.7.1. Last Stop BCH Searches for PALS

Even though elliptic pruning is not applicable, we can still employ a BCH search approach for distances from last stops to pickups. For this, we maintain a *last stop bucket* $B^{\uparrow}(v)$ for every $v \in V$. For every last stop $s_{k(\nu)}$, we generate an entry $(s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v)) \in B^{\uparrow}(v)$ at each vertex v in the upward CH search space rooted at $s_{k(\nu)}$. Then, for every pickup $p \in P_\rho$, we run an *individual (last stop) BCH query* that explores the reverse CH search space $G^{\downarrow}_p$ rooted at p and scans the last stop bucket at each settled vertex to compute the shortest-path distances from last stops to p. When the search scans an entry $(s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v)) \in B^{\uparrow}(v)$, it tries to improve the tentative distance $\tilde{\delta}(s_{k(\nu)}, p)$ with $\delta^{\uparrow}(s_{k(\nu)}, v) + \delta^{\downarrow}(v, p)$. Eventually, the shortest distance $\delta(s_{k(\nu)}, p)$ is found for every last stop $s_{k(\nu)}$.

Whenever the last stop of a vehicle changes, we traverse the forward CH search spaces of the old and new last stop to remove all old bucket entries and insert new entries, respectively. We stop each individual last stop BCH search as soon as the current distance no longer

admits a PALS insertion with cost smaller than the best known cost. Furthermore, we can bundle the BCH queries and use SIMD parallelism in a similar manner to bundled elliptic BCH searches (see Section 4.6.1).

Pruning Bucket Scans using Sorted Buckets. A remaining issue of this approach is the size of the last stop buckets. Without elliptic pruning, buckets contain many more entries, especially at vertices that have a high rank in the CH. Therefore, the queries have to scan large numbers of bucket entries, rendering the last stop BCH approach inefficient.

The future work section of [BSW21] suggests sorting the entries within each last stop bucket by their distance to address this issue. Suppose an individual BCH query rooted at $p \in P_\rho$ scans the bucket $B^\uparrow(v)$. For every entry $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v))$, let

$$c_{\min}(e) := c'(r, p, \delta^{\min}_{\text{pd}}, 0, t_{\text{req}}(r), \delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)).$$

We can stop each bucket scan early based on this lower bound:

▶ **Theorem 6.** *Let the entries of $B^\uparrow(v)$ be sorted by their upward distances in ascending order. Further, let $\hat{c}$ be an upper bound on the cost of the best insertion of the current request. Then, a BCH query rooted at pickup $p \in P_\rho$ can stop scanning $B^\uparrow(v)$ once it encounters an entry $e \in B^\uparrow(v)$ with $c_{\min}(e) > \hat{c}$.*

Proof. This can be shown by contradiction. Let $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v))$ with $c_{\min}(e) > \hat{c}$. Assume that the best insertion for request r is the PALS insertion $\mathcal{I} = (r, p, d, \nu', k(\nu'), k(\nu'))$ and that v is the highest ranked vertex on the only shortest up-down path from $s_{k(\nu')}$ to p. Then, $\delta(s_{k(\nu')}, p) = \delta^\uparrow(s_{k(\nu')}, v) + \delta^\downarrow(v, p)$ and the shortest path will be found by the BCH query using an entry $e' = (s_{k(\nu')}, \delta^\uparrow(s_{k(\nu')}, v)) \in B^\uparrow(v)$.

If $\delta^\uparrow(s_{k(\nu')}, v) < \delta^\uparrow(s_{k(\nu)}, v)$, then e' is scanned before e and the shortest path from $s_{k(\nu')}$ to p will be found. Otherwise,

$$\begin{aligned} c(\mathcal{I}) &= c'(r, p, \delta(p, d), \delta_{\text{psg}}(d), t^{\min}_{\text{dep}}(s_{k(\nu')}), \delta(s_{k(\nu')}, p)) \\ &\geq c'(r, p, \delta^{\min}_{\text{pd}}, 0, t_{\text{req}}(r), \delta^\uparrow(s_{k(\nu')}, v) + \delta^\downarrow(v, p)) \\ &\geq c'(r, p, \delta^{\min}_{\text{pd}}, 0, t_{\text{req}}(r), \delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)) \\ &= c_{\min}(e) > \hat{c} \end{aligned}$$

Thus, $c(\mathcal{I})$ is larger than the upper bound $\hat{c}$ on the best insertion cost and $\mathcal{I}$ cannot be the best insertion for r. Consequently, we do not have to scan any entries that come after e in $B^\uparrow(v)$. ◀

Updating the Upper Bound Cost. It is possible to simply use the cost of the best known insertion $c(\mathcal{I}^*)$ for the upper bound cost $\hat{c}$ needed for cost pruning. However, we can also improve $\hat{c}$ during the search. Each tentative distance $\tilde{\delta}(s_{k(\nu)}, p)$ found acts as an upper bound on the actual shortest distance $\delta(s_{k(\nu)}, p)$. Thus, whenever the tentative distance $\tilde{\delta}(s_{k(\nu)}, p)$ is updated, we can compute an upper bound

$$c_{\max} := c'(r, p, \delta(p, \text{dest}), 0, t^{\min}_{\text{dep}}(s_{k(\nu)}), \tilde{\delta}(s_{k(\nu)}, p))$$

on the cost of the best PALS insertion with ν and p. We update $\hat{c}$ to $c_{\max}$ if $c_{\max} < \hat{c}$. This technique finds inexact upper bounds on the cost of the best PALS insertion early which is helpful for the stopping criterion of bucket scans.

4.7.2. Collective Last Stop Searches for PALS

Finally, we propose a search approach based on the idea that we do not actually need to know the distance between every last stop and every pickup. If we knew the best PALS insertion $\mathcal{I}^*_{\text{pals}} = (r, p, d, \nu, k(\nu), k(\nu))$ in advance, we would only need to find $\delta(s_{k(\nu)}, p)$. Obviously, we do not know $\mathcal{I}^*_{\text{pals}}$ in advance but we find that it is possible to prune the distance queries for individual pickups (or actually PD-pairs) by comparing the queries to each other whenever they meet at a vertex. We introduce a collective BCH query that finds the best PALS insertion $\mathcal{I}^*_{\text{pals}}$ as well as the last stop distance $\delta(s_{k(\nu)}, p)$. In the following, we explain how labels representing PD-pairs are propagated through the CH search graph and how these labels can be pruned based on label domination.

Open and Closed Labels. A PD-pair label $(p, d, \delta^{\downarrow}(v, p))$ at a vertex $v \in V$ consists of the pickup $p \in P_\rho$, dropoff $d \in D_\rho$ and downwards distance $\delta^{\downarrow}(v, p)$. At each vertex $v \in V$, there is a set of *open* labels open(v) and a set of *closed* labels closed(v). An open label is a label that has not been settled yet. For each open label $l = (p, d, \delta^{\downarrow}(v, p))$, we store a lower bound $c_{\min}(l)$ for the cost of a PALS insertion that can be found for l in $G^{\downarrow}_v$

$$c_{\min}(l) := c'(r, p, \delta(p, d), \delta_{\text{psg}}(d), t_{\text{req}}(r), \delta^{\downarrow}(v, p))$$

Algorithm Outline. We give pseudocode for a collective BCH search in Algorithm 4.1. Our search maintains a priority queue Q that contains all open labels ordered increasingly by $c_{\min}$. Initially, at each pickup $p \in P_\rho$, an open label $(p, d, 0) \in \text{open}(p)$ is created for each $d \in D_\rho$ (line 7). As long as Q contains a label l with $c_{\min}(l) \leq \hat{c}$ for a known upper bound $\hat{c}$ on the cost of any insertion, our search proceeds with a next step (lines 9-12). In each step of the search, the label $l := \min(Q)$ is removed from Q and settled as described in the following.

Settling Open Labels. Settling an open label $l = (p, d, \delta^{\downarrow}(v, p))$ consists of three steps: First, we mark l closed at v, i.e., we move l from open(v) to closed(v) (line 14). Second, we search for a new best insertion by traversing all entries in the last stop bucket $B^{\uparrow}(v)$ (lines 15-18). For each entry $e = (s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v)) \in B^{\uparrow}(v)$, we compute the tentative distance $\tilde{\delta}(s_{k(\nu)}, p) = \delta^{\uparrow}(s_{k(\nu)}, v) + \delta^{\downarrow}(v, p)$ and a cost upper bound

$$c_{\max}(l, e) := c'(r, p, \delta(p, d), \delta_{\text{psg}}(d), t^{\min}_{\text{dep}}(s_{k(\nu)}), \tilde{\delta}(s_{k(\nu)}, p)).$$

If $c_{\max}(l, e) < \hat{c}$, we mark $\mathcal{I} = (r, p, d, \nu, k(\nu), k(\nu))$ as the best known PALS insertion, store the tentative distance $\tilde{\delta}(s_{k(\nu)}, p)$, and update $\hat{c} := c_{\max}(l, e)$. Note that $c_{\max}(l, e)$ is the exact cost of the PALS insertion $\mathcal{I} = (r, p, d, \nu, k(\nu), k(\nu))$ if $\tilde{\delta}(s_{k(\nu)}, p)$ is a shortest-path distance. Since the BCH search finds shortest up-down paths, we will thus eventually find the best PALS insertion. As before, we can stop each bucket scan early. For this purpose, we compute a vehicle-independent cost lower bound $c_{\min}(l, e)$ s.t. we can stop the search early if $c_{\min}(l, e) > \hat{c}$ using

$$c_{\min}(l, e) := c'(r, p, \delta(p, d), \delta_{\text{psg}}(d), t_{\text{req}}(r), \delta^{\uparrow}(s_{k(\nu)}, v) + \delta^{\downarrow}(v, p)).$$

Third, we propagate l to all neighboring vertices of v. For each neighboring vertex $w \in V$ with $(w, v) \in G^{\downarrow}$, we create a new open label $l' = (p, d, \ell^+(w, v) + \delta^{\downarrow}(v, p))$ at w (lines 19-21). Here, we employ cost pruning by discarding l' if the lower bound cost $c_{\min}(l')$ for this PD-pair and this distance exceeds $\hat{c}$ (line 23). Furthermore, we may be able to prune l' at v if it is dominated by an existing label at v (lines 24-25) as described in the following.

Algorithm 4.1 Collective BCH search used to find distances from last stops to pickups.

Input: P_ρ, D_ρ, $G^\downarrow = (V^\downarrow, E^\downarrow)$, $B^\uparrow(v)$ for $v \in V$
Output: (p^*, d^*) and $\delta(s_{k(\nu)}, p^*)$

procedure CollectiveBCH
  $Q :=$ PQ of labels with $key_Q(l) = c_{\min}(l)$
  $\forall v \in V : \text{open}(v) := \text{closed}(v) := \emptyset$
  $\hat{c} := c(\mathcal{I}^*)$
  for each $(p, d) \in P_\rho \times D_\rho$ **do**
    insertLabelAtVertex$(p, (p, d, 0))$
  while $Q \neq \emptyset$ **do**
    $l := Q$.deleteMin()
    if $c_{\min}(l) > \hat{c}$ **then return**
    settleLabel(l)

procedure settleLabel$(l = (p, d, \delta^\downarrow(v, p)))$
  open(v).remove(l); closed(v).insert(l)
  for each $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v)) \in B^\uparrow(v)$ **do**
    if $c_{\min}(l, e) > \hat{c}$ **then break**
    if $c_{\max}(l, e) < \hat{c}$ **then**
      $(p^*, d^*) := (p, d)$; $\hat{c} := c_{\max}(l, e)$
  for each $(u, v) \in E^\downarrow$ **do**
    $l' := (p, d, \ell^+(u, v) + \delta^\downarrow(v, p))$
    insertLabelAtVertex(u, l')

procedure insertLabelAtVertex(v, l')
  if $c_{\min}(l') > \hat{c}$ **then return**
  for each $l \in \text{open}(v) \cup \text{closed}(v)$ **do**
    if l dominates l' **then return**
  for each $l \in \text{open}(v)$ **do**
    if l' dominates l **then** open(v).remove(l)
  open(v).insert(l'); Q.insert(l')

Domination Pruning. Propagating a label through the entire search space for every PD-pair is too expensive. However, we find that we can compare labels at the same vertex and prune dominated labels in a technique we call *domination pruning*. Intuitively, a label l dominates a label l' at a vertex v if we know that any insertion found in the reverse CH search space $G_v^\downarrow$ rooted at v that uses l' has higher costs than the equivalent insertion using l.

To formalize this, we first define an upper bound for the cost of a PALS insertion found in $G_v^\downarrow$ for a label l. Let $l = (p, d, \delta^\downarrow(v, p))$ be a PD-pair label at a vertex $v \in V$. Let $w \in V_v^\downarrow$ and $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, w)) \in B^\uparrow(w)$. Then,

$$c_{\max}(l, v, e) := c'(r, p, \delta(p, d), \delta_{\text{psg}}(d), t_{\text{dep}}^{\min}(s_{k(\nu)}), \delta^\uparrow(s_{k(\nu)}, w) + \delta^\downarrow(w, v) + \delta^\downarrow(v, p)).$$

With this, we can formally define the domination relation between labels:

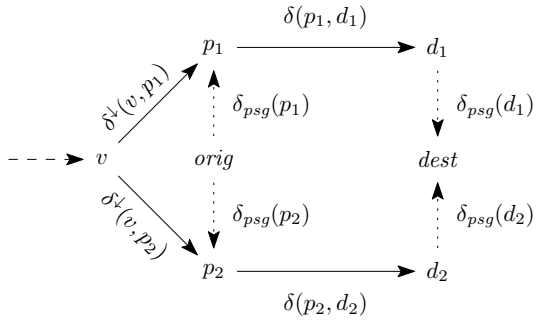


Figure 4.3 Depiction of a situation during a collective PALS search in which domination pruning may be applied for two labels $l_i = (p_i, d_i, \delta^{\downarrow}(v, p_i))$ for $i = 1, 2$ at v. Solid arrows indicate known parts of potential vehicle routes and dotted arrows indicate walking routes. Dashed arrow symbolizes unknown vehicle route to v.

▶ **Definition 7.** *A PD-pair label* l dominates *another label* l' *at a vertex* $v \in V$ *exactly if* $c_{\max}(l, v, e) < c_{\max}(l', v, e)$ *for every* $w \in V_v^{\downarrow}$ *and* $e \in B^{\uparrow}(w)$.

▶ **Theorem 8.** *If a label* l *dominates another label* l' *at* v*, we do not need to settle* l' *at* v.

Proof. This can be shown by contradiction. Assume $l_1 = (p_1, d_1, \delta^{\downarrow}(v, p_1))$ dominates $l_2 = (p_2, d_2, \delta^{\downarrow}(v, p_2))$ at v. Further, assume that $\mathcal{I} = (r, p_2, d_2, \nu, k(\nu), k(\nu))$ is the best PALS insertion. Let π be a shortest path from $s_{k(\nu)}$ to p_2. Wlog. π is an up-down-path in the CH consisting of an upwards prefix $\pi^{\uparrow}$ and a downwards suffix $\pi^{\downarrow}$. If $\pi^{\downarrow}$ does not contain v, then the collective search will not find π in $G_v^{\downarrow}$, and we do not have to settle l_2 at v.

Otherwise, $\pi^{\downarrow} = (w, \ldots, v, \ldots, p_2)$ with $w \in V_v^{\downarrow}$. Let $e = (s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, w)) \in B^{\uparrow}(w)$. Since π is a shortest path, we know that

$$c_{\max}(l_2, v, e) = c((r, p_2, d_2, \nu, k(\nu), k(\nu))).$$

However, l_1 dominates l_2 which means that

$$c((r, p_1, d_1, \nu, k(\nu), k(\nu))) \leq c_{\max}(l_1, v, e) < c_{\max}(l_2, v, e) = c((r, p_2, d_2, \nu, k(\nu), k(\nu)))$$

This contradicts $\mathcal{I}$ being the best PALS insertion. Hence, we do not have to settle label l_2 at v to find the best pair for ν. ◀

Efficiently Computing the Domination Relation. We find that it is not trivial to compute the domination relation efficiently because of the non-linearity of the cost function.

Consider the example depicted in Figure 4.3. Our algorithm needs to compute the domination relation between two labels $l_i = (p_i, d_i, \delta^{\downarrow}(v, p_i))$ for $i = 1, 2$ at v. For any bucket entry for a vehicle ν that we may scan in $G_v^{\downarrow}$, we need to decide whether ν should make the pickup and dropoff at p_1 and d_1 or p_2 and d_2.

In any case, ν passes the vertex v on its way to the pickup. Assume the vehicle arrives at v at some time t. We can then characterize the costs $c(\mathcal{I}_i, t)$ of the tentative insertions $\mathcal{I}_i = (r, p_i, d_i, \nu, k(\nu), k(\nu))$ for $i = 1, 2$ using t. We define the components of our cost function

(see Section 4.2) for this specific case:

$$\begin{aligned} t_{\text{trip}}(\mathcal{I}_i, t) &= t_{\text{arr}}(\mathcal{I}_i, t) + \delta_{\text{psg}}(d_i) - t_{\text{req}} \\ t^{+}_{\text{trip}}(\mathcal{I}_i, t) &= 0 \\ t_{\text{detour}}(\mathcal{I}_i, t) &= t_{\text{arr}}(\mathcal{I}_i, t) - t^{\min}_{\text{dep}}(s_{k(\nu)}) \\ t_{\text{walk}}(\mathcal{I}_i, t) &= \delta_{\text{psg}}(p_i) + \delta_{\text{psg}}(d_i) \end{aligned}$$

Here, the departure time at pickup p_i and the arrival time at dropoff d_i are defined as

$$\begin{aligned} t_{\text{dep}}(p_i, t) &:= \max\{t + \delta^{\downarrow}(v, p_i), t_{\text{req}} + \delta_{\text{psg}}(p_i)\} \\ t_{\text{arr}}(\mathcal{I}_i, t) &:= t_{\text{dep}}(p_i, t) + \delta(p_i, d_i). \end{aligned}$$

The constant term $t^{\min}_{\text{dep}}(s_{k(\nu)})$ in the detour time vanishes when we compute the cost difference $c(\mathcal{I}_1, t) - c(\mathcal{I}_2, t)$. Thus, t fully expresses the contribution of the vehicle and the cost difference is a function $\Delta_c(l_1, l_2, t)$ of only the two labels and t. Then, if $\Delta_c(l_1, l_2, t) < 0$ for all $t \geq t_{\text{req}}$, every insertion found in $G^{\downarrow}_v$ will be better with l_1 than with l_2, i.e., l_1 dominates l_2.

If the cost function for PALS insertions were to grow linearly with t, then $\Delta_c(l_1, l_2, t)$ would be constant with respect to t. In that case, we could simply check for domination using $\Delta_c(l_1, l_2, 0) < 0$. However, the cost function does not increase linearly with t, because the cost is constant with respect to t as long as the vehicle arrives at p_i earlier than the rider (see Section 4.5.2). If $t + \delta^{\downarrow}(v, p_i) \leq t_{\text{req}} + \delta_{\text{psg}}(p_i)$, then ν will wait for the rider at p_i for a certain wait time. The resulting maximum operation in $t_{\text{dep}}(p_i, t)$ introduces a point of non-linearity.

The rider arrival time at p_i differs between labels since $\delta_{\text{psg}}(p_i)$ differ. Thus, $\Delta_c(l_1, l_2, t)$ varies with t. Since we do not know which values of t are possible for insertions found in $G^{\downarrow}_v$, we cannot trivially determine whether l_1 dominates l_2.

Approximating the Domination Relation. Instead, we under-approximate the domination relation by computing a sufficient precondition. For this purpose, we find an upper bound $\Delta^{\max}_{\text{c}}(l_1, l_2) \geq \max_{t \geq t_{\text{req}}} \Delta_c(l_1, l_2, t)$ on the difference in insertion costs between any insertion that can be found for l_1 and l_2 in $G^{\downarrow}_v$. Then, l_1 dominates l_2 if $\Delta^{\max}_{\text{c}}(l_1, l_2) < 0$.

The cost upper bound is based on an upper bound on the difference in departure times at the pickup. For any $t \geq t_{\text{req}}$, we have

$$\begin{aligned} & t_{\text{dep}}(p_1, t) - t_{\text{dep}}(p_2, t) \\ \leq & \max\{t + \delta^{\downarrow}(v, p_1), t + \delta_{\text{psg}}(p_1)\} - (t + \delta^{\downarrow}(v, p_2)) \\ = & \max\{\delta^{\downarrow}(v, p_1), \delta_{\text{psg}}(p_1)\} - \delta^{\downarrow}(v, p_2). \end{aligned}$$

Thus, for any vehicle, the difference in the departure times at p_1 and p_2 is at most

$$\Delta_{\text{dep}}(l_1, l_2) := \max\{\delta^{\downarrow}(v, p_1), \delta_{\text{psg}}(p_1)\} - \delta^{\downarrow}(v, p_2).$$

Then, for every insertion found in $G^{\downarrow}_v$, we have the following upper bounds on the difference in detours and trip times between l_1 and l_2:

$$\begin{aligned} \Delta_{\text{trip}}(l_1, l_2) &:= \Delta_{\text{detour}}(l_1, l_2) + \delta_{\text{psg}}(d_1) - \delta_{\text{psg}}(d_2) \\ \Delta_{\text{detour}}(l_1, l_2) &:= \Delta_{\text{dep}}(l_1, l_2) + \delta(p_1, d_1) - \delta(p_2, d_2) \end{aligned}$$

Let $\Delta_{walk}(l_1, l_2)$ be the fix difference in walking costs. Putting it all together, we get

$$\Delta_{\mathrm{c}}^{\max}(l_1, l_2) := \Delta_{\mathrm{trip}}(l_1, l_2) + w_{\mathrm{detour}}\Delta_{\mathrm{detour}}(l_1, l_2) + w_{\mathrm{walk}}\Delta_{walk}(l_1, l_2)$$

During a collective BCH search, we can compute $\Delta_{\mathrm{c}}^{\max}(l_1, l_2)$ in constant time with information that is known at v without looking ahead in the search tree. Since we under-approximate domination, it is possible that l_1 actually dominates l_2 but our condition does not hold. However, we find that our domination criterion still manages to prune the vast majority of labels early.

Limitations. We remark that the insertion $\mathcal{I}$ found by the collective search is only guaranteed to be the best possible PALS insertion if $\mathcal{I}$ adheres to the service time hard constraint. Since our search ignores the service time constraint, it may return an insertion that breaks the constraint even if there are other PALS insertions that do not.

Therefore, if $\mathcal{I}$ breaks the service time constraint, we fall back to computing the distances from every last stop to every pickup using individual last stop BCH searches. The fallback individual BCH searches can make use of the good cost upper bounds found during the collective search. We find that this is only necessary in exceedingly rare cases.

4.8. Dropoff After Last Stop Insertions

A *dropoff-after-last-stop (DALS)* insertion $\mathcal{I} = (r, p, d, \nu, i, k(\nu))$ inserts the dropoff (but not the pickup) after the last stop of the vehicle's current route (cf. Figure 4.2). To compute the cost of a DALS insertion we need to compute the distance $\delta(s_{k(\nu)}, d)$ from the vehicle's last stop to the dropoff. This is similar to the shortest-path problem in the PALS case. We can utilize the approaches of bundled searches, BCH queries with sorted last stop buckets, and collective BCH queries with some minor differences.

Firstly, cost pruning is less effective than in the PALS case since the lower bounds on costs cannot include the PD-distance. Secondly, we cannot update the global cost upper bound $\hat{c}$ during the searches in the DALS case as we lack information about the cost of inserting the pickup earlier in the route. Finally, collective BCH searches have some more intricate differences between the PALS and DALS cases. We go into more detail about these differences in the rest of this section.

Collective BCH Searches for DALS. Collective BCH searches for the DALS case propagate labels for individual dropoffs through the search graph. Labels can be pruned based on a lower bound cost for each label or based on domination pruning. Domination between dropoff labels is a partial relation defined as follows.

▶ **Definition 9.** *Let $l_z = (d_z, \delta^{\downarrow}(v, d_z))$ for $z = 1, 2$ be two dropoff labels at $v \in V$. Let*

$$\begin{aligned}\Delta(l_1, l_2) &:= \delta^{\downarrow}(v, d_1) - \delta^{\downarrow}(v, d_2)\\ \Delta_{walk}(l_1, l_2) &:= \delta_{psg}(d_1) - \delta_{psg}(d_2)\\ \Delta_{trip}(l_1, l_2) &:= \Delta(l_1, l_2) + \Delta_{walk}(l_1, l_2)\end{aligned}$$

Then d_1 dominates d_2 *if*

$$\Delta_{trip}(l_1, l_2) + w_{detour}\Delta(l_1, l_2) + w_{walk}\Delta_{walk}(l_1, l_2) < 0$$

Assume that a label l_1 dominates another label l_2 at $v \in V$. The domination relation makes sure that the cost for any DALS insertion $\mathcal{I} = (r, p, d_z, \nu, i, k(\nu))$ found in $G_v^{\downarrow}$ will have smaller costs with d_1 than with d_2.

▶ **Theorem 10.** *If a dropoff label l_1 dominates another label l_2 at v, we do not need to settle l_2 at v.*

Proof. This can be shown by contradiction. Assume $l_1 = (d_1, \delta^{\downarrow}(v, d_1))$ dominates $l_2 = (d_2, \delta^{\downarrow}(v, d_2))$ at $v \in V$. Further, assume that $\mathcal{I}_2 = (r, p, d_2, \nu, i, k(\nu))$ is the best DALS insertion.

Let π be a shortest path from $s_{k(\nu)}$ to d_2. Wlog. π is an up-down path in the CH consisting of an upwards prefix $\pi^{\uparrow}$ and a downwards suffix $\pi^{\downarrow}$. If $\pi^{\downarrow}$ does not contain v, then the collective search will not find π in $G_v^{\downarrow}$, and we do not have to settle l_2 at v.

Otherwise, $\pi^{\downarrow} = (w, \dots, v, \dots, d_2)$ with $w \in V_v^{\downarrow}$. Since π is a shortest path, we know that

$$\delta(s_{k(\nu)}, d_2) = \delta^{\uparrow}(s_{k(\nu)}, w) + \delta^{\downarrow}(w, v) + \delta^{\downarrow}(v, d_2) \text{ and}$$
$$\delta(s_{k(\nu)}, d_1) \leq \delta^{\uparrow}(s_{k(\nu)}, w) + \delta^{\downarrow}(w, v) + \delta^{\downarrow}(v, d_1).$$

Consequently,

$$\delta(s_{k(\nu)}, d_1) - \delta(s_{k(\nu)}, d_2) \leq \delta^{\downarrow}(v, d_1) - \delta^{\downarrow}(v, d_2). \tag{$*$}$$

We consider the insertion $\mathcal{I}_1 = (r, p, d_1, \nu, i, k(\nu))$ and the cost difference $c(\mathcal{I}_1) - c(\mathcal{I}_2)$. Since the pickup detour of $\mathcal{I}_1$ and $\mathcal{I}_2$ are equal, we have $t^{+}_{\text{trip}}(\mathcal{I}_1) = t^{+}_{\text{trip}}(\mathcal{I}_2)$. Comparing the cost of both insertions then yields

$$\begin{aligned}
&c(\mathcal{I}_1) - c(\mathcal{I}_2) \\
&= (t_{\text{trip}}(\mathcal{I}_1) - t_{\text{trip}}(\mathcal{I}_2)) + w_{\text{detour}}(t_{\text{detour}}(\mathcal{I}_1) - t_{\text{detour}}(\mathcal{I}_2)) + w_{\text{walk}}(t_{\text{walk}}(\mathcal{I}_1) - t_{\text{walk}}(\mathcal{I}_2)) \\
&\overset{(*)}{\leq} \Delta_{\text{trip}}(l_1, l_2) + w_{\text{detour}}\Delta(l_1, l_2) + w_{\text{walk}}\Delta_{\text{walk}}(l_1, l_2) \\
&< 0
\end{aligned}$$

where the last inequality is due to domination. Hence, we do not have to settle l_2 at v. ◀

For each vehicle $\nu \in F$, the collective BCH search finds a single best dropoff d and the distance $\delta(s_{k(\nu)}, d)$. Since the cost induced by the dropoff is independent of the pickup p and pickup insertion index i, the optimal dropoff is unique for each vehicle and identical for all DALS insertions of that vehicle.

4.9. Experiments

Our source code[1] is written in C++20 and compiled with GCC 11.5 using `-O3`. We run our experiments on a machine that runs Rocky Linux 9.4 with an AMD EPYC 7702 processor (64 physical cores) and 1.0 TiB of RAM. Note that we only use a single processor core in this chapter. We use 32-bit distance labels and the AVX2 SIMD instruction set with 256-bit registers to compute up to 8 operations in one vector instruction.

We evaluate KaRRi on the `Berlin-1pct` (`B-1%`), `Berlin-10pct` (`B-10%`), `Ruhr-1pct` (`R-1%`), and `Ruhr-10pct` (`R-10%`) request sets [BSW21] that respectively represent 1% and

[1] Available at `https://github.com/molaupi/karri`.

Table 4.1 Key figures of our benchmark instances. Shows number of vertices ($|V|$), edges ($|E|$), vehicles (#veh.), and requests (#req.). Additionally, shows average number (rounded down) of pickups (N_ρ^p) and dropoffs (N_ρ^d) for walking radius $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$ on the `Berlin-1pct` and `Berlin-10pct` instances, and $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ on the `Ruhr-1pct` and `Ruhr-10pct` instances.

Instance	$\lvert V\rvert$ [$\cdot 10^3$]	$\lvert E\rvert$ [$\cdot 10^3$]	#veh.	#req.	$\rho = 0$s		$\rho = 150$s		$\rho = 300$s		$\rho = 450$s		$\rho = 600$s	
					N_ρ^p	N_ρ^d	N_ρ^p	N_ρ^d	N_ρ^p	N_ρ^d	N_ρ^p	N_ρ^d	N_ρ^p	N_ρ^d
`B-1%`	101	204	1000	16 569	1	1	15	16	55	55	122	122	216	217
`B-10%`	101	204	10 000	149 185	1	1	16	16	56	57	125	127	221	226
`R-1%`	988	2145	3000	49 707	1	1			51	48			164	160
`R-10%`	988	2145	30 000	447 555	1	1			51	48			164	160

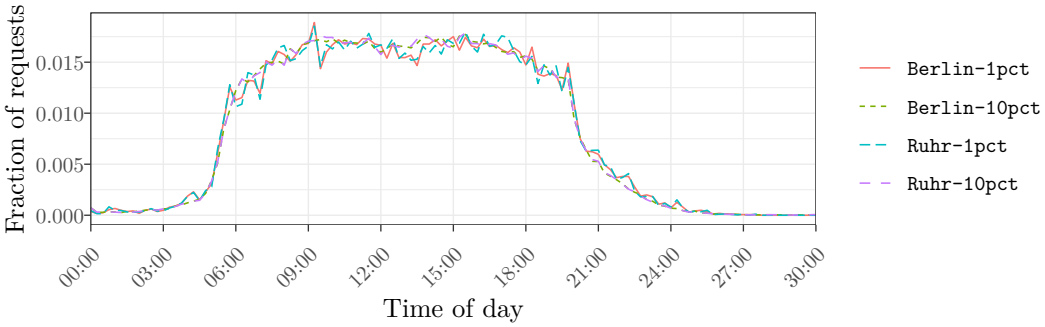


Figure 4.4 Distribution of trips over time for the `Berlin-1pct`, `Berlin-10pct`, `Ruhr-1pct`, and `Ruhr-10pct` instances. Shows fraction of total trips per 15-minute interval of the observation period.

10% of ride-pooling demand in the Berlin and Rhein-Ruhr metropolitan areas on a weekday. For the Berlin instances, Buchhold et al. [BSW21] run the agent-based transport simulation MATSim [HNA16] on the MATSim Open Berlin Scenario [ZKN19]. The simulation iteratively creates realistic travel patterns for a representative sample of the population containing 1% or 10% of the actual population[2]. For every trip, MATSim simulates the traveler's choice between different modes of transportation, including ride-pooling. The `Berlin-1pct` and `Berlin-10pct` request sets include all trips made with ride-pooling in the result of the simulation. For the Rhein-Ruhr request sets, the request times as well as the pickup and dropoff locations are randomly drawn according to distributions that lead to similar trip times and request density as the Berlin request sets (for details, see [BSW21]).

The underlying road networks are obtained from OpenStreetMap data as described in Section 10.2. In this chapter, we assume a constant walking speed of 5km/h for every traveler. We use the open-source library RoutingKit[3] to compute the contraction hierarchies of the road networks which takes less than a minute for our instances.

We consider walking as a mode of local transportation for riders. We scale the number

[2] Buchhold et al. [BSW21] do not simulate the entire population of Berlin due to prohibitive simulation times with the traditional MATSim implementation. However, a significantly faster new implementation of MATSim promises to alleviate this issue [LHN25]. Thus, this process of generating ride-pooling demand sets could be scaled up in the future.

[3] See `https://github.com/RoutingKit`.

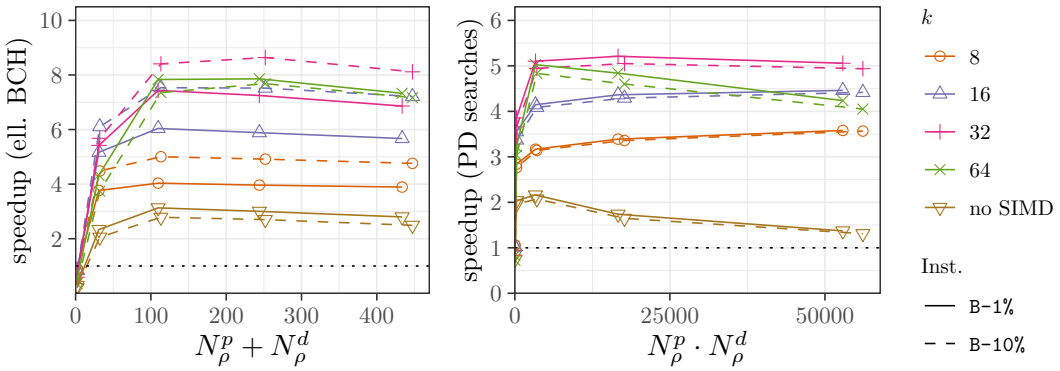


Figure 4.5 Mean speedups for bundling with SIMD instructions for elliptic BCH searches (left) and PD-distance searches (right) on the `Berlin-1pct` and `Berlin-10pct` instances with radius $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$. The x-axis shows the total number of meeting points ($N^p_\rho + N^d_\rho$) and the number of PD-pairs ($N^p_\rho \cdot N^d_\rho$) for the given radius, respectively. Considers $k \in \{8, 16, 32, 64\}$. Additionally shows running times without SIMD instructions with $k = 32$ (*no SIMD*). Note the different y-axes.

of pickups N^p_ρ and dropoffs N^d_ρ by using increasing maximum walk times (walking radius) $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$. We show the number of vertices, edges, vehicles, and requests as well as the numbers of pickups and dropoffs for different walking radii in Table 4.1. Figure 4.4 shows the distribution of requests over time. We run five iterations of every experiment, and report average running times.

Unless otherwise specified, we use the following default assumptions: We use the model parameters decided on in the parameter tuning experiments in Section 10.3. Vehicles have a default capacity of four, and the initial location of every vehicle is an edge drawn uniformly at random in the road network of the service area. The service time of every vehicle is set to encompass the entire observation period. We specify the number of runs per experiment and report average metric values over all runs.

4.9.1. Bundled Searches

In this section, we experimentally evaluate bundled searches in each of the described applications and find the optimal value of k for each of them. We conduct our experiments on the `Berlin-1pct` and `Berlin-10pct` instances with $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$.

Bundled Elliptic BCH Searches and PD-Distance Searches. We show experimental speedups for bundled elliptic BCH searches and bundled PD-distance searches in Figure 4.5.

We find that bundled searches with $k = 32$ lead to good speedups of up to 8.64 and 5.05 for elliptic BCH searches and PD-distance searches, respectively. Without vector instructions, we observe speedups of up to 3.13 and 2.07. For $k = 64$, speedups are smaller, particularly for the larger `Berlin-10pct` instance. This is due to the fact that the search trees of meeting points might not overlap in the beginning of the search. When considering many meeting points within the same batch, this can lead to redundant work. The value $k = 32$ strikes a balance between bundling effectively in the periphery of the sources where search trees overlap, and avoiding excessive redundant work in the beginning where there is less overlap.

Note that the SIMD parallelism is limited to eight searches per vector instruction, so values $k > 8$ do not allow more parallelism. The additional speedup for larger k is solely because of improved memory locality.

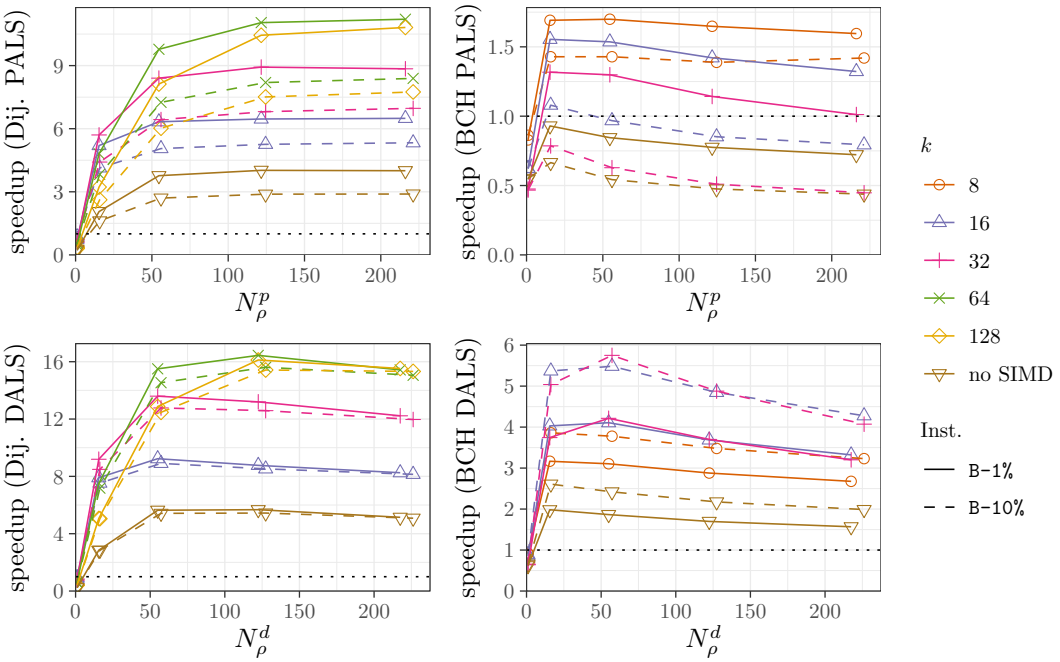


Figure 4.6 Mean speedups for bundling with SIMD instructions for Dijkstra searches (left) and individual BCH searches (right) in the PALS (top) and DALS (bottom) cases on the `Berlin-1pct` and `Berlin-10pct` instances with $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$. The x-axis shows the total number of pickups (N^p_ρ) or number of dropoffs (N^d_ρ). Considers $k \in \{16, 32, 64, 128\}$ for Dijkstra searches and $k \in \{8, 16, 32\}$ for BCH searches. Additionally shows speedups for bundling without SIMD for Dijkstra searches with $k = 64$ and BCH searches with $k = 8$ (*no SIMD*). Note the different y-axes.

Bundled Last Stop Searches. We depict speedups for bundled Dijkstra searches and individual BCH searches for the PALS and DALS cases in Figure 4.6.

We find that Dijkstra searches are well suited for bundling as we observe the smallest search times with $k = 64$. Since Dijkstra searches do not use shortcut edges, the searches for each individual source meet much earlier than BCH searches. Thus, the vast majority of the edge relaxations performed by the Dijkstra searches can be bundled well. This is evidenced by the fact that we see good speedups for bundled Dijkstra searches even without SIMD instructions. Larger $k > 64$ may be useful for larger numbers of sources but eventually we run into cache limitations as hundreds of bytes of distances need to be handled per vertex.

Contrarily, individual last stop BCH searches cannot be bundled as well due to two opposing properties: Firstly, most work is performed close to the sources. With sorted buckets, more bucket entries are scanned at vertices closer to the sources. Additionally, the cost based stopping criterion of last stop BCH searches limits the search radius. Secondly, due to the usage of shortcut edges, the search trees of individual searches only overlap at larger distances from the sources. Thus, bundling does not work well in the proximity of the sources and most work performed by last stop BCH searches is not well suited for bundling.

These factors have a stronger impact in the PALS case, as the stopping criterion is more effective (see Section 4.8). Thus, in the PALS case, bundling only achieves speedups of 1.7 on `Berlin-1pct` and 1.43 on `Berlin-10pct` for $k = 8$. Larger k consistently perform worse. Bundled searches without vector instructions are slower than non-bundled searches in the PALS case. In the DALS case, a larger search radius is explored, which allows better

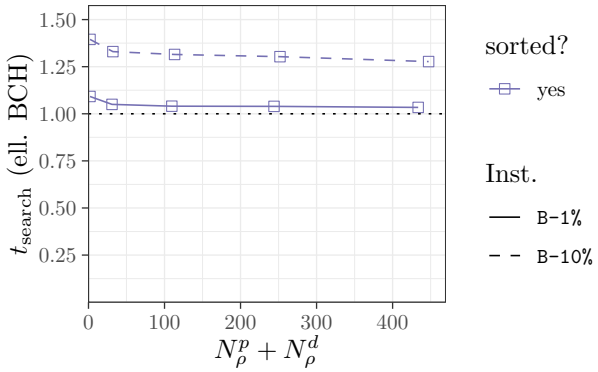


Figure 4.7 Mean speedups of elliptic BCH searches ($k = 1$) with sorted over unsorted buckets on the `Berlin-1pct` and `Berlin-10pct` instances for $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$.

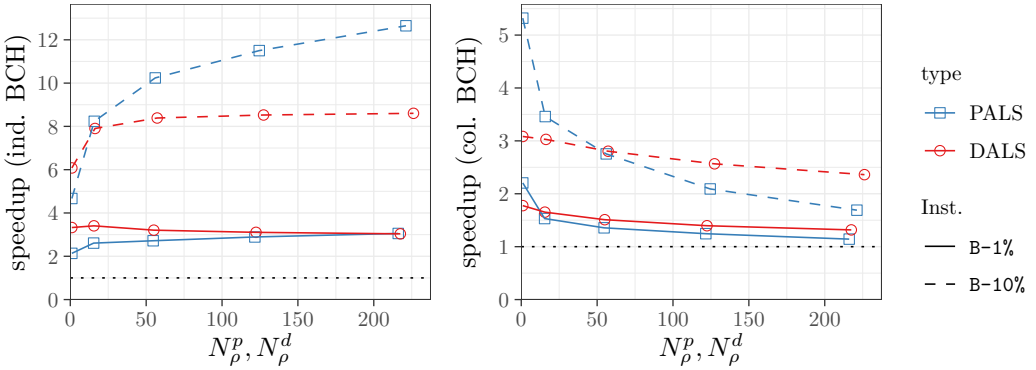


Figure 4.8 Mean speedups for individual (left, $k = 1$) and collective (right) BCH queries with sorted over unsorted buckets. Considers the PALS and DALS cases on the `B-1%` and `B-10%` instances for $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$. Note the different y-axes.

bundling with speedups of up to 4.21 and 5.75 for $k = 32$ on `Berlin-1pct` and `Berlin-10pct`, respectively. Even without SIMD, bundling achieves speedups of more than 2.5 for DALS.

4.9.2. Sorted Buckets

In the following, we analyze the effect of sorted buckets on elliptic BCH searches as well as individual and collective last stop BCH searches. We consider the reduction in the number of bucket entries scanned as well as the effects on the running time of the searches and the time for updating buckets. We experimentally compare all searches with sorted and unsorted buckets on `Berlin-1pct` and `Berlin-10pct` with $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$ and $k = 1$.

Sorted Buckets for Elliptic BCH Searches. The buckets for elliptic BCH searches are already strongly pruned using elliptic pruning. Therefore, sorting these buckets only elicits a major effect with a sufficiently large number of vehicles. As we can see in Figure 4.7, sorted buckets has almost no impact for the `Berlin-1pct` instance but leads to small speedups for the `Berlin-10pct` instance as it considers ten times more vehicles. On the larger input, sorted buckets reduce the number of entries scanned by about 40%, which leads to a decrease in the search time by up to 28%. For the largest radius $\rho = 600\text{s}$, sorting the buckets saves 10ms per request during the query, while updating buckets only becomes about 36µs

slower. In conclusion, sorted buckets are a valuable improvement for elliptic BCH searches, particularly with respect to the scalability to larger numbers of vehicles.

Sorted Buckets for Last Stop BCH Searches. In the following, we analyze the effect of sorted buckets on individual and collective last stop BCH searches. The speedups achieved with sorted buckets are shown in Figure 4.8.

For last stop BCH searches, sorted buckets are vital to reduce the number of bucket entries scanned since we cannot use elliptic pruning. For individual BCH searches, more than 97% and 98% fewer bucket entries are scanned with sorted buckets in the PALS and DALS cases, respectively. This reduces search times by factors of up to 12.65 and 8.60.

For collective searches, the number of bucket entries scanned decreases by similar rates of 94% and 96%. However, the resulting speedups are less pronounced, particularly for larger numbers of meeting points in the PALS case. We attribute this to the fact that collective searches spend comparatively more time on pruning the searches. This means that the searches need to spend less time scanning bucket entries, which limits the impact of sorted buckets. Notably, collective PALS searches generate initial labels for every PD-pair but prune almost all of them immediately. As the number of PD-pairs is proportional to ρ^4, this initialization can constitute up to 78% of the search time for larger values of ρ but sorted buckets have no effect on it. Consequently, we observe speedups of only 1.69 (PALS) and 2.36 (DALS) for $\rho = 600$s on `Berlin-10pct`. If we disregard the overhead for initial labels, these speedups increase to 4.19 and 2.66.

Maintaining sorted last stop buckets incurs an average overhead per request of at most 13µs while the reduction in search time is one to three orders of magnitude larger.

4.9.3. Collective BCH Searches

In this section, we analyze the performance of our collective last stop BCH queries (see Sections 4.7.2 and 4.8). In Table 4.2, we compare the search times and the times needed to enumerate candidate insertions for the three search approaches used for the PALS and DALS insertion types. Additionally, we show the number of relaxed edges and scanned bucket entries (or vertices in the case of Dijkstra-based queries). We report the results for $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ on the `B-1%` and `B-10%` instances. We use the optimal configuration for each combination of search type, insertion type, and radius.

At $\rho = 0$s, collective searches are slower than individual BCH searches as there is only a single pickup and dropoff so the overhead for explicitly maintaining labels is unwarranted. At $\rho = 300$s and $\rho = 600$s, collective searches offer the best search times, though. In the PALS case, search times for collective BCH are up to 5.67 times smaller than for individual BCH. In the DALS case, this relative speedup is even larger at up to 8.98.

We attribute the better scalability of collective searches to two main advantages: Firstly, collective searches can be pruned more precisely as we use lower bounds on the cost of specific PD-pairs or dropoffs instead of a general lower bound on the cost of every PD-pair or dropoff. This applies to the stopping criteria for bucket scans and for the searches as a whole. Secondly, collective searches consider all sources in one search, maximizing the amount of information available for pruning. Bundled searches can only consider k searches at once which means work may be repeated up to N_ρ^p/k times. Thus, the number of edge relaxations and bucket entry scans increases much faster with the number of pickups ($N_\rho^p, N_\rho^d \sim \rho^2$) for individual BCH searches than for collective BCH searches.

In addition, the enumeration times remain small for collective searches while they increase massively with ρ for individual BCH searches. This is due to the fact that collective searches

Table 4.2 Comparison of the PALS and DALS running times (in µs) of collective BCH searches (Coll.), individual BCH searches (BCH), and Dijkstra searches (Dij.) in their optimal configurations for three radii $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ on the `B-1%` and `B-10%` instances. Shows average number of edge relaxations ($\#_{\text{rel.}}$), number of bucket entries scanned ($\#_{\text{scans}}$), search time (t_{search}) and time for enumerating insertions (t_{enum}) per request. The smallest times per radius are marked in gray.

			Berlin-1pct				Berlin-10pct			
Type	ρ	Search	$\#_{\text{rel.}}$	$\#_{\text{scans}}$	t_{search}	t_{enum}	$\#_{\text{rel.}}$	$\#_{\text{scans}}$	t_{search}	t_{enum}
PALS	0	Coll.	40	10	3.0	0.0	19	14	2.7	0.0
		BCH	37	9	2.4	0.1	18	14	2.5	0.2
		Dij.	620	3	31.0	0.2	276	11	19.9	0.4
	300	Coll.	143	21	18.6	0.0	54	55	12.5	19.8
		BCH	1006	307	51.0	38.3	697	695	53.5	133.2
		Dij.	7157	24	509.9	43.3	5684	151	410.6	138.8
	600	Coll.	184	26	54.2	0.0	55	120	42.3	189.7
		BCH	4116	1690	207.6	547.7	2760	4591	240.2	2047.9
		Dij.	65 117	186	3972.8	557.0	58 306	1301	3614.4	1964.9
DALS	0	Coll.	209	278	21.7	0.6	191	1579	63.4	3.0
		BCH	214	278	13.6	0.4	197	1580	53.6	2.4
		Dij.	20 412	134	1125.2	8.6	16 043	1044	920.8	56.5
	300	Coll.	223	249	32.3	4.7	206	1368	72.6	19.8
		BCH	1255	823	81.8	83.8	1226	4748	224.3	275.3
		Dij.	29 353	182	2436.2	227.6	24 124	1439	2110.3	846.5
	600	Coll.	250	249	54.1	10.7	229	1366	96.4	42.1
		BCH	6543	4245	326.7	425.5	5885	23 498	865.4	1246.6
		Dij.	87 004	501	6180.4	1363.3	68 401	3761	5029.7	4092.7

filter candidate insertions during the search, identifying a single insertion in the PALS case or a small set of candidate vehicles and dropoffs in the DALS case. Contrarily, individual BCH searches first find all distances and then enumerate an insertion for each combination of candidate vehicle, pickup and dropoff. As the number of PD-pairs is proportional to ρ^4, enumeration times of individual BCH searches quickly grow very large.

Collective searches scale worse in the PALS case compared to the DALS case because many more initial labels need to be created. In the PALS case, one label is initialized for every PD-pair but most of these labels are pruned immediately based on cost or domination pruning. For instance, an average of 81.7% of initial labels are discarded right away for $\rho = 600\text{s}$ on the `B-10%` instance. Checking and pruning these labels makes up 88.6% of the search time of collective searches in the PALS case. In the DALS case, we only need to initialize one label per dropoff, which vastly reduces this overhead for pruning initial labels.

4.9.4. Comparison with Baseline Dispatcher

In this section, we compare KaRRi with the baseline dispatcher LOUD [BSW21]. For this comparison, we implemented KaRRi's cost function in the baseline dispatcher.

Running Times. We give the running times for the different phases of both our algorithm (K) and the baseline (B) on all instances in Table 4.4 (big table, moved to end of section).

First, we consider the scenario without meeting points ($\rho = 0$s) and compare KaRRi with the baseline dispatcher. Note that KaRRi has an overhead of up to 57µs for finding pickups and dropoffs even with $\rho = 0$s. This is because we include the time for computing the direct walking distance from origin to destination. Computing the PD-distances consists of a single CH query for $\rho = 0$s, so it is fast for both the baseline and KaRRi. For the elliptic BCH searches, KaRRi has a slight advantage due to the sorted buckets. The sorted buckets also filter candidate insertions which reduces the amount of time needed to enumerate *ordinary* and PBNS insertions. Our last stop BCH searches are well suited for $\rho = 0$s. They are up to one and two orders of magnitude faster than the baseline Dijkstra searches in the PALS and DALS cases, respectively. For the baseline, last stop searches make up between 63% and 96% of the total running time while our last stop searches make up at most 9% of the total running time of KaRRi. Maintaining sorted buckets leads to slightly increased update times. In total, we can reduce the average time per request by factors of about 4.7, 2.2, 31.5, and 8.1 for the `B-1%`, `B-10%`, `R-1%`, and `R-10%` instances, respectively, compared to the baseline.

Unfortunately, the source code of KaRRi's closest existing competitor STaRS+ [Mou+21] is not publicly available, making an experimental comparison difficult. Instead, we validate the effectiveness of our approach by comparing it with a naïve extension of the techniques used by our baseline algorithm. For this, we configured KaRRi to use no bundled searches or sorted buckets, to use Dijkstra searches for the PALS and DALS cases, and to use point-to-point CH queries to compute PD-distances. We report the running times of this extension (B*) for $\rho = 300$s and $\rho = 600$s on the `B-10%` instance and compare them to KaRRi.

We find that PD-distances can be computed around two orders of magnitude faster with our bucket based approach than with individual CH queries. Bundling the elliptic BCH searches and using sorted buckets makes them about one order of magnitude faster than the naïve extension. Our collective searches for the PALS and DALS cases beat the naïve approach by between two and three orders of magnitude.

Running Times with CCHs. We also equipped KaRRi with the possibility to use customizable CHs (CCHs) [DSW16], a technique that allows a CH to quickly be adapted to changing travel times in the road network at the cost of a slight increase in shortest-path query times. This extension to KaRRi was straightforward and all of our many-to-many routing techniques can still be used. We experimentally evaluated KaRRi-CCH by running it on `Berlin-1pct` and `Berlin-10pct` with $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ in the same configurations as with standard CHs. The resulting running times are shown in Table 4.4 (K-CCH).

KaRRi-CCH is slightly faster than KaRRi with CHs for the `Berlin-1pct` instance. In particular, CCHs are faster for the PD-distance queries and elliptic BCH queries for $\rho > 0$s. We attribute this to the fact that we can employ elimination tree searches, a specialized type of query for CCHs. These queries are often better suited for bundling than standard CH queries [BSW19], which explains the advantage for larger ρ. Since we use collective last stop searches, this does not come into effect for PALS and DALS.

For `Berlin-10pct`, KaRRi-CCH is faster than KaRRi for PD-distance searches but slower for elliptic BCH searches. We attribute this to the increased number of vehicles which lead to more bucket entries per elliptic bucket. Since elimination tree searches cannot be pruned as effectively as standard CH queries, they may scan the buckets of more vertices. This becomes more significant for `Berlin-10pct` due to the increased number of bucket entries.

Overall, the running time of KaRRi increases by at most 0.6ms per request when using

Table 4.3 Solution quality of KaRRi with different radii ($\rho \in \{0s, 300s, 600s\}$) on `B-1%`, `B-10%`, `R-1%`, and `R-10%`. For riders, we report the average wait and trip times (in mm:ss), as well as the percentage of travelers choosing ride-pooling. For vehicles, we give the average occupancy while driving and average total operation time (in hh:mm).

Inst.	ρ	wait	trip	% ride	occ	op
`B-1%`	0	3:49	17:07	98.5%	0.90	4:31
	300	3:18	16:08	98.9%	0.93	4:12
	600	3:18	16:08	98.9%	0.94	4:11
`B-10%`	0	2:44	15:48	98.4%	1.08	3:20
	300	2:26	15:09	98.8%	1.16	3:02
	600	2:26	15:08	98.8%	1.16	3:01
`R-1%`	0	5:42	20:35	83.0%	0.82	4:34
	300	4:55	19:05	84.4%	0.85	4:17
	600	4:52	18:55	84.5%	0.85	4:15
`R-10%`	0	3:22	16:40	87.8%	0.95	3:27
	300	2:50	15:35	89.4%	0.99	3:15
	600	2:50	15:32	89.4%	0.99	3:14

CCHs instead of standard CHs. At the same time, CCHs allow us to adapt our CH to changed travel times in the road network in less than 100ms. Thus, KaRRi-CCH combines fast dispatching with the ability to react to changing traffic conditions in real time.

Solution Quality. We give a limited analysis of the impact of meeting points on trip times and vehicle operation times like in the original paper on KaRRi [LS24]. Here, every traveler chooses between walking and ride-pooling based on our cost function. We conduct a much more detailed analysis of ride-pooling at scale based on KaRRi in Chapter 6.

In Table 4.3, we compare the solution quality of KaRRi with $\rho \in \{0s, 300s, 600s\}$. We give the average wait time and trip time of all travelers that use ride-pooling. Additionally, we report the average occupancy and operation time of each vehicle. At $\rho = 300s$, we observe improvements for both riders and vehicles. The average wait time and trip times decrease by up to 47 seconds and 90 seconds, respectively. At the same time, the average occupancy of vehicles increases and the average vehicle operation time decreases. Thus, meeting points lead to better pooling and improved rides. For `Berlin-1pct` and `Berlin-10pct`, almost all travelers choose ride-pooling even at $\rho = 0s$. However, for `Ruhr-1pct` and `Ruhr-10pct`, the rate of travelers choosing ride-pooling increases noticeably with meeting points.

At $\rho = 600s$, the quality improves only slightly compared to $\rho = 300s$. We believe this is due to the detours already being as efficient as necessary for the given demand. If there are more requests or fewer vehicles, a larger walking radius may lead to additional benefits.

4.10. Conclusion

KaRRi develops efficient many-to-many routing with bucket contraction hierarchies for dynamic ride-pooling. It improves on the state-of-the-art for the base problem and allows fast dispatching with meeting points, which promise benefits like a reduction in operating costs and air pollution. KaRRi's small running times open dynamic ride-pooling up to a variety of

extensions that promise to improve the quality of service, some of which are discussed in later chapters of this work.

We believe that there is more interesting work to be done for meeting points. Not all meeting points in a radius around the origin and destination may be viable or required candidates. A location may be unsuited for a pickup or dropoff due to unsafe conditions or lack of space. On the other hand, there may be locations that are suitable as meeting points but are never needed for optimal assignments, as they are dominated by better meeting points in the vicinity. Identifying the correct set of meeting points remains an open problem.

Additionally, meeting points provide an avenue for optimizing the number of stops that a vehicle needs to make. For example, when a new ride request is accepted, the dropoff of an existing rider may be changed on-the-fly to coincide with the pickup of the new rider. When considering pre-booked trips and these opportunities to transparently change existing trips, we could intermittently re-optimize vehicle schedules using a local search approach.

Using customizable contraction hierarchies with KaRRi allows routing with fast updates to edge travel times in the network. However, in the case of such an update e.g. due to congestion, the schedules of vehicles are affected as well. In the future, we would like to incorporate fast updates of the vehicle schedules to obtain a customizable ride-pooling dispatcher. It would also be interesting to study the potential degradation in solution quality due to updated travel times and explore opportunities for re-optimizing assignments.

Table 4.4 Running times (in µs) of different phases of the baseline (B), naïvely extended baseline (B*), KaRRi (K), and KaRRi-CCH (K-CCH) with different radii ($\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$) on `B-1%`, `B-10%`, `R-1%`, and `R-10%`. Shows mean times for finding P_ρ and D_ρ, PD-distance searches, elliptic BCH searches, enumerating *ordinary* and PBNS insertions, PALS and DALS searches, and updating routes and buckets as well as the mean total time per request.

Inst.	ρ	Alg.	find P_ρ, D_ρ	PD	Ell. BCH	Ord.& PBNS	PALS	DALS	update	total
`B-1%`	0	B	0	13	83	52	39	1129	88	1404
		K	53	14	72	21	3	14	122	298
		K-CCH	35	10	79	20	3	19	117	283
	300	K	278	155	372	133	21	37	121	1118
		K-CCH	259	105	312	132	22	62	120	1012
	600	K	814	998	1608	607	68	65	126	4286
		K-CCH	796	691	1075	605	67	88	126	3449
`B-10%`	0	B	0	13	276	168	23	890	85	1453
		K	57	14	225	134	3	55	167	654
		K-CCH	39	12	339	152	4	108	284	939
	300	B*	298	21 399	11 828	609	3132	33 034	79	70 379
		K	303	165	1069	553	40	92	177	2400
		K-CCH	302	116	1412	626	44	152	340	2992
	600	B*	877	349 210	45 827	2405	32 230	82 864	85	513 498
		K	867	1082	4514	2330	289	138	193	9414
		K-CCH	870	750	4848	2353	293	195	381	9691
`R-1%`	0	B	0	16	147	136	147	8751	94	9290
		K	17	14	103	32	4	18	107	295
	300	K	131	124	376	114	25	37	111	917
	600	K	348	581	1137	372	65	57	116	2675
`R-10%`	0	B	0	14	530	514	90	6428	82	7660
		K	19	16	396	267	3	77	170	948
	300	K	137	129	1464	600	33	106	193	2663
	600	K	347	597	4129	1481	201	137	205	7097

5 Parallel Batched Dynamic Ride-Pooling

Summary: Future ride-pooling systems may serve millions of customers with tens of thousands of vehicles. Current ride-pooling dispatchers do not scale to these numbers. We introduce Mt-KaRRi, a batch-based multi-threaded version of our dispatcher KaRRi. We show that a simple synchronization scheme suffices to process requests concurrently. In experiments, we show that our parallelization causes almost no loss in quality. We observe speedups of up to 53 with 96 threads, which allows real-time dispatching for an input instance representing the whole population of the Los Angeles metropolitan area.

Attribution: This chapter is based on parts of the paper "Advancing Dynamic Ride-Pooling Simulation – A Highly Scalable Dispatcher" [Lau+26], co-authored with Robin Andre, Kim Kandler, Peter Sanders, and Peter Vortisch. The parts of the paper relevant for this chapter were written by the author of this dissertation, with editing by Robin Andre, Peter Sanders, and Peter Vortisch. The algorithmic ideas for this chapter were developed by the author of this dissertation. The author of this dissertation conducted the implementation of the code and the experimental evaluation. Robin Andre and Kim Kandler contributed the demand data used in the experimental evaluation.

5.1. Introduction

Ride-pooling is seen as an important building block of future transportation systems in urban areas. In particular, cities without a well-established transit system could benefit from the promise of flexible shared transportation on car infrastructure. In the case of wide-scale adoption, we expect ride-pooling services to receive up to millions of travel requests per hour in large metropolitan areas. This level of demand may exceed the capabilities of sequential dispatchers. Our state-of-the-art sequential dispatcher KaRRi can process at most about 12 million requests per hour (see Table 4.4), but larger vehicle fleets or additional features like meeting points decrease the throughput.

In this chapter, we explore Mt-KaRRi, a multi-threaded version of KaRRi, that processes requests in batches and finds insertions for each request in a batch in parallel. This parallelization comes with the issue of synchronizing changes to the vehicle routes. If multiple requests in the same batch are assigned to the same vehicle, their insertions can conflict with each other. We propose a simple re-assignment procedure to synchronize assignments.

In an experimental evaluation, we analyze the scalability of our approach and the impact of batching on the solution quality. We show that small batches lead to negligible quality loss but provide sufficient parallelism to speed up the dispatching process. Our approach allows real-time dispatching with tens of millions of requests in a single morning rush hour.

5.1.1. Related Work

We give an overview on research into ride-pooling in Chapter 2. As described in Section 4.1.1, most work on efficient dispatching of ride-pooling vehicles focuses on reducing the number of potential assignments [HS02; Jaw+86; MRR95] or heuristically limiting the set of candidate vehicles [BMN17; HNA16; Hua+14; MZW13; MZW15].

Parallel computing remains underutilized in this space. Some publications on related vehicle routing problems use meta heuristics and tailor generic parallelization approaches for these meta heuristics to their use case (e.g. simulated annealing [CC02] and tabu search [Att+04]). Alonso-Mora et al. [Alo+17] parallelize the construction of an ILP for static ride-pooling and mention that black-box ILP solvers employ parallelism, which is likely implicitly utilized by most works on static ride-pooling that use ILP solving as a backend.

Ota et al. [Ota+15; Ota+17] identify two approaches to parallelization for dynamic ride-pooling. An *intra-request* parallelization attempts to enumerate potential assignments for a single request in parallel. This approach does not work well in combination with more recent advancements that reduce the number of potential assignments [BSW21; LS24], as the reduced number of assignments limits the available parallelism per request. An *inter-partition* parallelization divides a long observation period into independent sub-periods, so that the simulation can be performed in parallel for each of the sub-periods. This requires natural cuts in the demand where the ride-pooling activity in one sub-period does not affect the activity in the next sub-period. Ota et al. [Ota+15; Ota+17] simulate ride-pooling in New York City for a whole year and identify each day as a suitable sub-period with phases of little activity in the early morning hours of each day acting as the required natural cuts. Inter-partition parallelization can be applied for any long-term ride-pooling simulation, but it does not increase the throughput of a dispatcher, i.e., the rate at which requests can be answered. Thus, this approach cannot solve potential bottlenecks in production systems caused by many incoming requests in a short time period.

A third approach would be *inter-request* parallelization, i.e., processing multiple requests in parallel. This can increase the throughput of a dispatcher and is independent of the amount of work per request. However, as Ota et al. [Ota+15; Ota+17] point out, each processed request r_i in a sequence of requests $\langle r_1, \ldots, r_k \rangle$ affects the vehicle routes and thus the best assignments of future requests $\langle r_{i+1}, \ldots, r_k \rangle$. Therefore, an inter-request parallelization cannot result in the exact same solution as sequential processing for a given sequence of requests. However, dynamic ride-pooling can only be solved heuristically, so diverging from a sequential solution is acceptable if solution quality does not deteriorate. As a remaining issue, requests that are processed in parallel may affect each other's best assignments. For instance, if request r_1 and request r_2 are processed in parallel and their respective best assignments use the same vehicle ν, then synchronization is needed to find out whether both requests can be assigned to ν or whether one request needs to use a different vehicle.

Gidofalvi et al. [Gid+08] propose an inter-request parallelization that solves this issue by partitioning requests and existing vehicle trips by origin and destination location. They interpret each active new request as well as each existing vehicle passenger as a point in four-dimensional space (origin longitude, origin latitude, destination longitude, destination latitude). The 4D-space is partitioned into p blocks (where p is the number of processing units) such that each block has roughly the same number of requests. Then, only requests and existing passengers in the same partition are allowed to be grouped, which allows parallelization over the partitions. Partitioning travelers by origin and destination location is a promising approach for identifying riders that are unlikely to synergize well, but fully excluding pooling over partition boundaries may miss beneficial groupings. Additionally, a

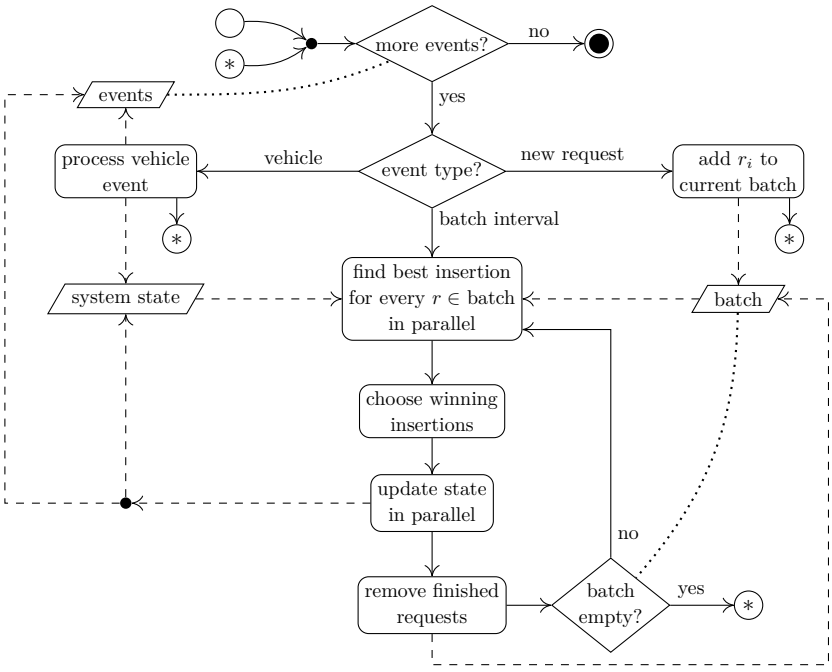


Figure 5.1 Workflow of Mt-KaRRi. Solid arrows indicate control flow. Dashed arrows indicate data reads and writes. Dotted lines indicate information for decisions. The node ⊛ signifies a loop to the next event.

single vehicle should be able to plan ahead and accept riders whose origin is close to the destination of current passengers. In this case, the vehicle route crosses partition boundaries, so it cannot be easily assigned to just one partition.

5.2. Parallelization of KaRRi

We introduce a thread-based parallelization of KaRRi called Mt-KaRRi. The new algorithm computes the best insertions for a batch of requests in parallel and performs a parallelized update of the route state that incorporates the changes for the whole batch. We illustrate the updated workflow of the event simulation with batching in Figure 5.1.

Batching Requests. To find batches of requests that can be dispatched in parallel, we partition the observation period into intervals of fixed length t_{batch}. The event simulation always contains a "batch interval" event at the end of the next interval. When a "new request" event occurs, the request is not immediately dispatched but only added to the current batch. Then, when a "batch interval" event occurs, the algorithm dispatches all requests in the current batch in parallel. This is akin to a rolling-horizon approach as outlined in Section 2.3.2, but we do not consider existing requests for re-optimization when dispatching the new requests.

For this parallel process, we start one worker thread on each processing unit of the processor. We assign each request in the batch to one worker thread that computes all shortest-path distances and the best insertion for this request, just as KaRRi would in the single-threaded setting. These computations only read the current route state and distances in BCH buckets and do not change this information. Therefore, there cannot be any race

conditions between threads in this part.

Conflicting Insertions However, conflicts can arise if the best insertions for multiple requests use the same vehicle. Assume that both the best insertions $\mathcal{I}_1$ for request r_1 and $\mathcal{I}_2$ for request r_2 use vehicle ν. Performing $\mathcal{I}_1$ or $\mathcal{I}_2$ can change the route of ν, so that the other is no longer the best insertion or even becomes infeasible.

We resolve these conflicts by postponing and finding new insertions for conflicting requests in an iterative manner. Assume that the best insertions for a set of requests $\{r_1, \dots, r_x\}$ all require the same vehicle. We arbitrarily pick one of the requests (here: r_1) and mark its insertion as winning. All other requests $r_2, \dots, r_x$ are postponed. After determining all winning insertions of the batch, we perform a parallelized collective update of the route state and bucket entries for these insertions. Then, we find new best insertions for all postponed requests. Since the insertion of r_1 is now represented in the route state, requests r_2 to r_x can no longer conflict with r_1. We iterate this method until all conflicts are resolved, which constitutes the inner loop at the bottom in Figure 5.1.

Choosing the batch length results in a trade-off between available parallelism and quality deterioration due to conflicts and increased wait times. However, we find that small values of t_{batch} lead to a small loss in quality while providing enough parallelism for the large instances where parallelization is most important (see our experiments in Figure 5.2). Note that many different conflict resolution strategies could be implemented. In fact, what we call a conflict here is actually a chance to pool requests, and a batch strategy can try to utilize synergies within a batch. This is akin to rolling-horizon strategies for dynamic ride-pooling, which have been shown to reach better solution quality than pure sequential insertion (see Section 2.3.2). In the future, we would like to find an efficient assignment procedure that utilizes batching for better solution quality compared to our current greedy approach.

5.3. Experiments

In this section, we experimentally analyze the quality and scalability of Mt-KaRRi.

As the ride-pooling use patterns (number of riders, distribution of vehicles, etc) have an impact on the running time of the dispatcher, it makes sense to analyze scalability in the most realistic setting available. Therefore, the scalability experiments in this section also use the mode choice model outlined in Chapter 6. Note the consequence that travelers may use other modes of transportation, which reduces the overhead for conflict resolution.

5.3.1. Setup

Our source code[1] is written in C++ 20 and compiled with GCC 11.5 using `-O3`. We use multi-threading primitives from the Intel OneAPI Threading Building Blocks library (version 2021.13.1). We use a machine running Rocky Linux 9.4 with an AMD EPYC 9684X processor (96 physical cores) and 1.6 TiB of RAM. Note that even on our largest input instances, we actually need only 60 GiB of RAM. We use 32-bit distance labels, so KaRRi's vectorized searches using the AVX2 SIMD instruction set with 256-bit registers allow for eight simultaneous searches.

Unless otherwise specified, we use the following default assumptions: We use the model parameters decided on in the parameter tuning experiments in Section 10.3 and $t_{\text{batch}} = 5\text{s}$.

[1] Available at `https://github.com/molaupi/karri/tree/mt_karri_batch_length_walking`.

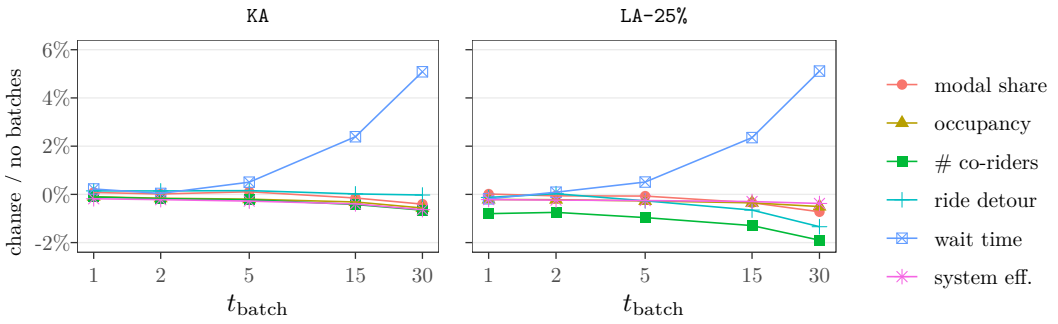


Figure 5.2 Change in quality for different t_{batch} compared to the non-batched version of KaRRi on `KA` and `LA-25%` with 50 000 and 125 000 vehicles, respectively. Shows the change for RP modal share, occupancy, and number of co-riders, as well as ride-detour, wait time, and system effectiveness. For explanations of the metrics, see Section 6.3.3. Note the logarithmic x-axis.

Vehicles have a default capacity of four, and the initial location of every vehicle is an edge drawn uniformly at random in the road network of the service area. The service time of every vehicle is set to encompass the entire observation period. We specify the number of runs per experiment and report average metric values over all runs.

Input Instances. We give an overview of the demand instances used in this section. A detailed description of how we generate demand data can be found in Section 6.3.1. We analyze our code on the road networks of the German cities of Karlsruhe and Stuttgart, as well as Los Angeles in the US. Table 6.2 provides the size of each network. An overview of the demand sets can be found in Table 6.1. Our request sets for Karlsruhe (`KA`) and Stuttgart (`ST`) cover a morning peak of a weekday comprising about 1 million and 1.4 million requests in three hours respectively. For Los Angeles (`LA-`p`%`), we have seven different request sets of increasing size. Each set is based on demand data for a representative sample of the population constituting $p\%$ of the total population for $p \in \{0.1, 1, 5, 10, 25, 50, 100\}$. The request set based on the full population (`LA-100%`) includes about 25 million requests in a nine hour time window with more than 6 million requests in a single peak hour.

We use meeting points for the `KA` and `ST` instances, but not for the LA instances as we lack a good pedestrian network for the observation area.

5.3.2. Effect of t_{batch} on Solution Quality

We compare the quality of assignments produced by Mt-KaRRi to the quality of sequential KaRRi. We use quality metrics designed for our detailed study of ride-pooling at scale in Chapter 6. Thus, a better explanation of these metrics can be found in Section 6.3.3.

We show the quality change for different values of t_{batch} compared to the non-batched version for the `KA` and `LA-25%` instances in Figure 5.2. The plots show averages of three runs.

The values of most metrics exhibit a linear relationship with the length of batch intervals (note the logarithmic x-axis). The waiting time increases the most, since a traveler may have to wait up to t_{batch} to be processed as part of the next batch. This also causes a slight drop in the modal share, occupancy, and number of co-riders for larger values of t_{batch}. Noticeably, even at $t_{\text{batch}} = 1$s, there is a negative offset in pooling (number of co-riders) compared to the non-batched version, which is more pronounced for the larger `LA-25%` instance. This can be attributed to the order in which requests are processed. Consider the latest request

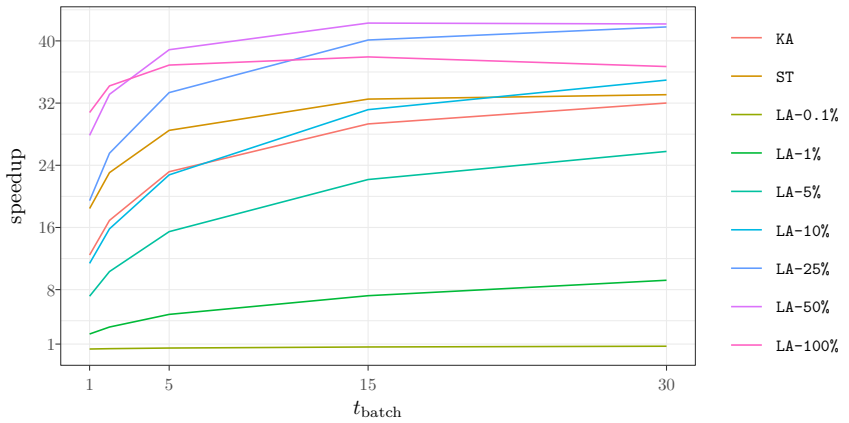


Figure 5.3 Average speedup of Mt-KaRRi with $T = 96$ threads over sequential KaRRi for batch lengths $t_{\text{batch}} \in \{1, 2, 5, 15, 30\}$. The `LA-`$p$`%` instances use $p \cdot 5000$ vehicles, and the `KA` and `ST` instances use 50 000 vehicles.

r of a batch. In the sequential setting, all other requests in the batch have already been processed and could be pooled with r. In the batched setting, the other requests have not been processed yet, and we might miss a synergy between r and the other requests. Thus, in the batched setting, there are always slightly more non-pooled rides than without batches.

5.3.3. Impact of Batch Interval t_{batch} on the Running Time

We first evaluate how the value of t_{batch} affects the running time of the batch-parallel dispatcher. For this, we show the speedup over sequential KaRRi for $T = 96$ threads and varying t_{batch} in Figure 5.3. We report results for a single run for `LA-50%` and `LA-100%`, averages of three runs for `LA-25%`, and averages of five runs for all other instances. As sequential KaRRi would have infeasible running times on large instances, we obtain a sample of the running times of KaRRi by running the parallel Mt-KaRRi and solving every one-hundredth request using the sequential algorithm. The speedups for the large instances (`LA-25%` and larger) use this sample as the sequential baseline.

We observe that even small values of t_{batch} suffice for good speedups with $T = 96$ threads. For the largest instances (`LA-50%` and `LA-100%`), batches are already large enough at $t_{\text{batch}} = 5$s to provide sufficient parallelism, with speedups of 36 or larger. Beyond $t_{\text{batch}} = 5$s speedups increase only slightly for large instances. For smaller instances, speedups are worse due to limited parallelism available in every batch. This is not a drawback in practice, as smaller instances already have shorter running times. Since there is a trade-off with quality, we choose to use $t_{\text{batch}} = 5$s as the default value for fast and high-quality parallel dispatching.

5.3.4. Weak Scaling

We experimentally evaluate the scalability of Mt-KaRRi to show that our dispatcher can solve large instances quickly. We first consider *weak scaling*, which describes how the running time varies for an increasing number of threads and a fixed problem size *per thread*. We conduct only a single run of experiments in this section, since scaling experiments take a long time, and one run suffices to see the trend in running times. We run sequential KaRRi and parallel Mt-KaRRi on `LA-`p`%` for $p \in \{1, 5, 10, 25, 50, 100\}$ to scale the number of requests proportionally to p. We also increase the number of vehicles proportionally. For Mt-KaRRi,

Table 5.1 Speedup of Mt-KaRRi over sequential KaRRi with proportionally increasing instance size (number of requests # *req.* and number of vehicles $|F|$) and number of threads (T), plus two additional configurations for the largest instance. Shows speedups for finding ride-pooling insertions (find ins.), updating the system state for accepted rides (update), and the total running time (total). Additionally shows average running time per request (in ms) for KaRRi (t_{seq}) and Mt-KaRRi (t_{par}).

instance	T	# req. [$\cdot 10^3$]	$\lvert F\rvert$ [$\cdot 10^3$]	find ins. (speedup)	update (speedup)	total (speedup)	t_{seq} [ms]	t_{par} [ms]
LA-1%	1	254	5	0.97	0.92	0.96	0.65	0.68
LA-5%	5	1271	25	4.32	3.71	4.21	2.31	0.55
LA-10%	10	2544	50	8.21	4.60	7.65	4.58	0.60
LA-25%	25	6359	125	16.92	8.37	15.68	12.60	0.80
LA-50%	50	12 719	250	29.10	9.32	26.71	30.68	1.15
LA-100%	96	25 432	500	39.43	13.69	36.89	68.00	1.84
LA-100%	96	25 432	5	70.84	5.14	49.72	2.06	0.04
LA-100%	96	25 432	50	60.54	10.62	53.65	19.00	0.35

we start with one thread for the LA-1% instance and increase the number of threads in proportion to p. We report the speedups of Mt-KaRRi over sequential KaRRi in Table 5.1. Remark two caveats: First, we use 96 threads for the LA-100% instance, since the machine used in the experiments only has 96 cores. Second, the sequential baseline for large instances (LA-50% and LA-100%) uses only a sample of requests as in Section 5.3.3.

We see good speedups for LA-p% with p threads for $p \leq 10$. Here, the efficiency (speedup divided by number of threads) is at least 0.82 for finding the best insertions. The work for updating the system state is not parallelized well and the efficiency is much lower. As most of the time is spent finding insertions, we still see good speedups of the total running time (efficiency ≥ 0.76).

The scalability appears to reach a limit for larger instances, with speedups of the total running time of only 15.68, 26.71 and 36.89 with 25, 50 and 96 threads, respectively. For an explanation, we see multiple contributing factors. Firstly, since we use the same batch length of $t_{\text{batch}} = 5$s for all instances, the number of requests per batch increases proportionally to p for the LA-p% instances. Thus, there are more conflicting insertions per batch, which leads to additional iterations in our conflict resolution approach and more re-computations of insertions. We can see the extent of this effect by normalizing running times with the total number of insertions computed instead of the number of requests. Even then, the speedups for LA-50% and LA-100% would be only 35.77 and 53.05, though. This brings us to the second reason for the limited scalability. Consider the running time per request for KaRRi (t_{seq}) and Mt-KaRRi (t_{par}) given in Table 5.1. For the sequential algorithm, the running time per request increases in proportion to the size of the instance. This is because there are more vehicles and more riders, which means that the BCH buckets have more entries and each shortest-path query becomes more complex. For the parallel algorithm, the increased number of threads is not able to compensate for this effect as we see an increasing running time per request with larger instances. We believe that the parallel algorithm is affected more by the increased complexity of the BCH data structure because the uncoordinated concurrent reads of the buckets lead to contention on the L3 cache, which is shared between all threads. To gain an understanding of the speedups possible without this drawback, we include the

Table 5.2 Average running times per request (in µs) for KaRRi without batching (*no batches*) and Mt-KaRRi with different numbers of threads (T) on `LA-1%` (left) and `LA-25%` (right) with 5000 and 50 000 vehicles, respectively. Shows average running time per request for finding the best ride-pooling insertion (ins.), updating the system state of the ride-pooling dispatcher (update), other work (other), and the total time (total).

	LA-1%				LA-25%			
T	ins.	update	other	total	ins.	update	other	total
no batches	553	89	3	646	8363	1123	5	9491
1	576	97	7	680	9460	256	7	9724
4	153	48	8	208	2206	93	7	2306
8	88	41	7	137	1253	68	7	1328
16	61	53	8	122	643	79	7	729
32	51	55	8	114	336	76	7	420
64	54	60	8	121	203	76	7	286
96	60	65	8	133	165	77	7	250

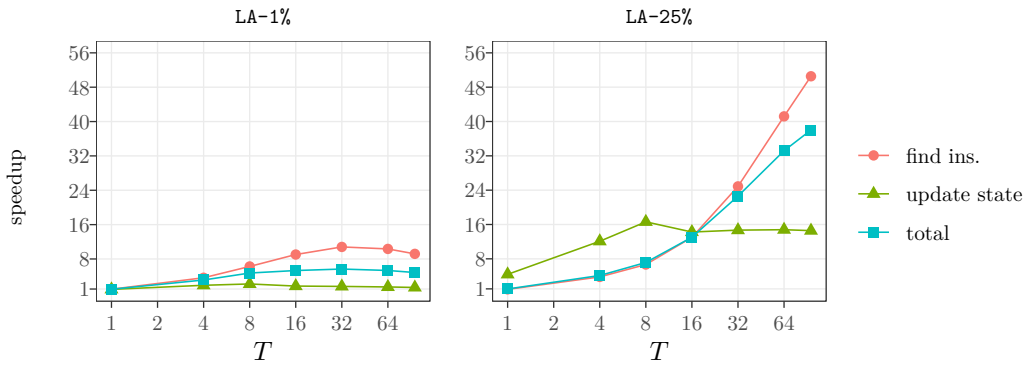


Figure 5.4 Speedups of Mt-KaRRi over sequential KaRRi with different number of threads (logarithmic x-axis) on `LA-1%` with 5000 vehicles (left) and `LA-25%` with 50 000 vehicles (right). Shows separate speedups for finding insertions, updating the system state, and the total running time.

last two rows in Table 5.1 where we consider `LA-100%` with smaller fleets. Although the speedups for updating the system state are still limited, we see much better speedups for finding insertions of 70.84 for 5000 vehicles and 60.54 for 50 000 vehicles, which leads to total speedups of 49.72 and 53.65.

Thus, a potential avenue for optimization is an improved BCH data structure and more coordinated access to this data structure to improve locality. For example, requests within the same batch may be ordered so that requests with origin and destination locations that are close to each other are processed at the same time. Due to this proximity in the road network, the BCH queries for these requests would access similar buckets, improving the locality of the memory reads.

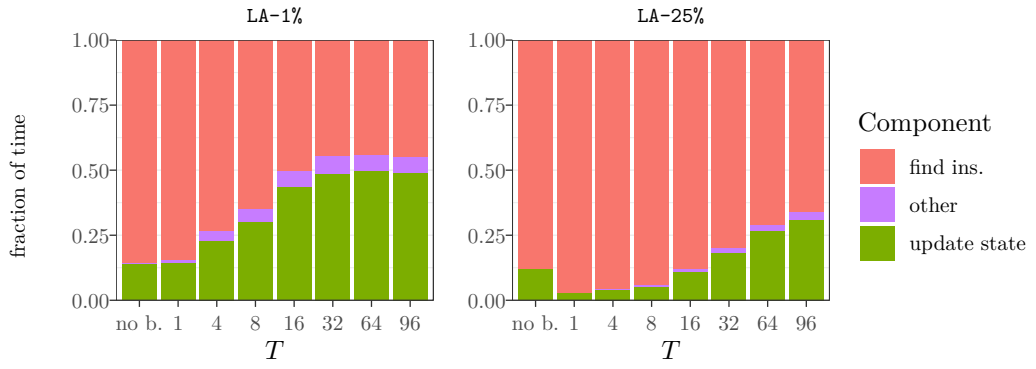


Figure 5.5 Fraction of running time per component of KaRRi (*no b.*) and Mt-KaRRi with T threads on `LA-1%` (left) and `LA-25%` (right) with 5000 and 50 000 vehicles, respectively.

5.3.5. Strong Scaling

We analyze the performance of Mt-KaRRi for an increasing number of threads on `LA-1%` and `LA-25%` with 5000 and 50 000 vehicles, respectively. We use $T \in \{1, 4, 8, 16, 32, 64, 96\}$ threads and evaluate the speedups compared to sequential KaRRi. We perform only a single run per configuration since the runs with smaller numbers of threads take a long time.

We show absolute running times in Table 5.2 and speedup plots in Figure 5.4. We only parallelized some parts of the system state updates, which is why there are no or very limited speedups for this component. We focused our efforts on the computation of insertions. For the smaller `LA-1%` instance, there is an average of 39.7 requests in every batch, which is not enough to utilize all threads or provide good load balancing (we use the default of $t_{\text{batch}} = 5$s here). Thus, the speedups for the computation of insertions is limited to a factor of at most 10.80 for `LA-1%`, which is reached at $T = 32$ threads The larger instance provides larger batches with an average of 981.6 requests per batch. With this, Mt-KaRRi with $T = 96$ threads computes insertions 50.54 times faster than KaRRi. The total speedup is at most 5.66 and 38.01 at $T = 32$ and $T = 96$ threads on the `LA-1%` and `LA-25%` instances, respectively.

As shown in Figure 5.5, the updates of the system state make up only about 14.3% and 11.9% of the total running time in the sequential setting for `LA-1%` and `LA-25%`, respectively. This is why we initially focused our parallelization efforts on the dominant task of computing insertions. However, with $T = 96$ threads the system state update contributes a significant part of the running time. In the future, we aim to improve the parallelization of these components to allow for good scalability to even larger numbers of threads.

5.4. Conclusion

We introduce Mt-KaRRi, a multi-threaded version of the dynamic ride-pooling dispatcher KaRRi. We show that a batching approach with a simple synchronization procedure based on re-assignments suffices for good scalability with a limited loss in solution quality compared to a sequential solution. In our experimental evaluation, we show that Mt-KaRRi can solve input instances with millions of requests per hour in real time.

In the future, we might achieve higher quality by processing batches of requests in a more informed way. For example, we could compute several possible assignments per request and solve a weighted matching problem to select vehicles. More sophisticated models might handle the case of assigning multiple requests to a single vehicle.

6 Analysis of Ride-Pooling in Large-Scale Transportation Systems with Multiple Modes of Transportation

Summary: Existing experimental studies on ride-pooling do not model whole transportation systems or use only small to medium size input instances, in part due to the limited scalability of ride-pooling dispatchers. This leaves a research gap in analyzing ride-pooling as an integrated part of metropolitan scale transportation systems.
We use our scalable dispatcher Mt-KaRRi to study ride-pooling at a huge scale with up to millions of requests per hour. We implement a mode choice model that lets us evaluate ride-pooling in competition with walking, transit, and private cars. We study the effects of the number of vehicles, the size of the customer base, the use of meeting points, and the location of requests (urban vs peripheral). Our experiments show that Mt-KaRRi is suited for the detailed analysis of ride-pooling at unprecedented scales.

Attribution: This chapter is based on parts of the paper "Advancing Dynamic Ride-Pooling Simulation – A Highly Scalable Dispatcher" [Lau+26], co-authored with Robin Andre, Kim Kandler, Peter Sanders, and Peter Vortisch. Most of the original paper was written by the author of this dissertation. The description of the mode choice model and of the extraction of request data from the demand sources were originally written by Robin Andre. The related work section of the original paper was written by Kim Kandler. Peter Sanders and Peter Vortisch contributed editing to all parts of the paper. Kim Kandler calibrated the mode choice model. The setup and design of the experimental study was developed jointly by Robin Andre and the author of this dissertation, with discussions and feedback by Peter Sanders and Peter Vortisch. The author of this dissertation implemented the code, performed the experiments, and wrote their evaluation.

6.1. Introduction

Ride-pooling promises to enrich transportation systems as a mode of transport that interpolates between the flexibility and availability of motorized individual transport (i.e., cars) and the resource efficiency (e.g. fuel, emissions, space) of shared public transport. As advancements in fields like automated driving and mobile computing make ride-pooling viable in practice, there is a need to study the effects of introducing ride-pooling into current transportation systems. It is particularly interesting to evaluate a scenario of large-scale adoption of ride-pooling in which ride-pooling serves as a viable mode of transportation for many travelers. With this, we might answer the question of whether shared ride-pooling trips can replace enough individual car trips to significantly reduce motorized traffic.

Our parallel dispatcher Mt-KaRRi (see Chapter 5) allows the simulation of ride-pooling on a microscopic level with tens of thousands of ride-pooling vehicles in urban areas with millions of travelers per hour. Thus, it is uniquely suited to study the effects of ride-pooling on the

transportation networks of entire metropolitan areas. In this chapter, we perform an extensive experimental evaluation of ride-pooling on large input instances. We put ride-pooling into the context of a wider transportation system by considering it in competition with other modes of transport in a mode choice simulation. We construct huge input instances using two different agent-based simulation frameworks and refine our ride-pooling model by extracting traveler-specific information. In our experiments, we analyze the impact of the size of the customer base, the use of meeting points, and the number of ride-pooling vehicles on the attractiveness and efficiency of ride-pooling. We further study the quality of ride-pooling for travelers that move between the urban core and the periphery of cities, which is a use case that is dominated by individual cars in current transportation systems.

6.1.1. Related Work

Existing work on ride-pooling dispatching algorithms mostly evaluates quality criteria like the percentage of riders for which a feasible ride is available, the total distance traveled by ride-pooling vehicles, the average vehicle occupancy, etc. Commonly, the quality of ride-pooling is compared to a baseline of using non-shared taxis with input instances based on recorded taxi trips [Alo+17; BMN17; JJP16; MZW13; Ruc+21; SX15]. However, this approach fails to represent the positive impact that ride-pooling may have on a whole transportation system. Ride-pooling is not just seen as a way to cover taxi demand more efficiently, but as a way to fundamentally affect current transportation systems [GYB21; MvAvW17]. Analyzing ride-pooling on taxi-based input instances runs the risk of underestimating the amount of available pooling and gives no indication of the larger effects on the whole transportation system such as a change in modal split or a reduction in total traffic volume.

To better understand these effects, leading research groups in transport simulation have integrated support for ride-pooling in their simulation software. Unfortunately, these approaches use ride-pooling dispatchers with limited scalability, which can become a bottleneck for the simulation of populations of metropolitan areas. In the following, we consider the scalability and computation time of two transport simulations that support ride-pooling. We refer to Zwick et al. [Zwi+22] for a comparison of the modeling accuracy of both approaches.

MATSim. The agent-based transport simulation MATSim [HNA16] supports ride-pooling as one of many possible extensions in its modular architecture [BSM16]. MATSim is capable of simulating whole populations and iteratively updates mobility plans of individual agents until it converges to realistic travel patterns. Each iteration includes a detailed mode choice simulation. When used with the ride-pooling extension, this step comprises calls to a dispatching algorithm that is based on an insertion heuristic [BMN17] (see Section 2.3.2) and uses Dijkstra's algorithm to compute shortest paths [BSW21].

Bischoff et al. [BMN17] evaluate ride-pooling using MATSim, but only consider ride-pooling in isolation and use an input instance representing a data set of recorded taxi trips instead of a whole population. In this limited setting, the simulation of about 27 000 requests in one day is reported to take 20 minutes. Leich and Bischoff [LB19] use the same setup to evaluate replacing fixed bus lines with ride-pooling. This study simulates a whole population of a borough of Berlin, Germany, but it only considers ride-pooling and public transit as available modes of transport.

Buchhold et al. [BSW20] report long computation times when using ride-pooling in a full MATSim simulation on a representative population sample of Berlin. They show that the simulation of ride-pooling in MATSim may act as a bottleneck with running times in the order of hours for a single MATSim iteration. Often, MATSim needs hundreds of iterations to

converge to realistic mobility plans. Thus, Buchhold et al. integrate their improved dispatcher LOUD into MATSim to reach running times per iteration in the order of minutes. This integration could be used to further investigate ride-pooling in MATSim.

LOUD is the basis of our dispatcher KaRRi and is already quite fast. However, lacking the advancements of KaRRi and the parallelization with Mt-KaRRi, LOUD does not scale to the very large input instances that we are interested in (see Sections 4.9.4 and 5.3.4). It should be noted that Buchhold et al. use the version of MATSim available in 2020 and that recent developments on MATSim may reduce computation times [LHN25].

mobiTopp. Wilkes et al. [Wil+21] combine the agent-based transport simulation mobiTopp [MKV13] with the ride-pooling dispatcher FleetPy [Eng+22]. FleetPy is based on the dynamic ride-pooling dispatcher by Alonso-Mora et al. [Alo+17] with refinements by Engelhardt et al. [EDB20] (see Section 2.3.2). As this dispatcher uses a rolling-horizon strategy with costly re-optimizations based on mixed integer programming, scalability may be limited. Engelhardt et al. show that the approach allows real-time dispatching for an input instance with thousands of vehicles, but it is unclear whether the algorithm scales to metropolitan-scale instances.

The Place of our Approach. Analyzing ride-pooling as part of a full transport simulation with well-calibrated models of travel demand and traveler behavior leads to reliable results. Our approach cannot reach this level of accuracy. Instead, we see the contribution of our work in demonstrating that routing does not have to be a bottleneck for evaluating ride-pooling in the largest available simulation models. Thus, our approach helps to prepare transport simulation for a future with greater acceptance and more pervasive implementation of ride-pooling in metropolitan areas.

6.2. Mode Choice

We begin by describing our model for the mode choice behavior of riders. Given a journey for multiple possible modes of transport, this model simulates the choice of a real traveler.

For every request, KaRRi computes the feasible insertion with the smallest cost according to its cost function (see Section 4.4). The dispatcher offers a ride to the traveler according to this insertion. If a feasible insertion exists at all, the original algorithm assumes that the traveler always accepts the ride offer. With this assumption, the ride-pooling system serves all travelers, but the vehicles also have to perform every ride, even those with very high overhead. Thus, vehicle routes and service quality tend to degenerate if demand exceeds the capabilities of the vehicle fleet.

However, it is unrealistic for travelers to always accept the ride offer and use ride-pooling, especially when ride offers get worse due to high utilization of the ride-pooling system. Therefore, in this chapter, we extend KaRRi with a simple mode choice mechanism that models a traveler's choice between walking, taking their own car, using traditional line-based public transit, and accepting the ride-pooling offer. In the KaRRi workflow (cf. Figure 4.1), this mode choice mechanism simply replaces the feasibility check and the system state is updated only if the rider chooses ride-pooling. In Mt-KaRRi (cf. Figure 5.1), travelers that do not choose ride-pooling do not participate in any insertion conflicts.

6.2.1. Choice Model

A discrete choice model is an algorithm to select an element a from a set of alternatives A. To model the decision process for each request, we assign a utility score $U_a \in \mathbb{R}$ to each mode choice, providing a quantitative representation of the factors driving the decision. In this work, we model the utility of each mode a as a linear combination of a mode-specific constant and a travel time factor $U_a = \alpha_a + t \cdot \beta_a + w \cdot \beta_a^w + acc \cdot \beta_a^{acc}$ where α_a captures the inherent preference, β_a the impact of travel time t on the choice attractiveness, β_a^w the impact of waiting time w and β_a^{acc} the impact of access and egress time to the mode (e.g. walking to a station). We use a logit model as in [Tra03] where the selection probability of an alternative is calculated by $P_a = \frac{e^{U_a}}{\sum_{i \in A} e^{U_i}}$. The parameters for the utility functions have been calculated from the German "Mobilität in Deutschland" survey from [NK18], so that the parameters best match the decisions observed in the travel diaries in the survey. We select a mode by calculating the choice probabilities and sampling from the cumulative distribution using a random number. We also take into account that a traveler may not have access to every mode. For example, a traveler may not own a private car or may not have viable public transportation nearby. In this case, we set the utility of the unavailable mode to $U_a = -\infty$, which leads to a mode probability of $P_a = 0$. This model uses the access, egress, and travel times produced by the routing algorithms at request time. In particular, any situational effects that arise after the model selection, such as unplanned ride-pooling detours, traffic congestion or public transport delays are not taken into account.

6.2.2. Application of Discrete Choice Model

The discrete choice model requires as input the travel time, waiting time and access and egress time from the origin to the destination of the traveler for every mode of transport considered. For ride-pooling, we can derive these values from the shortest paths in the vehicle and pedestrian network associated with the best insertion. We obtain the travel time for a private car and for walking by running a point-to-point CH query (cf. Section 3.1.2) from the origin to the destination in the vehicle and pedestrian network, respectively. For line-based public transit, we run a query using the public transit routing algorithm ULTRA [Bau+23] on a public transit network of the area considered. We use the bounded multi-criteria version of the algorithm [PS22] to obtain an approximately-optimal Pareto set of public transit journeys according to the criteria travel time, number of transfers, and transfer walking time. From this Pareto set, we choose the journey with the best utility according to the discrete choice model.

6.3. Input Data and Methodology

In this section, we describe how we construct our problem instances and which metrics we use to judge the performance and quality of our dispatcher.

6.3.1. Ride-Pooling Demand Data

We use three different demand sources to analyze the traffic mode: The cities of Stuttgart and Karlsruhe in Germany, as well as Los Angeles (LA) in the US. For LA we use a public scenario [MAT20] for the agent-based transport simulation MATSim [HNA16] with different available population scales of 0.1%, 1%, 10%, 25%, 50% and 100% as a baseline for the traffic demand. For the Karlsruhe instance, we use the demand data from [Wör+21], and for the

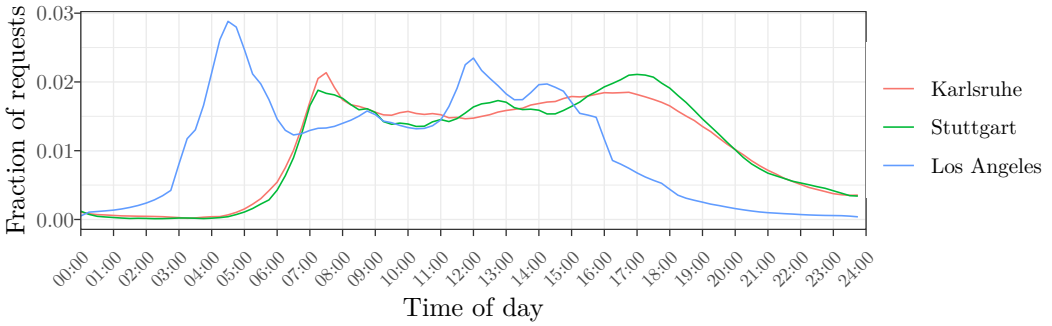


Figure 6.1 Distribution of trips over a whole weekday in our demand sets Karlsruhe (Tuesday), Stuttgart (Tuesday), and Los Angeles (Wednesday). Shows fraction of total trips per 15-minute interval of the day. For Los Angeles, the plot shows the distribution for the 100% population scale. The smaller scales have very similar distributions.

Table 6.1 Overview on demand sets.

Name	Network	#trips	period	#req./s	source
`KA`	Karlsruhe	975 773	6:00-9:00	90.35	mobiTopp
`ST`	Stuttgart	1 388 868	6:00-9:00	128.60	mobiTopp
`LA-0.1%`	Los Angeles	25 415	0:00-9:00	0.78	MATSim
`LA-1%`	Los Angeles	254 203	0:00-9:00	7.85	MATSim
`LA-5%`	Los Angeles	1 270 531	0:00-9:00	39.21	MATSim
`LA-10%`	Los Angeles	2 544 143	0:00-9:00	78.52	MATSim
`LA-25%`	Los Angeles	6 359 125	0:00-9:00	196.27	MATSim
`LA-50%`	Los Angeles	12 719 226	0:00-9:00	392.57	MATSim
`LA-100%`	Los Angeles	25 431 754	0:00-9:00	784.93	MATSim

Stuttgart instance we use the demand from [Wör+25]. Both use the agent-based simulation mobiTopp [MKV13]. Each of the input data provides a mobility plan per person for the entire duration of the simulation. For each input plan from the simulation frameworks we extract the travel demand by extracting a start time, end time and start and end location for each mobility plan. For the MATSim instances we use the referenced network links as start and end coordinate and for the mobiTopp instances we use the associated coordinates from the mobility plan. We show the resulting distribution of trips over a whole weekday in Figure 6.1. For all demand sets, we see a pronounced morning and afternoon peak. For the Karlsruhe and Stuttgart sets, the peaks are at about 07:30 am and 05:00 pm as expected. For these demand sets, we extract only the morning peak from 06:00 to 09:00 am, which gives us the instances called `KA` and `ST`, respectively. The peaks for the Los Angeles demand are notably earlier than for the other demands. We believe that the times in the MATSim scenario use an implicit offset of three hours. However, we could not find any documentation to this effect, so we decided to include a longer time period that covers the observed peak and the expected peak time. Our Los Angeles instances `LA-`p`%` for $p \in \{0.1, 1, 5, 10, 25, 50, 100\}$ contain all trips that the respective p% population input files specify to be issued between 00:00 and 09:00 am. For a summary of our problem instances, see Table 6.1. Note that the original demand data is based on minute intervals or quarter-hour intervals. We describe how we convert departure times to individual seconds for our dispatcher in Section 10.1.

Table 6.2 Overview on road networks.

Name	$\|V\|$	$\|E\|$	Network Structure
Karlsruhe	276 314	574 514	single center
Stuttgart	352 855	758 452	single center
Los Angeles	559 676	1 220 393	no strong center

Traveler-Specific Data. We derive additional information on the preferences of every individual traveler from the demand data sources to improve the transportation model. For this, we re-reference person attributes to the mobility plans in the source data.

As a first piece of additional data, we include car availability by assigning each traveler a probability to be able to use a private car. This decision will enable or disable the option to select car as a mode for a given trip. For the mobiTopp instances (`KA` and `ST`), we can derive the availability of a car for each traveler from existing household information. A traveler must possess a valid drivers license to be eligible for car usage. All drivers in a household who have an associated personal car will have an availability of 100%. The remaining drivers of the household can use a car with probability s/u where s is the number of shared cars and u is the number of unassigned drivers. For the LA instances, we assume that every rider has immediate access to a private car, which is not far from realistic for Los Angeles [UC 24].

We also add information on the walking speed and a maximum walking distance for every rider. We determine the maximum walking radius by taking the comfortable walking speed as measured by [Boh97]. To convert the walking speed into a distance, we multiply the speed with a fixed duration. We determine this duration by taking the access penalty parameter of ridesharing from the utility calculation (see Section 6.2) and set a fixed maximum so that the penalty from the access time is causing the utility to drop at most to 50% of the original utility. The resulting accepted walking distances are around 200–250 meters, which is within the recommended transit stop density of 300 meters for a high quality service area [Bun18].

As described in Section 4.4, we run bounded Dijkstra searches in G_{psg} to compute the possible pickup and dropoff locations for each request. Since we now use a separate maximum walking distance $l^{\max}_{\text{walk}}(r)$ and walking speed $v_{\text{walk}}(r)$ per traveler r, the Dijkstra searches use the length of a road segment as their distance metric instead of the travel time. We can stop the searches as soon as $l^{\max}_{\text{walk}}(r)$ is reached. We use the walking speed $v_{\text{walk}}(r)$ to transform distances into walking times for each pickup and dropoff location.

Vehicle and Pedestrian Networks. We generate road networks based on OpenStreetMap data as described in Section 10.2. We specify key metrics of the networks in Table 6.2.

The inner area of the `LA` network is a rectangular area stretching from 33°26'24"N to 34°19'48"N and from 116°57'0"W to 118°39'36"W. In accordance with the MATSim Open Los Angeles scenario, the outer area of `LA` is all of Southern California[1]. For `ST` and `KA`, the inner area is a 100km × 100km box aligned with the coordinate grid and centered around the respective city. The outer are is a 200km × 200km box. We obtain a representation of the public transit network for the `ST` and `KA` instances using a publicly available data set for transit schedules in all of Germany provided by DELFI[2].

[1] Parts of California south of about 35°45'N, following `https://download.geofabrik.de/north-america/us/california/socal.html`

[2] See `https://delfi.de/`.

(a) Germany **(b)** Stuttgart **(c)** Karlsruhe

Figure 6.2 Shows entire country of Germany (Figure 6.2a) and the areas immediately surrounding Stuttgart (Figure 6.2b) and Karlsruhe (Figure 6.2c). Urban areas are shaded dark, rural areas are shaded light. Red boxes in Figure 6.2a show position of Stuttgart (right box) and Karlsruhe (left box) within Germany. For Figures 6.2b and 6.2c, the inner outline indicates the limits of the city proper and the outer outline indicates the area in which trips are considered. The white area in the left of Figure 6.2c are parts of France for which we do not have RegioStaR data. No trips are made in this area.

6.3.2. Definition of Urban Core and Periphery

We partition the spatial topology of the German scenario instances based on the official German RegioStaR typology [Fed18]. This typology categorizes the municipalities based on settlement structure, centrality and spatial location. The typology can be differentiated at varying levels of precision to incorporate more information about the municipalities. We use the coarsest level RegioStaR-2, which separates between urban and rural regions. Figure 6.2 visualizes the RegioStaR-2 classification for all of Germany and for the areas immediately surrounding the cities of Stuttgart and Karlsruhe. Note that the area surrounding Karlsruhe contains parts of France for which we do not have a categorization. The `KA` demand set is limited to trips that lie entirely within Germany, though.

6.3.3. Performance and Quality Metrics

We measure the performance of the dispatcher by running time per request and the speedups observed by using multiple processors. To capture the system usage we measure the absolute number of ride-pooling users (*# riders*) and their relative share of total trips (*RP modal share*) by dividing the number of ridepooling trips by the number of total trips. We evaluate system-level efficiency by comparing the total time vehicles spend on the road to a baseline in which each ride-pooling user instead used their own car. Let T_{car} be the total driving time of private cars if every ride-pooling user used an individual car instead. Let T_{RP} be the total driving time of ride-pooling vehicles. We report the ratio $T_{\text{car}}/T_{\text{RP}}$ (*system effectiveness*). A system effectiveness of x means that the total vehicle operation time needed for all riders is reduced by a factor of x by using ride-pooling instead of private cars. We evaluate vehicle utilization using the time-weighted average occupancy (*occupancy*), which includes any time periods in which a vehicle is moving.

To assess waiting times, we report the average absolute waiting time of all ride-pooling trips (*wait time*). For the travel time, we compute the detour experienced by each rider, i.e., the difference between the in-vehicle time of the ride-pooling trip and the direct travel time

by car. Let δ_{car} be the direct travel time from a rider's origin to their destination. Let δ_{RP} be the total time the rider spends in a ride-pooling vehicle. We report the average of the ratios $\delta_{\text{RP}}/\delta_{\text{car}}$ for all riders (*ride detour*). In addition, we compute the average number of co-riders (*number of co-riders*) per ride-pooling user as a measure of the degree of pooling experienced by passengers.

6.4. Experiments

In this section, we describe the results of our detailed experimental evaluation of the impact of ride-pooling on large-scale transportation systems. We study the impact of the size of the customer base, the improvement gained with meeting points, and the influence of the fleet size on the solution quality. Finally, we compare the attractiveness of ride-pooling in urban areas and in rural areas.

6.4.1. Setup

We use the source code and setup for Mt-KaRRi (see Section 5.3.1). In addition to the 96-core machine described in Section 5.3.1, we use a second machine with an AMD EPYC 7702 processor (64 physical cores) and 1.0 TiB of RAM for some quality experiments. Since the hardware does not affect solution quality in our code, both machines are used interchangeably.

Unless otherwise specified, we use the following default assumptions: We use the model parameters decided on in the parameter tuning experiments in the appendix (see Section 10.3) and $t_{\text{batch}} = 5\text{s}$. Vehicles have a default capacity of four, and the initial location of every vehicle is an edge drawn uniformly at random in the road network of the service area. The service time of every vehicle is set to encompass the entire observation period. We specify the number of runs per experiment and report average metric values over all runs.

6.4.2. Impact of Customer Base

We analyze how the size of the customer base, i.e., the share of the population that considers using ride-pooling, affects the quality of the ride-pooling system. We use the `LA-`p`%` input instances (for $p \in \{0.1, 1, 5, 10, 25, 50, 100\}$, see Section 6.3.1) with fixed fleet sizes to represent different portions of the population making queries to our dispatcher. We report averages of three runs. Note that we only have limited information on individual travelers for the `LA` instances. Therefore, in this section, we do not use meeting points, since we do not know walking preferences of travelers.

Number of Riders. The plots in Figure 6.3 show the total number of riders and the ratio of travelers choosing ride-pooling over other modes of transport for a growing customer base.

The number of riders increases monotonically with a growing customer base. For the largest fleet, there are more than nine million total riders. For fleets of 50 000 or fewer vehicles, the number of riders increases only slowly. The RP modal share remains high for large fleets (up to 37% of the total population choose ride-pooling) but drops sharply for small fleets and a large customer base.

Pooling. In Figure 6.4, we show plots for three metrics describing the extent of shared rides in the ride-pooling system. First, we observe that the average number of co-riders increases with a larger customer base. This effect is relevant even for the largest vehicle fleets considered. Thus, a high rate of accepted rides and a large portion of shared rides are not

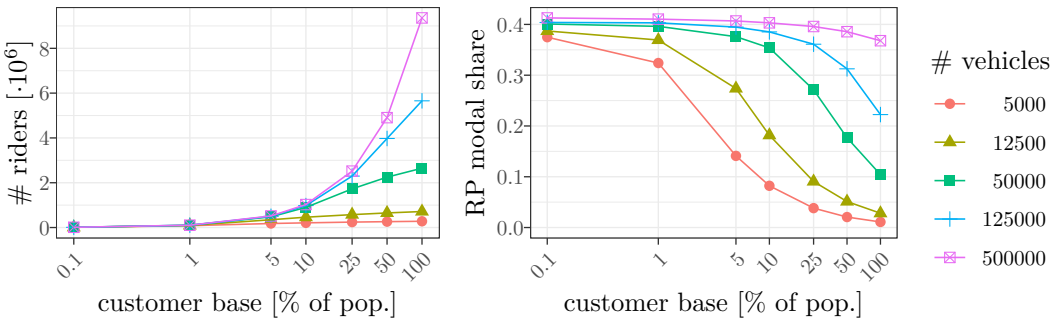


Figure 6.3 Ridership statistics for the LA-p% instances with $p \in \{0.1, 1, 5, 10, 25, 50, 100\}$ using different fleet sizes. The x-axis describes the share of the population willing to use ride-pooling on a logarithmic scale. The y-axis of each plot describes the number of riders in millions (left) and the RP modal share (right). Note the different y-axis scales. For explanations of the metrics, see Section 6.3.3.

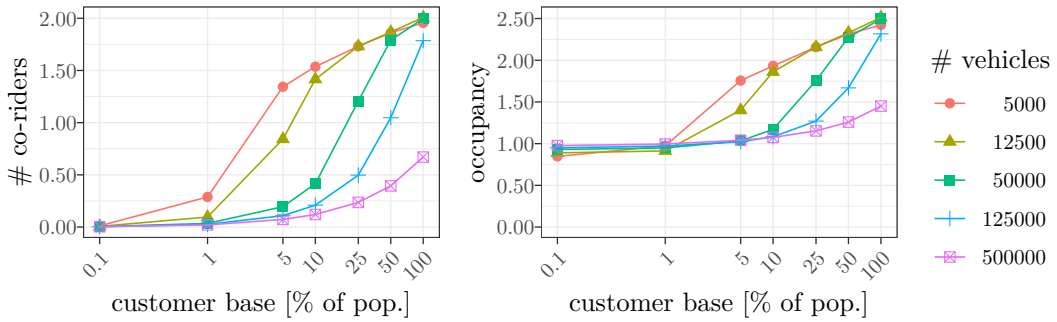


Figure 6.4 Pooling metrics on the LA-p% instances with $p \in \{0.1, 1, 5, 10, 25, 50, 100\}$ using different fleet sizes. The x-axis describes the share of the population willing to use ride-pooling on a logarithmic scale. The y-axis shows the number of co-riders (left) and occupancy (right). Note the different y-axis scales. For explanations of the metrics, see Section 6.3.3.

mutually exclusive. For example, more than five million travelers choose ride-pooling for the largest customer base and 125 000 vehicles. At the same time, every rider has an average of 1.79 co-riders along their trip.

This trend of increased pooling is also reflected in the average vehicle occupancy. For a small customer base, the occupancy is below one because of dead mileage spent moving to a request's origin. However, for large customer bases, this effect is compensated for by improved pooling. For 125 000 vehicles and the largest customer base, an average of 2.32 riders occupy each vehicle while driving. Even for the largest vehicle fleet considered, there are about 1.45 riders on average.

Quality of Rides, Resource Usage. The plots in Figure 6.5 show the quality of accepted ride-pooling rides and the usage of vehicle resources relative to the size of the customer base.

We observe growing ride detours for larger customer bases. This is caused by the previously noted higher level of pooling and the detours made for shared rides. Since smaller fleets require more pooling than large ones, small fleets also lead to larger in-vehicle times. With 500 000 vehicles, the average in-vehicle time is at most 18% longer than a direct car trip, but with 125 000 vehicles, the average in-vehicle time is up to 68% longer. Similarly,

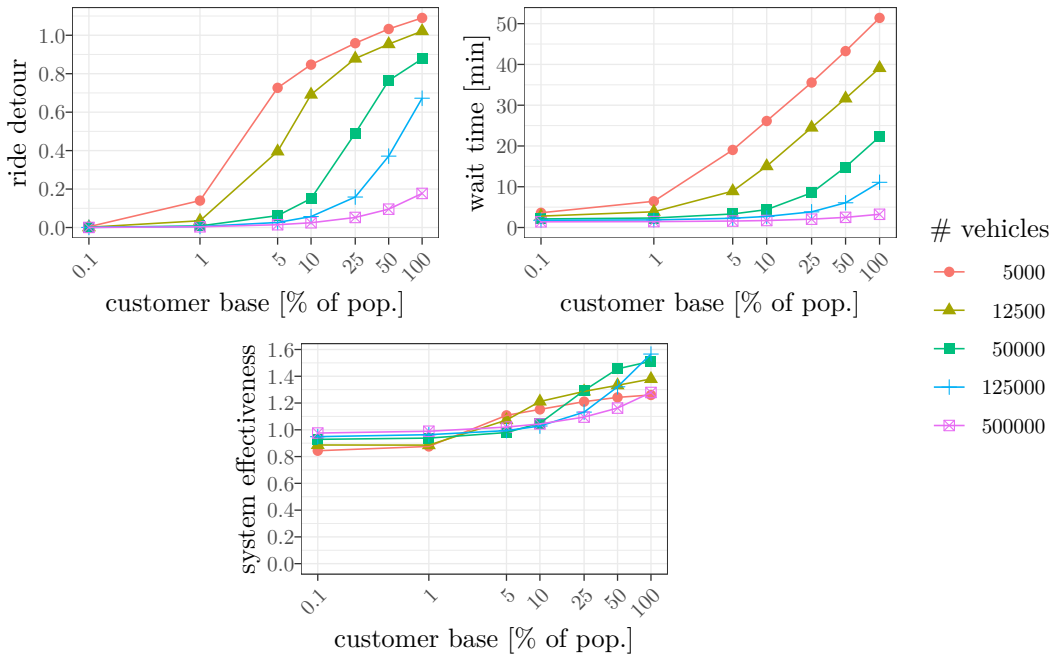


Figure 6.5 Quality metrics on the `LA-p%` instances with $p \in \{0.1, 1, 5, 10, 25, 50, 100\}$ using different fleet sizes. The x-axis describes the share of the population willing to use ride-pooling on a logarithmic scale. The y-axis of each plot shows the ride detour, wait time, and system effectiveness, respectively. Note the different y-axis scales. For an explanation of the metrics, see Section 6.3.3.

wait times increase with larger customer bases because vehicles may have to make stops for other riders before picking up a new rider. Due to our randomized mode choice model, even some of the rides with very long wait times are accepted.

Finally, we consider the system effectiveness as a measure of the resources saved compared to private cars. With small customer bases, there is little pooling, so the dead mileage spent to pick up new riders dominates the driving time saved through pooling. Here, ride-pooling can be assumed to use more vehicle resources than private cars (system effectiveness smaller than one). However, the improved pooling that comes with a larger customer base inverts this ratio such that ride-pooling vehicles spend less time driving than private cars would. For large vehicle fleets, ride-pooling uses up to 1.57 times less total vehicle time on the road than private cars.

Conclusion on Customer Base. As expected, we find that larger customer bases increase the number of travelers who use ride-pooling and can increase the potential for shared rides, which leads to significant savings in vehicle resources. However, the quality of rides deteriorates significantly if a small fleet tries to handle a high demand for rides. In these situations, the dispatcher should potentially make the decision to no longer offer new rides to avoid this degradation of quality.

6.4.3. Impact of Meeting Points

For the rest of the experimental section, we use our dispatcher with meeting points, i.e., riders walking to pickup locations and from dropoff locations (see Section 4.2). We motivate

Table 6.3 Effect of meeting points with walking on dispatcher quality for the KA and ST instances with 50 000 vehicles. Shows average number of pickup locations ($|P|$) and dropoff locations ($|D|$) considered, average pickup walking time (*p-walk*), dropoff walking time (*d-walk*), wait time (*wait*), in-vehicle time (*in-veh.*) and total trip time (*trip*) of riders, as well as average number of co-riders along each rider's trip (*co-r.*), total vehicle drive time (*drive*), and RP modal share (*RP*). Row marked "no" shows results without meeting points. Row marked "yes" shows results with traveler-specific meeting points determined using walking speed and maximum walking distance defined for each individual traveler (see Section 6.3.1). Rows marked "Δ[%]" give change relative to no walking in percent.

inst.	walk	$\|P\|$	$\|D\|$	p-walk [mm:ss]	d-walk [mm:ss]	wait [mm:ss]	in-veh. [mm:ss]	trip [mm:ss]	co-r. [abs.]	drive [h]	RP [%]
KA	no	1.00	1.00	00:00	00:00	05:46	18:21	24:07	1.50	55 093	40.85
	yes	14.92	14.85	00:53	00:12	04:00	17:06	22:11	1.57	49 868	41.09
	Δ[%]					-30.63	-6.85	-8.01	4.32	-9.48	0.60
ST	no	1.00	1.00	00:00	00:00	16:02	28:57	44:58	1.96	75 786	30.77
	yes	19.04	16.10	01:18	00:21	11:44	26:36	39:58	2.06	70 416	32.52
	Δ[%]					-26.85	-8.10	-11.12	5.00	-7.08	5.70

this using the following experiments that show the benefits of meeting points for the solution quality of our ride-pooling dispatcher. The set of meeting points for each rider is generated as described in Section 6.3.1. Table 6.3 shows the quality without and with meeting points for the KA and ST instances. We report average results for five runs of each experiment.

Given traveler-specific walking speeds and distances, an average of between 15 and 19 meeting points are considered for each request. On average, each rider walks about five times farther in the beginning of their trip than in the end, as vehicles are more likely to be constrained by existing riders at the pickup location than at the dropoff location. For both instances, the wait time shrinks considerably when using meeting points. The average time between issuing a request and being picked up is reduced from a pure wait of 05:46 min (16:02 min) to a combination of walking and waiting comprising 05:05 min (12:05 min) for the KA (ST) instance. Additionally, the average in-vehicle time for riders decreases by 6.85% for KA and 8.10% for ST, leading to average total trip times that are 8.01% and 11.12% smaller than without meeting points for KA and ST. The level of pooling improves when using meeting points, with the average number of co-riders increasing by 4.32% for the KA instance and 5.00% for the ST instance. This is also reflected in a decrease in total vehicle drive time by 9.48% for KA and 7.08% for ST.

The modal share of ride-pooling stays about the same for the KA instance, but increases by 5.70% for the ST instance. This difference is caused by the KA instance having fewer requests and a smaller network. Since we use fleets of the same size for both instances, the fleet is under more stress in the ST instance and the quality of rides is worse. Therefore, for the ST instance, meeting points have a greater impact on the quality of ride-pooling, which leads to a greater improvement in modal share.

In conclusion, meeting points improve both rider trips and vehicle resource usage. They are particularly helpful if the fleet is experiencing high demand. Thus, we always consider meeting points if possible. In a real-world system, a dispatcher could offer both rides with and without walking and include incentives for walking (e.g. an additional fare discount).

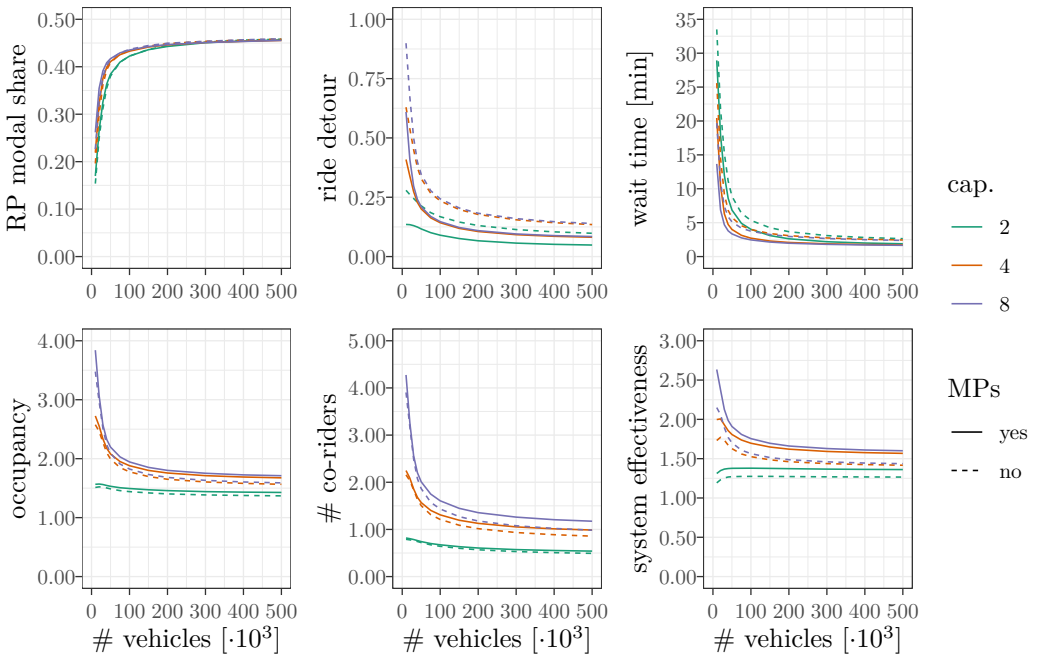


Figure 6.6 Quality on KA with fleets $|F| \in \{10, 20, 30, 40, 50, 75, 100, 150, 200, 300, 400, 500\} \cdot 10^3$ and $\text{cap}(\nu) \in \{2, 4, 8\}$ for every $\nu \in F$ with and without meeting points (*MPs*). The x-axis describes the number of vehicles in thousands. The y-axis of each plot shows one of our quality metrics (see Section 6.3.3).

6.4.4. Impact of Fleet

Figures 6.6 and 6.7 depict the solution quality relative to the size of the fleet and seating capacity for the KA and ST instances, respectively. We report average results for three runs.

First, note that the use of meeting points leads to improvements in all metrics. Although the modal share of ride-pooling is not affected much, in-vehicle times and wait times become shorter, pooling is improved, and the system effectiveness is around 10% better than without meeting points for every fleet size. Thus, we focus on the case with meeting points.

We find that the share of travelers using ride-pooling increases with a growing number of vehicles. This can be attributed to an improved quality of rides as evidenced by smaller ride detours and wait times. Until about 50 000 vehicles for KA and 100 000 vehicles for ST, the quality of rides and modal share improve a lot. Even larger fleets provide only a small benefit to the quality of rides and the modal share. For example, for KA, the modal share of ride-pooling is 41.1% at 50 000 vehicles of capacity four (with meeting points) and only improves to 45.5% for 500 000 vehicles.

For small vehicle fleets, shared rides are vital, and we see high occupancy rates of more than 1.5, 2.5, and 3.5 and an average number of co-riders of up to 0.75, 2.25, and 4.25 for capacity two, four, and eight, respectively. The level of pooling drops with growing vehicle fleets, as more vehicles are available to serve riders individually, like a taxi. However, even for the largest fleets, the occupancy rate is at least 1.67 for a capacity of four or eight.

As expected, less pooling leads to worse system effectiveness for large fleets. Intriguingly, the system effectiveness initially improves with growing fleet size for some configurations even though the level of pooling decreases. This effect is more pronounced for weaker configurations, i.e., with smaller capacity or without meeting points. We believe that the

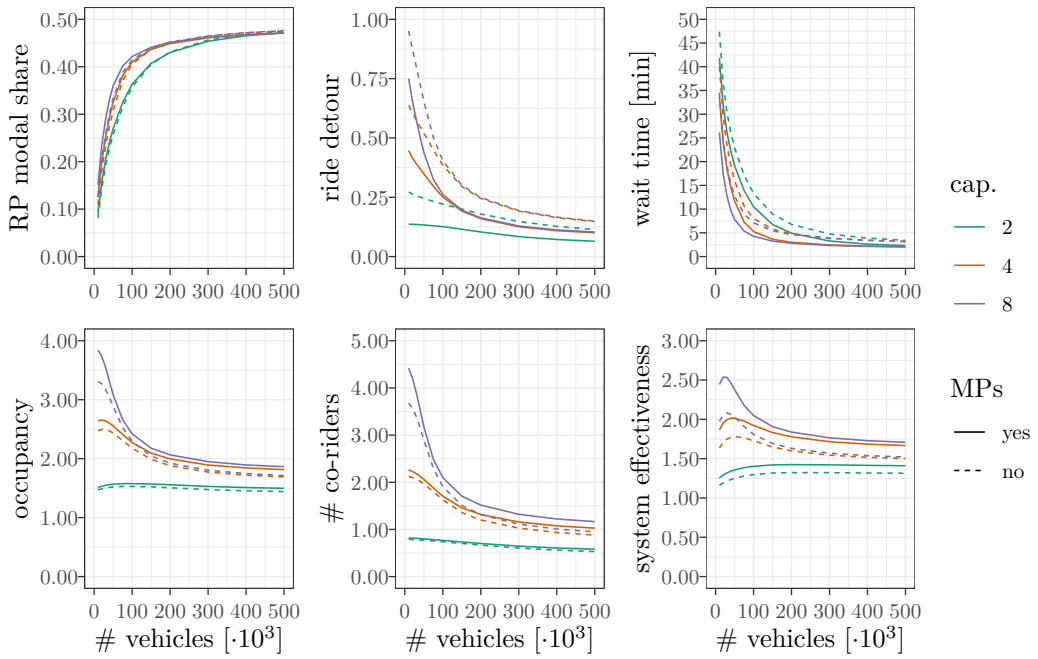


Figure 6.7 Quality on ST with fleets $|F| \in \{10, 20, 30, 40, 50, 75, 100, 150, 200, 300, 400, 500\} \cdot 10^3$ and $\text{cap}(\nu) \in \{2, 4, 8\}$ for every $\nu \in F$ with and without meeting points (*MPs*). The x-axis describes the number of vehicles in thousands. The y-axis of each plot shows one of our quality metrics (see Section 6.3.3).

reduced effectiveness for the smallest fleets is due to ineffective pooling, where new rides are found that have viable detours, but do not actually group riders with sufficiently similar itineraries. In this case, there are long detours that leads to long vehicle operation times, in-vehicle times, and wait times. For larger fleets, it is more likely that non-shared rides are available and rides are only pooled if there is a good match.

In Figure 6.8, we show the occupancy of vehicles over time for a variant of the KA instance that entails an entire day (from Tue 3 am to Wed 3 am). We observe that the number of vehicles driving with three or four passengers follows the level of demand with a morning and afternoon peak. When there is less demand in the middle of the day and in the evening, there is less stress on the system and more trips with one or two passengers can be made.

6.4.5. Comparison between Urban Core and Periphery

In this section, we evaluate the quality of ride-pooling for trips that move between the core and the periphery of a city. This use case is particularly interesting for ride-pooling because peripheral areas provide fewer opportunities to use traditional public transit compared to urban areas. However, urban areas attract a significant number of journeys from surrounding areas, e.g. for commuting to work or school. Thus, the improved flexibility offered by ride-pooling may be a good opportunity to reduce the dominant use of private cars for these trips, and bundle the travelers into fewer vehicles for more efficient journeys.

We conduct experiments on the KA and ST instances with 50 000 vehicles. For every origin and destination location in the request set, we decide whether it is urban or rural according to the RegioStaR-2 classification of Germany (for details, see Section 6.3.2). Requests fall into one of four categories: urban-urban, rural-rural, urban-rural, and rural-urban. We

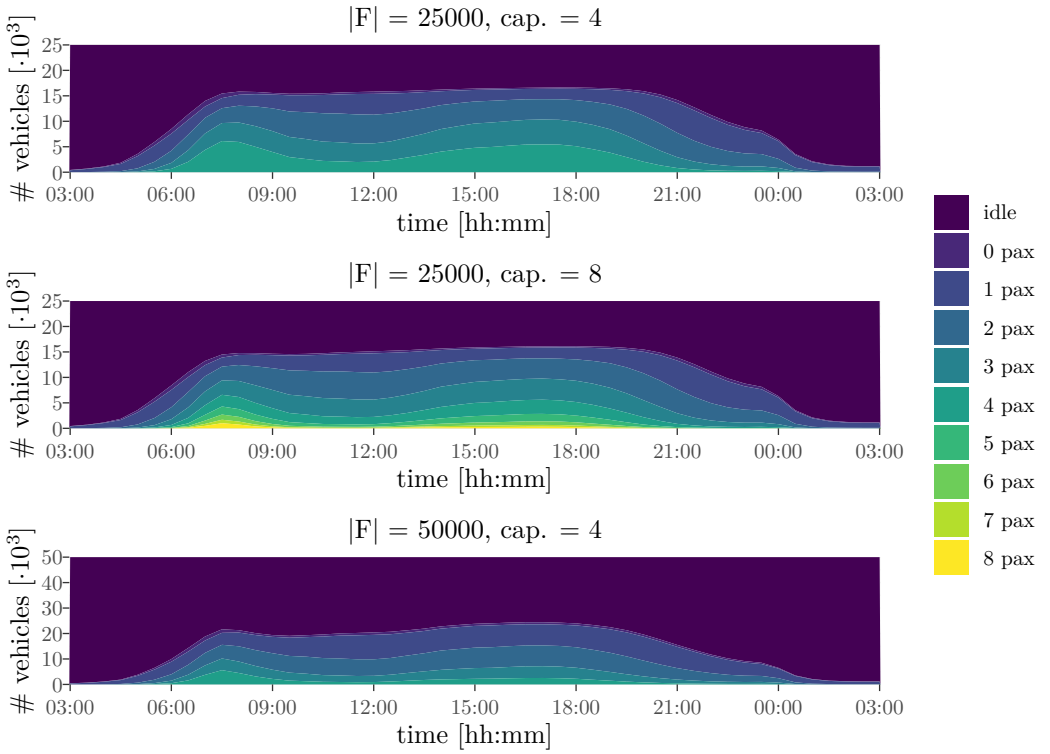


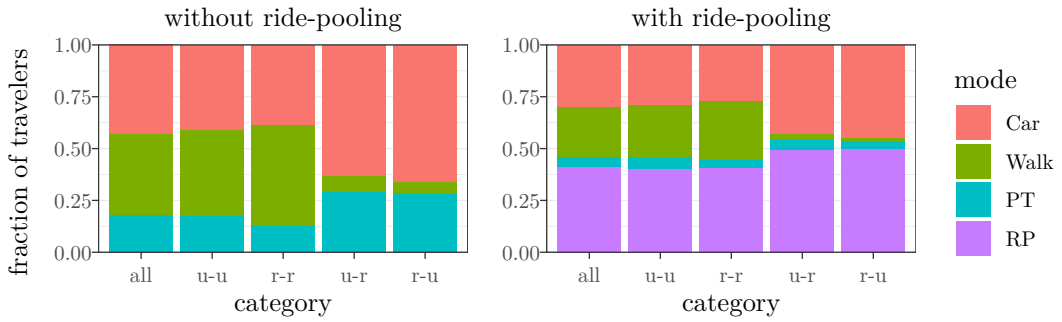
Figure 6.8 Shows distribution of vehicles over number of passengers and time for a full day variant of `KA` with 25 000 vehicles of capacity 4 (top), 25 000 vehicles of capacity 8 (middle), and 50 000 vehicles of capacity 4 (bottom). The height of each color stripe at a given point on the x-axis represents the number of vehicles that are currently idle (*idle*) or driving with k passengers (k *pax*).

run the dispatcher for all requests and then analyze the results by category, so requests of different categories affect each other's results like in a real system. We report average results for three runs of each experiment.

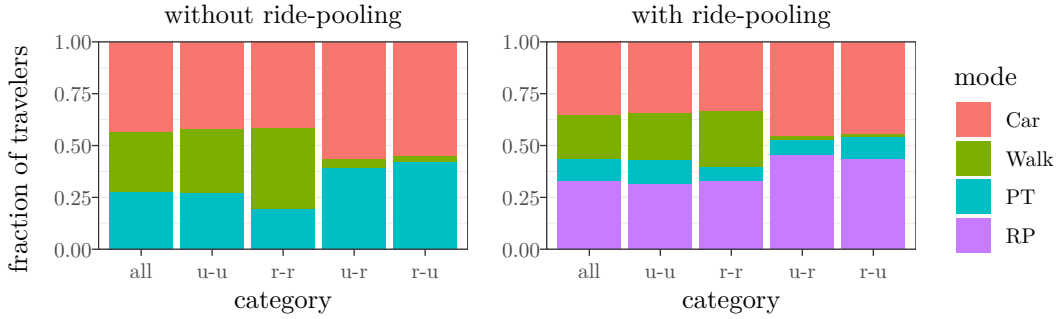
In Figure 6.9, we show the distribution of travelers across modes for all requests and for each of the four categories of requests in the `KA` and `ST` instances. The plots on the left show the mode split when ride-pooling is not available. For the urban-urban and rural-rural categories, many travelers choose to walk since travel distances are small. In the urban-urban setting, about 18% and 27% of travelers choose public transit for `KA` and `ST`, respectively. In the rural-rural setting, these numbers are lower at about 13% and 19%. The urban-rural and rural-urban categories contain journeys with much larger travel distances, so few travelers walk. Here, the modal share of the car is up to 66%, while public transport accounts for more than 28% of trips for `KA` and up to 42% for `ST`.

The plot on the right of Figure 6.9 shows the mode split including ride-pooling. We find that ride-pooling takes on the largest share of any mode in all categories. Ride-pooling seems to replace public transport as a more convenient alternative to private cars, and the share of transit falls to a single digit percentage in all categories. Importantly, ride-pooling is the most attractive in the urban-rural and rural-urban categories where cars previously dominated due to a lack of good alternatives. Here, the introduction of ride-pooling causes the mode share of the car to fall from 66% to less than 45%.

To understand the reason for the improved attractiveness of ride-pooling in urban-to-rural



(a) Modal split for `KA`.



(b) Modal split for `ST`.

Figure 6.9 Fraction of travelers per mode of transportation (PT = public transit, RP = ride-pooling) for `KA` (top, Figure 6.9a) and `ST` (bottom, Figure 6.9b) without ride-pooling (left) and with ride-pooling (right, 50 000 vehicles). Shows distribution across modes for all requests (all), and subsets of urban-to-urban (u-u), rural-to-rural (r-r), urban-to-rural (u-r), and rural-to-urban (r-u) requests according to the RegioStaR-2 classification of Germany (see Section 6.3.2).

and rural-to-urban requests, we provide data on the quality of ride-pooling rides by category in Table 6.4. Note that rides in the urban-rural and rural-urban categories have slightly higher wait times and in-vehicle times relative to the shortest-path distance. This seems to apply worse ride quality for these categories. We find that the increased modal share in the urban-rural and rural-urban categories are not caused by better ride quality, but can instead be explained by longer travel distances compared to other categories (column δ). In our model, the probability of choosing ride-pooling grows with increasing travel distance since the wait time loses significance relative to the in-vehicle time. For long trips, travelers decide that the discomfort of having to wait for a ride-pooling vehicle is outweighed by the fact that they do not have to operate a car themselves. This supports the idea that ride-pooling can reduce the number of inefficient private car journeys between the center and the periphery of an urban area. In the future, a multimodal combination of ride-pooling and public transit may optimally leverage both modes for these trips.

Fairness. Finally, we analyze the fairness of ride quality across our categories. We focus on the `KA` instance. In Figure 6.10, we show plots on the cumulative distribution of wait times and ride detours relative to the direct car travel time, as well as the wait time and ride detour added after the rider accepted the offer.

First, consider the distribution of wait times. We find that wait times have a similar

Table 6.4 Key rider quality metrics for Mt-KaRRi on the `KA` and `ST` instances using $|F| = 50\,000$ vehicles. Rows show results for subsets of requests classified by origin and destination locale according to the RegioStaR-2 classification of Germany's low-level subdivisions into urban and rural areas (see Section 6.3.2). Shows number of requests for each category, average car travel time from origin to destination of all requests (δ) and only ride-pooling riders (δ_{RP}), average wait time (wait), in-vehicle time (ride), and trip time (trip) of ride-pooling rides, as well as the RP modal share (RP).

inst.	type	# req.	δ [mm:ss]	δ_{RP} [mm:ss]	wait [mm:ss]	ride ($/\delta_{\mathrm{RP}}$) [mm:ss]	trip [mm:ss]	RP [%]
`KA`	all	975 772	11:56	13:29	04:00	17:06 (1.20)	22:11	41.09
	urban-urban	703 190	11:12	12:44	03:57	16:07 (1.20)	21:10	40.14
	rural-rural	184 507	09:21	10:49	03:44	13:30 (1.18)	18:14	40.64
	urban-rural	36 562	21:09	21:13	04:56	27:26 (1.27)	33:29	49.43
	rural-urban	51 513	24:37	24:10	04:36	31:15 (1.27)	36:55	49.73
`ST`	all	1 388 867	16:11	18:51	11:44	26:36 (1.35)	39:58	32.52
	urban-urban	1 096 587	15:14	17:32	11:24	24:48 (1.34)	37:52	31.22
	rural-rural	171 481	12:20	15:19	11:28	20:53 (1.31)	34:01	32.81
	urban-rural	39 519	33:03	34:05	14:21	48:12 (1.43)	64:03	45.17
	rural-urban	81 280	29:01	29:34	13:57	42:20 (1.43)	57:46	43.25

distribution for all categories with a slightly more pronounced tail for the urban-rural and rural-urban categories. This can again be attributed to the longer travel distance of requests in these categories. Here, the travel time dominates the utility of each mode in the mode choice. Thus, more rides with longer wait times will be accepted than in the other categories where travel times tend to be shorter.

We see similar results when considering the ride detour. Note that the ride detour is smaller than zero if the traveler walks part of the total distance on their way to a meeting point. This effect is more pronounced for urban-urban and rural-rural journeys where the total distance is smaller. In addition, a larger fraction of rides have a ride detour of zero (i.e., non-shared rides) in these categories. Due to the longer total travel distance in the urban-rural and rural-urban categories, larger detours are allowed and the riders occupy the vehicle for a longer time, so shared rides become more likely. This results in slightly increased ride detours compared to the other categories.

For the majority of rides, the wait time does not get larger than the expected wait time, because it is unlikely that a detour is made between the points in time when the ride is accepted and when the rider is picked up. However, it is slightly more common in the urban-rural and rural-urban categories due to the longer wait times of accepted ride offers.

The added ride detour compared to the original ride offer reflects our insight on the total ride detour. In the urban-urban and rural-rural categories, fewer riders experience additional detours than in the urban-rural and rural-urban categories. In the latter categories, almost two thirds of rides experience additional detours. However, in about 90% of all rides, the additional detour stays below half the direct car travel time.

In conclusion, we find that journeys in the urban-urban and rural-rural categories experience slightly better rides than those in the urban-rural and rural-urban categories, in particular due to fewer shared rides. However, journeys within the urban environment do not monopolize the ride-pooling fleet to the detriment of journeys in the periphery as we see

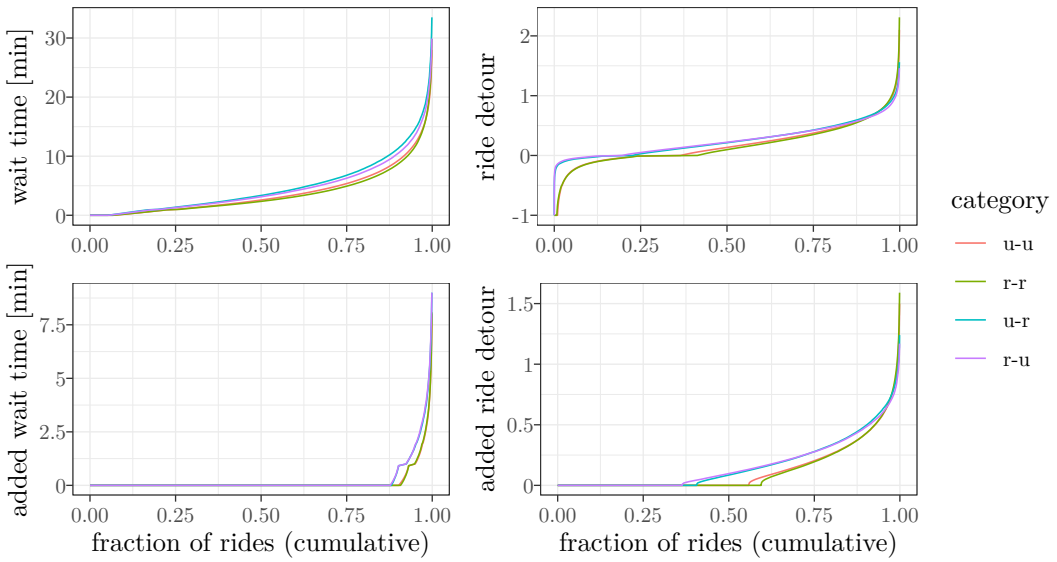


Figure 6.10 Top: Cumulative distribution of wait times and ride detours of performed rides on `KA` with 50 000 vehicles. Bottom: Cumulative distribution of wait time and ride detour added to the originally offered ride after the rider has accepted the ride. We omit the top 0.1% of values to improve readability. Each line shows the distribution for one category of requests.

similar distributions of wait times and ride detours for all categories. Our mode choice model ignores wait times and ride detours added after accepting a ride, but riders would come to expect the additional inconvenience in a production system. Since these added wait and travel times are more pronounced for urban-rural and rural-urban trips, this effect should be considered in more detail when designing a ride-pooling system, e.g., with respect to the service area and the pricing scheme.

6.5. Conclusion

We use our dynamic ride-pooling dispatcher Mt-KaRRi to conduct experiments on ride-pooling at an unprecedented scale with millions of traveler requests per hour. In a simulation study, we find that a larger customer base can improve system effectiveness but may lead to system-wide degradation of ride-quality if every customer is always offered a ride. Our experiments suggest that the use of meeting points, i.e., riders walking small distances to a pickup/dropoff location, improves pooling and can reduce the stress on a vehicle fleet in situations with high demand. Furthermore, we show that more and/or larger vehicles lead to improved ride quality at the cost of decreased system effectiveness, but that a careful choice of the vehicle fleet can provide near-optimal rides while retaining good effectiveness. An analysis of ride-pooling in peripheral areas suggests that it can be an attractive alternative to private cars in areas with limited choices of transportation modes.

We consider Mt-KaRRi to be well-suited for further, more detailed studies on ride-pooling at a large scale. A tight coupling of a traffic simulation (e.g., mobiTopp [MKV13]) and Mt-KaRRi would allow for a direct injection of travel times and fleet status and thus a qualitatively improved mobility simulation.

Part II.

Transfers and Multimodal Journeys

7 Dynamic Ride-Pooling with Transfers

Summary: Allowing transfers between ride-pooling vehicles could reduce vehicle costs while maintaining service quality for riders, particularly in congested ride-pooling systems. Due to the added complexity of finding good assignments with transfers, there has not been much progress on this problem. We introduce KaRRiT, an efficient dispatcher that finds single-transfer journeys. We create an exact variant of KaRRiT that exhaustively looks for optimal transfers using engineered one-to-all shortest-path techniques and problem-specific pruning of the solution space. Additionally, we propose a heuristic variant that only considers important vertices in the road network for transfers. Although our engineering is effective, the exact variant of KaRRiT takes hundreds of milliseconds per request, which is likely too slow for production systems. The heuristic variant achieves running times that are two orders of magnitude larger than the best no-transfer algorithm. Preliminary quality experiments show that transfers can improve system effectiveness at a small cost to rider trip times.

Attribution: This chapter is based on the paper "Exact and Heuristic Dynamic Taxi Sharing with Transfers using Shortest-Path Speedup Techniques" [BL25], co-authored with Johannes Breitling. The paper is based on the bachelor thesis of Johannes Breitling [Bre25a], but fully reworks the structure and introduces new ideas (here: Section 7.5 and parts of Section 7.4.4). The paper was mainly written by the author of this dissertation with small contributions and editing by Johannes Breitling. The algorithmic ideas were developed jointly by Johannes Breitling and the author of this dissertation. Johannes Breitling initially implemented the code for their thesis. The author of this dissertation refined and extended the code for the paper. The author of this dissertation conducted the experimental evaluation.

7.1. Introduction

Recent developments in autonomous vehicles increase the attractiveness of *ride-pooling* in which a fleet of taxi-like vehicles is intelligently controlled to transport travelers without fixed stops or schedules. These systems attempt to bundle riders and maximize the usage of each vehicle's capacity for more resource efficient journeys compared to private cars or traditional taxis. The advantages of such systems have been extensively studied in numerous simulation studies [Aga+11; BMN17; FK14; Kue+23; Wil+21; Zwi+22] and real-world field tests [Gar+15; Gil+20; JSM19; KFK21; TW08; Wec+18; Yu+17; Zhu21]. In Chapter 6, we show the effects of ride-pooling on large input instances using our dispatcher KaRRi.

The positive impact of ride-pooling could further be improved upon by allowing riders to transfer between vehicles during their journey. This additional option may allow the vehicle dispatcher to reduce vehicle operation times and increase the occupancy rates of vehicles without negatively affecting rider trip times. Thus, transfers could provide both economical and ecological benefits to ride-pooling systems. However, current studies into ride-pooling largely lack the option of transfers due to large computation times. Ride-pooling is already a difficult problem without transfers [MZW13; Sav85] but transfers lead to an even more

complex problem as the number of possible assignments of a rider to one or more vehicles increases combinatorially.

Based on our dispatcher KaRRi (see Chapter 4), we propose the first exact dispatching algorithm for dynamic ride-pooling with transfers that is able to scale to realistic city-scale input instances. For this purpose, we extend the model for traditional ride-pooling to allow journeys with at most one transfer. We find that a main issue of dynamic ride-pooling with transfers is the exploding number of shortest-path (SP) distances in the road network that need to be known to choose the best assignment for a rider. We focus on computing these distances on-the-fly when a rider request comes in, as this serves as the basis of a dynamic dispatcher that uses exact and up-to-date travel time information. We analyze the SP problems at hand and employ state-of-the-art speedup techniques for SPs in road networks to efficiently solve them. Also, we propose techniques to prune the number of assignments that need to be considered and explore both instruction-level and thread-level parallelization. In addition to an exact algorithm, we describe a heuristic approach that sacrifices some solution quality for improved running times.

In an experimental evaluation, we show that the proposed measures lead to viable dispatching times for realistic demand on the road network of Berlin, Germany. We give first indications that transfers improve vehicle operation times and occupancy rates at the cost of slightly increased rider trip times. Our approach lays the groundwork for more precise studies of ride-pooling with transfers on large urban road networks with realistic request sets. This analysis is left to future work in cooperation with application experts.

7.1.1. Related Work

Although ride-pooling has received a lot of attention recently (see Chapter 2), there has not been much work on ride-pooling with transfers. Most approaches consider a fixed set of transfer points that is known in advance (e.g. charging stations for electric cars) and find solutions using mixed-integer programming [Col14; Hou+16; Wan+23]. These approaches assume that the distance between any pair of vertices is known without having to compute it.

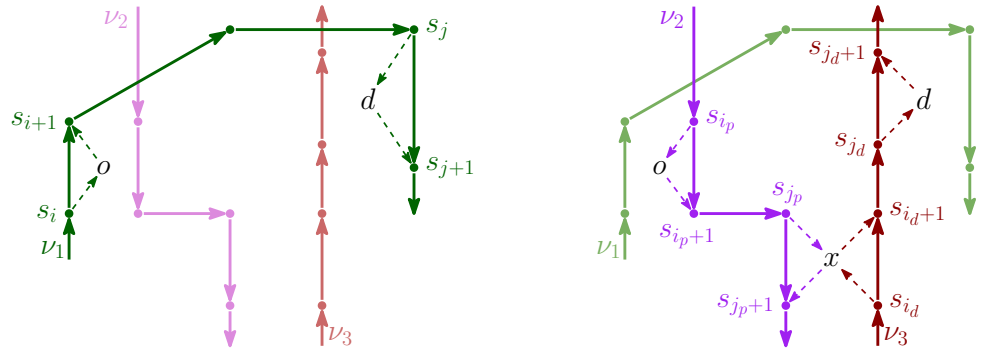
The approach that is most closely related to our work is an extension of the LOUD dispatcher that locates feasible transfer points based on three different heuristics [Wil23a]. Their results show a reduction in total operation cost due to transfers. Note, though, that the cost model differs from the one we use since vehicle wait times are not considered part of a vehicle's detour and rider trip times are only taken into account as constraints.

To the best of our knowledge, there are no existing approaches for dynamic ride-pooling with optimal transfers that are able to scale to realistic instances of large urban areas.

7.2. Ride-Pooling Model with Transfers

The model for ride-pooling with transfers is largely equivalent to the base model outlined in Section 3.2. Our dispatcher still maintains a fleet of vehicles and attempts to find the best insertion for every incoming request according to a cost function. We briefly reiterate some important details of the base model before describing our model for transfers.

Recall that $G = (V, E)$ represents the road network considered. A request has the form $r = (\text{orig}, \text{dest}, t_{\text{req}})$, where $\text{orig} \in V$ is the origin location of the traveler, $\text{dest} \in V$ is the destination location of the traveler, and t_{req} is the time at which the request is issued. An insertion without transfers has the form $\mathcal{I} = (r, \nu, i, j)$, which means that the origin orig of request r is inserted as a new stop between stops s_i and s_{i+1} of vehicle ν, and the destination dest is inserted as a new stops between stops s_j and s_{j+1} with $j > i$.



(a) No-transfer insertion (r, ν_1, i, j).

(b) Transfer insertion $(r, x, \nu_2, \nu_3, i_p, j_p, i_d, j_d)$.

Figure 7.1 Illustration of routes of three vehicles ν_1, ν_2, and ν_3 and a request $r = (o, d, t_{\text{req}})$. Full arrows show current routes while dashed arrows denote detours made for a possible insertion. Figure 7.1a depicts an insertion without transfer using ν_1. Figure 7.1b illustrates a single-transfer insertion using pickup vehicle ν_2 and dropoff vehicle ν_3 with a transfer at x.

Single-Transfer Insertions. In this chapter, we focus on efficiently finding optimal *single-transfer journeys*, where a rider changes vehicles at most once. This extension induces a significant combinatorial increase in the number of possible insertions compared to the traditional case without transfers, which poses the main challenge of our work.

Consider a request $r = (\text{orig}, \text{dest}, t_{\text{req}})$. A *single-transfer insertion* takes the form $\mathcal{I}_{transfer} = (r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$. The *pickup vehicle* ν_p picks up the new rider at orig immediately after stop $s_{i_p}(\nu_p)$ and drops them off at the transfer location x immediately after stop $s_{j_p}(\nu_p)$. The *dropoff vehicle* ν_d picks the rider up at x for the second leg of the trip immediately after stop $s_{i_d}(\nu_d)$ and drops them off at dest immediately after stop $s_{j_d}(\nu_d)$. Figure 7.1 compares a no-transfer insertion and a single-transfer insertion.

7.2.1. Cost Computation of Single-Transfer Insertions

We use the cost function defined for KaRRi (see Section 4.2) without changes and apply the same constraints to transfer insertions. Here, we describe additional intricacies that come with computing the cost of insertions with transfers.

Vehicles Waiting at the Transfer Location. After processing a transfer insertion where the dropoff vehicle arrives at the transfer location x sooner than the pickup vehicle, the dropoff vehicle has to wait at x for the arrival of the transferring rider. This is similar to a vehicle arriving at a meeting point sooner than the rider (see Section 4.5). Since KaRRi already handles this case, these additional vehicle wait times do not require a change of our model.

Dependencies between Vehicle Schedules. Transfers introduce dependencies between vehicles' schedules. In a transfer, the dropoff vehicle can only leave the transfer point once the transferring rider arrives in the pickup vehicle. Thus, any delay to the arrival of the pickup vehicle at the transfer point may also delay the departure of the dropoff vehicle. In effect, an insertion may affect vehicles that do not directly participate in the insertion due to previously introduced transfers. To account for this, we explicitly memorize the dependency between pickup and dropoff vehicle whenever a transfer insertion is performed. Then, when computing the cost for a new insertion $\mathcal{I}_{transfer} = (r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$, we

find any vehicles that depend on ν_p or ν_d, and potentially propagate the detours caused by $\mathcal{I}_{transfer}$ to their routes. For any such dependent vehicle, we obtain an added operation time and added trip time for existing riders, which we consider in the total cost of $\mathcal{I}_{transfer}$.

7.3. Algorithm Overview

We describe the use of detour ellipses for transfer insertions and identify two distinct types of transfer insertions. Based on this, we give an overview of the structure of our algorithm.

7.3.1. Detour Ellipses for Transfers

The central challenge of computing single-transfer insertions is the fact that the pickup vehicle ν_p and the dropoff vehicle ν_d can move freely in the road network, which makes every location in the road network a potential transfer point. Without further limitations, the number of transfer insertions that need to be checked would be at least linear in the size of the road network. However, we can reduce the number of viable transfer locations for each request by taking into account the constraints on vehicle detours imposed by existing riders.

As mentioned in Section 4.3.2, the constraints on vehicle routes induce a detour leeway $\lambda(s_i, s_{i+1})$ between any two consecutive stops s_i and s_{i+1}. This leeway defines the ellipse $\mathcal{E}(s_i) \subseteq V$ containing all locations to which a detour between s_i and s_{i+1} can be made without breaking any constraints. Thus, for any feasible transfer insertion $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$, the transfer point x must lie in $\mathcal{E}(s_{j_p}(\nu_p))$ if $j_p < k(\nu_p)$ and in $\mathcal{E}(s_{i_d}(\nu_d))$ if $i_d < k(\nu_d)$. If we know the detour ellipses of stop pairs along the routes of relevant vehicles, we can deduce a limited set of viable transfer locations for which transfer insertions need to be constructed.

In the following, we describe how detour ellipses can be used to enumerate transfer insertions of different types. For now, we assume that all necessary detour ellipses are known. We explain how to compute a detour ellipse on-the-fly in Section 7.4.1. We describe how to compute the necessary shortest-path distances in Sections 7.4.2 and 7.4.3.

7.3.2. Types of Transfers

We distinguish between two types of transfer insertions that differ in how detour ellipses constrain the set of potential transfer points.

Ordinary Transfers. An *ordinary transfer* insertion is any insertion $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$ where both $j_p < k(\nu_p)$ and $i_d < k(\nu_d)$, i.e., for which the transfer point x is inserted before the last stop in the routes of both the pickup vehicle and the dropoff vehicle. Let $F_p(r) = \{(\nu, i) \in F \times \mathbb{N} \mid i \in \{0, \dots, k(\nu)\} \text{ and orig} \in \mathcal{E}(s_i(\nu))\}$ denote the set of vehicles and associated stops in the vehicle route such that ν can perform a pickup of r at orig immediately after stop s_i without breaking any of the vehicle's constraints. Analogously, let $F_p(r) = \{(\nu, j) \in F \times \mathbb{N} \mid j \in \{0, \dots, k(\nu)\} \text{ and dest} \in \mathcal{E}(s_j(\nu))\}$ be the set of candidate dropoff vehicles and associated stops.

In ordinary transfer insertions, the transfer point needs to be in the intersection of two ellipses $\mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$. Thus, for any $(\nu_p, i_p) \in F_p(r)$ and any $(\nu_d, j_d) \in F_p(r)$, every insertion $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$ for every $j_p = i_p, \dots, k(\nu_p) - 1$, every $i_d = 0, \dots, j_d$, and every $x \in \mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$ is feasible. Figure 7.2 shows an ordinary transfer insertion using vehicle ν_1 and ν_2 and a transfer after stops s_{j_p} and s_{i_d}. Any point in the overlap of the orange and purple ellipses is a viable transfer location for these vehicles and stops.

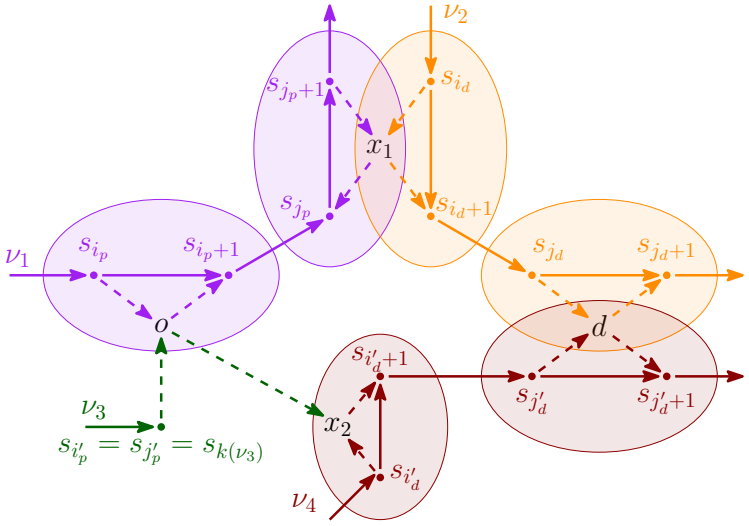


Figure 7.2 Illustration of routes of four vehicles ν_1, ν_2, ν_3, and ν_4, and a request $r = (o, d, t_{\text{req}})$. Depicts an ordinary transfer insertion $(r, x_1, \nu_1, \nu_2, i_p, j_p, i_d, j_d)$ and an after-last-stop transfer insertion $(r, x_2, \nu_3, \nu_4, k(\nu_3), k(\nu_3), i'_d, j'_d)$. Shaded areas indicate detour ellipses. Solid arrows are current routes. Dashed arrows are possible detours.

After-Last-Stop (ALS) Transfers. Any transfer insertion which is not ordinary has either $j_p = k(\nu_p)$ or $i_d = k(\nu_d)$. This means, either the pickup vehicle brings the new rider to the transfer location after its current last stop $s_{k(\nu_p)}$ or the dropoff vehicle picks up the new rider at the transfer location after its current last stop $s_{k(\nu_d)}$. Therefore, we call these insertions *after-last-stop transfer* insertions. Note that transfer insertions with both $j_p = k(\nu_p)$ and $i_d = k(\nu_d)$ will always have a higher cost than the non-transfer insertion $(r, \nu_p, i_p, k(\nu))$ in our model, so we do not have to consider this case.

Focus on the case $j_p = k(\nu_p)$ and $i_d < k(\nu_d)$. In this case, the pickup vehicle is not bound by any detour ellipse since it will have already dropped off all other passengers when it reaches s_{j_p}. Thus, there are no rider constraints at this point. However, the viable transfer points are still limited to the detour ellipse $\mathcal{E}(s_{i_d}(\nu_d))$. Therefore, for any vehicle that can perform a pickup at any index i_p along its route and any $(\nu_d, j_d) \in F_p(r)$, the set of feasible insertions contains $(r, x, \nu_p, \nu_d, i_p, k(\nu_p), i_d, j_d)$ for every $i_d = 0, \ldots, j_d$, and every $x \in \mathcal{E}(s_{i_d}(\nu_d))$. Note that this may include the case of $i_p = k(\nu_p)$, i.e., a *paired* ALS insertion.

Analogously, for any dropoff vehicle ν_d that can perform a dropoff after its last stop $s_k(\nu_d)$ and any $(\nu_p, i_p) \in F_p$, the insertion $(r, x, \nu_p, \nu_d, i_p, j_p, k(\nu_d), k(\nu_d))$ is feasible for every $j_p = i_p, \ldots, k(\nu_p) - 1$, and every $x \in \mathcal{E}(s_{j_p}(\nu_p))$.

In Figure 7.2, a paired ALS transfer insertion is depicted where vehicle ν_3 extends its route after its last stop to pick up the rider at o and bring them to a transfer location x_2 within the ellipse $\mathcal{E}(s_{i'_d}(\nu_4))$. Any location within this ellipse would be a feasible choice for x.

7.3.3. Structure of Algorithm

Whenever a new request $r = (\text{orig}, \text{dest}, t_{\text{req}})$ is issued, the dispatching algorithm is started with the current route state. The best insertion returned by the algorithm is used to update the route state before the next request is processed.

Our dispatching algorithm is outlined in Algorithm 7.1. We always allow a rider to use no-transfer or transfer insertions. Thus, to start with, we use the base KaRRi algorithm to find

Algorithm 7.1 Algorithm Outline. Comments indicate description of implementation.

1: **Input:** $r = (\text{orig}, \text{dest}, t_{\text{req}})$ **Output:** best insertion
2: $\mathcal{I}_{\text{none}} := \text{KaRRi}(r)$ ▷ no-transfer (Chapter 4)
3: $F_p(r), F_p(r) := \text{findPickupAndDropoffVehicles}(r)$ ▷ Section 7.4.2
4: $\mathbb{E} := \text{computeDetourEllipses}(F_p(r), F_p(r))$ ▷ Section 7.4.1
5: $\mathcal{I}_{\text{ord}} := \text{findBestOrdinaryTransferInsertion}(r, F_p(r), F_p(r), \mathbb{E})$ ▷ Section 7.4.2
6: $\mathcal{I}_{\text{alsp}} := \text{findBestALSTransferInsertionPVeh}(r, F_p(r), F_p(r), \mathbb{E})$ ▷ Section 7.4.3
7: $\mathcal{I}_{\text{alsd}} := \text{findBestALSTransferInsertionDVeh}(r, F_p(r), F_p(r), \mathbb{E})$ ▷ Section 7.4.3
8: **return** $\arg\min_{\mathcal{I} \in \{\mathcal{I}_{\text{none}}, \mathcal{I}_{\text{ord}}, \mathcal{I}_{\text{alsp}}, \mathcal{I}_{\text{alsd}}\}} c(\mathcal{I})$

the best no-transfer insertion. Then, we compute the sets $F_p(r)$ and $F_p(r)$ of potential pickup and dropoff vehicles. We compute the detour ellipse for every stop along the routes of these vehicles where a transfer may be made as described in the previous section. Subsequently, we enumerate all ordinary transfer insertions, all ALS transfer insertions for pickup vehicles, and all ALS transfer insertions for dropoff vehicles.

7.4. Exact Transfer Points

In this section, we describe how we can efficiently implement each step of the algorithm mentioned in Section 7.3.3 to solve dynamic ride-pooling with optimal transfers. More precisely, for each request $r = (\text{orig}, \text{dest}, t_{\text{req}})$, we aim to find the single-transfer insertion $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$ with the smallest total cost (at the time of dispatching r).

7.4.1. Computing Detour Ellipses On-the-Fly

Finding exact transfer insertions requires us to compute the detour ellipses $\mathcal{E}(s_i)$ of many stop pairs (s_i, s_{i+1}). To find out whether a vertex $v \in V$ lies within $\mathcal{E}(s_i)$ we need to know the distances $\delta(s_i, v)$ and $\delta(v, s_{i+1})$ in order to check if $\delta(s_i, v) + \delta(v, s_{i+1}) \leq \lambda(s_i, s_{i+1})$. In effect, we need to compute the distances $\delta(s_i, v)$ and $\delta(v, s_{i+1})$ for every $v \in V$.

These two one-to-all shortest path problems can simply be solved by running a forward Dijkstra search rooted at s_i and a reverse Dijkstra search rooted at s_{i+1}. The Dijkstra searches can be stopped once a vertex $v \in V$ with $d[v] > \lambda(s_i, s_{i+1})$ is settled. During the searches, we memorize which vertices have been settled. For every vertex settled by both searches, we check whether $\delta(s_i, v) + \delta(v, s_{i+1}) \leq \lambda(s_i, s_{i+1})$ to determine if $v \in \mathcal{E}(s_i)$.

As an alternative, we can use the one-to-all speedup technique PHAST (see Section 3.1.4). We run a forward PHAST search rooted at s_i as well as a reverse PHAST search rooted at s_{i+1}. Then, we check whether $v \in \mathcal{E}(s_i)$ for every $v \in V$ using the computed distances. Note that PHAST queries cannot be pruned using $\lambda(s_i, s_{i+1})$ like the Dijkstra searches.

As proposed by the authors of PHAST, the queries can be accelerated using both instruction- and thread-level parallelism (cf. Section 3.1.4). In contrast to PHAST, it is notoriously difficult to apply multi-threading to speed up Dijkstra queries with good scalability [MS03]. Similarly, bundling Dijkstra queries for vectorized edge relaxations only works well if the sources are close to each other in the graph and thus have overlapping shortest-path trees. Unfortunately, this is not the case for arbitrary stops.

Note that both approaches provide the shortest-path distances between vehicle stops and transfer points that are later required to compute the cost of an insertion.

7.4.2. Optimal Ordinary Transfers

To find all ordinary transfers, we need to compute the intersection of the detour ellipses of stops of pickup vehicles in $F_p(r)$ and dropoff vehicles in $F_p(r)$, as described in Section 7.3.2.

The LOUD no-transfer dispatcher provides a way to compute the sets $F_p(r)$ and $F_p(r)$ of potential pickup and dropoff vehicles. For both orig and dest, we run a forward and reverse BCH search that identifies the vehicles and stops at which a detour can be made to perform the pickup or dropoff. Elliptic pruning speeds up these queries and limits the size of $F_p(r)$ and $F_p(r)$ (cf. Section 4.3.2). These BCH searches also provide the distances needed to compute the detour made for the pickup and the dropoff.

To facilitate intersecting ellipses, we sort every ellipse by vertex ID. Then, any intersection $\mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$ can be constructed with a linear sweep over $\mathcal{E}(s_{j_p}(\nu_p))$ and $\mathcal{E}(s_{i_d}(\nu_d))$.

If $i_p \neq j_p$ and $i_d \neq j_d$, the distances between stops and transfer points computed during the ellipse reconstruction suffice to calculate the cost of the insertion. In the case of a *paired* insertion, i.e., $i_p = j_p$ or $i_d = j_d$, we need to additionally know the distances from orig to the transfer point x or the distance from x to dest, respectively. For paired insertions, we first assume $\delta(\text{orig}, x) = 0$ or $\delta(x, \text{dest}) = 0$ and compute a lower bound for the cost of the insertion. If this lower bound is worse than the best known cost, we can safely discard it. Otherwise, we compute the distance $\delta(\text{orig}, x)$ or $\delta(x, \text{dest})$ using a point-to-point CH query.

7.4.3. Optimal After-Last-Stop Transfers

As described in Section 7.3.2, in an ALS insertion, a transfer point may be any location within the detour ellipse of the non-ALS vehicle. Here, we focus on the case that the pickup vehicle ν_p is the ALS vehicle and brings the rider to a transfer point in an ellipse of a dropoff vehicle ν_d. Then, we need to know the distances from the last stop $s_{k(\nu_p)}$ to every location in this ellipse. Extending this to all possible pickup vehicles in $F_p(r)$ and dropoff vehicles in $F_p(r)$, we get a many-to-many shortest path problem where the set of sources $\mathcal{L}$ contains all last stops of pickup vehicles, while the set of targets $\mathcal{T}$ is the union of all eligible ellipses, i.e.,

$$\mathcal{L} = \{s_{k(\nu_p)} \mid (\nu_p, \_) \in F_p(r)\}, \qquad \text{and} \qquad \mathcal{T} = \bigcup_{(\nu_d, j_d) \in F_p(r)} \bigcup_{i_d \in \{0, \dots, j_d\}} \mathcal{E}(s_{i_d}(\nu_d)).$$

We utilize RPHAST for this problem. We run the selection phase once for $\mathcal{T}$ and then run a query from every $l \in \mathcal{L}$. We can bundle and vectorize these queries (cf. Section 3.1.4).

A pickup vehicle may also perform both the pickup and the trip to the transfer after its current last stop in a paired ALS insertion $(r, x, \nu_p, \nu_d, k(\nu_p), k(\nu_p), i_d, j_d)$. In this case, we need to know the distance from orig to every transfer point x. Thus, we run a single RPHAST query from orig to $\mathcal{T}$. Additionally, we need to identify vehicles ν_p that need to be considered for a pickup after their last stop as $F_p(r)$ is not guaranteed to contain all of them. For this purpose, we utilize BCH searches as proposed by KaRRi (cf. Section 4.7). We construct bucket entries for every last stop. Then, a single reverse BCH query rooted at orig computes the distances from all last stops to orig. We prune the set of viable vehicles by comparing cost lower bounds based on these distances to the best known cost.

Now consider the case that the dropoff vehicle goes to the transfer after its last stop while the pickup vehicle incorporates the transfer somewhere along its existing route. Then, the ALS insertion will always be paired since the dropoff must necessarily be performed after the transfer. Thus, we can use the same techniques outlined for paired ALS insertions above.

7.4.4. Speeding up Enumeration of Insertions

Computing the cost for a transfer insertion takes much longer than for a no-transfer insertion. Thus, we describe three ways to speed up the computation of insertions and their cost. We focus on ordinary insertions but all techniques are applicable to ALS insertions, too.

Cost Bounds. For each request r, our algorithm first finds the best no-transfer insertion. The cost of this no-transfer insertion serves as an upper bound for the best cost for r.

Consider a set of possible transfer insertions $(r, x, \nu_p, \nu_d, i_p, j_p, i_d, j_d)$ for fixed ν_p, ν_d, i_p, j_p, i_d, and j_d, and $x \in \mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$. We can obtain a lower bound on the cost of any insertion in this set by applying the cost function with lower bounds on the distances from and to any transfer point x. If this lower bound cost is already worse than the best no-transfer cost, we do not have to consider any of the individual insertions in the set. To get lower bounds on the distances from and to any feasible transfer point, we simply memorize the smallest such distances seen while intersecting the ellipses.

Pareto-Dominance between Transfer Points. We find that many transfer points can never lead to a best insertion as there are other transfer points which are guaranteed to lead to better insertions. We devise a measure of Pareto-dominance between transfer points within the same ellipse intersection that allows us to exclude these dominated transfer points.

▶ **Definition 11.** *Let* $x_1, x_2 \in \mathcal{E}(s_{j_p}(\nu_p)) \cap \mathcal{E}(s_{i_d}(\nu_d))$. *Then,* x_1 dominates x_2 *if*

$$\delta(s_{j_p}, x_1) + \delta(x_1, s_{j_p+1}) < \delta(s_{j_p}, x_2) + \delta(x_2, s_{j_p+1}), \tag{7.1}$$

$$\delta(s_{i_d}, x_1) + \delta(x_1, s_{i_d+1}) < \delta(s_{i_d}, x_2) + \delta(x_2, s_{i_d+1}), \text{ and} \tag{7.2}$$

$$\delta(s_{j_p}, x_1) + \delta(x_1, s_{i_d+1}) < \delta(s_{j_p}, x_2) + \delta(x_2, s_{i_d+1}). \tag{7.3}$$

▷ Claim 12. If x_1 dominates x_2, then

$$c(\mathcal{I}_1 = (r, x_1, \nu_p, \nu_d, i_p, j_p, i_d, j_d)) < c(\mathcal{I}_2 = (r, x_2, \nu_p, \nu_d, i_p, j_p, i_d, j_d)).$$

Proof. The insertions $\mathcal{I}_1$ and $\mathcal{I}_2$ differ only in the transfer point. Thus, if the vehicle detour as well as the rider trip time incurred for x_1 is smaller than for x_2, the cost of $\mathcal{I}_1$ will be smaller than that of $\mathcal{I}_2$.

Conditions (7.1) and (7.2) ensure that the vehicle detour is smaller for x_1 than for x_2. This also guarantees that the added trip times for existing passengers of ν_p and ν_d will not be larger for x_1 than for x_2.

For the rider trip time, it suffices to make sure that the arrival time at s_{i_d+1} is smaller for x_1 than for x_2. This is more complex than the issue of detours, though, since the departure time at the transfer point is determined by whether the pickup vehicle or the dropoff vehicle arrives sooner. Assume that the pickup vehicle departs at s_{j_p} at time $t_{\text{dep}}(s_{j_p})$ after making the detour for the pickup. Similarly, let $t_{\text{dep}}(s_{i_d})$ describe the time at which the dropoff vehicle leaves s_{i_d}. Then, the arrival time of the pickup and dropoff vehicles at transfer point x can be characterized as $t^p_{\text{arr}}(x) := t_{\text{dep}}(s_{j_p}) + \delta(s_{j_p}, x)$ and $t^d_{\text{arr}}(x) := t_{\text{dep}}(s_{i_d}) + \delta(s_{i_d}, x)$, respectively. The trip time until s_{i_d+1} is $t_{\text{trip}}(x) := \max\left\{t^p_{\text{arr}}(x), t^d_{\text{arr}}(x)\right\} + \delta(x, \mathrm{s}_{i_d+1})$. Thus, we need to show that $t_{\text{trip}}(x_1) < t_{\text{trip}}(x_2)$ if x_1 dominates x_2. We consider two cases:

Case 1 : Assume $t^p_{\text{arr}}(x_1) \leq t^d_{\text{arr}}(x_1)$. Then,

$$\begin{aligned} t_{\text{trip}}(x_1) &= t^d_{\text{arr}}(x_1) + \delta(x_1, s_{i_d+1}) \\ &= t_{\text{dep}}(s_{i_d}) + \delta(s_{i_d}, x_1) + \delta(x_1, s_{i_d+1}) \\ &\overset{(7.2)}{<} t_{\text{dep}}(s_{i_d}) + \delta(s_{i_d}, x_2) + \delta(x_2, s_{i_d+1}) \\ &= t^d_{\text{arr}}(x_2) + \delta(x_2, s_{i_d+1}) \leq t_{\text{trip}}(x_2). \end{aligned}$$

Case 2 : Assume $t^p_{\text{arr}}(x_1) > t^d_{\text{arr}}(x_1)$. Then,

$$\begin{aligned} t_{\text{trip}}(x_1) &= t^p_{\text{arr}}(x_1) + \delta(x_1, s_{i_d+1}) \\ &= t_{\text{dep}}(s_{j_p}) + \delta(s_{j_p}, x_1) + \delta(x_1, s_{i_d+1}) \\ &\overset{(7.3)}{<} t_{\text{dep}}(s_{j_p}) + \delta(s_{j_p}, x_2) + \delta(x_2, s_{i_d+1}) \\ &= t^p_{\text{arr}}(x_2) + \delta(x_2, s_{i_d+1}) \leq t_{\text{trip}}(x_2). \end{aligned}$$

◀

In road networks with heterogeneous travel speeds, there can easily be locations that are not well accessible to both the pickup and the dropoff vehicle (e.g., side roads within a neighborhood), which leads to them being dominated by locations on more easily accessible roads in the vicinity. Note that a test for domination between two transfer points can be computed quickly. Thus, it is worth filtering transfer points based on domination before performing the much more expensive calculation of insertion costs.

Parallelization. Computing the cost of all feasible insertions can be trivially parallelized. We iterate over pairs of pickup and dropoff vehicles in $F_p(r) \times F_p(r)$ in parallel with the same thread computing the cost for all insertions of one vehicle pair. Each thread keeps a thread-local best insertion seen. When all threads are done, the best of the thread-local insertions is chosen. Pareto-dominance and cost bounds can still be applied by each thread.

7.5. Heuristic Transfer Points

In this section, we describe a way to reduce running times by heuristically choosing a subset of transfer points based on CHs. CHs aim to order vertices by their importance for shortest paths in the road network during construction. Therefore, we can assume that vertices of high CH rank can be reached easily and may be good candidates for transfer points.

In our heuristic, we only consider the k percent of vertices in the network with the highest ranks as transfer points. Let $V_{sorted} = \langle v_1, \ldots, v_n \mid v_j \in V, \text{rk}(v_j) \geq \text{rk}(v_{j+1}) \rangle$. Let $V_{k\%}$ denote the subset of the first $nk/100$ vertices. When computing the potential transfer points for a request, for every ellipse $\mathcal{E}(s_i)$, candidate vertices are now limited to $\mathcal{E}(s_i) \cap V_{k\%}$.

This restriction provides two advantages with regard to computation time. Firstly, the one-to-all PHAST queries used to reconstruct detour ellipses (see Section 7.4.1) can now stop after scanning only the top $k\%$ of vertices. Secondly, the average size of each restricted detour ellipse will be much smaller which reduces the number of transfer insertions that need to be tried. As a trade-off, the heuristic may negatively affect the solution quality since potentially good transfer locations that are not in the top $k\%$ of vertices may be missed. The parameter k allows an interpolation between the reduced running time and loss in quality.

7.6. Experimental Evaluation

We experimentally evaluate our approach on realistic input instances for dynamic ride-pooling. In this section, we refer to our approach as *KaRRiT* (KaRRi with Transfers).

7.6.1. Experimental Setup and Benchmark Instances

Our source code[1] is written in C++20 and compiled with GCC 11.5 using `-O3`. We use two machines for separate experiments, both running Rocky Linux 9.5. Machine A has 64 GiB of memory and a single 16-core AMD Ryzen 9 3950X processor at 3.5GHz. Machine B has 512 GiB of memory and a single 32-core Intel Xeon Gold 6314U processor at 2.3 GHz. We use 32-bit distance labels and the AVX2 SIMD instruction set with 256-bit registers to compute up to 8 operations in one vector instruction.

We evaluate KaRRiT on the `Berlin-1pct` (`B-1%`), and `Berlin-10pct` (`B-10%`) request sets [BSW21] that, respectively, represent ride-pooling demand for 1% and 10% of the population of the Berlin metropolitan area on a weekday. Both sets cover a time window of 30 hours and follow a realistic distribution of demand on a weekday regarding both time and space. The `Berlin-1pct` and `Berlin-10pct` request sets contain 16569 and 149185 requests, respectively. Images on the distribution of requests over the observation period can be found in Figure 4.4. For a more detailed description of the request sets and the associated road network, see Section 4.9. We use 1000 vehicles for `Berlin-1pct` and 10000 vehicles for `Berlin-10pct`. Each vehicle has a capacity of four and a service time interval covering the entire 30 hours. The initial locations of all vehicles are drawn uniformly at random. We compute a contraction hierarchy of the road network using the open-source library RoutingKit[2]. This takes less than a minute for Berlin.

Unless otherwise specified, we use the following default assumptions: We use the model parameters decided on in the parameter tuning experiments in Section 10.3. Vehicles have a default capacity of four, and the initial location of every vehicle is an edge drawn uniformly at random in the road network of the service area. The service time of every vehicle is set to encompass the entire observation period. Although KaRRiT supports meeting points (cf. Chapter 4), we do not use them in these experiments due to the already very large complexity of ride-pooling with transfers.

Note that we do not use a mode choice model as described in Chapter 6, as it is unclear how transfers affect the preferences of travelers. We compute the trip with the lowest cost using walking, ride-pooling without a transfer, or ride-pooling with a transfer, and assume the traveler accepts the trip. The Berlin input instances are configured to represent only ride-pooling demand (instead of all travel demand), which makes this an acceptable assumption. Our evaluation focuses on the running time of our dispatcher. A more detailed quality analysis is left for future work.

7.6.2. Analysis of Optimizations for the Exact Algorithm

We analyze the impact of individual features of our algorithm on the running time. For this, we run experiments on machine A. We use the `Berlin-1pct` instance as running times for a non-optimized implementation are infeasible on `Berlin-10pct`. We consider the four main components of our algorithm (lines 4-7 in Algorithm 7.1) separately.

---

[1] Available at `https://github.com/molaupi/karri/tree/karrit`.
[2] `https://github.com/RoutingKit`.

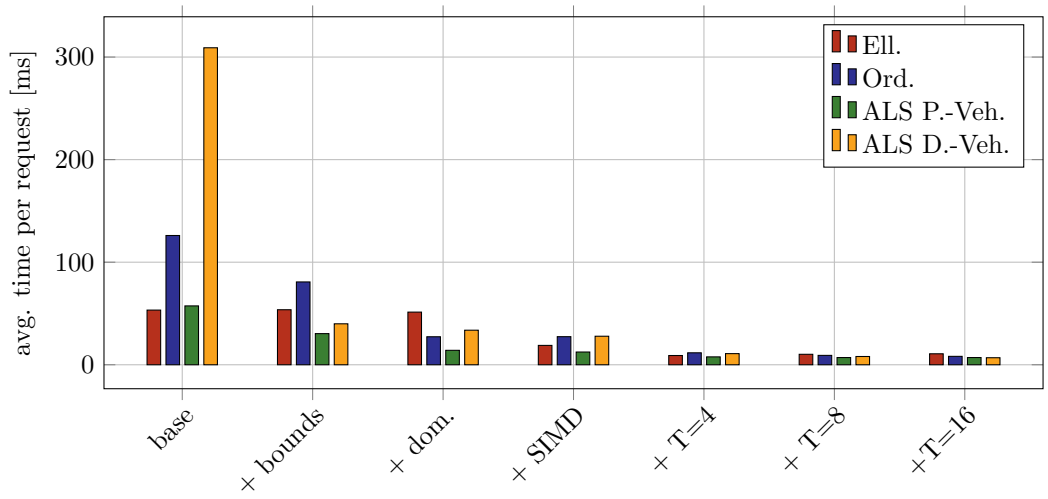


Figure 7.3 Average runtime per request of components of KaRRiT in incrementally more efficient configurations. The configuration "base" uses PHAST and RPHAST wherever possible but none of the other optimizations described in Section 7.4. The other configurations add the specified optimization to the configuration to their left. (The configurations "T=n" each add multi-threading with n threads to the "+ SIMD" configuration.)

To start with, utilizing PHAST speeds up the computation of detour ellipses by a factor of about 2.5 compared to the Dijkstra-based implementation (cf. Section 7.4.1). Thus, we start from this baseline of using PHAST and RPHAST, and illustrate the speedups achieved with our additional measures in Figure 7.3.

Applying cost bounds (see Section 7.4.4) has the greatest effect on ALS insertions for the dropoff vehicle with a speedup of about 7.74. The enumeration of ALS insertions for the pickup vehicle and ordinary insertions become 1.89 and 1.56 times faster, respectively.

Additionally checking transfer points for Pareto-dominance speeds up the enumeration of ordinary transfer insertions, ALS transfer insertions for the pickup vehicle, and ALS transfer insertions for the dropoff vehicle by factors of 2.96, 2.15, and 1.19, respectively. This shows that transfer point dominance and cost bounds work well in combination. In fact, the two methods seem to synergize as the speedups for transfer point domination are highest for the components where cost bounds provide the least benefit.

Bundling PHAST queries and employing SIMD vector instructions to run up to eight searches simultaneously speeds up ellipse reconstruction by a factor of 2.72. However, it has almost no effect on the ALS steps because their runtime is dominated by work not affected by SIMD speedups like enumerating insertions or the RPHAST selection phase.

Using four threads for parallel PHAST queries and enumeration of insertions leads to mediocre speedups, ranging from 1.62 for ALS transfer insertions for the pickup vehicle, to 2.56 for the dropoff vehicle. For the PHAST queries used during ellipse reconstruction, this can be attributed to the fact that PHAST can only settle vertices in parallel within individual CH levels. Since our road network is comparatively small, the sizes of CH levels are limited and synchronization overhead may be large. For the enumeration of transfers, speedups are again held back by work that we did not parallelize, e.g., the RPHAST selection phase. These effects contribute to the lack of scalability to larger numbers of threads. With 16 threads we hardly see any speedups compared to 4 threads. The only component that benefits from more threads is the enumeration of ALS transfer insertions for the dropoff vehicle, which spends a lot of time on trivially parallelizable cost computation.

Table 7.1 Comparison of running times and key performance metrics for main components of exact (E) and heuristic (H) variants of KaRRiT on the `Berlin-1pct` and `Berlin-10pct` instances. Shows average running time per request for each step as well as average total running time per request in milliseconds. Additionally, shows number of insertions tried (ins) for ordinary and ALS, as well as number of potential transfer points ($|\mathcal{T}|$) for ALS (both in thousands).

inst.	alg.	Ell.	Ord.		ALS P.-Veh.			ALS D.-Veh.			Total
		time [ms]	ins [$\cdot 10^3$]	time [ms]	$\lvert\mathcal{T}\rvert$ [$\cdot 10^3$]	ins [$\cdot 10^3$]	time [ms]	$\lvert\mathcal{T}\rvert$ [$\cdot 10^3$]	ins [$\cdot 10^3$]	time [ms]	time [ms]
`B-1%`	E	10.4	3.6	8.6	60.6	2.1	7.8	31.3	14.0	6.9	34.0
	H	2.7	0.3	1.0	1.2	0.3	0.9	0.8	1.6	0.8	5.7
`B-10%`	E	58.2	42.8	85.6	148.6	80.5	37.0	93.9	189.5	73.4	255.2
	H	16.0	5.1	9.3	2.8	9.9	2.8	1.5	26.3	3.7	32.4

7.6.3. Effect of Heuristic on Running Time

We compare the running time of the heuristic variant (H) of KaRRiT (see Section 7.5) with the exact variant (E). The experiments were run on machine B with 32 threads, all optimizations, and $k = 10\%$ for H. Running times per request for different steps are shown in Table 7.1.

Restricting transfer points to vertices of high CH rank using H reduces the running time of the ellipse reconstruction by a factor of about four compared to the exact solution. As discussed in Section 7.5, this is caused by the PHAST queries having to perform only one tenth of the work, settling the top $k = 10\%$ of vertices in the CH. Since the upper CH levels are small and only vertices within the same CH level can be settled in parallel, the ellipse reconstruction in H does not benefit from multi-threading. This limits the speedup over E to four instead of being closer to ten. Due to reduced ellipse sizes, H computes the cost of about eight times fewer insertions than E. In the ordinary transfer step, the smaller ellipse sizes also reduce the time needed for intersecting ellipses. Additionally, the set of targets $\mathcal{T}$ for RPHAST in the ALS transfer components is around two orders of magnitude smaller for H than for E, which speeds up the selection and query phases of RPHAST.

For E, the set of transfer points $\mathcal{T}$ includes almost all locations in the road network[3]. Thus, a PHAST query without an expensive RPHAST selection phase may perform better. For H, the number of transfer points is small enough to warrant using RPHAST.

Note that the number of insertions tried can exceed the number of transfer points because every pickup or dropoff vehicle may be matched with any transfer point. In turn, the transfer point pruning strategies outlined in Section 7.4.4 reduce the number of insertions tried. Interestingly, these pruning strategies appear to be more successful for E than for H as the ratio between the number of insertions tried and the number of potential transfer points is much larger for H than for E. This may be explained by the fact that H already heuristically selects good transfer points based on CH rank such that they may be harder to prune using cost bounds or especially transfer point domination.

[3] It is possible that $|\mathcal{T}| > |V|$ since our implementation of KaRRiT uses edges, not vertices, as transfer locations. We still always have $|\mathcal{T}| \leq |E|$.

Table 7.2 Comparison of dispatch quality on `Berlin-1pct` and `Berlin-10pct` between the baseline without transfers KaRRi (K) and KaRRiT in the exact (E) and heuristic (H) variants. Shows average rider wait time and trip time in minutes and seconds, average trip time relative to shortest orig-dest path, average vehicle operation time in hours and minutes, and vehicle occupancy rate averaged over time. Last column gives average total running time of the dispatcher per request.

inst.	alg.	wait [mm:ss]	trip [mm:ss]	trip (rel.)	drive [hh:mm]	occ	total [ms]
B-1%	K	03:30	16:30	1.46	04:00	0.886	0.2
	H	03:29	16:32	1.47	03:59	0.894	5.7
	E	03:30	16:35	1.47	03:58	0.898	34.0
B-10%	K	02:43	15:33	1.72	02:55	1.061	0.4
	H	02:45	15:52	1.75	02:49	1.109	32.4
	E	02:48	16:02	1.78	02:47	1.127	255.2

7.6.4. Impact of Transfers on Solution Quality

We evaluate the impact of allowing transfers on the solution quality of dynamic ride-pooling by experimentally comparing KaRRiT in the exact (E) and heuristic (H) variants to the no-transfer baseline KaRRi. Results for experiments run on machine B are shown in Table 7.2. Note that we give only preliminary results on dispatch quality, since the focus of this work is on the algorithmic aspects. In the future, we plan to use our new fast algorithm to consider the effect of transfers in more detail with a larger variety of input instances.

Considering the running time of the algorithms, KaRRi is up to two orders of magnitude faster than the heuristic variant, and three orders faster than the exact variant. The running time of H can still be considered practical while E is too slow for real-world application.

On `Berlin-1pct`, transfers appear to have almost no effect on the solution quality. This can be attributed to the fact that the density of requests is small with about one request per minute. Thus, it is unlikely that two riders share a vehicle (as evidenced by the low vehicle occupancies). Moreover, a vehicle may always be available to take a passenger to their destination directly, which often outperforms any transfer insertions. In additional experiments with artificially increased request densities, we were able to confirm that request density is in fact a deciding factor for the viability of transfers.

As `Berlin-10pct` also provides a much denser request set, we focus on this instance here. When comparing the exact solution E to KaRRi, we see decent improvements in occupancy rates and slight improvements in vehicle operation times. Average occupancy rates increase by 5.9%, and vehicle operation times decrease by 4.6%. This is a significant benefit with respect to the use of space on the roads and the operating costs of the ride-pooling provider. As a drawback, rider trip times increase by 3.1% compared to KaRRi. This matches the expectation that transfers may be good for efficiency at a cost of rider satisfaction.

While these gains may not justify the large running time of E, the heuristic H retains most of the benefits. The improvement for vehicles are about one third smaller than for E, but rider trip times are also less severely affected. Since H is an order of magnitude faster than E, it may therefore be a more viable candidate for a production system.

7.7. Conclusion

KaRRiT provides an efficient algorithmic approach to find optimal single-transfer journeys for the dynamic ride-pooling problem with on-the-fly distance computation. We explore the usage of state-of-the-art shortest-path speedup techniques and propose new pruning techniques for the large solution space. While we only show first results on the benefits of transfers on dense request sets, we find that our approach is suited to conduct experiments on city-scale real-world instances. We aim to use this new opportunity to design more precise studies on the service quality and resource usage of ride-pooling with transfers in cooperation with application experts. This analysis should include considerations on the viability of transfer locations with regard to aspects like safety, accessibility, and efficiency of transfers.

In the future, we would like to improve the efficiency of our exact algorithm, in particular by introducing multi-threading to non-parallelized regions of the algorithm or by introducing a prunable version of the PHAST algorithm to speed up the ellipse reconstruction process. As an alternative to computing all ellipses on-the-fly, we hope to be able to identify good candidates for pairs of pickup and dropoff vehicles by considering the routes of vehicles as sequences of line segments in three dimensional space of location and time. If two routes come close to each other in this setting, the vehicles could be good candidates for a transfer.

Further, various extensions of the problem could be explored: Allowing multi-transfer journeys in dynamic ride-pooling, especially three-leg journeys with high-capacity trunk vehicles and smaller feeder vehicles, may improve dispatch quality. Allowing pre-booking or the batching of requests in a rolling-horizon approach would open up the possibility for local optimizations and could improve dispatch quality by reducing the impact of the online characteristics of the problem.

8 Multimodal Routing for Ride-Pooling and Public Transportation

Summary: The greatest potential of ride-pooling (RP) in future transportation systems lies in complementing public transit (PT), as an integration of both modes would benefit from the efficiency of PT and the flexibility of RP. Though similar systems of restricted scope have been studied for decades, there has not been a lot of work on integrated routing of RP and PT.
We combine our RP dispatcher KaRRi and the multimodal PT framework ULTRA to obtain PaRRot, a routing algorithm that finds combined journeys which use both RP and PT. We identify important differences to traditional multimodal routing with PT and walking, and show that our scenario requires multi-criteria routing that optimizes RP cost and rider trip time. We modify KaRRi to efficiently compute RP trips to all PT stations. A multi-criteria PT query then uses these access trips to find complete multimodal journeys. Experiments show that PaRRot finds combined RP-PT journeys with an overhead of only 2.5ms *compared to a solution without combined journeys. We show that combined journeys can improve system-wide travel times and can reduce the total operation time of the RP vehicle fleet. We find that combined journeys are very useful for riders who previously had no good alternative to a RP-only trip.*

Attribution: This chapter builds upon the student research project "Algorithm Engineering for Software-Defined Public Transportation (PTaxi: Combining Taxi-sharing and Public Transit)" by Ha Linh Nguyen [Ngu25], which was influenced by some previous work by Patrick Steil. A previous bachelor thesis on the topic by Louis Tiede [Tie22a] followed a different approach and has little overlap with this chapter. The text of this chapter was fully written by the author of this dissertation. The algorithm structure was jointly conceptualized by Ha Linh Nguyen and the author of this dissertation, with discussions and feedback by Peter Sanders. The author of this dissertation additionally contributed the analysis of problem properties in Sections 8.4.1 and 8.4.2 as well as the speedup techniques in Section 8.5. Ha Linh Nguyen implemented a preliminary version of computing access RP trips. The author of this dissertation extended the implementation to compute full combined journeys and implemented more-criteria routing as well as techniques for speeding up the computation of access RP trips. The author of this dissertation conducted the experimental evaluation.

8.1. Introduction

The economies of scale of public transportation make it an efficient mode of transportation with respect to many sustainability indicators like fuel use, emissions, use of public land, social equity, and more [Lit21]. Thus, sustainable transportation systems need to be designed in a way that makes public transportation more attractive than motorized individual transportation (such as personal cars) for the majority of travelers. Municipalities can work

towards this goal by making cars less attractive, e.g. by introducing congestion charges or removing parking spaces. At the same time, alternatives to the car need to be made more attractive to uphold or improve the overall quality of transportation, e.g. by expanding cycling infrastructure or planning new developments around transit (*transit-oriented development*).

Improving coverage of public transit is an important step towards sustainable cities [Che+13; VB19], but public transit cannot cover all travel demand. Due to the aforementioned economies of scale, a minimum level of demand is required to justify introducing a new bus line or increasing the frequency of service on an existing part of the network. Some areas in a city may not provide this level of demand. Thus, there is a risk that measures to reduce car traffic might make transportation less accessible and less convenient in some areas.

In these areas, on-demand shared transportation could fill the gap. Fast, convenient, and affordable on-demand transportation is seen as a way to complement fixed public transit and improve the overall efficiency of a transit system [Car+23]. In areas, where there is too little demand for fixed transit, travelers could instead use an on-demand service to access a transit hub in the vicinity. This paradigm has been studied under the terms *flexible transit* or *demand-responsive transit* [Kof04]. Traditionally, these terms refer to any transit system that contains a component that does not follow a fixed schedule or route, but responds to explicitly stated current demand levels. For example, a bus might only visit a certain stop if a traveler indicated that they would like to board the bus at this station, e.g. by pressing a button or calling the bus agency. In the past, these systems have been limited in scope, both in theory and in practice [Kof04].

Modern technology, most importantly internet-capable mobile phones with location services, have made the idea of on-demand mobility much more attractive. Ride-pooling is considered an important future mode of transportation, as it promises to effectively respond to the existing travel demand while being more efficient than non-pooled taxis. Ride-pooling cannot achieve the same levels of efficiency as public transportation, though. Thus, it is natural to integrate ride-pooling as the missing on-demand mode of transport for sustainable transportation systems based on public transit.

In this work, we combine state-of-the-art routing solutions for ride-pooling and public transportation to obtain an algorithm that is capable of finding multimodal journeys with fixed transit and on-demand ride-pooling in real time. We base our approach on the ULTRA framework for multimodal PT journey planning [Bau+23] and focus on the case of using ride-pooling as a *feeder mode* in the beginning and end of a public transit journey. We identify key differences compared to traditional feeder modes that make it impossible to use many existing preprocessing techniques. However, we show that our ride-pooling dispatcher KaRRi (see Chapter 4) can be extended to efficiently find feeder trips using ride-pooling. On the basis of these feeder trips, a single many-to-one more-criteria PT query can find the optimal combined journey. In an experimental evaluation of our approach PaRRot, we show that we can find multimodal journeys in milliseconds on a large urban network with tens of thousands of ride-pooling vehicles and hundreds of thousands of requests per hour. We compare our approach to a transportation system without combined journeys and find that the introduction of combined journeys improves the average trip time and the operational costs for the ride-pooling fleet.

8.1.1. Related Work

We give an overview on related work in the fields of routing for public transportation and planning of multimodal journeys.

Routing Algorithms for Public Transportation. Routing in public transportation (PT) networks is a well studied problem with many mature algorithms that solve different variants of the problem [Bas+16]. Most commonly, PT routing algorithms solve the earliest-arrival problem. Given a source location, target location, and departure time, the algorithm is tasked with finding a PT journey from the source to the target that arrives as soon as possible using PT connections in a fixed timetable. On the most basic level, this problem can be solved by running general shortest-path algorithms on a graph representation of the timetable [BJ04; SWW00]. State-of-the-art methods preprocess the timetable data without modeling it as a graph explicitly [Bas+10; BHS16; Dib+18; DPW15a; Wit15]. These algorithms make use of the fact that the timetable can be represented with more structure than the graph formulation to allow for faster access of relevant data and pruning of the search. For example, they achieve better memory locality by processing the entire trip of a public transit vehicle as a whole instead of considering it as individual connections between subsequent stations. There are many recent advances in improving the query times or preprocessing times of PT routing algorithms that use additional information on the structure of the network, e.g. using goal-directed search [Gro+26] or a hierarchical representation of the network [Ste25].

For many PT algorithms, *more-criteria* variants have been proposed where the goal is to find a set of journeys that are Pareto-optimal with respect to multiple criteria. A journey J for a given source, target, and departure time is Pareto-optimal for a set of criteria if there is no other journey J' for the same source, target, and departure time that is at least as good as J in every criterion and better than J in at least one criterion. Beyond earliest arrival time, common objectives are the number of transfers [Dib+18; DPW15a; Pyr+08; Wit15], fare price [DPW15a; MS07], or total walking time [PS22; PV19]. Generally, multiple criteria require algorithms to maintain and propagate more information than earliest-arrival variants. For every station S, algorithms need to know not just the journey with the best known earliest arrival time, but a set of Pareto-optimal known journeys. Often, these are represented as a bag of journey labels at each station which can dynamically grow or shrink during the execution of the algorithm [DPW15a; Mül+07; Pyr+08]. These aspects make more-criteria routing for arbitrary criteria much more expensive than earliest-arrival routing [DPW15a; PS22]. In the worst case, the number of Pareto-optimal journeys can be exponential in the network size. However, usual criteria often do not lead to exponential work in practice. In many works, algorithms optimize the earliest arrival time in conjunction with one additional criterion. The number of transfers (i.e., the number of times that a traveler has to transfer from one public transit vehicle to another during their journey) should be considered a special case for this additional criterion, since some approaches are able to optimize the number of transfers "for free" as they scan the PT network in rounds with exactly one transfer after each round [DPW15a; Wit15].

Multimodal Journey Planning. Most journey planning algorithms consider transfers not just at the exact same station, but they allow travelers to walk from a station to other nearby stations for a transfer. Traditionally, the set of permissible walking transfers is precomputed and represented in a transfer graph [Bas+10; DMS08; DPW15a]. This can be considered a restricted form of *multimodal* journey planning, since a journey can consist of two modes of transportation, public transit and walking, in an interleaved manner.

Unrestricted multimodal journey planning allows the use of a transfer mode for arbitrary trips between stations. The transfer mode (e.g. walking, bike-share, etc) does not use a schedule, which requires an approach that differs from planning the PT legs of the journey. For a simple extension of the traditional model, the transitive closure of the

transfer graph can be used. However, in dense urban areas, this results in an excessive number of transfers [WZ17]. Thus, journey planners opt to run shortest-path queries on an underlying network for the transfer mode to find unrestricted transfers on-the-fly. These approaches preprocess the network to limit the running time of the shortest-path queries. This can be achieved by contracting the network to a core that contains shortcuts to skip over unimportant vertices [Del+13; DPW15b] or by using a two-hop labeling [PV19]. The ULTRA framework [Bau+23] goes one step further by precomputing a set of transfer shortcuts between stations that provably suffice to find any Pareto-optimal journey in the network. These shortcuts comprise a transfer graph, which removes the need for on-the-fly shortest-path queries in a transfer network and allows many journey planning algorithms to employ unrestricted transfers without additional changes.

Combining Dynamic Ride-Pooling and Public Transportation. *Dynamic ride-pooling* (RP) is a novel transportation mode in which taxi-like vehicles are dynamically dispatched to transport travelers without fixed routes or a fixed schedule. For a summary of research on ride-pooling, see Chapter 2. In a multimodal combination of public transportation and ride-pooling (PT+RP), journeys can consist of interleaved legs of public transportation and ride-pooling. The proposed benefit of PT+RP is that ride-pooling may augment a PT network for journeys that are not well covered to make PT more flexible while maintaining its economy of scale [Göd+23; Kos+24; Xu+25]. Xu et al. [Xu+25] find that ride-pooling may be better suited for multimodal integration with PT than non-pooled ride-hailing, since the share of multimodal trips is observed to be higher for ride-pooling than for ride-hailing in a real trip data set. In the rest of this section, we focus on the PT+RP routing problem which asks to find the best combined journey for given travel requests.

PT+RP routing differs significantly from the aforementioned multimodal planning with unrestricted transfers. Many of the preprocessing techniques used for unrestricted transfers cannot be applied, as they assume fixed travel times in the network of the transfer mode. Although some approaches consider taxis as a transfer mode [Del+13; DPW15b], they do not take into account that the travel times of taxis (and RP) inherently depend on the volatile current state of the vehicle routes. Due to this added complexity, we focus on the case where RP is only used as a *feeder mode*, i.e., RP may be used at the beginning and at the end of a trip (with PT in between), but not to transfer between PT stations. This is by far the most studied scenario for the PT+RP problem [Lor+23; LQ10; LWB96].

A traditional paradigm related to PT+RP is *flexible transit* [Kof04]. In flexible transit, a PT network is augmented with certain dynamic features. For example, a bus route might only be serviced if a traveler previously announced that they would like to use the route (e.g. by calling the bus agency). An interesting special case of flexible transit is the *demand responsive connector* (DRC) [Kof04; LQ10]. Here, a demand responsive transport vehicle (DRT, see Section 2.2.3) serves as a feeder to a PT system that has fixed lines and a fixed schedule. In the DRC routing problem, a number of requests with an origin location and latest service time are given. The task is to find a route for the DRC vehicle to pick up riders and bring them to the same transit station. The route should optimize goals such as the distance driven, operational cost, or number of serviced travelers. In dynamic settings, not all requests need to be known in advance. Solution approaches for the DRC follow usual trends for vehicle routing problems [TV14]: Small instances can be solved by formulating mathematical programs and applying appropriate solver software [Li+21; LQ10; LS17; ZG26]. For larger instances, insertion heuristics that greedily insert requests one-by-one can be used [LLW05; LQ10; LS17]. DRCs share some characteristics with PT+RP, since in both

approaches travelers use a dynamically scheduled vehicle to access a PT network. However, the scope of a DRC is much smaller, as there is usually only one vehicle that operates in a limited area with a single transit station. Additionally, DRCs are usually not concerned with a traveler's whole journey, but only with an access trip to the first PT station. PT+RP considers the whole network and a fleet of vehicles, which promises to reduce operational costs and improve trip quality. Lee and Savelsbergh [LS17] show that even a DRC with more than one station has significant advantages over a traditional single-station DRC.

An early work on PT+RP routing by Liaw et al. [LWB96] considers dial-a-ride vehicles (DARP, see Section 2.2.1) that bring riders to buses with fixed lines. They focus on a fixed pre-booking period in which requests may come in. During this period, a greedy heuristic tries to accommodate as many requests as possible in the planned DARP schedule. Then, between the pre-booking period and the execution of the plan, the schedule is post-optimized using simulated annealing. Results show that the combination of DARP and fixed-line buses improve upon solutions using DARP only. This early work gives important directions for future research like restricting dynamic vehicles to act as a feeder mode (instead of a full transfer mode), and using unimodal solutions to prune the search for multimodal solutions.

Stiglic et al. [Sti+18] combine ride matching (see Section 2.2.3) with PT. The approach is concerned with matching single drivers to single riders where both the driver and the rider have origin and destination locations. The authors extend a previous formulation of ride matching with meeting points as a bipartite matching problem [Sti+15]. In the extended variant, drivers may drop riders off at transit stations instead of their destination. Additionally, drivers may end their trip at a transit station with a park-and-ride facility. An experimental study concludes that the number of matched rider requests can be increased significantly and that more than 30% of rider trips use a combination with PT. As the observed ride matching problem is much simpler than ride-pooling, an exact solution for known demand of 2000 requests can be solved in a minute.

A Closely Related Approach. Lorente et al. [Lor+23] present an approach that has a very similar understanding of the PT+RP routing problem as this work, since they consider a dynamic version of the problem where requests arrive in an online fashion. RP may be used as a feeder mode in the beginning and in the end of a journey. Alternatively, travelers are allowed to walk to and from stations.

The authors first identify an important difficulty with using RP for an egress trip that takes a traveler from a final PT station S to their destination. It is pointed out that the choice of the best vehicle depends on the unknown future state of the vehicle fleet at the time that the traveler arrives at S. To solve this issue, the algorithm uses a heuristic for the quality of an egress RP trip at the time of processing a request. The computation of the actual egress trip is delayed to a later point in time when the rider reaches their final PT station. The authors propose a vehicle relocation strategy that moves a vehicle to the final PT station before the traveler arrives there, so that a good egress trip will be available.

To route PT+RP journeys, the algorithm uses a rolling-horizon strategy (see Section 2.3.2) where all requests that arrive in a certain time interval are accumulated to form a batch of requests. The authors propose a three-phase routing algorithm to find combined PT+RP journeys for all requests in a batch. In the first phase, for each request, the algorithm heuristically determines a set of access stations where the traveler may board a fixed PT trip and a set of egress stations where the traveler leaves the PT system. Additionally, a set of candidate vehicles is determined for each request based on its access and egress stations. In the second phase, for each request, the algorithm computes every potential RP assignment

for the candidate stations and vehicles as well as potential PT journeys that connect access and egress stations of the request. Here, the aforementioned quality heuristic is used to decide the quality of combined journeys with egress trips. In the third phase, a mixed integer program (MIP) is formed that represents the possible journeys for all requests in the batch. The solution of this MIP chooses assignments that optimize a linear cost function that takes into account different optimization goals like the number of travelers served, their fares, and the detour experienced by existing RP passengers.

The paper includes a study on a large input instance of Barcelona that allows travelers to use RP-only, PT-only, or combined journeys. The results show that combined journeys have lower fares and travel times than RP-only as they avoid congestion in the inner city.

The algorithm by Lorente et al. [Lor+23] is an intricate and well-developed solution for PT+RP routing. The paper points out many important difficulties and directions for future research. However, there are some aspects we would like to improve upon in our solution. Firstly, the optimization algorithm is initialized with heuristically chosen access stations, egress stations, and vehicles. Although the heuristics are not described in detail, the paper suggests that only stations and vehicles in a certain distance radius around a request's origin location are considered. In a fully dynamic system, all stations should be eligible, and the set of beneficial stations should be determined based on the state of the RP vehicle fleet and the available RP trips. Secondly, the algorithm precomputes pairwise station-to-station PT journeys to be used as the middle section of combined journeys. To limit the number of precomputed journeys, these are allowed to have at most two PT legs (i.e., one transfer between different PT vehicles). This restriction can cause the algorithm to miss optimal combined journeys. We will show that there is no need to precompute PT journeys, as modern PT routing algorithms can handle on-the-fly queries for combined journeys without becoming a bottleneck. Finally, the processing times of the proposed algorithm are too long for an interactive application in practice. Even when using the powerful Gurobi solver to solve the MIP with a time limit relative to the batch size, the PT+RP routing algorithm takes about 0.5s per request on average. Due to the batched approach, the response time for each traveler is in the order of seconds. We show that an approach based on an insertion heuristic can achieve much smaller response times to allow real-time dispatching.

8.2. Problem Statement

Our goal is to find multimodal journeys that are comprised of stretches of public transportation (PT) and ride-pooling (RP). For this, we describe the public transportation and ride-pooling models used, formalize combined journeys, and extend our RP cost function accordingly.

8.2.1. Public-Transportation Model

We formalize a public transportation network, the concept of a journey in this network, and the PT routing problem.

Public Transportation Network. A *public transportation network* represents all available public transportation in a given area over a given time frame. It contains of a set of *stations* $\mathcal{S}$ at which travelers can board or alight a public transportation vehicle. Note that works on public transportation routing often use the term "stop" instead of "station". We use "station" for public transportation to distinguish it from stops made by RP vehicles. Similarly, we use upper-case S as the symbol for stations to avoid an overload of the lower-case s used for RP stops.

A *connection* describes a way to move from one station to another in the PT network without an intermittent stop at any other station. A connection has a source station with a departure time and a target station with an arrival time.

A *PT-vehicle trip* T is a contiguous sequence of connections performed by the same PT vehicle. It encompasses a fixed sequence of stations $\langle S_1, \dots, S_{|T|} \rangle$ with monotonically increasing arrival times $t_{\text{arr}}(T, S_i)$ and departure times $t_{\text{dep}}(T, S_i)$. If two PT-vehicle trips serve the same sequence of stations (only at different times), the PT-vehicle trips are considered part of the same *route*, unless they overtake each other[1]. We make sure to use the full term "PT-vehicle trip" to refer to this part of a PT network, since we also use "trip" in the context of ride-pooling (e.g. "trip cost", Section 4.2).

Journey. We now formalize journeys in the PT-network. We focus on journeys that start and end at stations here, and later explain how to extend this to any start or end location in an underlying transportation network.

A *journey* is a sequence of legs $J = \langle L_1, \dots, L_{|J|} \rangle$. Each *leg* L is a sub-segment of a PT-vehicle trip $T(L)$ from a boarding station $S_{\text{board}}(L)$ to an alighting station $S_{\text{alight}}(L)$. The PT-vehicle trip implies the departure time $t_{\text{dep}}(L) = t_{\text{dep}}(T(L), S_{\text{board}}(L))$ at the boarding station and the arrival time $t_{\text{arr}}(L) = t_{\text{arr}}(T(L), S_{\text{alight}}(L))$ at the alighting station. Between legs, travelers *transfer* from one station to another using a transfer mode of transportation. We focus on walking as a transfer mode. Possible transfers are represented in a *transfer graph* $G_{\text{transfer}} = (\mathcal{S}, E_{\text{transfer}})$. Every transfer edge $(S, S') \in E_{\text{transfer}}$ describes a possible transfer from station S to station S'. The walking time for a transfer edge $e \in E_{\text{transfer}}$ is $\ell_{\text{transfer}}(e)$. Let $t_{\text{arr}}(J) = t_{\text{arr}}(L_{|J|})$. For consistency, we require that a transfer exists between subsequent legs, i.e., $(S_{\text{alight}}(L_i), S_{\text{board}}(L_{i+1})) \in E_{\text{transfer}}$ for $i = 1, \dots, |J| - 1$. Furthermore, the,traveler must be able catch the trip of the following leg in time, i.e., $t_{\text{arr}}(L_i) + \ell_{\text{transfer}}(S_{\text{alight}}(L_i), S_{\text{board}}(L_{i+1})) \leq t_{\text{dep}}(L_{i+1})$ for $i = 1, \dots, |J| - 1$.

Public-Transportation Routing. Given a station-to-station query (S_s, S_t, t_{q}) with a source station $S_s \in \mathcal{S}$, target station $S_t \in \mathcal{S}$, and departure time t_{q}, the *earliest-arrival problem* asks to find a journey from S_s to S_t that departs no earlier than t_{q} and has the earliest possible arrival at S_t. Formally, among all journeys

$$\mathcal{J}(S_s, S_t, t_{\text{q}}) = \{J = (L_1, \dots, L_{|J|}) \mid S_{\text{board}}(L_1) = S_s, S_{\text{alight}}(L_{|J|}) = S_t, t_{\text{dep}}(L_1) \geq t_{\text{q}}\},$$

we aim to find $\arg\min_{J \in \mathcal{J}(S_s, S_t, t_{\text{q}})} t_{\text{arr}}(J)$.

In more-criteria journey planning, there are multiple objectives that need to be optimized For example, one might want to additionally minimize the total walking time. Let $\Gamma = (\gamma_1, \dots, \gamma_k)$ denote a set of criteria to be minimized with $\gamma_i(J)$ being the value of γ_i for a journey J. The *more-criteria problem* for Γ asks to find the set of journeys that are Pareto-optimal for Γ, which we explain in the following: Consider two journeys J_1, J_2 from S_s to a station S. We say that J_1 dominates J_2 if $\gamma_i(J_1) \leq \gamma_i(J_2)$ for all $i \in \{1, \dots, k\}$ and there is at lest one $i \in \{1, \dots, k\}$ such that $\gamma_i(J_1) < \gamma_i(J_2)$. A journey J to station S is Pareto-optimal for Γ if there is no other journey to S that dominates J. Thus, the goal is to find maximal $\mathcal{J}' \subseteq \mathcal{J}(S_s, S_t, t_{\text{q}})$ such that for every $J' \in \mathcal{J}'$, there is no $J \in \mathcal{J}(S_s, S_t, t_{\text{q}})$ that dominates J'.

[1] A PT-vehicle trip T overtakes another PT-vehicle trip T' if both serve the same sequence of stations and there are two indices $i < j$ such that (a) $t_{\text{arr}}(T, S_i) \geq t_{\text{arr}}(T', S_i)$ or $t_{\text{dep}}(T, S_i) \geq t_{\text{dep}}(T', S_i)$ and (b) $t_{\text{arr}}(T, S_j) < t_{\text{arr}}(T', S_j)$ or $t_{\text{dep}}(T, S_j) < t_{\text{dep}}(T', S_j)$.

Initial and Final Transfers. Transfers are usually based on an underlying network of the transfer mode (e.g. a network of footpaths for walking) into which stations are embedded. In our case, we use the passenger network $G_{\text{psg}} = (V_{\text{psg}}, E_{\text{psg}})$ that is also used for meeting points in ride-pooling. We map each station $S \in \mathcal{S}$ of the PT network to the closest location $\text{loc}(S) \in V_{\text{psg}}$ using the geographical coordinates of the station. We write S instead of $\text{loc}(S)$ if the context makes it sufficiently clear that we mean the location of the station.

If the origin or destination of a traveler are not at a station but at some other location in the transfer mode network, the transfer mode may be used for an *initial transfer* from the origin to a station or a *final transfer* from a station to the destination. To model a journey with initial and final transfers, consider a source $s \in V_{\text{psg}}$ and a target $t \in V_{\text{psg}}$. We consider an extended transfer graph $G'_{\text{transfer}} = (\mathcal{S} \cup \{s, t\}, E'_{\text{transfer}})$ where E'_{transfer} contains the original transfer edges E_{transfer} plus an initial transfer edge (s, S) as well as a final transfer edge (S, t) for every $S \in \mathcal{S}$. The walking distances $\ell_{\text{transfer}}(s, S)$ and $\ell_{\text{transfer}}(S, t)$ are the shortest-path distances from s to S and from S to t in G_{psg}. Now, a full journey from s to t is like a station-to-station journey $J = \langle L_1, \ldots, L_{|J|} \rangle$ described above, but it uses the initial transfer edge $(s, S_{\text{board}}(L_1))$ in the beginning and the final transfer edge $(S_{\text{alight}}(L_{|J|}), t)$ in the end. If a departure time t_{q} is given, the journey is only feasible if $t_{\text{q}} + \ell_{\text{transfer}}(s, S_{\text{board}}(L_1)) \leq t_{\text{dep}}(S_{\text{board}}(L_1))$. The arrival time of the journey is now $t_{\text{arr}}(J) = t_{\text{arr}}(L_{|J|}) + \ell_{\text{transfer}}(L_{|J|}, t)$.

8.2.2. Ride-Pooling Model

We use the model for ride-pooling outlined in Sections 3.2 and 4.2. We use insertions with meeting points as described in Section 4.2 to model RP trips to and from PT stations. Recall that such an insertion for a request r has the form $\mathcal{I} = (r, p, d, \nu, i, j)$, where p is one of the possible pickup locations $P(r)$, d is one of the possible dropoff locations $D(r)$, and $\nu \in F$ is a ride-pooling vehicle. The index i means that the pickup is inserted as a new stop between the existing stops s_i and s_{i+1} in the route of ν. Similarly, j denotes that the dropoff is inserted between $s_j(\nu)$ and $s_{j+1}(\nu)$.

8.2.3. Multimodal Model

In this work, we aim to find journeys that contain legs of ride-pooling and PT-vehicle trips. We focus on ride-pooling as a feeder mode, which means ride-pooling may only be used at the beginning and end of a journey with public transportation in between. This paradigm is the most studied use of ride-pooling in multimodal systems [Lor+23; LQ10; LWB96].

Combined Journey. A *combined journey* consists of an RP trip, a sequence of PT legs (with transfers), and another RP trip. Either the first or last RP trip may be omitted, but not both, as this would just be a regular PT journey. We call the RP trip in the beginning the *access RP trip* and the RP trip in the end the *egress RP trip*. In the literature, these are sometimes referred to as first-mile and last-mile trips. We do not use the terms initial and final transfers here, as ride-pooling is not the same as the transfer mode of the PT network used to transfer from one station to another. The *access station* of the combined journey is at the end of the access RP trip and the beginning of the first PT leg. The *egress station* is at the end of the last PT leg and the beginning of the egress RP trip. We allow walking between PT legs (as transfers) as well as at the beginning of the access RP trip and at the beginning and the end of the egress RP trip (as meeting points). We do not consider meeting

points at the end of an access RP trip, i.e., ride-pooling vehicles have to drop travelers off exactly at the station.

Given a request $r = (\text{orig}, \text{dest}, t_{\text{req}})$, we formalize a combined journey as a tuple $C = (r, S_{\text{access}}, S_{\text{egress}}, \mathcal{I}_{\text{access}}, J, \mathcal{I}_{\text{egress}})$. The access and egress stations are given by S_{access} and S_{egress}. The insertion $\mathcal{I}_{\text{access}} = (r_{\text{access}}, p, S_{\text{access}}, \nu_{\text{access}}, i_{\text{access}}, j_{\text{access}})$ with $r_{\text{access}} = (\text{orig}, S_{\text{access}}, t_{\text{req}})$ and $p \in P(r)$ describes the access RP trip. The PT part of the combined journey is given as the PT journey $J = (L_1, \ldots, L_{|J|})$ with $|J| \geq 1$, $S_{\text{board}}(L_1) = S_{\text{access}}$, and $S_{\text{alight}}(L_{|J|}) = S_{\text{egress}}$. The insertion $\mathcal{I}_{\text{egress}} = (r_{\text{egress}}, S_{\text{egress}}, d, \nu_{\text{egress}}, i_{\text{egress}}, j_{\text{egress}})$ with $r_{\text{egress}} = (S_{\text{egress}}, \text{dest}, t_{\text{arr}}(L_{|J|}))$ and $d \in D(r)$ describes the egress RP trip.

Cost Function. A good combined journey strikes a balance between the utility of the journey for the rider (particularly in comparison to single-mode journeys), and the operational cost for the RP vehicles (detours and added trip time for existing riders). Our dispatcher needs to identify a single combined journey to offer to the traveler.

We extend the ride-pooling cost function of KaRRi (see Section 4.2) to combined journeys as a heuristic for overall journey quality. We then aim to find the combined journey with minimal cost. Given a combined journey $C = (r, S_{\text{access}}, S_{\text{egress}}, \mathcal{I}_{\text{access}}, J, \mathcal{I}_{\text{egress}})$, the cost function takes on the form

$$c(C) = c_{\text{traveler}}(C) + t^{+}_{\text{trip}}(\mathcal{I}_{\text{access}}) + t^{+}_{\text{trip}}(\mathcal{I}_{\text{egress}}) + w_{\text{detour}} \cdot (t_{\text{detour}}(\mathcal{I}_{\text{access}}) + t_{\text{detour}}(\mathcal{I}_{\text{egress}})).$$

The terms $t^{+}_{\text{trip}}(\mathcal{I}_{\text{access}})$ and $t^{+}_{\text{trip}}(\mathcal{I}_{\text{egress}})$ describe the added trip times of existing riders of ν_{access} and ν_{egress} incurred by $\mathcal{I}_{\text{access}}$ and $\mathcal{I}_{\text{egress}}$. The terms $t_{\text{detour}}(\mathcal{I}_{\text{access}})$ and $t_{\text{detour}}(\mathcal{I}_{\text{egress}})$ denote the added vehicle operation times of ν_{access} and ν_{egress}, with model parameter w_{detour} determining their impact in relation to trip times. For exact definitions of these terms, see Section 4.5. The *traveler cost*

$$c_{\text{traveler}}(C) = t_{\text{trip}}(C) + w_{\text{walk}} \cdot t_{\text{walk}}(C) + \omega_{\text{transfer}} \cdot N_{\text{transfer}}(C)$$

represents the quality of the journey for the traveler. It considers the total trip time of the traveler $t_{\text{trip}}(C)$, the total walking time for meeting points and transfers $t_{\text{walk}}(C)$, and the total number of transfers $N_{\text{transfer}}(C) = |J| + 1$. The model parameters w_{walk} and ω_{transfer} decide the severity of the inconvenience of walking and the number of transfers, respectively.

8.3. Preliminaries

In this section, we give an overview on the public transportation routing algorithms used in this work. Our ride-pooling approach is a variation of KaRRi, which we do not describe here but instead refer to Chapter 4.

8.3.1. RAPTOR

We choose to use the well-established PT routing algorithm RAPTOR [DPW15a] as the basis of our algorithm. The algorithm is suited for networks of the size of metropolitan areas, and has a simple structure with a straight-forward extension for more-criteria routing. If required, our approach could easily use other faster PT routing algorithms.

RAPTOR solves the earliest-arrival problem for a query (S_s, S_t, t_{q}) by proceeding in rounds. For each station S in the network, let $t^{i}_{\text{arr}}(S)$ describe the earliest arrival time known for S after i rounds. Initially, set $t^{0}_{\text{arr}}(S_s) = t_{\text{q}}$ and $t^{0}_{\text{arr}}(S) = \infty$ for $S \neq S_s$. Round i consists of the following steps. The algorithm first initializes $t^{i}_{\text{arr}}(S) := t^{i-1}_{\text{arr}}(S)$ for every station S.

Let $\mathcal{S}_{i-1}$ be the set of stations for which an improved arrival time was found in round $i-1$. For the first round, use $\mathcal{S}_0 = \{S_s\}$. The algorithm scans each route that contains at least one station in $\mathcal{S}_{i-1}$. To scan a route, the stations of the route are processed in order. When a station S in $\mathcal{S}_{i-1}$ is reached, the algorithm finds the earliest PT-vehicle trip of the route that the traveler may board at S with departure time at least $t^{i-1}_{\text{arr}}(S)$. Consider a station S' that comes after S in the route. The PT-vehicle trip boarded at S defines a new possible arrival time at S'. If this improves the best known arrival time $t^{i}_{\text{arr}}(S')$, it is updated and S' is added to $\mathcal{S}_i$. If $S' \in \mathcal{S}_{i-1}$, we know an arrival time $t^{i-1}_{\text{arr}}(S')$ at S' from the previous round. If this arrival time is earlier than the arrival time $t^{i}_{\text{arr}}(S')$ in our current scan of the route, we may be able to proceed with an earlier trip for the scan based on $t^{i-1}_{\text{arr}}(S')$. After scanning routes, the algorithm enters the *transfer phase*. For every station S in $\mathcal{S}_i$, and every outgoing edge (S, S') in the transfer graph, there is a new possible arrival time at S' using $t^{i}_{\text{arr}}(S)$ and the transfer time encoded in the graph. If the new arrival time improves upon $t^{i}_{\text{arr}}(S')$, it is updated and S' is added to $\mathcal{S}_i$.

After round i, the algorithm has computed the earliest arrival time with i or fewer legs for every station. The algorithm ends after round k if $\mathcal{S}_k$ is empty. For faster convergence, a station S may be omitted from $\mathcal{S}_i$ if its arrival time is later than the best known arrival time at the target. To retrieve the legs of the best journey, parent pointers are maintained during the search. Due to the round-based structure, RAPTOR can easily keep track of Pareto-optimal journeys with respect to arrival time and number of transfers.

McRAPTOR is a variation of RAPTOR that solves the more-criteria problem by storing a set of Pareto-optimal *labels* for every station instead of just the earliest arrival time. Each label at a station S represents a possible journey to S with the according criteria values. When scanning routes, each label at $S \in \mathcal{S}_{i-1}$ implies a possible PT-vehicle trip on each route of S. The route may need to be scanned for multiple Pareto-optimal PT-vehicle trips. When a station S' is reached, a new label is generated based on the current PT-vehicle trip. If the new label is dominated by known labels at S', it is discarded. Otherwise, it is added to the labels of S' and any labels dominated by the new label are discarded. The relaxation of transfers is adapted analogously. Thus, after round i, the algorithm has computed the set of Pareto-optimal journeys with i or fewer legs for every station.

8.3.2. Unlimited Transfers

Traditionally, PT routing algorithms use a fixed transfer graph with transfers restricted to stations that are close to each other with respect to the transfer mode. In multimodal journey planning, the goal is to allow unlimited transfers. This means that a traveler can transfer from a station to every other station that can be reached using the transfer mode in an underlying network (e.g. a network of footpaths). The simplest approach for this is to use the transitive closure of the fixed transfer graph. However, this transitive closure leads to extremely dense transfer graphs that paralyze the transfer phase. MCR [Del+13] extends McRAPTOR to unlimited transfers by running Dijkstra's algorithm in the underlying network during the transfer phase. To achieve feasible running times, MCR contracts the network to a core that contains only vertices and edges that are required for shortest paths between stations.

ULTRA. ULTRA [Bau+23] proposes a different approach based on a preprocessing phase in which a new transfer graph is constructed. This transfer graph contains an edge between two stations S and S' if there is an unlimited transfer between S and S' that is required for a Pareto-optimal journey. The edges in this graph are called shortcuts.

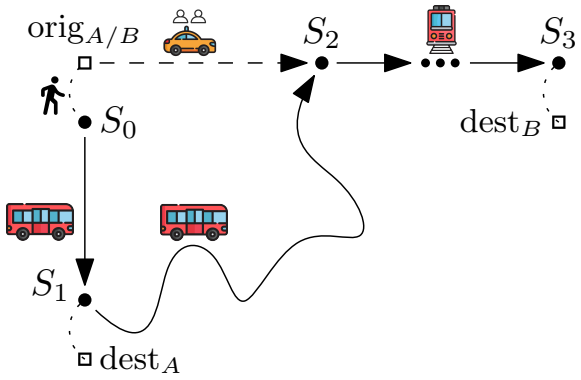


Figure 8.1 Illustration of two journeys via public transit for travelers A and B going from orig$_{A/B}$ to dest$_A$ and dest$_B$, respectively. Traveler A has a satisfactory PT-only journey going from station S_0 to S_1. Traveler B profits from ride-pooling as they can take an access RP trip to the train station S_2 to avoid an inconvenient bus ride.

The preprocessing phase uses MCR to compute all Pareto-optimal journeys in the network with at most two legs and an unlimited transfer in between. The authors show that the unlimited transfers of these journeys contain all shortcuts needed for any longer Pareto-optimal journey. ULTRA proposes many optimizations to improve the running time of this preprocessing phase.

The resulting transfer shortcut graph can be used as a transfer graph in any PT routing algorithm, including McRAPTOR. This leads to much faster query times than computing unlimited transfers on-the-fly like MCR. If the origin or destination of a query are not located at a station, the authors of ULTRA propose to use bucket contraction hierarchy (BCH) queries (see Section 3.1.3) to compute unlimited initial and final transfers.

8.4. Algorithm Overview

Combining ride-pooling and public transportation journey planning comes with significant new challenges compared to routing for each mode in isolation. We describe two main challenges and outline the structure of our algorithm.

8.4.1. Difference to Traditional Feeder Modes

Journey planners for public transport usually allow travelers to use the transfer mode to get from an origin location to a PT station and from a PT station to the destination. These trips are called *feeder trips*. In our scenario, ride-pooling can perform feeder trips, but not transfers, so we call it a *feeder mode*. There are two important differences between traditional feeder modes (such as walking, cycling, or e-scooters) and ride-pooling.

Number of Access Stations. Traditional feeder modes are slower than public transport for sufficiently long travel distances. Thus, only a small number of PT stations will ever be relevant for journeys with optimal arrival times. In contrast, ride-pooling can often be faster than public transport and more stations may be relevant as access stations.

Consider the example illustrated in Figure 8.1. Two travelers A and B start at the same location orig$_{A/B}$ and travel to different destinations dest$_A$ and dest$_B$. Assume that no ride-pooling is available. There is one bus stop S_0 close to orig$_{A/B}$ that the travelers

can walk to. The first destination dest_A is close to bus stop S_1 in the next town over and can be reached via a bus route that also serves S_0. The second destination dest_B is much further away. A PT journey requires traveler B to take a bus trip at S_0 to the main station S_2 of the closest large city and take a train from there to S_3. The bus trip is slow since the route serves multiple smaller towns on its way. However, the bus is still faster than walking directly to S_2, so S_0 remains the only relevant access station.

If we introduce ride-pooling into this scenario, traveler A does not benefit, since they already have a good journey to t_A using only walking and public transport. Traveler B does benefit from ride-pooling, though, as they are now able to use an access RP trip directly to S_2 that is faster than the bus and may allow them to catch an earlier train at S_2. Thus, with ride-pooling, S_2 becomes a relevant access station.

Degree of Preprocessing. Traditional feeder modes are fully flexible and independent. The travel time of a traditional feeder mode depends only on the origin and destination location, and is not affected by the departure time or the behavior of other travelers. This allows journey planners to fully or partially preprocess the travel times of feeder trips, i.e., trips from the origin to PT stations and from PT stations to the destination. For example, ULTRA uses BCH queries (see Section 3.1.3) to compute access and egress travel times using the feeder mode. The algorithm precomputes the bucket entries for these BCH queries once and stores them alongside the PT network data.

For ride-pooling, the best access and egress RP trips always depend on the departure time and the current routes of the RP vehicles. As we consider the dynamic version of the RP problem, this information is not available a priori, so preprocessing RP feeder trips is not as straightforward as for traditional feeder modes[2]. In particular, egress RP trips pose an issue as pointed out by Lorente et al. [Lor+23]. Assume that we are processing a new request at time t_{req}. For a combined journey, the traveler might arrive at the egress station S_{egress} at some time $t_{\text{arr}}(S_{\text{egress}})$ far in the future. For the egress RP trip from S_{egress} to the destination, we would have to pre-book an RP trip with earliest possible departure time $t_{\text{arr}}(S_{\text{egress}})$. However, we do not know the state of the RP system at time $t_{\text{arr}}(S_{\text{egress}})$. We can likely find a vehicle ν that is close to S_{egress} at time t_{req} that can be assigned to the egress trip, but the cost of this RP trip would not adequately represent the quality of this insertion. We cannot determine how useful ν may be for other insertions in the time between t_{req} and $t_{\text{arr}}(S_{\text{egress}})$ and whether a different vehicle ν' may be the better choice for the pre-booked egress trip. At time t_{req}, we can only estimate the cost of a future egress RP trip.

Algorithms for PT routing handle an increased number of access stations without any difficulty, as they can initialize the search at many stations with departure times that vary by query and station. As a consequence, the main difficulty of our problem is the computation of access and egress RP trips.

8.4.2. Combined Journeys Require More-Criteria Routing

In the dynamic ride-pooling problem, we find the RP trip with optimal cost. In this section, we show that we need to optimize for two criteria when finding combined journeys: arrival time and a modified cost function that does not include trip time.

[2] Journey planning algorithms also rely on preprocessing for transfers between stations (see Section 8.3.2). Thus, using RP as a transfer mode is difficult, and we focus on the use as a feeder mode for now.

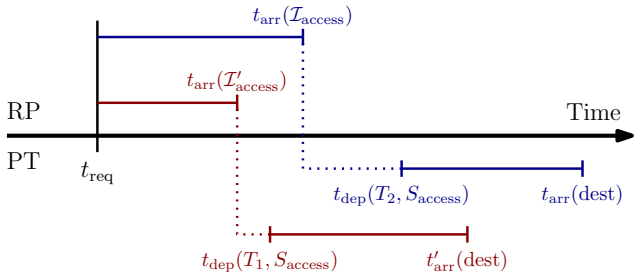


Figure 8.2 Timeline for two possible access RP trips $\mathcal{I}_{\text{access}}$ and $\mathcal{I}'_{\text{access}}$ to the same access station S_{access}. Top shows access RP trips, while bottom shows PT-vehicle trips to the destination. Dotted lines are waiting time at S_{access}. Even if $c(\mathcal{I}_{\text{access}}) < c(\mathcal{I}'_{\text{access}})$, the traveler may arrive at the destination much earlier with $\mathcal{I}'_{\text{access}}$, which could lead to smaller overall costs.

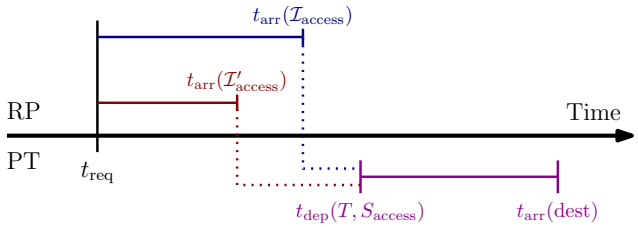


Figure 8.3 Timeline for two possible access RP trips $\mathcal{I}_{\text{access}}$ and $\mathcal{I}'_{\text{access}}$ to the same access station S_{access}. Top shows access RP trips, while bottom shows PT-vehicle trip to the destination. Dotted lines are waiting time at S_{access}. Even if $t_{\text{arr}}(\mathcal{I}'_{\text{access}}) < t_{\text{arr}}(\mathcal{I}_{\text{access}})$ and $c(\mathcal{I}'_{\text{access}}) < c(\mathcal{I}_{\text{access}})$, the traveler may arrive the destination at the same time for both trips, which makes the non-trip cost the deciding factor in overall cost.

Optimization Criteria for Access Trips. First, we show that it does not suffice to find access RP trips with optimal cost, but that access trips need to consider the arrival time too.

Consider the situation depicted in Figure 8.2. There are two access trips $\mathcal{I}_{\text{access}} = (r, p, S_{\text{access}}, \nu, i, j)$ and $\mathcal{I}'_{\text{access}} = (r, p', S_{\text{access}}, \nu', i', j')$ to the same access station S_{access}. The insertion $\mathcal{I}'_{\text{access}}$ arrives at S_{access} earlier than $\mathcal{I}_{\text{access}}$. Assume that the cost of the access trip only is smaller for $\mathcal{I}_{\text{access}}$ than for $\mathcal{I}'_{\text{access}}$, so $c(\mathcal{I}_{\text{access}}) < c(\mathcal{I}'_{\text{access}})$. This is possible since only the trip time term $t_{\text{trip}}(\mathcal{I})$ in the cost of a RP insertion is directly related to $t_{\text{arr}}(\mathcal{I})$.

As illustrated in the diagram, there are two PT-vehicle trips T_1 and T_2 that can bring the traveler to their destination. If the traveler uses $\mathcal{I}_{\text{access}}$, they do not arrive at S_{access} in time for T_1, but have to take T_2. In this case, they arrive at dest at time $t_{\text{arr}}(\text{dest})$. If the traveler instead uses $\mathcal{I}'_{\text{access}}$, they arrive early enough at S_{access} to use T_1, and arrive at the destination at time $t'_{\text{arr}}(\text{dest}) < t_{\text{arr}}(\text{dest})$. Thus, the trip time is shorter if the traveler uses $\mathcal{I}'_{\text{access}}$. The difference in trip time and in trip cost of the combined journey is $\Delta = t_{\text{arr}}(\text{dest}) - t'_{\text{arr}}(\text{dest})$. If $\Delta > c(\mathcal{I}'_{\text{access}}) - c(\mathcal{I}_{\text{access}})$, the overall cost of the combined journey with $\mathcal{I}'_{\text{access}}$ is smaller than with $\mathcal{I}_{\text{access}}$, even though $c(\mathcal{I}_{\text{access}}) < c(\mathcal{I}'_{\text{access}})$. In conclusion, we need to include the arrival time as an optimization criterion for access RP trips, so our access trips do not ignore beneficial earlier PT-vehicle trips.

Modified Cost Criterion. Second, we show that we cannot use the full cost as an optimization criterion for access RP trips, but instead we need to use a modified cost function that does not include the trip cost for the rider.

Consider the situation depicted in Figure 8.3. As in Figure 8.2, there are two access RP trips $\mathcal{I}_{\text{access}}$ and $\mathcal{I}'_{\text{access}}$ to the same access station S_{access}, and $\mathcal{I}'_{\text{access}}$ arrives at S'_{access} earlier than $\mathcal{I}_{\text{access}}$. This time, let the cost of $\mathcal{I}'_{\text{access}}$ be smaller, so $c(\mathcal{I}'_{\text{access}}) < c(\mathcal{I}_{\text{access}})$. Thus, $\mathcal{I}_{\text{access}}$ dominates $\mathcal{I}'_{\text{access}}$ with respect to full cost c and arrival time t_{arr}. In this new scenario, the traveler can catch the next PT-vehicle trip T using either of the access RP trips. Thus, the arrival time at the destination and overall trip time is the same for both access RP trips.

Let $c_{-t}(\mathcal{I}) = c(\mathcal{I}) - t_{\text{trip}}(\mathcal{I})$ for $\mathcal{I} \in \{\mathcal{I}_{\text{access}}, \mathcal{I}'_{\text{access}}\}$ describe the cost of each access trip without considering the trip cost for the new rider. We call $c_{-t}(\mathcal{I})$ the *non-trip cost* of $\mathcal{I}$. Assume that $c_{-t}(\mathcal{I}_{\text{access}}) < c_{-t}(\mathcal{I}'_{\text{access}})$.

Consider the total cost of the combined journeys C and C' that result from using either access trip (see our cost function in Section 8.2.3). The trip costs and transfer costs, as well as the walking costs within the PT portions of the combined journeys are identical. Thus, the overall cost difference is the difference in non-trip costs of the access RP trips $c(C) - c(C') = c_{-t}(\mathcal{I}_{\text{access}}) - c_{-t}(\mathcal{I}'_{\text{access}}) < 0$. In effect, $\mathcal{I}_{\text{access}}$ leads to a smaller overall cost. Therefore, we need to find Pareto-optimal access trips with respect to the arrival time t_{arr} and the cost without trip time c_{-t}.

We extend the concept of non-trip cost to combined journeys as $c_{-t}(C) = c(C) - t_{\text{trip}}(C)$. The non-trip cost of a combined journey includes the non-trip costs of the access and egress insertions, as well as the walking cost and penalties for transfers (see Section 8.2.3). Note that the arrival time implies the trip time $t_{\text{trip}}(C) = t_{\text{arr}}(C) - t_{\text{req}}$, so we can always compute the full cost given the non-trip cost and the arrival time.

Optimization Criteria for PT Query. Third, we show that it does not suffice to optimize only for the earliest possible arrival time during the PT query. Consider two combined journeys $C = (r, S_{\text{access}}, S_{\text{egress}}, \mathcal{I}_{\text{access}}, J, \mathcal{I}_{\text{egress}})$ and $C' = (r, S'_{\text{access}}, S'_{\text{egress}}, \mathcal{I}'_{\text{access}}, J', \mathcal{I}'_{\text{egress}})$. Assume that C and C' are identical starting at a station S. More specifically, assume $S_{\text{egress}} = S'_{\text{egress}}$, $\mathcal{I}_{\text{egress}} = \mathcal{I}'_{\text{egress}}$, and that the PT sections J and J' share an identical suffix $J^c = \langle L^c_1, \ldots, L^c_{|J^c|} \rangle$, so $J = \langle L_1, \ldots, L_k \rangle \circ J^c$ and $J' = \langle L'_1, \ldots, L'_{k'} \rangle \circ J^c$. Here, $S = S_{\text{board}}(L^c_1)$ is the first common station. Assume that C arrives at S earlier than C', so $t_{\text{arr}}(L_k) < t_{\text{arr}}(L'_{k'})$.

Then, the argumentation is similar to the previous paragraph: Both journeys are identical from S to the destination, so the arrival time at the destination is the same. The overall cost difference is determined by the non-trip costs of the combined journeys, which do not depend on the arrival time at S. Thus, it is possible that $c(C) > c(C')$ even though C arrives at S earlier than C'. In effect, we need to use a more-criteria PT routing algorithm that finds Pareto-optimal journeys with respect to non-trip cost and arrival time.

8.4.3. Structure of Our Algorithm

Our algorithm PaRRot (Public Transit Ride-Pooling Router) processes each request in five phases, which we outline here.

Initialization. Our algorithm considers combined RP-PT journeys as one possible mode of transportation along with walking, pure ride-pooling, and pure public transport (with walking as a transfer mode). We first compute the best journeys for every mode other than RP-PT. For the walking distance, we run a CH query in the passenger network G_{psg}. For the best ride-pooling trip, we run our dispatcher KaRRi (see Chapter 4). For PT-only, we run ULTRA-RAPTOR (see Section 8.3) to find the journey with the earliest arrival time using unrestricted walking transfers.

Afterwards, we compute the cost of the RP-only trip c_{RP} and the cost of the PT-only journey c_{PT} according to our cost function for combined journeys. The value $\hat{c} = \min\{c_{\text{RP}}, c_{\text{PT}}\}$ is an upper bound for the cost of any relevant combined journey, which we will use to prune various queries. This approach of pruning eligible combined journeys based on uni-modal journeys has previously been applied in an early work by Liaw et al. [LWB96].

Computing Heuristic Egress RP Trips. After initialization, we compute the egress RP trip. As described in Section 8.4.1, the egress RP trip requires pre-booking, possibly far into the future. We cannot determine which vehicle will be well suited for the egress trip, so we instead use a heuristic for the cost of the egress trip, like previously proposed by Lorente et al. [Lor+23] (see Section 8.1.1). In our experiments on KaRRi, we observe that the cost of a ride-pooling trip linearly correlates with the direct distance $\delta(\text{orig}, \text{dest})$ from origin to destination. Thus, we propose using a heuristic egress cost of $c^{\text{heu}}_{\text{egress}}(S) = m_{\text{cost}} \cdot \delta(S, \text{dest}) + b_{\text{cost}}$ for each station $S \in \mathcal{S}$. The model parameters m_{cost} and b_{cost} can be calibrated using KaRRi. Note that $c^{\text{heu}}_{\text{egress}}(S)$ is the full cost of the egress RP trip instead of the non-trip cost used in the PT router. This is okay, as we are only interested in the smallest full cost at the destination.

To compute the distances $\delta(S, \text{dest})$ needed for the heuristic, we run BCH queries (see Section 3.1.3) in the vehicle network G. For every station S, we generate a forward bucket entry $(S, \delta^{\uparrow}(S, v))$ in a bucket $B(v)$ of every vertex v in the upward CH search space of S. We sort the entries of every bucket $B(v)$ for $v \in V$ by their distance $\delta^{\uparrow}(S, v)$. We call these buckets *station source buckets*. We generate station source buckets once for the network as a preprocessing step and write them to disk. When finding a combined PT query for a request $r = (\text{orig}, \text{dest}, t_{\text{req}})$, we run a reverse BCH query rooted at dest using the station source buckets to find the required distances $\delta(S, \text{dest})$. We know that any distance with $\delta(S, \text{dest}) > \hat{c}$ can only lead to infeasible combined journeys. We can prune scans of the sorted buckets based on this upper bound (see Section 4.3.1).

We compute heuristic egress trips first, since knowing the egress trips gives us a simple way to potentially improve the cost upper bound $\hat{c}$. For every station S with a valid egress trip distance $\delta(S, \text{dest})$, we proceed as follows. If S was not visited during the PT-only query, move on to the next station. Otherwise, the result of the PT-only query includes a PT journey J from orig to S. We can combine this journey J and the heuristic egress RP trip into a combined journey C without an access RP trip. Since C is a valid combined journey, its cost is an upper bound for the smallest cost of any combined journey. Thus, if $c(C) < \hat{c}$, we update $\hat{c} := c(C)$. Note that C is valid, but it is not necessarily the best combined journey without an access trip for S, since the PT journey J derived from the PT-only query is not necessarily optimal with regards to our cost function.

Computing Access RP Trips. We modify KaRRi to compute the best access RP trips to every station. We prune the searches using $\hat{c}$, so many stations will not have a feasible access trip. Nonetheless, we compute the access trips for thousands of stations, so this step becomes the main bottleneck of our algorithm. We describe methods for speeding up the computation of access trips in the next section.

More-Criteria PT Query. As explained in Section 8.4.2, we require more-criteria PT routing to find Pareto-optimal journeys with respect to arrival time and cost according to our cost function for combined journeys. For this, we run an ULTRA query with McRAPTOR (see Section 8.3).

In addition to ULTRA's usual initial and final transfers by walking, we allow access and egress RP trips. During the initialization of the algorithm, we create an initial McRAPTOR label for every Pareto-optimal access RP trip at the respective station. At the end of each McRAPTOR round, the algorithm combines every updated label with the egress RP trip of the station to create a candidate combined journey to the destination. Note that we also allow arrivals at the destination by PT-vehicle trip or by walking as candidates for the best combined journey.

At the destination, we only keep track of the journey with the smallest total cost c^*. We can use c^* for target pruning: During the search, any label l with $c(l) > c^*$ can be discarded and does not need to be propagated further.

For our query, we need to make sure that the ULTRA transfer shortcut graph is constructed for the criteria of arrival time t_{arr} and non-trip cost c_{-t}. We call this set of criteria $\Gamma_1 = (t_{\text{arr}}, c_{-t})$. An existing implementation of ULTRA for McRAPTOR[3] computes all shortcuts needed for Pareto-optimal journeys with the criteria arrival time t_{arr}, walking time t_{walk}, and number of transfers N_{transfer}. We call this different set of criteria $\Gamma_2 = (t_{\text{arr}}, t_{\text{walk}}, N_{\text{transfer}})$. Instead of implementing preprocessing for Γ_1, we show that the shortcuts for Γ_2 contain all shortcuts needed for Γ_1.

Assume that a journey J from a station S_s to another station S_t is Pareto-optimal for Γ_1, but is dominated by a different journey J' for Γ_2. Thus, $t_{\text{arr}}(J') \leq t_{\text{arr}}(J)$, $t_{\text{walk}}(J') \leq t_{\text{walk}}(J)$, and $N_{\text{transfer}}(J') \leq N_{\text{transfer}}(J)$ with at least one strict inequality. Then,

$$\begin{aligned} c_{-t}(J') =& w_{\text{walk}} t_{\text{walk}}(J') + \omega_{\text{transfer}} N_{\text{transfer}}(J') \\ \leq& w_{\text{walk}} t_{\text{walk}}(J) + \omega_{\text{transfer}} N_{\text{transfer}}(J) = c_{-t}(J). \end{aligned}$$

With this, J' dominates J for Γ_1, which is a contradiction. Therefore, if a combined journey is Pareto-optimal for Γ_1, it is also Pareto-optimal for Γ_2. In effect, the shortcuts needed for Γ_1 are a subset of the shortcuts for Γ_2 and we can safely use the existing code. For a future optimization, we could implement an ULTRA preprocessing phase that computes shortcuts for Γ_1, which may reduce the number of edges in the transfer shortcut graph.

Mode Choice. Simulating mode choice behavior for multimodal trips using RP and PT is a complex problem, which requires advanced models such as cross-nested logits [Kos+24]. Therefore, we cannot use our mode choice model from Chapter 6 here. As we are mostly interested in the algorithmic contributions and the running time of our algorithm, we do not implement the more complex model, but use a simplified mode choice mechanism. For every mode, we compute the journey with the smallest cost according to our cost function (see Section 8.2.3). Then, the rider chooses the mode with the smallest overall cost. As our cost function conflates operational costs and rider trip quality, this approach does not truly represent rider mode choice, but is akin to all travelers blindly following the journey suggested by a mobility-as-a-service provider. In the future, we want to improve upon this model in cooperation with experts from the transportation science community. In particular, including a model for fares would give more reliable results.

If a traveler chooses a combined RP-PT journey $C = (r, S_{\text{access}}, S_{\text{egress}}, \mathcal{I}_{\text{access}}, J, \mathcal{I}_{\text{egress}})$, we first insert the access RP trip $\mathcal{I}_{\text{access}}$ into the route of the respective vehicle ν_{access}. Let $s \in R(\nu_{\text{access}})$ be the stop introduced for the access station S_{access}. We enforce a latest permissible arrival time at s, so that no additional detour can cause the traveler to miss their

[3] Available at `https://github.com/kit-algo/ULTRA`.

intended first PT trip from S_{access}. This uses the same mechanism for latest permissible arrival times that is already in place for the RP constraints (see Section 3.2).

Note that we only know the heuristically estimated cost of the egress RP trip $\mathcal{I}_{\text{egress}}$ at this point. Since we want to avoid pre-booking far into the future, we fix the egress station S_{egress} now, but defer the dispatching process for the egress trip until the traveler is close to reaching S_{egress}. The traveler arrives at S_{egress} at time $t_{\text{arr}}(J)$. We use KaRRi to find $\mathcal{I}_{\text{egress}}$ at time $t_{\text{arr}}(J) - t_{\text{buf}}$ where t_{buf} is a model parameter for a small time interval. We use $t_{\text{buf}} > 0$ to allow vehicles to make their way to S_{egress} before the traveler arrives so that the wait time at S_{egress} is minimized and the ride can possibly be pooled with other travelers arriving at a similar time. We still have to pre-book trips but the time between assigning the rider and their earliest possible departure time is limited to at most t_{buf}. Thus, if t_{buf} is not too large, the vehicle routes known at $t_{\text{arr}}(J) - t_{\text{buf}}$ likely already extend to $t_{\text{arr}}(J)$ and we can make a more informed decision than at the original request time.

8.5. Finding Access RP Trips Efficiently

When finding combined RP-PT journeys, we are faced with the novel challenge of computing access RP trips. In traditional dynamic ride-pooling, we need to solve a point-to-point ride-pooling problem to find the trip from the origin to the destination that has the smallest cost. Now, we deal with a more-criteria *one-to-many* ride-pooling problem where we try to find the Pareto-optimal access RP trips to every station. More precisely, given a request $r = (\text{orig}, \text{dest}, t_{\text{req}})$, every potential access station S_{access} induces a request $r_S = (\text{orig}, S_{\text{access}}, t_{\text{req}})$, and we look for every insertion $\mathcal{I} = (r_S, p, S_{\text{access}}, \nu, i_{\text{access}}, j_{\text{access}})$ that is Pareto-optimal with respect to arrival time and cost. Recall that i_{access} and j_{access} describe the stop index in the route of ν where the pickup at p and the dropoff at S_{access} are to be inserted as new stops.

In some ways, we can treat the set of stations like the set of potential dropoffs $D(r)$ in ride-pooling with meeting points. Some of the techniques used for meeting points in KaRRi can be extended to the new problem. However, there are many more stations in the network than there are dropoffs in KaRRi, and stations are not localized around the destination like dropoffs. Thus, in many cases, we need new methods to efficiently find insertions and the shortest-path distances needed to determine their cost.

8.5.1. Direct Distances from Pickups to Stations

Consider an access trip $\mathcal{I}_{\text{access}} = (r, p, S_{\text{access}}, \nu_{\text{access}}, i_{\text{access}}, j_{\text{access}})$ with $i_{\text{access}} = j_{\text{access}}$. Let $a := i_{\text{access}} = j_{\text{access}}$. The pickup at p and the dropoff at S_{access} are inserted between the same two existing stops s_a and s_{a+1} in the route of ν, which would cause the vehicle to go directly from p to S_{access} before moving on to s_{a+1}. Thus, we need to know the distance $\delta(p, S_{\text{access}})$ from the pickup $p \in P(r)$ to the access station S_{access}. In KaRRi, the distances from pickups to dropoffs are computed by generating target buckets for the dropoffs and running BCH queries from each pickup. Computing the distances from pickups to stations is even simpler as the locations of the stations do not change. We generate *station target buckets* for each station once as a preprocessing step and store these buckets as part of the PT network. During a PaRRot query, we run a BCH query using the station target buckets from every pickup. We can sort bucket entries and prune bucket scans using $\hat{c}$. This is analogous to computing the distances from each station to the destination for our egress trip heuristic (see Section 8.4.3).

8.5.2. Ordinary Access Insertions

An access insertion $\mathcal{I}_{\text{access}} = (r, p, S_{\text{access}}, \nu_{\text{access}}, i_{\text{access}}, j_{\text{access}})$ is *ordinary* if the dropoff at the access station is inserted somewhere along the route of the vehicle ν_{access} (and not after its current last stop $s_{k(\nu_{\text{access}})}$), i.e, if $j_{\text{access}} < k(\nu_{\text{access}})$, where $k(\nu_{\text{access}})$ is the current number of stops of ν_{access}. To compute the cost of an ordinary access insertion, we need to know the distances $\delta(s_{j_{\text{access}}}, S_{\text{access}})$ and $\delta(S_{\text{access}}, s_{j_{\text{access}}+1})$.

In KaRRi, we run one elliptic BCH query per dropoff $d \in D(r)$ to determine the analogous distances $\delta(s_j, d)$ and $\delta(d, s_{j+1})$. These elliptic BCH queries can be bundled well, as the dropoffs are highly localized. We cannot simply extend this approach, since there are too many stations and bundling would be less efficient.

However, we can exploit that the location of every station is fixed and does not depend on the request like the location of a dropoff. We once again utilize the concept of detour ellipses (see Section 4.3.2). Recall that rider time constraints lead to latest permissible arrival times at every stop. These constraints define the detour leeway $\lambda(s_a, s_{a+1})$, which describes how long a detour between stops s_a and s_{a+1} may be without violating any rider constraints. The set of all locations to which a detour can be made is called the detour ellipse $\mathcal{E}(s_a)$. A station S can only be feasible for a dropoff between $s_{j_{\text{access}}}$ and $s_{j_{\text{access}}+1}$ if it lies in the detour ellipse $\mathcal{E}(s_{j_{\text{access}}})$. Thus, for every stop $s_a \in R(\nu)$ of every vehicle $\nu \in F$, we precompute the subset of *elliptic stations* $\mathcal{S}_{\mathcal{E}}(s_a) = \{S \in \mathcal{S} \mid \delta(s_a, S) + \delta(S, s_{a+1}) \leq \lambda(s_a, s_{a+1})\}$ that lie in $\mathcal{E}(s_a)$. Along with each elliptic station $S \in \mathcal{S}_{\mathcal{E}}(s_a)$, we store the distances $\delta(s_a, S)$ and $\delta(S, s_{a+1})$ for retrieval during the query.

Precomputing Stations in Detour Ellipses. We compute the elliptic stations for a new stop as follows. Assume that we are about to insert a new stop s_{new} between s_a and s_{a+1}. Let $\mathcal{E}_{\text{old}} = \mathcal{E}(s_a)$ and $\mathcal{S}_{\text{old}} = \mathcal{S}_{\mathcal{E}}(s_a)$ before inserting s_{new}. The insertion of a new stop can only decrease the size of a detour ellipse. Thus, after the insertion, the new ellipses between s_a and s_{new} as well as between s_{new} and s_{a+1} are subsets of the previous ellipse, so $\mathcal{E}(s_a) \subseteq \mathcal{E}_{\text{old}}$ and $\mathcal{E}(s_{\text{new}}) \subseteq \mathcal{E}_{\text{old}}$. Consequently, $\mathcal{S}_{\mathcal{E}}(s_a) \subseteq \mathcal{S}_{\text{old}}$ and $\mathcal{S}_{\mathcal{E}}(s_{\text{new}}) \subseteq \mathcal{S}_{\text{old}}$. For every $S \in \mathcal{S}_{\text{old}}$, we compute $\delta(S, s_{\text{new}})$ and $\delta(s_{\text{new}}, S)$ by running BCH queries from s_{new} using the station source buckets and station target buckets (see Sections 8.4.3 and 8.5.1). Then, for each station $S \in \mathcal{S}_{\text{old}}$, we can use the computed distances and the previously stored distances to check whether $\delta(s_a, S) + \delta(S, s_{\text{new}}) \leq \lambda(s_a, s_{\text{new}})$ and whether $\delta(s_{\text{new}}, S) + \delta(S, s_{a+1}) \leq \lambda(s_{\text{new}}, s_{a+1})$. This results in an updated $\mathcal{S}_{\mathcal{E}}(s_a)$ and the new $\mathcal{S}_{\mathcal{E}}(s_{\text{new}})$. This process can easily be adapted for the case of inserting a new stop after the current end of the route. In this case there is no previous $\mathcal{S}_{\mathcal{E}}(s_{k(\nu)})$, so we have to use $\mathcal{S}_{\text{old}} = \mathcal{S}$ and run another BCH query from the old last stop $s_{k(\nu)}$. Every BCH query mentioned here can be pruned using the leeway between the respective stops. We can also update $\mathcal{S}_{\mathcal{E}}(s_a)$ if the leeway of s_a shrinks due to a change in latest permissible arrival times. For every $S \in \mathcal{S}_{\mathcal{E}}(s_a)$, we check whether S exceeds the new leeway using the stored distances. If so, we remove S from $\mathcal{S}_{\mathcal{E}}(s_a)$.

Enumerating Ordinary Access Insertions. During a PaRRot query, we first find the set of vehicles that can perform a pickup. Like in Section 7.3.2, let

$$F_p(r) = \{(\nu, p, i) \in F \times P(r) \times \mathbb{N} \mid i \in \{0, \ldots, k(\nu)\} \text{ and } p \in \mathcal{E}(s_i(\nu))\}$$

denote the set of vehicles and associated stops in the vehicle route such that ν can perform a pickup of r at p immediately after stop s_i without breaking any of the vehicle's constraints. We obtain $F_p(r)$ as a result of the elliptic BCH queries performed for

every $p \in P(r)$. For every $(\nu_{\text{access}}, p, i_{\text{access}}) \in F_p(r)$, we generate an access insertion $\mathcal{I}_{\text{access}} = (r, p, S_{\text{access}}, \nu_{\text{access}}, i_{\text{access}}, j_{\text{access}})$ for every $j_{\text{access}} = i_{\text{access}}, \ldots, k(\nu_{\text{access}}) - 1$ and every $S_{\text{access}} \in \mathcal{S}_{\mathcal{E}}(s_{j_{\text{access}}}(\nu_{\text{access}}))$. We know all required distances to compute the cost and arrival time of $\mathcal{I}_{\text{access}}$: The distances $\delta(s_{i_{\text{access}}}, p)$ and $\delta(p, s_{i_{\text{access}}+1})$ have been computed by the elliptic BCH queries. The distances $\delta(s_{j_{\text{access}}}, S_{\text{access}})$ and $\delta(S_{\text{access}}, s_{j_{\text{access}}+1})$ are stored as part of $\mathcal{S}_{\mathcal{E}}(s_{j_{\text{access}}}(\nu_{\text{access}}))$. If $i_{\text{access}} = j_{\text{access}}$, the required distance $\delta(p, S_{\text{access}})$ has been precomputed as described in Section 8.5.1.

Only if $i_{\text{access}} = 0$, we work with incomplete information. In this pickup-before-next-stop case (see Figure 4.2), the distance $\delta(s_0, p)$ computed by the elliptic BCH searches is only a lower bound on the distance, since the vehicle ν_{access} has to be re-routed from its current unknown position instead of going directly from s_0 to p. We proceed as described in Section 4.6.3. We can use the lower bound $\delta(s_0, p)$ to compute a lower bound for the cost and arrival time of each access insertion $\mathcal{I}_{\text{access}}$. If $\mathcal{I}_{\text{access}}$ is not dominated by any other known trip to S_{access}, we compute the current position of ν_{access} and the exact distance from s_0 to p via this current position. This allows us to consider $\mathcal{I}_{\text{access}}$ with exact cost and arrival time as a candidate access trip to S_{access}.

Pruning Candidate Insertions. In large input instances, there can be thousands of stations per ellipse and hundreds of pickup vehicles, so we may need to enumerate many candidate access insertions. We can apply pruning based on the global cost bound $\hat{c}$ to reduce the number of candidate insertions.

We sort the elliptic stations $\mathcal{S}_{\mathcal{E}}(s_a)$ of each stop s_a by increasing detour $\delta(s_a, S) + \delta(S, s_{a+1})$. Consider a fixed $(\nu_{\text{access}}, p, i_{\text{access}}) \in F_p(r)$ and $j_{\text{access}} > i_{\text{access}}$. We compute a lower bound c_{pickup} on the cost incurred just by this information. This lower bound includes the vehicle cost and added trip time for existing riders caused by the detour for p as well the trip time until departure at $s_{j_{\text{access}}}$. We then iterate over the stations in $\mathcal{S}_{\mathcal{E}}(s_{j_{\text{access}}})$ in order. Assume that we are processing a station $S_{\text{access}} \in \mathcal{S}_{\mathcal{E}}(s_{j_{\text{access}}})$. We add the vehicle cost and added trip time for existing passengers caused by the detour for S_{access} to c_{pickup} to obtain c_{station}. Since the stations are ordered by increasing detour, c_{station} is a lower bound for the cost of any remaining station in $\mathcal{S}_{\mathcal{E}}(s_{j_{\text{access}}})$. Thus, if $c_{\text{station}} > \hat{c}$, we are done for j_{access}.

8.5.3. After-Last-Stop Access Insertions

Access insertions may also append new stops for stations to the end of an existing vehicle route. We call these access insertions $\mathcal{I}_{\text{access}} = (r, p, S_{\text{access}}, \nu_{\text{access}}, i_{\text{access}}, j_{\text{access}})$ with $j_{\text{access}} = k(\nu_{\text{access}})$ *after-last-stop (ALS) insertions.* As in KaRRi (see Figure 4.2), either the pickup is also inserted after the last stop ($i_{\text{access}} = k(\nu_{\text{access}})$, *PALS*) or the pickup is inserted somewhere along the existing route ($i_{\text{access}} < k(\nu_{\text{access}})$, *DALS*).

PALS Access Insertions. To compute the cost of a PALS access insertion $\mathcal{I}_{\text{access}} = (r, p, S_{\text{access}}, \nu_{\text{access}}, k(\nu_{\text{access}}), k(\nu_{\text{access}}))$, we only need to know the distances $\delta(s_{k(\nu_{\text{access}})}, p)$ from the last stop to the pickup and $\delta(p, S_{\text{access}})$ from the pickup to the station. The latter is computed as described in Section 8.5.1. For the former, we can proceed similarly to KaRRi (see Section 4.7). We run a reverse BCH query from every pickup $p \in P(r)$ using last stop source buckets. We cannot use collective BCH searches since we look for the best insertions per station and we may not apply a domination criterion for labels of different stations. We can still apply pruning based on $\hat{c}$ and sorted buckets, though.

DALS Access Insertions. For DALS access insertions, the approach has to change significantly. In KaRRi, we run a reverse BCH query from every dropoff, which may be bundled or collective. This is not feasible for stations as there are many more stations than dropoffs.

Instead, we proceed similarly to the process for ordinary access insertions. We again use the set of vehicles F_p that can perform a pickup somewhere along the existing route. Then, for every vehicle ν_{access} that occurs in F_p, we run a forward BCH query from $s_{k(\nu_{\text{access}})}$ using the station target buckets described in Section 8.5.1. As for ordinary access stations, we can compute a lower bound c_{pickup} on the cost induced by the pickup information. We use c_{pickup} and $\hat{c}$ to prune bucket scans during the BCH queries.

8.6. Experiments

In this section we experimentally evaluate PaRRot and compare it against a solution without combined journeys.

8.6.1. Setup and Input Instances

Our source code of PaRRot[4] is implemented in C++20 and is based on an open-source implementation of ULTRA[5] and our implementation of KaRRi[6]. We compile our code with GCC 11.5 using optimization level `-O3` and `-march=native`. We run our experiments on two machines that both run Rocky Linux 9.4. Machine A has an AMD EPYC 7702 processor (64 physical cores) and 1.0 TiB of RAM. Machine B has an AMD EPYC 7713 Processor (64 physical cores) and 2.0 TiB of RAM. Note that we only need less than 10 GiB of RAM and use a single processor core in this chapter. We use 32-bit distance labels and the AVX2 SIMD instruction set with 256-bit registers to compute up to 8 operations in one vector instruction.

We evaluate PaRRot on the `KA` input instance described in Section 6.3.1. This input instance describes 975 773 requests in a morning peak from 6:00-9:00 am in the metropolitan region of Karlsruhe, Germany. The vehicle road network contains 276 314 vertices and 574 514 edges. The pedestrian road network has 667 831 vertices and 1 727 978 edges. The public transit network contains 49 800 stops, 26 971 routes, and 218 691 trips (for one day).

We precompute contraction hierarchies for the road and pedestrian networks using RoutingKit[7], which takes a few minutes. We also precompute the ULTRA transfer shortcut graph for the criteria arrival time, walking time, and number of transfers, as proposed in Section 8.4.3. This takes 40 minutes on a 32-core machine and results in a transfer shortcut graph with about 730 000 edges.

Unless stated otherwise, we use the following default assumptions and model parameters: The initial location of each RP vehicle is picked uniformly at random. Each RP vehicle has a seating capacity of four. The service time of every RP vehicle encompasses the whole observation period. We parametrize our cost function with $w_{\text{walk}} = 1$, which means that any duration t spent walking contributes twice as much cost as spending t in a vehicle or waiting. We use $\omega_{\text{transfer}} = 0$, so transfers do not make a PT journey less desirable. We choose this value to improve the chances of PT journeys and combined journeys compared

[4] Available at `https://github.com/molaupi/PaRRot`.
[5] Available at `https://github.com/kit-algo/ULTRA`.
[6] Available at `https://github.com/molaupi/karri`.
[7] Available at `https://github.com/RoutingKit`.

to RP-only journeys, which do not require transfers. In a more realistic mode choice model, this value should be set according to traveler preferences. We set the egress dispatch buffer to $t_{\text{buf}} = 15\,\text{min}$, as preliminary experiments showed that this is sufficient for providing small wait times for egress trips. The model parameters w_{detour} (part of the cost function), as well as m_{cost} and b_{cost} (define the egress cost heuristic) are evaluated in Section 8.6.2. We explain good choices for their default values in the according parts of that section.

We do not consider meeting points, except in Section 8.6.4 and Section 8.6.5.

Default Quality Metrics. A number of quality metrics are used in multiple experiments. We call these default quality metrics and explain their meaning here. Any metrics not listed here are explained where they are used.

The metric *drive* refers to the total time driven by all ride-pooling vehicles in hours. The metric *trip* reports the average trip time for all travelers in the road network. Values labeled *occ.* give the average number of passengers (occupancy) in each vehicle over time, ignoring any times in which the vehicle is idle. The metric *rides* describes the average number of riders that every vehicle serves during the entire three-hour observation period.

8.6.2. Parameter Tuning

We begin by tuning the model parameters of PaRRot. We experimentally find good values for the parameters m_{cost} and b_{cost} of the linear egress cost heuristic, the detour weight parameter w_{detour} of our cost function, and for the buffer time t_{buf} for dispatching egress trips. In this section, we use machine A.

Tuning the Cost Heuristic for Egress RP Trips. In Section 8.4.3, we explain that we need to use a heuristic for estimating the cost of egress ride-pooling trips, since pre-booking RP assignments far into the future leads to meaningless costs. Our cost estimate has the form $c^{\text{heu}}_{\text{egress}}(S) = m_{\text{cost}} \cdot \delta(S, \text{dest}) + b_{\text{cost}}$, where $\delta(S, \text{dest})$ is the distance from station $S \in \mathcal{S}$ to the destination dest of the request. Here, we show that there is in fact a linear correlation between cost and direct distance, and that the parameters m_{cost} and b_{cost} can reasonably be calibrated based on RP-only data.

We proceed as follows: First, we run PaRRot without combined journeys, i.e., only allowing walking, PT-only, or RP-only journeys. Then, for every request $r = (\text{orig}, \text{dest}, t_{\text{req}})$ that used RP, we extract the direct distance $\delta(\text{orig}, \text{dest})$ as well as the cost $c(\mathcal{I})$ of the request's best RP-only insertion $\mathcal{I}$ according to our cost function. We fit a linear regression model to the relationship between $c(\mathcal{I})$ and $\delta(\text{orig}, \text{dest})$ by the method of least squares to obtain m_{cost} and b_{cost}. We then use these values in a run of PaRRot with combined journeys. For every request r' that uses a combined journey $C = (r', S_{\text{access}}, S_{\text{egress}}, \mathcal{I}_{\text{access}}, J, \mathcal{I}_{\text{egress}})$ with an egress RP trip, we record the estimated cost $c^{\text{heu}}_{\text{egress}}(S_{\text{egress}})$ at the time of processing r' as well as the actual cost $c(\mathcal{I}_{\text{egress}})$ at the time of dispatching the egress RP trip. In Table 8.1, we show statistics on the linear regression and the ratio between estimated and actual cost of egress trips for the `KA` instances with three different fleet sizes.

First, consider the R^2 value of the linear regression (coefficient of determination). In statistics, the R^2 value describes how much of the variation of the dependent variable (in our case: $c(\mathcal{I})$) can be explained by the value of the independent variable (in our case: $\delta(\text{orig}, \text{dest})$). With an R^2 value of at least 0.57, our linear model can be expected to estimate costs decently well. The values are not closer to one because some parts of the cost function are not necessarily related to the distance of the request. For example, the added trip time of existing passengers depends on the number of passengers in the vehicle and their detour

Table 8.1 Statistics on tuning of cost heuristic for egress RP trips with $|F| \in \{5000, 10\,000, 20\,000\}$ vehicles. The R^2 value is the statistical coefficient of determination. It describes the influence of the direct distance $\delta(\text{orig}, \text{dest})$ on the cost $c(\mathcal{I})$ of a RP-only insertion $\mathcal{I}$ (between 0 for no influence and 1 for full dependence). Columns m_{cost} and b_{cost} show the slope and intercept that result from linear regression on the relation between $c(\mathcal{I})$ and $\delta(\text{orig}, \text{dest})$. The remaining columns describe the ratio of estimated cost $c^{\text{heu}}_{\text{egress}}(S_{\text{egress}})$ over actual cost $c(\mathcal{I}_{\text{egress}})$ of an egress RP trip. The columns show the p%-percentiles for $p \in \{5, 25, 50, 75, 95\}$ and the geometric mean of this ratio.

	lin. regression			observed ratio $c^{\text{heu}}_{\text{egress}}(S_{\text{egress}})/c(\mathcal{I}_{\text{egress}})$					
$\lvert F\rvert$	R^2	m_{cost}	b_{cost}	5%	25%	50%	75%	95%	g. mean
5000	0.57	3.68	1399	0.67	0.84	1.02	1.37	2.68	1.13
10 000	0.68	3.54	1038	0.71	0.89	1.09	1.54	3.13	1.24
20 000	0.76	3.45	772	0.73	0.89	1.10	1.62	3.30	1.27

caused by the new insertion. These other influences are more pronounced for vehicle fleets that experience more trips per vehicle, which also explains why the R^2 values is smaller for fewer vehicles. The slope of the linear estimator m_{cost} is about 3.5 because we parametrize our cost function with $w_{\text{detour}} = 3$, so that the added detour of a vehicle contributes 3 times more to the cost of an insertion than the trip time of a rider. We analyze different values of w_{detour} in the next paragraph. The intercept b_{cost} is a constant offset in "cost units", which can be interpreted as seconds of trip time. We find that b_{cost} is close to the average waiting time of the RP-only trips used to fit the model (1295s for 5000 vehicles, 875s for 10 000 vehicles, and 593s for 20 000 vehicles).

The distribution of the ratio between the estimated cost $c^{\text{heu}}_{\text{egress}}(S_{\text{egress}})$ and the observed cost $c(\mathcal{I}_{\text{egress}})$ of egress RP trips shows that our linear estimator works well. With a median between 1.02 and 1.10, and a geometric mean between 1.13 and 1.27, we slightly overestimate the cost of egress trips. A 5th percentile of at least 0.67 means that we rarely severely underestimate the actual cost. This is desirable in the sense that we do not want to assign artificially low costs to combined RP-PT journeys when comparing them to the existing modes of transportation. However, with a 75th percentile of up to 1.62 and a 95th percentile of more than 3, we may often be too pessimistic. In the future, an improved model based on actually performed egress trips could alleviate this issue. Additionally, if the arrival at an egress station is not too far into the future, we could immediately compute the exact cost of the egress trip instead of using the estimator.

Tuning the Detour Weight Parameter w_{detour}. Recall that our cost function has the basic form $c(C) = \textit{trip costs} + w_{\text{detour}} \cdot \textit{detour}$. The detour weight parameter w_{detour} defines a weight for the contribution of the vehicle detour to the total cost of an insertion relative to the trip time of a rider. In our parameter tuning experiments for KaRRi (see Figure 10.1), we establish $w_{\text{detour}} = 1$ as a good choice for KaRRi. However, PaRRot favors higher values of w_{detour}, as they might incentivize more efficient combined journeys over RP-only journeys. In the following, we evaluate the influence of w_{detour} on the quality of PaRRot.

In Table 8.2, we show different quality metrics for rider trips and vehicle operation with varying w_{detour} for three different fleet sizes. To begin with, the results affirm that values of $w_{\text{detour}} > 1$ are beneficial for combined journeys. The modal share of combined journeys (column *co.*) is highest for $w_{\text{detour}} = 2$ or $w_{\text{detour}} = 3$. As is to be expected, the total vehicle drive time (*drive*) becomes smaller for larger values of w_{detour}, since the costs of RP

Table 8.2 Performance of PaRRot for different values of the detour weight parameter $w_{\text{detour}} \in \{1, 2, 3, 4\}$ with $|F| \in \{5000, 10000, 20000\}$ vehicles. Shows modal share of combined journeys in percent (*co.*), default metrics (*drive*, *trip*, and *occ*, see Section 8.6.1), and PaRRot running time per request in milliseconds (*time*).

$\lvert F\rvert$	w_{detour}	co. [%]	drive [h]	trip [hh:mm]	occ. [# pax]	time [ms]
5000	1	3.27	16 131	40:02	2.67	9.85
	2	3.45	12 161	39:31	2.67	6.00
	3	3.56	9322	39:15	2.62	4.47
	4	3.31	7193	39:10	2.55	3.82
10 000	1	5.96	26 890	37:57	2.71	13.30
	2	7.05	19 888	37:16	2.66	7.28
	3	7.14	14 576	37:12	2.57	5.00
	4	5.87	10 323	37:39	2.48	4.06
20 000	1	9.09	42 698	33:59	2.66	17.53
	2	11.93	29 995	33:43	2.58	8.58
	3	11.35	20 136	34:44	2.47	5.45
	4	8.40	13 064	36:15	2.40	4.25

trips increase relative to walking and PT-only journeys, which are not affected by w_{detour}. Although ride-pooling is usually faster than public transportation, the average trip time (*trip*) actually decreases slightly for larger w_{detour} up to some threshold. Thus, larger values of w_{detour} seem to restrict ride-pooling to only those trips that really profit from it compared to PT or walking. We can also see this in the slightly reduced average vehicle occupancy (*occ.*). With fewer RP resources spent on non-essential RP trips, there is more room for the vehicle fleet to work on requests that are not served well by public transit. The increased share of combined journeys and the reduced trip time indicate that the multimodal approach can help to balance out essential and non-essential RP trips. If w_{detour} becomes too large, ride-pooling is restricted too much, and trip times grow again.

We pick $w_{\text{detour}} = 4$ for 5000 vehicles, $w_{\text{detour}} = 3$ for 10 000 vehicles, and $w_{\text{detour}} = 2$ for 20 000 vehicles to minimize the trip time. These values also maximize the modal share of combined journeys for 10 000 and 20 000 vehicles. The PaRRot running time per request for these choices ranges from 3.82 to 8.58 milliseconds. Interestingly, for all three fleet sizes, the chosen value of w_{detour} leads to about 1.5 hours of drive time per vehicle, which is half of the observation period of three hours. From the perspective of a mobility-as-a-service provider, scheduling requests so that RP vehicles drive half of the time may be a good heuristic to avoid overloading the fleet and losing the ability to respond well to future requests. Note that drive time only includes actual time spent driving. In our scenario, vehicles additionally spend an average of 25-30 minutes making stops to pick up and drop off riders (with an enforced minimum stop time of 1 minute per stop).

8.6.3. Fleet Size

We analyze the impact of the number of RP vehicles in the RP fleet on the quality and running time of PaRRot. We consider a varying fleet size $|F| \in \{5000, 10\,000, 20\,000, 40\,000, 80\,000\}$ on the `KA` instance. Every vehicle has a capacity of four. We use the values of w_{detour} found

Table 8.3 Quality and performance of PaRRot for varying of vehicles $|F| \in \{5000, 10\,000, 20\,000, 40\,000, 80\,000\}$. Left side shows quality: Modal share of ride-pooling (*RP*) and combined journeys (*co.*) in percent, and default metrics (*drive*, *trip*, *occ.*, and *rides*, see Section 8.6.1). Right side shows performance (averages per request): Running time in ms (*time*), number of bucket entries scanned by elliptic BCH queries (*entries*), and number of candidate pickup vehicles ($|F_p|$).

$\|F\|$	RP [%]	co. [%]	drive [h]	trip [hh:mm]	occ. [# pax]	rides [# pax]	time [ms]	entries	$\|F_p\|$
5000	7.28	3.31	7193	39:10	2.55	20.7	3.8	9811	63
10 000	13.14	7.14	14 576	37:12	2.57	19.8	5.0	22 366	146
20 000	24.47	11.93	29 995	33:43	2.58	17.8	8.6	65 344	445
40 000	33.44	17.09	40 231	29:48	2.42	12.3	9.7	69 206	470
80 000	29.38	14.20	35 232	31:53	2.50	5.3	9.6	68 481	473

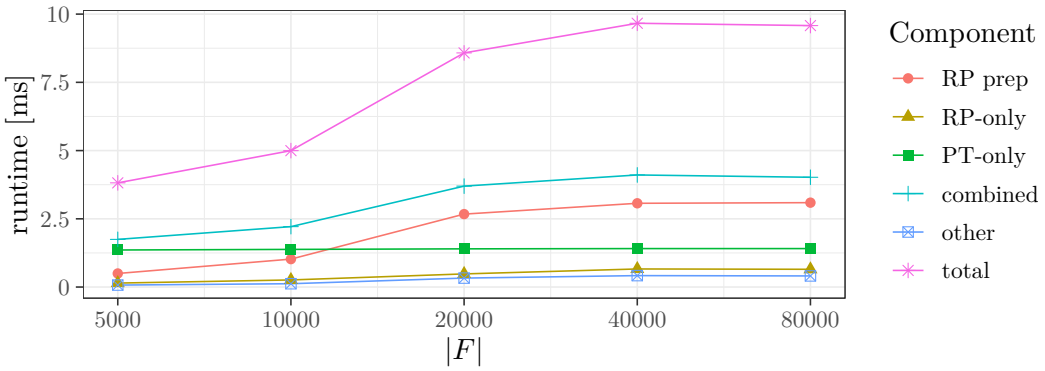


Figure 8.4 Runtime per request (in ms) for fleet size $|F| \in \{5000, 10\,000, 20\,000, 40\,000, 80\,000\}$. All vehicles have a capacity of four. Shows running time of RP preparation phase (*RP prep*), finding RP-only journeys (*RP-only*), finding PT-only journeys (*PT-only*), finding combined journeys (*combined*), updating the system state (*other*), and total running time (*total*). Note the logarithmic x-axis.

in Section 8.6.2 for 5000, 10 000, and 20 000 vehicles. For 40 000 and 80 000 vehicles, we use $w_{\text{detour}} = 2$, the same value as for 20 000 vehicles. In this section, we use machine A.

Quality. In Table 8.3, we give an indication for the quality of trips with different fleet sizes. We focus on the left side of the table here, as the last three columns give useful insights for the performance, which we discuss later. To begin with, consider the modal share of ride-pooling (column *RP*) and combined journeys (column *co.*). As expected, larger vehicle fleets can provide better RP journeys to more travelers. Thus, the modal share of ride-pooling (column *RP*) and combined journeys (column *co.*) increase with a growing number of vehicles. However, this comes at a cost of increased RP vehicle operation time (column *drive*). The average trip time decreases by more than nine minutes when increasing the fleet size from 5000 to 40 000 vehicles, while the average occupancy per vehicle decreases slightly.

Note that the average trip time increases from 40 000 to 80 000 vehicles. This is because our choice of $w_{\text{detour}} = 2$ is too strict for 80 000 vehicles, which can also be seen in the much smaller average number of rides per vehicle (column *rides*). With a proper parametrization, we would expect the observed trend to continue.

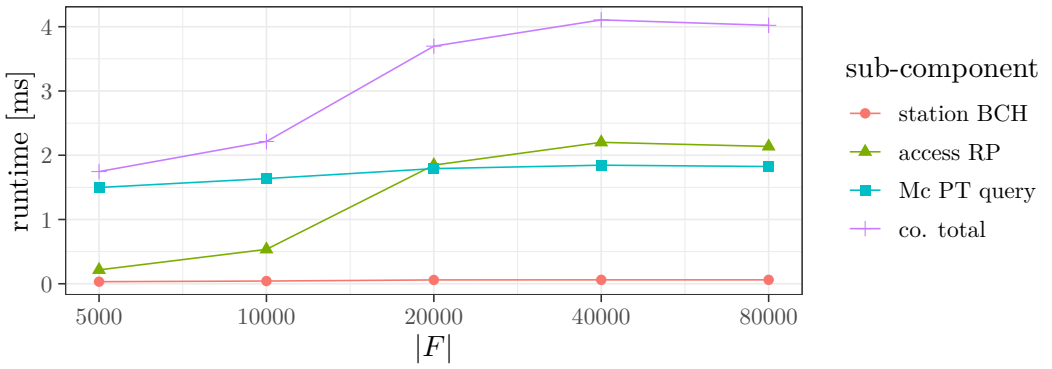


Figure 8.5 Runtime per request (in ms) for finding combined journeys with varying fleet size $|F| \in \{5000, 10\,000, 20\,000, 40\,000, 80\,000\}$. All vehicles have a capacity of four. Shows runtime of station BCH queries (*station BCH*), search for access RP trips (*access RP*), more-criteria PT query (*Mc PT query*), and total time (*co. total*). Note the logarithmic x-axis.

Running Time. In Figure 8.4, we show the total running time per request of PaRRot and its components on the y-axis for varying fleet size on the x-axis (logarithmic).

The component *RP prep* describes the running time for work whose result is used for RP-only and combined journeys. The main part of *RP prep* are elliptic BCH searches (see Section 4.6.1), which are used to compute feasible pickup and dropoff vehicles for RP-only insertions and for access RP trips (see Section 8.5.2). The running time of *RP prep* increases with larger vehicle fleets, as the elliptic BCH searches need to scan more bucket entries (column *entries* in Table 8.3). The average running time of this phase is at most 3ms.

RP-only contains KaRRi's efficient last stop BCH searches (see Sections 4.7 and 4.8) and the enumeration of possible insertions. The running time of this phase scales linearly with the number of vehicles and is at most 0.66ms per request. The *PT-only* query is unaffected by the number of vehicles, as it does not interact with the RP fleet. It takes an average of about 1.4ms per request. The component *other* involves updating the system state for an insertion. It contributes at most 0.42ms to the total running time.

The component *combined* represents all work performed for finding combined journeys. We show the running times of its sub-components in Figure 8.5.

The *station BCH* queries compute the distances from each pickup location to every station and from every station to each dropoff location (see Sections 8.4.3 and 8.5.1). Their running time increases slightly with a larger number of vehicles, as the station BCH queries for egress distances are pruned based on the egress cost heuristic. Since larger vehicle fleets lead to smaller RP costs, the heuristic gives smaller cost estimates (cf. Table 8.1), which delays pruning. Even for 80 000 vehicles, this phase takes only 0.06ms.

The running time for finding access RP trips (*access RP*) increases with growing fleet size, and is the predominant reason for the overall increase in time spent searching for combined journeys. The number of possible access trips scales linearly with the number of vehicles that can pick a traveler up ($|F_p|$), which we show in the last column of Table 8.3.

The running time of the *Mc PT query* increases slightly with the number of vehicles, as there are possibly more access and egress stations that can be reached when more RP vehicles are available. The running time of this phase lies between 1.50 and 1.85ms.

Table 8.4 Quality of PaRRot for $|F| = 10\,000$ vehicles and varying meeting point walking radius $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}\}$. Shows modal share of ride-pooling (*RP*) and combined journeys (*co.*) in percent, and default metrics (*drive*, *trip*, *occ.*, and *rides*, see Section 8.6.1). The columns labeled *trip / direct car* show the geometric mean of the ratio of the observed trip times over hypothetical direct car trips from origin to destination. Column *all* considers all travelers, column *RP* considers RP-only travelers, and column *co.* considers travelers who use a combined journey.

							trip / direct car		
ρ	RP [%]	co. [%]	drive [h]	trip [hh:mm]	occ. [# pax]	rides [# pax]	all	RP	co.
0	13.14	7.14	14 576	37:12	2.57	19.8	3.16	3.67	2.83
150	16.73	7.98	14 258	36:33	2.77	24.1	3.10	3.41	2.75
300	19.16	8.29	13 768	36:03	2.89	26.8	3.06	3.26	2.70

8.6.4. Meeting Points

In this section, we analyze the running time and quality of PaRRot with meeting points. These running times shown are for experiments on machine B.

Quality. Table 8.4 shows the solution quality when considering all meeting points in a walking radius $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}\}$ on the `KA` instance with 10 000 vehicles. We find that meeting points substantially increase the share of travelers who use RP-only trips, and slightly increase the share of travelers who use combined journeys. However, this does not lead to an increase in total vehicle time driven or average trip times. In fact, the total vehicle drive time decreases by 2.2% for $\rho = 150\text{s}$ and by 5.5% for $\rho = 300\text{s}$. The average trip time for all requests decreases by more than one minute for a walking radius of $\rho = 300\text{s}$. The average occupancy and the number of rides increase, which is a sign of improved pooling, as the drive time does not grow. To get a better understanding of how much meeting points decrease the trip times of RP-only and combined journeys, we compare the trip times to hypothetical journeys using a personal car. For every request, we compute the shortest path distance from origin to destination and normalize the trip time by this value. The last three columns of Table 8.4 report the geometric mean of normalized trip times for all requests, RP-only journeys, and combined journeys. The normalized trip time decreases only slightly for combined journeys, but RP-only journeys benefit greatly with a reduction of the normalized trip time by more than 11% for $\rho = 300\text{s}$. This is in line with our previous findings for meeting points in Sections 4.9.4 and 6.4.3. The improvement for combined journeys is smaller because their trip time still depends on the transit schedule. For example, an access RP trip may become cheaper in terms of RP vehicle costs, but as long as the traveler uses the same PT connection, the trip time does not change.

Running Time. We now consider the running time of PaRRot with different numbers of meeting points. Figure 8.6 gives an overview of the running time for every component of PaRRot. The total running time increases by a factor of 3.06 from $\rho = 0\text{s}$ to $\rho = 150\text{s}$ and by a factor of 4.13 from $\rho = 150\text{s}$ to $\rho = 300\text{s}$. The running times of the *RP-only* and *combined* components increase the most. To explain this, Table 8.5 shows the number of insertions tried for RP-only trips and access RP trips by type of insertion (see Figure 4.2).

For *RP-only*, the number of ordinary insertions grows a lot with larger walking radius ρ.

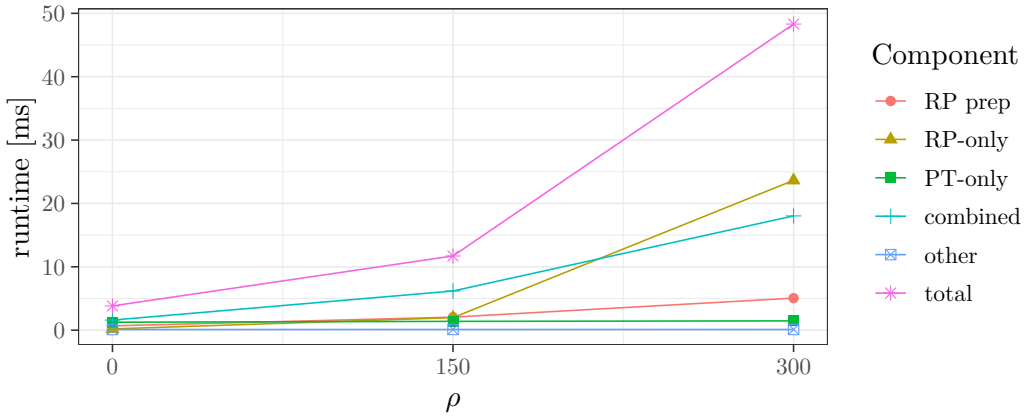


Figure 8.6 Running time of components of PaRRot (y-axis) for $|F| = 10\,000$ vehicles and varying meeting point walking radius $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}\}$ (x-axis). Shows running time of RP preparation phase (*RP prep*), finding RP-only journeys (*RP-only*), finding PT-only journeys (*PT-only*), finding combined journeys (*combined*), updating the system state (*other*), and total running time (*total*).

Table 8.5 Number of insertions tried by PaRRot for different insertion types for RP-only insertions (*RP-only*) and access RP insertions (*access*). Considers $|F| = 10\,000$ vehicles and varying meeting point walking radius $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}\}$. Columns N_ρ^p and N_ρ^d state the average number of pickup and dropoff locations in the walking radius. Shows average number of ordinary insertions (*ord.*), pickup-before-next-stop insertions (*PBNS*), pickup-after-last-stop insertions (*PALS*), dropoff-after-last-stop insertions (*DALS*), and total number of insertions per request.

type	ρ	N_ρ^p	N_ρ^d	ord.	PBNS	PALS	DALS	total
RP-only	0	1	1	118	7	1	82	208
	150	20	19	65 655	3920	1	1576	71 151
	300	72	70	917 504	41 342	1	4713	963 559
access	0	1	1	1415	111	399	1445	3370
	150	20	19	32 177	2178	361	38 959	73 674
	300	72	70	101 235	5209	382	138 341	245 167

This is because for every vehicle $\nu \in F$ that can perform the pickup and dropoff of a new request, we need to try every combination of pickup and dropoff location. Thus, the number of ordinary insertions grows with the number of PD-pairs $N_\rho^p \cdot N_\rho^d$. For $\rho = 300\text{s}$, the time for enumerating ordinary insertions dominates the total running time of the *RP-only* phase.

For *access* RP trips, the number of insertions grows in proportion to the number of pickups, since for every access RP trip, we need to try every possible pickup location. Thus, the running time of the *combined* phase grows linearly with the number of pickups N_ρ^p.

8.6.5. Impact of Combined Journeys

In this section, we compare our solution PaRRot to a baseline without combined RP-PT journeys. For the baseline, we run our RP dispatcher KaRRi and a PT-only query like in PaRRot, but simply leave out the steps for finding combined journeys. The running time results in this section are for machine B.

Table 8.6 Performance of components of unimodal baseline (alg = U) and PaRRot (alg = P) for $|F|$ vehicles and meeting point walking radius ρ. Shows average running time per request (in milliseconds) for RP preparation needed for PT-only and combined journeys (*RP prep*), finding RP-only journeys (*RP-only*), finding PT-only journeys (*PT-only*), finding combined journeys (*combined*), finding walking journeys and updating the system state (*other*), as well as total running time (*total*). Last two rows show average number of entries scanned by elliptic BCH searches (*entries*) and average number ordinary RP-only insertions tried (*ord ins*).

$\|F\|$	ρ	alg	RP prep	RP-only	PT-only	combined	other	total	entries	ord ins
10 000	0	U	0.72	0.19	0.99	0.00	0.09	1.99	32 688	173
		P	0.68	0.20	1.25	1.59	0.10	3.83	22 366	118
40 000	0	U	3.18	0.55	1.22	0.00	0.51	5.45	137 490	594
		P	2.10	0.54	1.47	3.22	0.37	7.71	69 206	268
10 000	300	U	7.09	42.63	1.30	0.00	0.11	51.12	156 768	1 688 949
		P	5.06	23.68	1.48	18.03	0.11	48.36	101 158	917 503

Running Time. We first consider the running time of PaRRot in comparison to the baseline. Table 8.6 provides the running times of both algorithms for 10 000 and 40 000 vehicles without meeting points, as well as for 10 000 vehicles with meeting points. We would expect that the running time of the components *RP prep*, *RP-only*, *PT-only*, and *other* would be roughly the same for the baseline and PaRRot, since the steps are identical. We find that this is not the case. The *RP prep* phase is faster for PaRRot than for the baseline. Additionally, when using meeting points, the *RP-only* phase is much faster for PaRRot than for the baseline.

This is caused by a smaller number of bucket entries that need to be scanned by elliptic BCH searches (column *entries*), and a smaller number of ordinary RP-only insertions that need to be tried (column *ord ins*). We believe that both of these effects are due to stricter deadlines in RP routes imposed by access RP trips. In an access RP trip, the traveler needs to arrive at the access station in time to catch their PT connection. When a dropoff at an access station is inserted into a vehicle route as a new stop s_j, the traveler's arrival deadline has to be propagated to every earlier stop s_a with $a < j$ in the route. Thus, vehicles that perform access RP trips have little room for detours, which causes their detour ellipses to be small. In effect, these vehicles have few elliptic bucket entries (cf. Section 4.3.2). The tight time constraints also lead to a smaller number of vehicles that can make both the pickup and the dropoff for an ordinary RP-only insertion without breaking a time constraint.

In all three configurations, the fraction of access RP trips among all RP trips is about 21%. As deadlines are propagated through routes, this relatively small number of access RP trips is enough to cause the observed reduction in the number of bucket entries scanned and in the number of insertions tried. The effect on the running time of the *RP-only* phase is only noticeable when using meeting points, because the number of insertions also scales with the number of PD-pairs as described in Section 8.6.4.

The overall running time for PaRRot is at most 2.26ms larger than for the baseline. Thus, using PaRRot to add combined journeys to an existing router for multiple modes of transport comes with very little overhead. When considering meeting points, we slightly improve the total running time per request compared to the baseline. As discussed, this is due to pruning based on tight deadlines for access RP trips.

Table 8.7 Quality of unimodal baseline ($alg = \mathrm{U}$) and PaRRot ($alg = \mathrm{P}$) for $|F| = 10\,000$ vehicles with walking radius $\rho = 300$s and for $|F| = 40\,000$ vehicles with walking radius $\rho = 0$s. Columns labeled *modal split* show modal share of walking (*walk*), public transportation (*PT*), ride-pooling (*RP*), and combined journeys (*co.*) in percent. Also shows default metrics (*drive*, *occ.*, and *trip*, see Section 8.6.1).

$\|F\|$	alg	modal split				drive	occ.	trip
		walk [%]	PT [%]	RP [%]	co. [%]	[h]	[# pax]	[hh:mm]
10 000	U	24.81	51.65	23.53	0.00	13 726	2.96	37:02
	P	25.36	47.18	19.16	8.29	13 768	2.89	36:03
40 000	U	17.81	36.81	45.38	0.00	43 976	2.57	30:45
	P	18.15	31.32	33.44	17.09	40 231	2.42	29:48

Improvement in Solution Quality. We compare the solution quality of PaRRot to the baseline without combined journeys. We discuss these results with the viewpoint of introducing combined journeys to a system that did not previously have them. As an overview of the impact of combined journeys, consider Table 8.7. We compare the quality of PaRRot and the baseline for a small fleet of 10 000 vehicles with meeting points, and for a larger fleet of 40 000 vehicles without meeting points. Combined journeys have a modal share of 8.2% for the smaller fleet and 17.09% for the larger fleet. Travelers who use combined journeys in PaRRot mostly use public transit or ride-pooling in the baseline, as illustrated in Figure 8.7.

With 10 000 vehicles, the total vehicle drive time increases by 0.3% when introducing combined journeys. This is due to the fact that the majority of travelers on combined journeys previously used PT (see Figure 8.7). These travelers previously did not incur any vehicle drive time but now require access or egress RP trips. With 40 000 vehicles, the total drive time instead decreases by about 8.5%. Here, more travelers that previously used RP switch to combined journeys. Each of these travelers now incurs less vehicle operation time, since they take PT for part of their journey. The average vehicle occupancy falls slightly in both scenarios. We interpret this as fleets becoming less overloaded with non-essential ride-pooling trips. Average trip times for all travelers decrease by about one minute. This represents a reduction by 2.7% with 10 000 vehicles and by 3.1% for 40 000 vehicles.

In the following, we consider the trip times of travelers in more detail to better understand which kinds of requests benefit from combined journeys. In Table 8.8, we show the trip time per mode of transportation in comparison to the travel time of a hypothetical direct trip by car from the origin to the destination (denoted δ_{OD} in the table). We call this travel time the *OD-distance*. The average trip time per rider is given by the column *trip*. Columns *abs* and *rel* describe the absolute difference and relative ratio between trip time and OD-distance, which we call *absolute and relative trip detour*. The left side and right side of the table show results for the baseline and for PaRRot, respectively.

First, compare the top left (baseline with 10 000 vehicles) to the bottom left (baseline with 40 000 vehicles). The OD-distance of travelers that use RP is a lot longer with 40 000 vehicles than with 10 000 vehicles, which means that RP is chosen more often for long distance requests when more vehicles are available.

Now, compare the left side (baseline) to the right side (PaRRot). We begin by considering the changes for PT-only and RP-only journeys. Introducing combined journeys decreases the average trip time of PT-only journeys and RP-only journeys by at least two minutes with

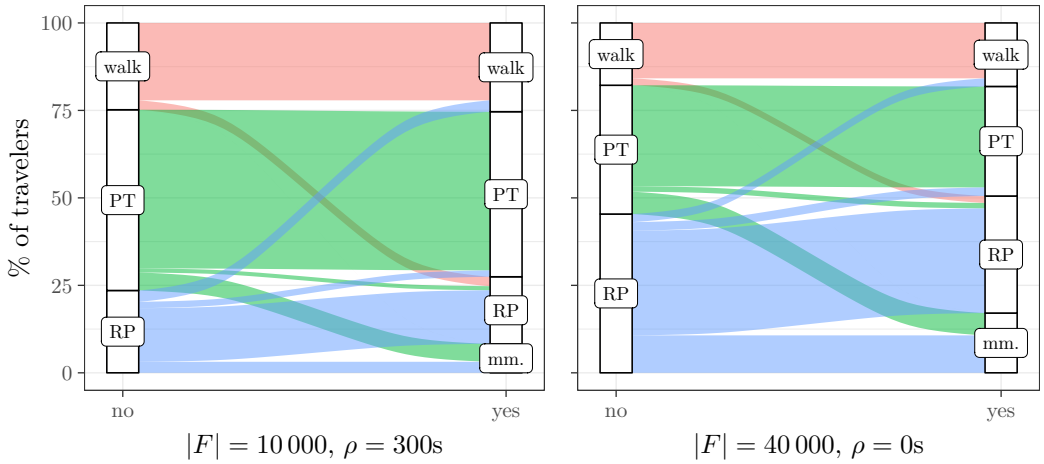


Figure 8.7 Visualization of change of modal split when introducing combined journeys for $|F| = 10\,000$ vehicles with $\rho = 300$s (left plot) and $|F| = 40\,000$ vehicles with $\rho = 0$s (right plot). Heights of boxes on left side (labeled *no* on x-axis) show share of travelers per mode (y-axis) without combined journeys. Heights of boxes on right side (labeled *yes* on x-axis) show share of travelers per mode (x-axis) with combined journeys (*co.* means combined). Height of each color stripe shows share of travelers that use mode at the left end of the stripe if combined journeys are not allowed and use mode at the right end of the stripe if combined journeys are allowed.

10 000 vehicles, and by at least five minutes with 40 000 vehicles. This is in part because the OD-distance of travelers that choose PT-only or RP-only journeys becomes smaller. However, the absolute trip detour for PT-only and RP-only journeys is also reduced by between one and three minutes, which means these journeys become noticeably faster.

Now consider the trips of combined journeys (row *co.*). The OD-distance of combined journeys is much higher than for all other modes, which means that combined journeys are particularly attractive for travelers that need to go a long distance. Combined journeys provide good trip quality for these travelers, as the absolute trip detour is actually smaller than for PT-only journeys, and the relative trip detour is the smallest out of any mode.

This is important for riders who were forced to use RP previously because they have a long OD-distance (so they cannot walk), and bad access to transit (so they cannot use PT). We find that this description matches the group of travelers who switch from RP to combined journeys: The average OD-distance for these travelers is 17:31min with 10 000 vehicles and 20:36min with 40 000 vehicles, which is much higher than the overall average for RP-only journeys. The average PT-only trip time for these travelers is 90:49min with 10 000 vehicles and 83:04min with 40 000 vehicles, which is above the overall average of 57:12min.

In conclusion, combined journeys provide a useful mode of transportation that helps improve the overall quality of the transportation network by filling a gap for riders that previously had no alternative but to use an RP-only trip. This reduces overall trip time, since ride-pooling resources are freed up. If the number of vehicles is sufficiently large, the change from RP-only to combined journeys leads to savings in total vehicle operation time. For small fleets, fewer travelers had the option to use ride-pooling to begin with, so the introduction of combined journeys draws in more travelers from public transportation. While trip times still improve, the total vehicle operation time stays about the same.

Table 8.8 Trip times for trips found by unimodal baseline and PaRRot for $|F| = 10\,000$ vehicles with walking radius $\rho = 300$s and for $|F| = 40\,000$ vehicles with walking radius $\rho = 0$s. Each row gives values for requests that use the given mode. Column labeled δ_{OD} shows average direct travel time from origin to destination by car for requests that chose this mode. Column labeled *trip* shows average trip time for trips with this mode. Column *abs* gives average absolute difference between trip time and car travel time. Column *rel* gives geometric mean of ratio between trip time and car travel time.

		baseline				PaRRot			
$\|F\|$	mode	δ_{OD} [hh:mm]	trip [hh:mm]	abs [hh:mm]	rel	δ_{OD} [hh:mm]	trip [hh:mm]	abs [hh:mm]	rel
10 000	walk	03:25	09:50	06:25	2.90	03:28	10:03	06:34	2.92
	PT	17:15	54:48	37:33	3.19	16:37	51:23	34:47	3.13
	RP	09:00	26:43	17:43	3.05	07:34	24:33	16:58	3.26
	co.					20:37	54:56	34:19	2.70
	all	11:53	37:02	25:09	3.08	11:53	36:03	24:10	3.06
40 000	walk	02:46	07:05	04:19	2.69	02:47	07:07	04:21	2.69
	PT	16:51	47:31	30:40	2.87	15:29	42:26	26:57	2.83
	RP	11:25	26:25	15:01	2.43	08:37	21:23	12:46	2.57
	co.					21:19	47:11	25:51	2.25
	all	11:53	30:45	18:52	2.63	11:53	29:48	17:55	2.61

8.6.6. Qualitative Comparison with Lorente et al. [Lor+23]

We compare the qualities of PaRRot to the approach by Lorente et al. [Lor+23] that we describe in Section 8.1.1.

Lorente et al. focus on finding a solution with the best possible quality in reasonable time. A rolling-horizon approach (cf. Section 2.3.2) is a natural way to achieve this goal, as it has been shown to result in high quality solutions in RP-only dispatchers [Alo+17; EDB20]. They use a detailed model that incorporates aspects such as the traveler fare and priorities for different requests. Solving a mathematical program in the optimization step for each batch has the potential to more successfully identify candidates for pooling than our insertion heuristic (cf. Section 2.3.2).

In their experimental evaluation, Lorente et al. show that up to almost 20% of travelers use combined journeys, which is similar to our scenario with 40 000 vehicles. However, in our results, the modal share of RP-only is about two times larger than in the results by Lorente et al. This is likely due to their more precise model, particularly considering travel fares, and perhaps due to finding more rider synergies in the mathematical program. Additionally, the observation area in our experiments (Karlsruhe and surroundings, Germany) is not as well covered by public transit as the observation area in the other work (Barcelona, Spain), which increases the need for ride-pooling in certain trips. As a similarity, both works identify that large distances requests are the most likely candidates for combined journeys.

We also briefly revisit the goals for our approach in comparison to Lorente et al., which we outline at the end of Section 8.1.1. Our algorithm PaRRot considers all public transport stations as access stations. We do not have to rely on a heuristic sample of access stations, because we prune our queries for access RP trips based on the best non-combined journeys known. Furthermore, our on-the-fly more-criteria PT query is exhaustive and does not limit the complexity of the PT leg of combined journeys. Finally, our running times in the order of

milliseconds per request allow for more responsive queries than the rolling-horizon approach proposed by Lorente et al., which takes seconds to return a journey to the traveler.

8.7. Conclusion

In this chapter, we design PaRRot, an efficient algorithm for multimodal journey planning with ride-pooling and public transportation. We find that multimodal planning with ride-pooling differs substantially from traditional modes like walking or cycling, as ride-pooling trips always depend on the changing vehicle routes. We show that we need more-criteria routing to find optimal journeys with respect to a cost function that considers RP vehicle operation costs and rider trip times. We extend our ride-pooling dispatcher KaRRi to find one-to-many RP trips from a traveler's origin to the stations in the PT network. We introduce new preprocessing and pruning techniques to speed up this process. In an experimental evaluation, we show that PaRRot finds multi-modal journeys in less than ten milliseconds. Since we can reuse information and prune based on the results of KaRRi, the additional overhead for finding combined journeys over a KaRRi query is less than 2.5ms. We show that the introduction of combined journeys can reduce average trip times of all travelers by one minute and improve the total drive time of ride-pooling vehicles by 8.5%.

In this chapter, we focus on the scalability of our approach. In the future, we would like to improve our model of traveler behavior to get a more reliable understanding of the effect of combined journeys in a transportation system. In particular, it would be interesting to find out whether the degree of pooling is higher on trips to PT stations. Additionally, the fact that combined journeys are useful for travelers that go long distances, but have bad access to transit, makes combined journeys a promising solution for peripheral areas of cities (cf. Section 6.4.5). In the future, we want to analyze the potential of combined journeys in these areas in more detail.

Based on these insights, we might be able to further tune the performance of our algorithm. For example, we observe very high loads on vehicles in our experiments with an average occupancy of almost three. These high loads render some of the pruning techniques of KaRRi and PaRRot ineffective. Thus, we could optimize our algorithms for this case, e.g. by considering pruning methods based on vehicle capacities in the shortest-path queries.

9 Conclusion

In this work, we propose algorithmic approaches to dynamic ride-pooling that improve the state of the art in terms of response time and scalability. We show that algorithmic speedup techniques can facilitate fast routing even for extensions of the problem like meeting points, transfers, and multimodal integration, which serve to reduce operational costs and improve trip quality of on-demand shared transportation. In this concluding chapter, we summarize the results of our work and describe ideas for future improvements of our approach.

9.1. Summary of Results

We summarize the results of this work by chapter.

In Chapter 4, we introduce our ride-pooling dispatcher KaRRi that uses efficient many-to-many routing with bucket contraction hierarchies to solve ride-pooling with meeting points. We introduce new pruning techniques and exploit the locality of meeting points for bundled shortest-path queries. KaRRi computes insertions with optimal meeting points in milliseconds, which is as fast as the previous state of the art without meeting points. At the same time, meeting points reduce both rider trip times and vehicle operation costs by 10%.

In Chapter 5, we propose Mt-KaRRi, a multi-threaded version of our dispatcher. A simple synchronization procedure allows us to concurrently process multiple requests. We show that the throughput of our algorithm scales well with the number of threads used (speedups of up to 53 with 96 threads), and that our parallelization does not incur a substantial loss in solution quality (less than 1% in all metrics considered). Mt-KaRRi can process input instances with millions of requests per hour in real time, which is a throughput that is two orders of magnitude higher than previous approaches.

In Chapter 6, we study ride-pooling as a part of a full transportation system on large-scale travel demand in metropolitan areas. We include a model for rider mode choice and evaluate ride-pooling in competition with walking, public transportation, and private cars. We show the effects of ride-pooling at an unprecedented scale in terms of number of travelers and number of vehicles. With this, we show that KaRRi and Mt-KaRRi are well suited for the study of ride-pooling and for future large-scale production systems.

In Chapter 7, we introduce KaRRiT, an extension of KaRRi that finds trips with transfers between ride-pooling vehicles. The exact variant of KaRRiT finds optimal transfer locations and uses new pruning techniques and parallelization to handle the large solution space. A heuristic variant of KaRRiT further speeds up the process. In experiments, we find that transfers reduce the total vehicle operation time compared to KaRRi. While our speedup techniques are effective, the exact variant of KaRRiT is up to three orders of magnitude slower than KaRRi, which is likely too slow for production systems. The heuristic variant is one order of magnitude faster than the exact variant, but retains most of the quality advantage, which makes it a more promising candidate for large-scale applications.

In Chapter 8, we integrate KaRRi in a framework for multimodal journey planning to obtain PaRRot, a routing algorithm that efficiently finds journeys that combine ride-pooling and public transportation. We identify new challenges for multimodal planning caused by

the dynamic nature of ride-pooling. To solve these issues, we extend KaRRi for one-to-many ride-pooling queries that consider all stations as targets, and use multi-criteria public-transit routing to find combined journeys that optimize a cost function. PaRRot finds optimal combined journeys in less than ten milliseconds for large transportation networks with tens of thousands of ride-pooling vehicles. The overhead compared to a baseline without combined journeys is less than 2.5ms. At the same time, introducing combined journeys improves average trip times and reduces the operation time of ride-pooling vehicles by up to 8.5%. PaRRot finds similar trends in solution quality as the state of the art, but PaRRot's response times are one to two orders of magnitude smaller.

9.2. Future Work

In this section, we describe some directions for future research that we consider to be particularly important based on our results.

Computation Times and Scalability. KaRRi scales well to large numbers of requests and vehicles, but some of the proposed pruning techniques do not work well when the number of riders per vehicle is very high (outlined in Sections 6.4.3 and 8.6.5). It is possible that a real-world dispatcher would avoid this level of congestion on the fleet with measures like high fares or denying service to some requests. However, if these situations do occur in practice, we should improve KaRRi for congested fleets, e.g. by encoding vehicle capacity limits into the BCH bucket entries for better pruning of candidate vehicles.

The synchronization approach used by our multi-threaded dispatcher Mt-KaRRi is based on iteratively finding new assignments for conflicting requests. This can become a bottleneck as it increases the span of the parallel algorithm (the length of the critical path). A solution that computes multiple alternative insertions for each request in a batch could avoid having to recompute an assignment from scratch.

We believe that transfers have great potential for improving the quality of ride-pooling, and that our experiments in Section 7.6.4 do not fully appreciate this potential, as they use instances with rather sparse demand. Thus, we would like to further optimize KaRRiT, our dispatcher for ride-pooling with transfers. In particular, we can likely improve upon the brute-force reconstruction of detour ellipses. For example, we can likely identify good transfer locations by finding out where vehicle routes approach each other (in time and space) as a preprocessing step.

Re-optimization. As first pointed out by Psaraftis [Psa80], the solution quality of insertion heuristics benefits from re-optimization steps, which we currently do not employ. As a first step to improving the solution quality of our dispatchers, we want to compare KaRRi to approaches with better quality (e.g. based on the rolling-horizon strategy). Then, we could optimize our sequential (non-batched) and batched approaches as follows.

For the sequential approach, we can run re-optimization procedures intermittently at fixed points during the dispatching process. We can model the current state of routes and rider journeys as a discrete optimization problem and develop fast (approximate or heuristic) solution methods. In particular, it would be interesting to find reduction rules for the optimization problem based on different problem constraints.

For the batched approach used by Mt-KaRRi, we could try to move closer to a rolling-horizon strategy, but still focus on scalability and a fast response time. We would like to

find an approach that improves upon the greedy insertion heuristic, but avoids having to solve a mathematical program, as this would become a bottleneck for scalability.

Improving Model Accuracy. Our current model of the behavior of travelers and ride-pooling service providers lacks sophistication. Most importantly, we do not consider money as an incentive for travelers (fares) or for operators (operation costs, revenue). To improve upon this, we would like to integrate a model for fares, with fair splitting of costs between riders, as well as a decision model for when a dispatcher should reject a request due to the ride being in opposition to goals like profitability or sustainability.

Furthermore, our model of road networks is insufficient, as we assume that every vehicle can travel at the speed limit at any time. In urban environments, it is important to consider congestion effects that can substantially increase travel times. We made a first step towards this goal by incorporating customizable contraction hierarchies into KaRRi, which allow us to easily use up-to-date edge travel times in the network. However, changes in travel times may require re-optimizing the routes of ride-pooling vehicles. In the future, we would like to enable KaRRi to handle updated travel times with minimal loss in solution quality.

Finally, for a much more precise approach, we could integrate our algorithms with an agent-based transport simulation like mobiTopp [MKV13] or MATSim [HNA16]. These simulations are designed to accurately model traveler behavior, congestion, and many more aspects of urban transportation. Wilkes et al. [Wil+21] show how a ride-pooling dispatcher can serve as the back-end in a mobility simulation. The scalability of KaRRi and its variants would unlock the ability to consider meeting points, transfers, and multimodal routing in these simulations at a large scale.

Appendix

10 Technical Appendix

In this chapter, we provide technical details on the input data used in our ride-pooling simulations. Furthermore, we describe parameter tuning experiments for the model parameters of KaRRi and Mt-KaRRi.

10.1. Departure Times

Our dispatchers support departure times at a resolution of one second. At this resolution, the distribution of travel times in some of our input demand sets is skewed. For example, when we divide the Los Angeles demand from Section 6.3 into intervals of 15 minutes, more than half of all travel requests in an average interval occur in the first second of the interval. Presumably, this is an artifact of the transport simulations and the underlying activity plans, which are not meant to be used second-by-second.

Therefore, we generate a set of new request times that prevents these unnatural spikes but adheres to the original distribution at a resolution of 15-minute intervals as follows. First, we divide the observation period into 15-minute intervals, shifting the start of each interval by 7.5 minutes so that each spike is in the middle of an interval. For each interval, we count the number of requests. Let n_i be the number of requests in interval i. The mean number of requests per second in batch i is $m_i = n_i/900$. We assign the mean as a weight to the middle second $900i$ of each batch i and interpolate linearly between these values. More precisely, for second $x = 900i + c$ with $0 \leq c < 900$, we assign weight $w(x) = m_i + \frac{m_{i+1}-m_i}{900} \cdot c$. The sum of weights $\sum_x w(x)$ is equal to the number of requests. Thus, we could simply assign $w(x)$ requests to second x. However, this would be an unrealistically smooth distribution. Therefore, we randomly sample new request times from a cumulative distribution for every second x of the observation period with probabilities $p(x) = x/\sum_{x'} w(x')$. We sort the sampled request times and assign them to the requests in the original order. While the expected number of requests per second x is still $w(x)$, this randomization adds some nuance to the distribution. This is particularly relevant for the parallel Mt-KaRRi dispatcher (see Chapter 5) as it processes requests in batches representing a fixed time interval. Without the randomization, subsequent batches would always contain very similar numbers of requests.

10.2. Generating Road Networks

In this section, we explain how we construct the vehicle road network $G = (V, E)$ and the pedestrian road network $G_{\text{psg}} = (V_{\text{psg}}, E_{\text{psg}})$ of a ride-pooling service area in practice. We also describe how we map the locations of each travel request onto the generated networks.

10.2.1. Generating Vehicle and Pedestrian Networks

OpenStreetMap (OSM) (`https://openstreetmap.org`) provides detailed map data for many densely populated regions of the world, particularly in Western industrialized countries, including Germany and the United States. We base our road networks on publicly accessible OSM data obtained from `https://download.geofabrik.de`. Next to regular roads, the

OSM data includes footpaths, cycle paths, bridleways, etc., which we can use to determine a network of paths that are accessible to pedestrians. For this, we rely on a classification of roads and paths into different categories used by OSM.

Vehicle-Accessible and Pedestrian-Accessible Roads and Paths. Every recorded road and path is called a *highway* in OSM. Every highway is attributed with a type that gives a high-level classification of its properties such as `motorway` or `residential` but also `footway` or even `steps`. We ignore highways of more exotic types like `escape` (off-ramps for runaway trucks in extended downhill stretches of mountain roads) or `via_ferrata` (a mountaineering route with cables, stemples, ladders, etc) . The common classifications provide a reasonable indication of vehicle and pedestrian accessibility. For a vehicle network G, we include any OSM highways (links) of the following types [`motorway`, `trunk`, `primary`, `secondary`, `tertiary`, `unclassified` (this type does *not* indicate missing information), `residential`, and `living_street`]. We exclude `service`, which describes access roads that a transit vehicle should not enter. For a pedestrian network G_{psg}, we include any OSM highways of the following types [`tertiary`, `unclassified`, `residential`, `living_street`, `service`, `pedestrian`, `track`, `footway`, `bridleway`, `steps`, `path`, and `cycleway`]. We include `bridleway` and `cycleway` because multi-use paths for horses/bicycles and pedestrians are often tagged as such. Links of the types: [`tertiary`, `living_street`] are present in both G and G_{psg}, which can be used as meeting points for vehicles and passengers. We tried to restrict this further to only roads that are tagged as having sidewalks, but we found this information to be incomplete in a lot of places.

During the extraction of the road networks from OSM data, we store a mapping between the corresponding edges in both networks. Then, any edge with a valid such mapping represents a road segment that is accessible to both motor vehicles and pedestrians.

Inner and Outer Network. We partition the geographical area for each ride-pooling input instance into an *inner* and *outer* area. The inner network represents the actual service area. The origin and destination location of each travel request as well as the initial location of each vehicle must lie within this area. During service, the shortest path between two vehicle stops may contain roads outside of the service area. For example, a motorway ring outside of the core city may represent the fastest way to get from one side of town to the other. Therefore, we include an outer area which extends beyond the service area.

Our vehicle network includes all vehicle-accessible OSM highways (see above) within the inner area. For the outer area, the vehicle network only includes important roads of type `tertiary` or higher, as less important roads in the outer area are not needed for shortest paths between locations in the inner area. The pedestrian network includes all pedestrian-accessible OSM highways in the inner area and no highways in the outer area.

Edge Weights (Travel Times and Distances). Our algorithm needs to know a travel time $\ell(e)$ for every edge $e \in E$. We obtain these travel times based on the length of the road segment and the speed limit encoded in the OSM map data. If no speed limit is specified in OSM, we use reasonable default values based on the highway type. For the pedestrian graph G_{psg}, we encode the length of each path segment instead of the travel time since each rider can specify their own walking speed.

10.2.2. Mapping Rider Requests onto Generated Networks

Our input request sets are based on raw demand data (usually from transport simulations) that do not use the same representation of the road network of a respective area as the networks generated by us. The MATSim scenarios Berlin, Ruhr (see Section 4.9), and Los Angeles (see Section 6.3) use OSM data in the background but do not explicitly distinguish between vehicle and pedestrian networks. The Stuttgart and Karlsruhe demands (see Section 6.3) are based on proprietary networks.

Thus, for every demand set, we need to map the O-D pairs from the source network into the networks in our representation. As a preprocessing step, we perform the following mapping once for every demand and write the request set for the output network to disk. We call the network on which a demand is based the input network and we call our representation the output network. To map the O-D pairs, we use the coordinate information stored for each vertex in the input and output networks. We construct a k-d tree [Ben75], a nearest-neighbor index that contains the coordinates of all vertices in the output network as points in two-dimensional Euclidean space. Given a traveler's origin or destination location v in the input network and the associated coordinates, we query the k-d tree to find the closest vertex to v in the output network. Note that we interpret WGS-84 coordinates as points in Euclidean space to be able to use a k-d tree. This may skew the results for large networks but we deem this effect tolerable for metropolitan-size road networks that are not too close to the earth's poles. Replacing the k-d tree with a data structure suited for geodesic distances, e.g., a vantage-point tree [Yia93], would alleviate this issue entirely.

We want to ensure that the origin and destination locations in the output network are accessible both by pedestrians and motor vehicles to guarantee that a rider's origin and destination are valid pickup/dropoff locations, and that every traveler can potentially walk and reach other meeting points. We restrict the k-d tree to only contain vertices that are incident to at least one edge that is accessible in the vehicle and the pedestrian road network. Note that this may cause some skew in results depending on the request sets and the OSM data on which the output network is built.

10.3. Tuning of KaRRi Model Parameters

In this appendix, we analyze the effect of the model parameters of Mt-KaRRi on the solution quality. We consider the metrics outlined in Section 6.3.3. We run experiments where one parameter is varied, while all other parameters are fixed at their default value, which is specified at the end of the respective section. We perform these experiments for the `KA` and `LA-10%` instances (see Section 6.3.1) with a fleet of 25 000 vehicles.

10.3.1. Detour Cost Parameter

The vehicle cost parameter w_{detour} determines the impact of the vehicle detour on the cost of an insertion.

At $w_{\text{detour}} = 0$, the cost of an insertion ignores the vehicle detour. Thus, ride detours are small but routes become inefficient with little pooling and bad system effectiveness. For `KA`, the modal share of ride-pooling decreases with increasing w_{detour} as ride detours and wait times become longer. For `LA-10%`, ride detours also grow with larger w_{detour} but the wait time initially falls with larger w_{detour}, so that the maximum modal share is reached at $w_{\text{detour}} = 1$. The decrease in wait times is caused by detours contributing to the wait time of currently waiting riders, so fewer and more efficient detours at larger w_{detour} lead

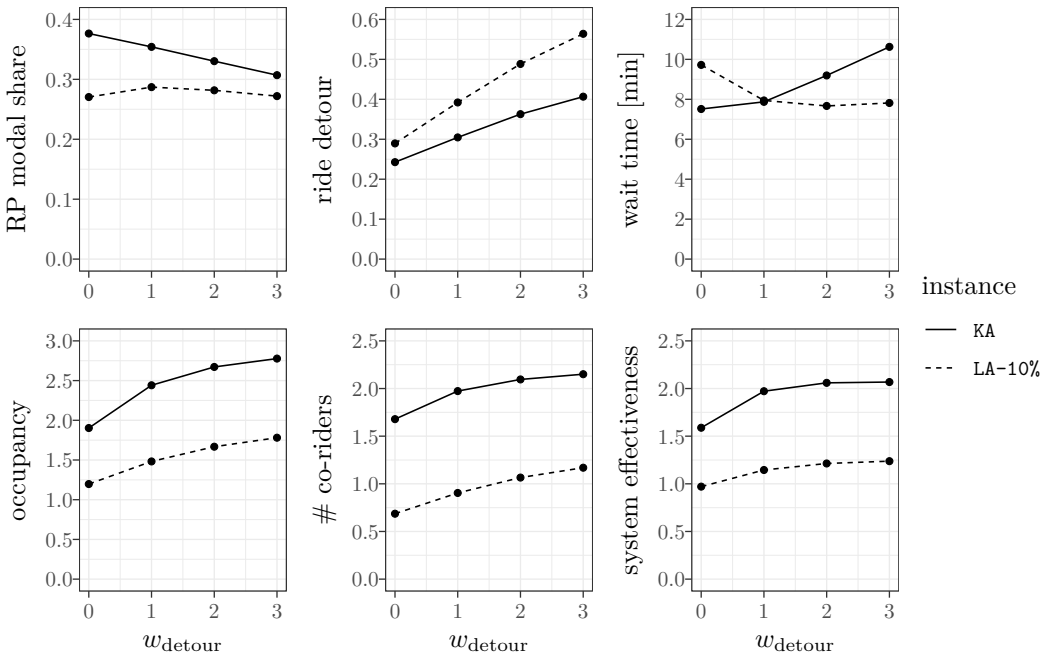


Figure 10.1 Shows RP modal share, ride detour, wait time, occupancy, average number of co-riders, and system effectiveness for varying detour cost parameters $w_{\text{detour}} \in \{0, 1, 2, 3\}$ on the `KA` and `LA-10%` instances with 25 000 vehicles. For an explanation of the metrics, see Section 6.3.3. Note the different y-axis scales.

to smaller average wait times. We do not see this effect in the `KA` instance, as the level of pooling is much higher than for `LA-10%`, and the longer wait times associated with shared rides compensate for the advantage of fewer and more efficient detours.

Increasing w_{detour} to values greater than one improves system effectiveness only slightly, but further decreases the attractiveness of ride-pooling. Thus, we use $w_{\text{detour}} = 1$ in our experiments to strike a balance between service quality and vehicle resource usage.

10.3.2. Trip Time Constraint Parameters

The trip time constraint parameters α and β define a threshold $t^{\max}_{\text{trip}}(r) = \alpha \cdot T_t + \beta$ for the total trip time of a traveler r relative to the tentative trip time T_t of the ride that rider r accepted. If an insertion causes the trip time of any existing rider r to exceed $t^{\max}_{\text{trip}}(r)$, the insertion is considered infeasible. Thus, the trip time constraint limits the detour that is allowed for existing riders.

We show quality results for varying α and β in Figure 10.2. Larger values of α and β define a less restrictive trip time constraint and allow more insertions to vehicles with existing passengers. This is reflected in an increased modal share of ride-pooling and slightly longer ride detours, as well as shorter wait times due to a larger supply of possible rides in the vicinity. The level of pooling increases with a greater average occupancy and number of co-riders since detours for shared rides are more likely to be feasible. The increase in pooling causes more efficient use of vehicles and an improved system effectiveness.

With sufficiently large α and/or β, each metric approaches an asymptote. We choose to use $\alpha = 1.4$ and $\beta = 10\text{min}$, which is close to this asymptote, to prioritize system-wide quality expressed by a large modal share and good system effectiveness. Note that this differs

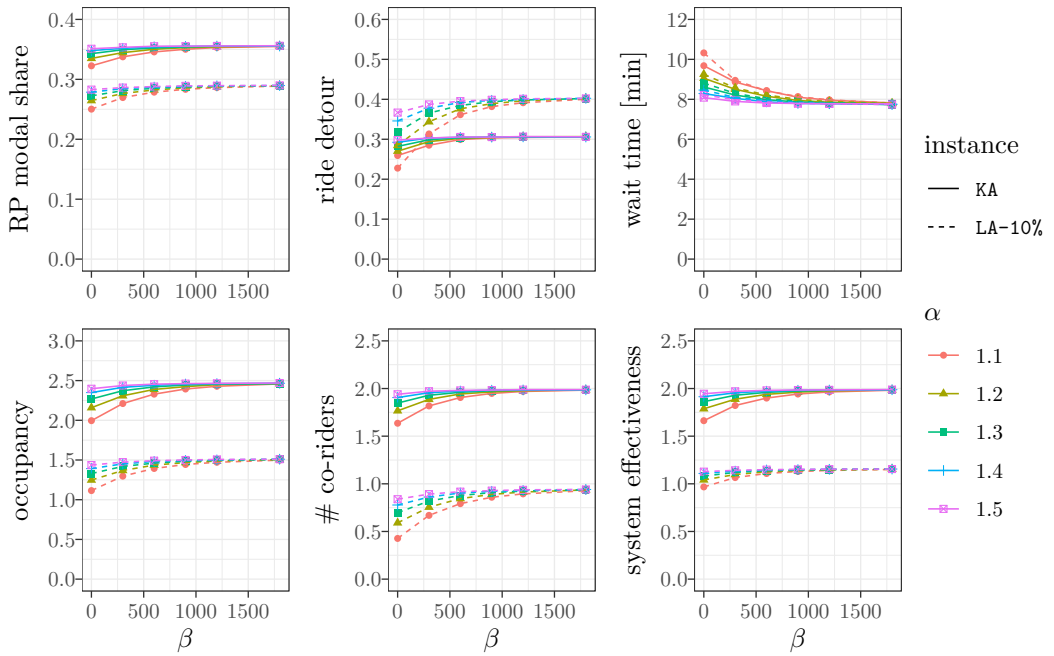


Figure 10.2 Shows RP modal share, ride detour, wait time, occupancy, average number of co-riders, and system effectiveness for varying trip time constraint parameters $\alpha \in \{1.1, 1.2, 1.3, 1.4, 1.5\}$ and $\beta \in \{0, 5, 10, 15, 20\}$ (in minutes) on the `KA` and `LA-10%` instances with 25 000 vehicles. For an explanation of the metrics, see Section 6.3.3. Note the different y-axis scales.

from not using a trip time constraint at all since the constraint maintains a guaranteed service quality for every individual rider.

10.3.3. Maximum Waiting Time Parameter

The maximum waiting time $t_{\text{wait}}^{\max}$ comprises a threshold for the waiting time of a rider, i.e., the time between issuing the request and being picked up by a vehicle. Assume that an existing rider r accepted a ride with a tentative pickup time of T_p. If an insertion causes the pickup time of r to exceed $T_p + t_{\text{wait}}^{\max}$, the insertion is considered infeasible.

We show the effect of $t_{\text{wait}}^{\max}$ on the solution quality of Mt-KaRRi in Figure 10.3. The maximum wait time parameter has only a small effect on every metric. This is likely due to the wait time constraint only having an effect in the rare case that an insertion causes a detour on the way to picking up an already assigned rider. A value of $t_{\text{wait}}^{\max} = 0$ does not allow for this case at all. This makes too many insertions infeasible, and we actually see longer wait times and a slightly smaller modal share than for larger values of $t_{\text{wait}}^{\max}$.

We choose $t_{\text{wait}}^{\max} = 10\text{min}$ to maximize modal share.

10.3.4. Walking Cost Parameter

The walking cost parameter w_{walk} determines the impact of walking on the cost of an insertion. The walking time includes the time the rider spends walking from their origin to the pickup location of the insertion and from the dropoff location of the insertion to their destination. With greater values of w_{walk}, a short walking time becomes more important.

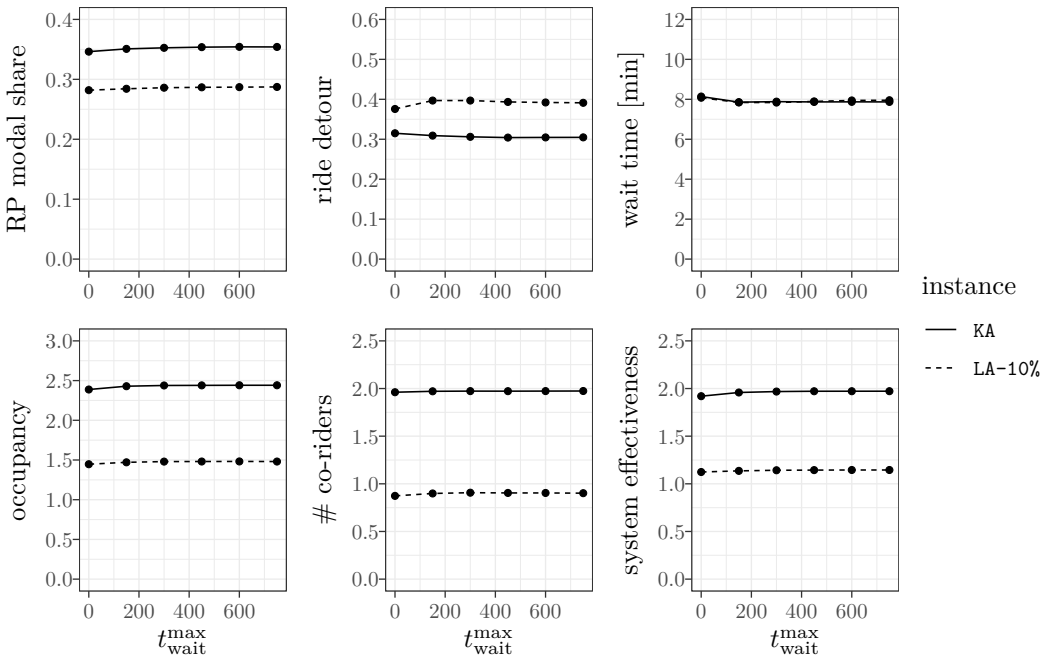


Figure 10.3 Shows RP modal share, ride detour, wait time, occupancy, average number of co-riders, and system effectiveness for varying maximum wait time $t_{\text{wait}}^{\max} \in \{0, 2.5, 5, 7.5, 10, 12.5\}$ (in minutes) on the `KA` and `LA-10%` instances with 25 000 vehicles. For an explanation of the metrics, see Section 6.3.3. Note the different y-axis scales.

The plots in Figure 10.4 show how different quality metrics are affected by the walking cost parameter w_{walk}. The plot only considers the `KA` instance, since we do not allow meeting points for `LA-10%`.

The best RP modal share and average occupancy are achieved at $w_{\text{walk}} = 1$. This comes at a cost of increased ride detours and wait times, and slightly worse system effectiveness compared to $w_{\text{walk}} = 0$. All metrics become worse for $w_{\text{walk}} > 1$. Note that the wait time does not include the walking time to the pickup location.

It is important to also consider the effect on the actual walking times. The left plot in Figure 10.5 shows the average absolute walking times of riders for varying w_{walk}. The walking time falls with increasing w_{walk}. At $w_{\text{walk}} = 0$, each rider walks for 121s on average, which drops to around 78s at $w_{\text{walk}} = 1$.

The y-axis of the right plot in Figure 10.5 shows the median of the ratios between the walking time and the shortest-path travel time by car. The plot shows that at $w_{\text{walk}} = 0$, in the median, the walking time of a rider is almost 16% of the entire trip from origin to destination by car. This ratio decreases to less than 10% for $w_{\text{walk}} = 1$.

To ensure the maximum RP modal share and limited walking times, we choose to use $w_{\text{walk}} = 1$ in our experiments.

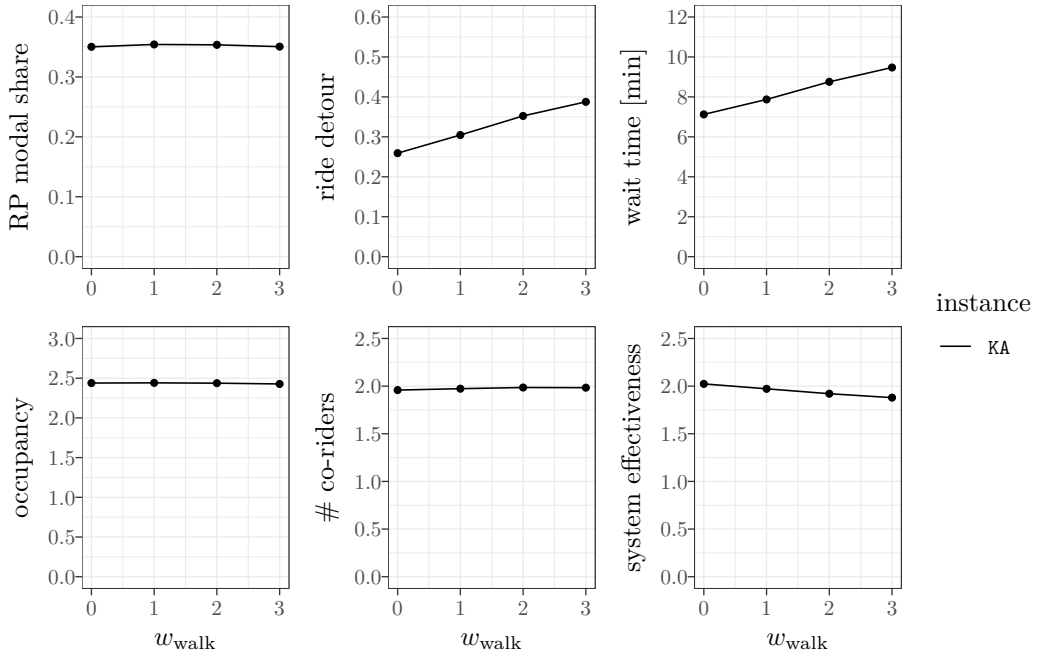


Figure 10.4 Shows RP modal share, ride detour, wait time, occupancy, average number of co-riders, and system effectiveness for varying walking cost parameters $w_{\text{walk}} \in \{0, 1, 2, 3\}$ on the KA instance with 25 000 vehicles. To determine meeting points, we use the rider-specific walking speed and maximum walking distance given by the KA instance. For an explanation of the metrics, see Section 6.3.3. Note the different y-axis scales.

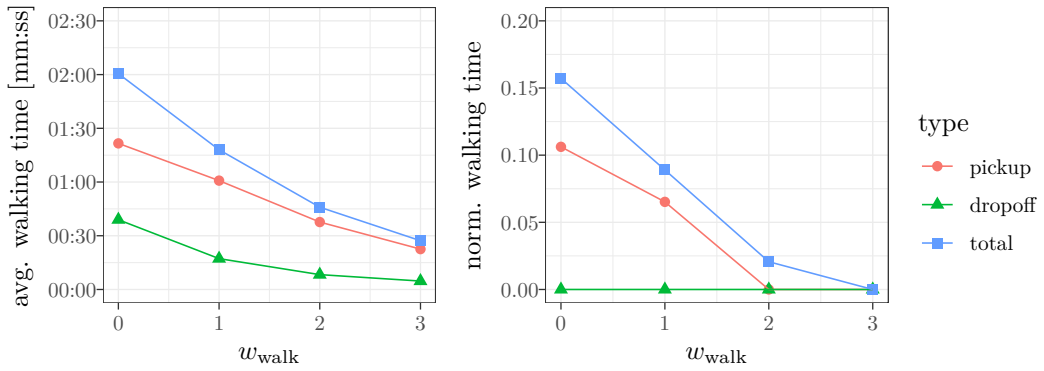


Figure 10.5 Shows absolute walking times (left) and normalized walking times (right) for varying walking cost parameters $w_{\text{walk}} \in \{0, 1, 2, 3\}$ on the KA instance with 25 000 vehicles. The normalized walking time on the right is defined as the median of the ratios between the walking time and the shortest-path travel time from origin to destination by car for every rider.

Acronyms

ALS after-last-stop. 83–86, 89, 90

BCH bucket contraction hierarchy. 16, 17, 20, 21, 25–28, 32–37, 40, 41, 43–48, 51, 85, 103, 104, 107, 109–112, 116, 117, 120, 140

CH contraction hierarchy. 15–17, 26, 29, 33–38, 41, 48, 49, 85, 87, 89, 90, 106, 107

DALS dropoff-after-last-stop. 28, 40, 41, 44–48, 51, 111, 112, 119

KaRRi Karlsruhe Rapid Ride-Pooling. v–viii, xii, xiv, 2–4, 21, 23, 25, 27–29, 31, 33–35, 37, 39, 41, 43, 45, 47–52, 54–60, 63, 79–81, 83, 84, 91, 93, 94, 101, 106, 107, 109–112, 114, 117, 119, 124–127, 131, 133, 134, 136, 141, 143
KaRRiT Karlsruhe Rapid Ride-Pooling with Transfers. vi, viii, 3, 79, 88–92, 125, 126, 140, 141, 143

LOUD Local Buckets Dispatching. 26, 27, 34, 47, 63

Mt-KaRRi Multi-Threaded Karlsruhe Rapid Ride-Pooling. v, vii, viii, 2, 4, 52, 54–61, 63, 68, 76, 77, 125, 126, 131, 133, 135, 141, 143

OP ordinary paired. 28, 31–33
Ord ordinary. 32, 33, 48, 51

PALS pickup-after-last-stop. xi, xii, 28, 34–40, 44–48, 51, 111, 119
PaRRot Public Transit Ride-Pooling Router. vi, viii, 4, 93, 94, 106, 109, 110, 112–121, 123–126, 142, 143
PBNS pickup-before-next-stop. 28, 32, 33, 48, 51, 119
PHAST PHAST hardware-accelerated shortest path trees. 16, 17, 84, 87, 89, 90, 92
PT public transportation. 93–104, 106–109, 112–124

RP ride-pooling. 5, 12, 56, 67–69, 71, 75, 76, 93, 96–98, 100, 101, 103–109, 112–124, 143
RPHAST restricted PHAST hardware-accelerated shortest path trees. 17, 85, 89, 90

List of Algorithms

4. Dynamic Ride-Pooling with Meeting Points
4.1. Collective BCH for Pickup-After-Last-Stop 37

7. Dynamic Ride-Pooling with Transfers
7.1. KaRRiT Algorithm Outline 84

List of Figures

3. Preliminaries
3.1. Example CH 15

4. Dynamic Ride-Pooling with Meeting Points
4.1. Workflow of KaRRi 27
4.2. Insertion types 28
4.3. Illustration of domination pruning. 38
4.4. Distribution of trips over time for Berlin and Ruhr instances 42
4.5. Speedups with bundling for elliptic BCH searches and PD-distance searches . 43
4.6. Speedups with bundling for last stop searches 44
4.7. Speedups with sorted buckets for elliptic BCH searches 45
4.8. Speedups with sorted buckets for last stop searches 45

5. Parallel Batched Dynamic Ride-Pooling
5.1. Workflow of Mt-KaRRi. 54
5.2. Effect of batch length on solution quality, relative to non-batched KaRRi . . 56
5.3. Effect of batch length on speedup over sequential KaRRi 57
5.4. Speedups of Mt-KaRRi over the non-batched sequential version 59
5.5. Composition of running times by Mt-KaRRi component for different numbers of threads 60

6. Analysis of Ride-Pooling in Large-Scale Transportation Systems with Multiple Modes of Transportation
6.1. Distribution of trips over a whole day in our large-scale demand sets. 65
6.2. RegioStaR-2 typology of Germany, Stuttgart and Karlsruhe. 67
6.3. Effect of customer base on ridership. 69
6.4. Effect of customer base on pooling. 69
6.5. Effect of customer base on trip quality and vehicle resource usage. 70
6.6. Effect of fleet size on quality. 72
6.7. Effect of fleet size on quality. 73
6.8. Vehicle occupancy over time. 74
6.9. Modal split classified by urban/rural origin and destination. 75
6.10. Distribution of wait times and ride detours by urban/rural origin and destination 77

7. Dynamic Ride-Pooling with Transfers
7.1. Illustration of no-transfer and transfer insertion for same request 81
7.2. Illustration of detour ellipses and transfers 83
7.3. Running times per component of KaRRiT 89

8. Multimodal Routing for Ride-Pooling and Public Transportation
8.1. Illustration of how ride-pooling increases the number of access stations 103
8.2. Illustration on optimization criteria for access trips 105
8.3. Illustration on need for modified cost criterion 105

8.4. Impact of fleet size on running times of PaRRot. 116
8.5. Impact of fleet size on running times of search for combined journeys in PaRRot. 117
8.6. Impact of meeting points on running times of PaRRot. 119
8.7. Change of modal split when introducing combined journeys. 122

10. Technical Appendix

10.1. Solution quality for varying detour cost parameters. 134
10.2. Solution quality for varying trip time constraint parameters. 135
10.3. Solution quality for varying maximum wait time. 136
10.4. Solution quality for varying walking cost parameters. 137
10.5. Walking times for varying walking cost parameters. 137

List of Tables

4. Dynamic Ride-Pooling with Meeting Points
4.1. Key figures of benchmark instances 42
4.2. Comparison of running times of different last stop search approaches 47
4.3. Comparison of solution quality of KaRRi and baseline 49
4.4. Comparison of running times of KaRRi and baseline 51

5. Parallel Batched Dynamic Ride-Pooling
5.1. Weak scaling of Mt-KaRRi 58
5.2. Strong scaling of Mt-KaRRi 59

6. Analysis of Ride-Pooling in Large-Scale Transportation Systems with Multiple Modes of Transportation
6.1. Overview on demand sets. 65
6.2. Overview on road networks. 66
6.3. Effect of meeting points on quality of trips and usage of vehicle resources. . . 71
6.4. Quality of trips classified by urban/rural origin and destination. 76

7. Dynamic Ride-Pooling with Transfers
7.1. Comparison of running times of KaRRiT and baseline 90
7.2. Comparison of quality of KaRRiT and baseline 91

8. Multimodal Routing for Ride-Pooling and Public Transportation
8.1. Statistics on tuning of cost heuristic for egress RP trips. 114
8.2. Impact of detour weight w_{detour} on quality of PaRRot. 115
8.3. Impact of fleet size on quality of PaRRot. 116
8.4. Impact of meeting points quality of PaRRot. 118
8.5. Number of insertions for different meeting point radii in PaRRot. 119
8.6. Comparison of running time of PaRRot with unimodal baseline. 120
8.7. Comparison of quality of PaRRot with unimodal baseline. 121
8.8. Comparison of trip times of PaRRot with unimodal baseline. 123

Publications and Supervised Theses

In Conference Proceedings

- J. Breitling and M. Laupichler. "Exact and Heuristic Dynamic Taxi Sharing with Transfers using Shortest-Path Speedup Techniques". In: *Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*. edited by J. Sauer and M. Schmidt. Schloss Dagstuhl, 2025, 15:1–15:22. DOI: 10.4230/OASIcs.ATMOS.2025.15
- M. Laupichler and P. Sanders. "Fast Many-to-Many Routing for Dynamic Taxi Sharing with Meeting Points". In: *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2024, pages 74–90. DOI: 10.1137/1.9781611977929.6
- *Not relevant for this dissertation:* T. Bläsius, A. Feilhauer, M. Jung, M. Laupichler, P. Sanders, and M. Zündorf. "Synergistic Traffic Assignment". In: *Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025, Detroit, MI, USA, May 19-23, 2025*. Edited by S. Das, A. Nowé, and Y. Vorobeychik. International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2025, pages 352–360. DOI: 10.5555/3709347.3743549
- *Not relevant for this dissertation:* M. Farhan, H. Koehler, Q. Wang, J. Wang, M. Laupichler, and P. Sanders. "Customization Meets 2-Hop Labeling: Efficient Routing in Road Networks". In: *Proceedings of the VLDB Endowment* 18.10 (2025), pages 3326–3338. DOI: 10.14778/3748191.3748198

Technical Reports

- M. Laupichler, R. Andre, K. Kandler, P. Sanders, and P. Vortisch. *Advancing Dynamic Ride-Pooling Simulation – A Highly Scalable Dispatcher*. Technical report. Karlsruhe Institute of Technology, May 2026. DOI: 10.48550/arXiv.2605.11798

Theses

- M. Laupichler. "Asynchronous n-Level Hypergraph Partitioning". Master Thesis. Karlsruhe Institute of Technology, 2021
- M. Laupichler. "Alias Analysis using Abstract Interpretation". Bachelor Thesis. Karlsruhe Institute of Technology, 2019

Supervised Theses

- L. Buchholz. “Vehicle Path Planning in Constrained Environments”. Master Thesis. Karlsruhe Institute of Technology, 2026
- H. Csöre. “Fast Dynamic Traffic Assignment using Engineered Shortest-Path Speedup Techniques”. Master Thesis. Karlsruhe Institute of Technology, 2026
- J. Breitling. “Single Transfer Journeys in Dynamic Taxi Sharing”. Bachelor Thesis. Karlsruhe Institute of Technology, 2025
- A. Kuchenbecker. “Optimizing Meeting Points in Dynamic Taxi Sharing”. Bachelor Thesis. Karlsruhe Institute of Technology, 2024
- J. Körte. “Intermodal Tour Planning for Delivery and Pickup Activities for CEP Service Providers”. Bachelor Thesis. Karlsruhe Institute of Technology, 2024
- T. L. Gladbach. “Reoptimization in Dynamic Taxi Sharing with Meeting Points”. Bachelor Thesis. Karlsruhe Institute of Technology, 2024
- H. L. Nguyen. “Multi-Threaded Dynamic Taxi Sharing with Meeting Points”. Bachelor Thesis. Karlsruhe Institute of Technology, 2024
- K. Frisen. “Heuristische Optimierung der globalen Zielfunktion im dynamischen Ridesharing”. Bachelor Thesis. Karlsruhe Institute of Technology, 2023
- M. Jung. “Inverse Traffic Assignment”. Bachelor Thesis. Karlsruhe Institute of Technology, 2023
- M. Willich. “Improving Vehicle Detour in Dynamic Ridesharing using Transfer Stops”. Master Thesis. Karlsruhe Institute of Technology, 2023
- L. J. Morlok. “Dynamic Ridesharing with Relaxed Optimality”. Bachelor Thesis. Karlsruhe Institute of Technology, 2023
- Z. Cai. “Exploring Inverse Traffic Assignment”. Bachelor Thesis. Karlsruhe Institute of Technology, 2022
- L. Tiede. “Integrating Ridesharing and Public Transit Routing”. Bachelor Thesis. Karlsruhe Institute of Technology, 2022

Bibliography

[Abr+11] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. "A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Volume 6630 LNCS. Springer, Berlin, Heidelberg, 2011, pages 230–241. DOI: 10.1007/978-3-642-20662-7_20 (cited on page 22).

[Aga+11] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang. "Dynamic Ride-Sharing: A Simulation Study in Metro Atlanta". In: *Transportation Research Part B: Methodological* 45 (9 2011), pages 1450–1464. ISSN: 01912615. DOI: 10.1016/j.trb.2011.05.017 (cited on pages 6, 13, 20, 79).

[Aga+12] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang. "Optimization for Dynamic Ride-Sharing: A Review". In: *European Journal of Operational Research* 223 (2 2012), pages 295–303. ISSN: 03772217. DOI: 10.1016/j.ejor.2012.05.028 (cited on pages 8, 13).

[Aïv+16] U. M. Aïvodji, S. Gambs, M.-J. Huguet, and M.-O. Killijian. "Meeting points in ridesharing: A privacy-preserving approach". In: *Transportation Research Part C: Emerging Technologies* 72 (Nov. 2016), pages 239–253. ISSN: 0968090X. DOI: 10.1016/j.trc.2016.09.017 (cited on page 24).

[Alo+17] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus. "On-Demand High-Capacity Ride-Sharing via Dynamic Trip-Vehicle Assignment". In: *Proceedings of the National Academy of Sciences of the United States of America* 114 (3 2017), pages 462–467. ISSN: 10916490. DOI: 10.1073/pnas.1611675114 (cited on pages 1, 9, 11, 12, 22, 53, 62, 63, 123).

[AO14] K. Aissat and A. Oulamara. "A Priori Approach Of Real-Time Ridesharing Problem With Intermediate Meeting Locations". In: *Journal of Artificial Intelligence and Soft Computing Research* 4 (4 2014), pages 287–299. ISSN: 2083-2567. DOI: 10.1515/jaiscr-2015-0015 (cited on pages 21, 24).

[Att+04] A. Attanasio, J.-F. Cordeau, G. Ghiani, and G. Laporte. "Parallel Tabu search heuristics for the dynamic multi-vehicle dial-a-ride problem". In: *Parallel Computing* 30 (3 Mar. 2004), pages 377–387. ISSN: 01678191. DOI: 10.1016/j.parco.2003.12.001 (cited on pages 11, 53).

[Bad+21] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus, R. Berriel, T. M. Paixão, F. Mutz, L. de Paula Veronese, T. Oliveira-Santos, and A. F. D. Souza. "Self-Driving Cars: A Survey". In: *Expert Systems with Applications* 165 (2021). ISSN: 09574174. DOI: 10.1016/j.eswa.2020.113816 (cited on page 20).

[Bas+07] H. Bast, S. Funke, P. Sanders, and D. Schultes. "Fast Routing in Road Networks with Transit Nodes". In: *Science* 316 (5824 2007), pages 566–566. ISSN: 0036-8075. DOI: 10.1126/science.1137521 (cited on pages 14, 23).

[Bas+10] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger. "Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns". In: *European Symposium on Algorithms*. Edited by M. Berg and U. Meyer. Springer, Sept. 2010, pages 290–301. DOI: 10.1007/978-3-642-15775-2_25. URL: http://link.springer.com/10.1007/978-3-642-15775-2_25 (cited on page 95).

[Bas+16] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. "Route planning in transportation networks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9220 LNCS (2016), pages 19–80. ISSN: 16113349. DOI: 10.1007/978-3-319-49487-6_2 (cited on pages 14, 95).

[Bau+10] R. Bauer, T. Columbus, B. Katz, M. Krug, and D. Wagner. "Preprocessing Speed-Up Techniques Is Hard". In: *Algorithms and Complexity, 7th International Conference, CIAC 2010, Rome, Italy, May 26-28, 2010. Proceedings*. Edited by T. Calamoneri and J. Díaz. Lecture Notes in Computer Science. Springer, 2010, pages 359–370. DOI: 10.1007/978-3-642-13073-1_32 (cited on page 15).

[Bau+23] M. Baum, V. Buchhold, J. Sauer, D. Wagner, and T. Zündorf. "ULTRA: Unlimited Transfers for Efficient Multimodal Journey Planning". In: *Transp. Sci.* 57.6 (2023), pages 1536–1559. DOI: 10.1287/TRSC.2022.0198 (cited on pages 25, 64, 94, 96, 102).

[BD10] R. Bauer and D. Delling. "SHARC: Fast and Robust Unidirectional Routing". In: *ACM Journal of Experimental Algorithms (JEA)* 14 (2010). ISSN: 1084-6654. DOI: 10.1145/1498698.1537599 (cited on page 26).

[Bel+25] J. Belz, C. Böhnen, T. Brand, I. Dubernet, R. Follmer, A. Galich, D. Gruschwitz, N. Höing, K. Jäkel, K. Jearwattanakanok, K. Köhler, T. Kuhnimhof, L. van Nek, C. Nobis, M. Plener, M. Porschen, M. Ruppenthal, M. Schelewsky, J. Schuppan, M. Schürig, I. Seitz, and B. Wawrzyniak. *Mobilität in Deutschland – MiD Ergebnisbericht*. Technical report. commissioned by the Federal Ministry for Digital and Transport. infas Institute for Applied Social Science, German Aerospace Center, and IVT Research GmbH, Nov. 2025. DOI: 10.71786/dlr.vf-9hvj-sh11. URL: https://elib.dlr.de/221542/ (cited on page 1).

[Ben75] J. L. Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (Sept. 1975), pages 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007 (cited on page 133).

[BFR15] F. Bistaffa, A. Farinelli, and S. Ramchurn. "Sharing Rides with Friends: A Coalition Formation Algorithm for Ridesharing". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 29 (1 2015). ISSN: 2374-3468. DOI: 10.1609/aaai.v29i1.9242 (cited on page 13).

[BHS16] H. Bast, M. Hertel, and S. Storandt. "Scalable Transfer Patterns". In: *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Jan. 2016, pages 15–29. ISBN: 978-1-61197-431-7. DOI: 10.1137/1.9781611974317.2 (cited on page 95).

[BJ04] G. S. Brodal and R. Jacob. “Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries”. In: *Electronic Notes in Theoretical Computer Science* 92 (Feb. 2004), pages 3–15. ISSN: 15710661. DOI: 10.1016/j.entcs.2003.12.019 (cited on page 95).

[BL25] J. Breitling and M. Laupichler. “Exact and Heuristic Dynamic Taxi Sharing with Transfers using Shortest-Path Speedup Techniques”. In: *Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*. Edited by J. Sauer and M. Schmidt. Schloss Dagstuhl, 2025, 15:1–15:22. DOI: 10.4230/OASIcs.ATMOS.2025.15 (cited on pages 2, 14, 79, 144).

[BM16] J. Bischoff and M. Maciejewski. “Simulation of City-wide Replacement of Private Cars with Autonomous Taxis in Berlin”. In: *Procedia Computer Science* 83 (Jan. 2016), pages 237–244. ISSN: 1877-0509. DOI: 10.1016/J.PROCS.2016.04.121 (cited on page 12).

[BMB13] A. Bar-Yosef, K. Martens, and I. Benenson. “A model of the vicious cycle of a bus line”. In: *Transportation Research Part B: Methodological* 54 (Aug. 2013), pages 37–50. ISSN: 0191-2615. DOI: 10.1016/J.TRB.2013.03.010 (cited on page 1).

[BMN17] J. Bischoff, M. Maciejewski, and K. Nagel. “City-Wide Shared Taxis: A Simulation Study in Berlin”. In: *IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017. DOI: 10.1109/ITSC.2017.8317926 (cited on pages 6, 20, 22, 53, 62, 79).

[Boh97] R. W. Bohannon. “Comfortable and maximum walking speed of adults aged 20—79 years: reference values and determinants”. In: *Age and Ageing* 26.1 (1997), pages 15–19 (cited on page 66).

[Bre25a] J. Breitling. “Single Transfer Journeys in Dynamic Taxi Sharing”. Bachelor Thesis. Karlsruhe Institute of Technology, Mar. 2025. DOI: 10.5445/IR/1000182339 (cited on page 79).

[BRN16] K. Braekers, K. Ramaekers, and I. V. Nieuwenhuyse. “The vehicle routing problem: State of the art classification and review”. In: *Computers & Industrial Engineering* 99 (Sept. 2016), pages 300–313. ISSN: 0360-8352. DOI: 10.1016/J.CIE.2015.12.007 (cited on page 7).

[BRT14] T. Bektaş, P. P. Repoussis, and C. D. Tarantilis. “Chapter 11: Dynamic Vehicle Routing Problems”. In: *MOS-SIAM Series on Optimization* (Nov. 2014), pages 299–347. DOI: 10.1137/1.9781611973594.CH11. URL: https://epubs.siam.org/doi/10.1137/1.9781611973594.ch11 (cited on pages 10, 11).

[BSB26] J. Baudru, T. Stützle, and H. Bersini. “Optimization of Carpool Service Problem via Transfer Points Genetic-Based Algorithms”. 2026 (cited on page 24).

[BSM16] J. Bischoff, N. Soeffker, and M. Maciejewski. *A Framework for Agent-Based Simulation of Demand Responsive Transport Systems*. Technical report. Presented at the OR 2016 conference. Technical University of Berlin, 2016. DOI: 10.14279/depositonce-5760 (cited on page 62).

[BSW19] V. Buchhold, P. Sanders, and D. Wagner. “Real-Time Traffic Assignment using Engineered Customizable Contraction Hierarchies”. In: *ACM Journal of Experimental Algorithms (JEA)* 24 (2019). ISSN: 1084-6654. DOI: 10.1145/3362693 (cited on pages 26, 33, 48).

[BSW20] V. Buchhold, P. Sanders, and D. Wagner. *Fast, Exact and Scalable Dynamic Ridesharing (Full Paper)*. Technical report. Karlsruhe Institute of Technology, Nov. 2020. DOI: 10.1137/1.9781611976472.8. URL: https://arxiv.org/abs/2011.02601v2 (cited on pages 62, 63).

[BSW21] V. Buchhold, P. Sanders, and D. Wagner. "Fast, Exact and Scalable Dynamic Ridesharing". In: *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2021, pages 98–112. ISBN: 9781611976472. DOI: 10.1137/1.9781611976472.8 (cited on pages 2, 10, 11, 13, 14, 17, 18, 20–22, 26, 32–35, 41, 42, 47, 53, 62, 88).

[Bun18] Bundesinstitut für Bau-, Stadt- und Raumforschung (BBSR). *Verkehrsbild Deutschland 2018: Die Angebotsqualität im Öffentlichen Verkehr (ÖV)*. Analysen Kompakt 08/2018. Accessed: 2025-11-28. Bonn, Germany: Bundesinstitut für Bau-, Stadt- und Raumforschung (BBSR), 2018. URL: https://www.bbsr.bund.de/BBSR/DE/veroeffentlichungen/analysen-kompakt/2018/ak-08-2018-dl.pdf (cited on page 66).

[Car+23] F. Carrese, S. Sportiello, T. Zhaksylykov, C. Colombaroni, S. Carrese, M. Papaveri, and S. M. Patella. "The Integration of Shared Autonomous Vehicles in Public Transportation Services: A Systematic Review". In: *Sustainability 2023, Vol. 15, Page 13023* 15 (17 Aug. 2023), page 13023. ISSN: 2071-1050. DOI: 10.3390/SU151713023. URL: https://www.mdpi.com/2071-1050/15/17/13023/htm%20https://www.mdpi.com/2071-1050/15/17/13023 (cited on pages 1, 94).

[CC02] Z. Czech and P. Czarnas. "Parallel simulated annealing for the vehicle routing problem with time windows". In: *Proceedings 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*. IEEE Comput. Soc, 2002, pages 376–383. ISBN: 0-7695-1444-8. DOI: 10.1109/EMPDP.2002.994313. URL: http://ieeexplore.ieee.org/document/994313/ (cited on page 53).

[Che+13] M. Chester, S. Pincetl, Z. Elizabeth, W. Eisenstein, and J. Matute. "Infrastructure and automobile shifts: positioning transit to reduce life-cycle environmental impacts for urban sustainability goals". In: *Environmental Research Letters* 8 (1 Mar. 2013). ISSN: 1748-9326. DOI: 10.1088/1748-9326/8/1/015041 (cited on page 94).

[CL03] J. F. Cordeau and G. Laporte. "A tabu search heuristic for the static multi-vehicle dial-a-ride problem". In: *Transportation Research Part B: Methodological* 37 (6 July 2003), pages 579–594. ISSN: 0191-2615. DOI: 10.1016/S0191-2615(02)00045-0 (cited on page 9).

[CL07] J. F. Cordeau and G. Laporte. "The Dial-a-Ride Problem: Models and Algorithms". In: *Annals of Operations Research* 153 (1 2007), pages 29–46. ISSN: 02545330. DOI: 10.1007/s10479-007-0170-8 (cited on page 7).

[Coh+03] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. "Reachability and Distance Queries via 2-Hop Labels". In: *SIAM J. Comput.* 32.5 (2003), pages 1338–1355. DOI: 10.1137/S0097539702403098. URL: https://doi.org/10.1137/S0097539702403098 (cited on page 14).

[Col14] B. Coltin. "Multi-Agent Pickup And Delivery Planning With Transfers". PhD thesis. Carnegie Mellon University, USA, 2014. DOI: 10.1184/R1/6720740.v1 (cited on page 80).

[Cor06] J. F. Cordeau. “A Branch-and-Cut Algorithm for the Dial-a-Ride Problem”. In: *Operations Research* 54 (3 2006), pages 573–586. ISSN: 0030364X. DOI: 10.1287/opre.1060.0283 (cited on page 9).

[Dau+24] A. Dauer, T. G. Dias, J. P. de Sousa, and B. de Athayde Prata. “Configurations and features of demand responsive transports”. In: *Transportation Research Procedia* 78 (Jan. 2024), pages 71–78. ISSN: 2352-1465. DOI: 10.1016/J.TRPRO.2024.02.010 (cited on page 8).

[Del+13] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. “PHAST: Hardware-Accelerated Shortest Path Trees”. In: *Journal of Parallel and Distributed Computing* 73 (7 2013), pages 940–952. ISSN: 07437315. DOI: 10.1016/j.jpdc.2012.02.007 (cited on pages 16, 17, 26, 96, 102).

[Del+17] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. “Customizable Route Planning in Road Networks”. In: *INFORMS Transportation Science* 51 (2 2017). DOI: 10.1287/trsc.2014.0579 (cited on pages 14, 23, 26).

[DGW11] D. Delling, A. V. Goldberg, and R. F. Werneck. “Faster Batched Shortest Paths in Road Networks”. In: *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*. LIPIcs, 2011. ISBN: 978-3-939897-33-0. DOI: 10.4230/OASIcs.ATMOS.2011.52 (cited on pages 17, 26).

[Dib+18] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. “Connection Scan Algorithm”. In: *ACM Journal of Experimental Algorithmics* 23 (Nov. 2018), pages 1–56. ISSN: 1084-6654. DOI: 10.1145/3274661 (cited on page 95).

[Dij59] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *Numerische Mathematik* 1 (1959) (cited on pages 14, 22).

[DMS08] Y. Disser, M. Müller–Hannemann, and M. Schnee. “Multi-criteria Shortest Paths in Time-Dependent Train Networks”. In: *Workshop on Experimental Algorithms*. Springer, 2008, pages 347–361. DOI: 10.1007/978-3-540-68552-4_26 (cited on page 95).

[DPW15a] D. Delling, T. Pajor, and R. F. Werneck. “Round-Based Public Transit Routing”. In: *Transportation Science* 49 (3 Aug. 2015), pages 591–604. ISSN: 0041-1655. DOI: 10.1287/trsc.2014.0534 (cited on pages 95, 101).

[DPW15b] J. Dibbelt, T. Pajor, and D. Wagner. “User-Constrained Multimodal Route Planning”. In: *ACM Journal of Experimental Algorithmics* 19 (Feb. 2015), pages 1–19. ISSN: 1084-6654. DOI: 10.1145/2699886 (cited on page 96).

[DR18] F. Duarte and C. Ratti. “The Impact of Autonomous Vehicles on Cities: A Review”. In: *Journal of Urban Technology* 25 (4 2018), pages 3–18. ISSN: 1063-0732. DOI: 10.1080/10630732.2018.1493883 (cited on pages 6, 20).

[DS14] K. F. Doerner and J.-J. Salazar-González. “Chapter 7: Pickup-and-Delivery Problems for People Transportation”. In: *Vehicle Routing* (Nov. 2014), pages 193–212. DOI: 10.1137/1.9781611973594.CH7. URL: https://epubs.siam.org/terms-privacy (cited on page 7).

[DSW16] J. Dibbelt, B. Strasser, and D. Wagner. “Customizable Contraction Hierarchies”. In: *ACM Journal of Experimental Algorithmics* 21 (2016), pages 1–49. ISSN: 1084-6654. DOI: 10.1145/2886843 (cited on pages 22, 48).

[EDB20] R. Engelhardt, F. Dandl, and K. Bogenberger. *Speed-up Heuristic for an On-Demand Ride-Pooling Algorithm*. Technical report. Technical University of Munich, July 2020. DOI: 10.48550/arXiv.2007.14877. URL: https://arxiv.org/abs/2007.14877v1 (cited on pages 11, 12, 63, 123).

[Eng+22] R. Engelhardt, F. Dandl, A.-A. Syed, Y. Zhang, F. Fehn, F. Wolf, and K. Bogenberger. "FleetPy: A Modular Open-Source Simulation Tool for Mobility On-Demand Services". In: (July 2022). URL: https://arxiv.org/abs/2207.14246v1 (cited on page 63).

[Eur18] Eurostat Task Force on Passenger Mobility. *Passenger Mobility Statistics - Report on Surveys*. Technical report. Accessed: 2026-02-11. Eurostat, 2018. URL: https://circabc.europa.eu/ui/group/097ec987-3401-48d2-8cff-925d158b6eb1/library/25861439-946c-4932-a8af-502d3eee7243/details (cited on page 1).

[EVR09] B. Eksioglu, A. V. Vural, and A. Reisman. "The vehicle routing problem: A taxonomic review". In: *Computers & Industrial Engineering* 57 (4 Nov. 2009), pages 1472–1483. ISSN: 0360-8352. DOI: 10.1016/J.CIE.2009.05.009 (cited on page 7).

[FBA21] A. Fielbaum, X. Bai, and J. Alonso-Mora. "On-demand ridesharing with optimized pick-up and drop-off walking locations". In: *Transportation Research Part C: Emerging Technologies* 126 (2021), page 103061. ISSN: 0968090X. DOI: 10.1016/j.trc.2021.103061 (cited on pages 2, 22, 23).

[Fed18] Federal Ministry of Transport and Digital Infrastructure (BMVI). *RegioStaR – Regional Statistical Area Typology for Mobility and Transport Research*. Technical report. Accessed: 2025-11-28. Berlin, Germany: Federal Ministry of Transport and Digital Infrastructure (BMVI), 2018. URL: https://www.bmdv.bund.de/SharedDocs/DE/Anlage/G/regiostar-raumtypologie-englisch.pdf (cited on page 67).

[FK14] D. J. Fagnant and K. M. Kockelman. "The Travel and Environmental Implications of Shared Autonomous Vehicles, Using Agent-Based Model Scenarios". In: *Transportation Research Part C: Emerging Technologies* 40 (2014), pages 1–13. ISSN: 0968090X. DOI: 10.1016/j.trc.2013.12.001 (cited on pages 6, 20, 79).

[FK18] D. J. Fagnant and K. M. Kockelman. "Dynamic Ride-Sharing and Fleet Sizing for a System of Shared Autonomous Vehicles in Austin, Texas". In: *Transportation* 45 (1 2018), pages 143–158. ISSN: 0049-4488. DOI: 10.1007/s11116-016-9729-z (cited on pages 12, 20).

[Ful+17] L. Fulton, D. J. Mason, D. Meroux, and U. C. Davis. *Three Revolutions in Urban Transportation*. Technical report. Institute for Transportation Development and Policy, 2017 (cited on pages 1, 6).

[Fur+13] M. Furuhata, M. Dessouky, F. Ordóñez, M. E. Brunet, X. Wang, and S. Koenig. "Ridesharing: The State-of-the-Art and Future Directions". In: *Transportation Research Part B: Methodological* 57 (2013), pages 28–46. ISSN: 01912615. DOI: 10.1016/j.trb.2013.08.012 (cited on pages 8, 13).

[Gar+11] T. Garaix, C. Artigues, D. Feillet, and D. Josselin. "Optimization of occupancy rate in dial-a-ride problems via linear fractional column generation". In: *Computers & Operations Research* 38 (10 Oct. 2011), pages 1435–1442. ISSN: 0305-0548. DOI: 10.1016/J.COR.2010.12.014 (cited on page 9).

[Gar+15] E. Gargiulo, R. Giannantonio, E. Guercio, C. Borean, and G. Zenezini. "Dynamic Ride Sharing Service: Are Users Ready to Adopt it?" In: *Procedia Manufacturing* 3 (2015), pages 777–784. ISSN: 23519789. DOI: 10.1016/j.promfg.2015.07.329 (cited on pages 6, 20, 79).

[Gar+25] A. Garus, K. Albano, B. G, and C. B. *On the Energy Intensity of Road Transport in the Presence of Connected and Automated Mobility*. Technical report. Joint Research Council, 2025. DOI: 10.2760/8978379. URL: https://data.europa.eu/doi/10.2760/8978379, (cited on pages 1, 6, 7).

[Gei+10] R. Geisberger, D. Luxen, S. Neubauer, P. Sanders, and L. Volker. "Fast Detour Computation for Ride Sharing". In: *OpenAccess Series in Informatics*. Volume 14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010, pages 88–99. ISBN: 9783939897200. DOI: 10.4230/OASIcs.ATMOS.2010.88 (cited on page 13).

[Gei+12] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. "Exact Routing in Large Road Networks using Contraction Hierarchies". In: *INFORMS Transportation Science* 46 (3 2012). DOI: 10.1287/trsc.1110.0401 (cited on pages 14–16, 21, 22).

[Gid+08] G. Gidofalvi, T. B. Pedersen, T. Risch, and E. Zeitler. "Highly scalable trip grouping for large-scale collective transportation systems". In: *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. ACM, Mar. 2008, pages 678–689. ISBN: 9781595939265. DOI: 10.1145/1353343.1353425 (cited on page 53).

[Gil+20] M. Gilibert, I. Ribas, C. Rosen, and A. Siebeneich. "On-demand Shared Ride-hailing for Commuting Purposes: Comparison of Barcelona and Hannover Case Studies". In: *Transportation Research Procedia*. Volume 47. Elsevier B.V., 2020, pages 323–330. DOI: 10.1016/j.trpro.2020.03.105 (cited on pages 6, 20, 79).

[GJ02] M. R. Garey and D. S. Johnson. *Computers and intractability : A guide to the theory of NP-completeness*. W.H. Freeman, 2002, page 340. ISBN: 0716710455 (cited on page 6).

[GK22] K. M. Gurumurthy and K. M. Kockelman. "Dynamic ride-sharing impacts of greater trip demand and aggregation at stops in shared autonomous vehicle systems". In: *Transportation Research Part A: Policy and Practice* 160 (June 2022), pages 114–125. ISSN: 09658564. DOI: 10.1016/j.tra.2022.03.032 (cited on pages 2, 22).

[GKR16a] P. Goel, L. Kulik, and K. Ramamohanarao. "Optimal Pick up Point Selection for Effective Ride Sharing". In: *IEEE Transactions on Big Data* 3 (2 2016), pages 154–168. DOI: 10.1109/tbdata.2016.2599936 (cited on pages 21, 24).

[GKR16b] P. Goel, L. Kulik, and K. Ramamohanarao. "Privacy-Aware Dynamic Ride Sharing". In: *ACM Transactions on Spatial Algorithms and Systems* 2 (1 2016), pages 1–41. ISSN: 2374-0353. DOI: 10.1145/2845080 (cited on pages 13, 24).

[Göd+23] J. Gödde, L. Ruhrort, V. Allert, and J. Scheiner. "User characteristics and spatial correlates of ride-pooling demand – Evidence from Berlin and Munich". In: *Journal of Transport Geography* 109 (May 2023), page 103596. ISSN: 0966-6923. DOI: 10.1016/J.JTRANGEO.2023.103596 (cited on page 96).

[Gra16] Grab Holdings Inc. *Share a Ride and the Fare with GrabShare, Grab's New Commercial Carpool Service*. https://www.grab.com/sg/press/tech-product/share-ride-fare-grabshare-grabs-new-commercial-carpool-service/. Accessed 2026-05-12. Dec. 2016 (cited on page 6).

[Gro+26] E. Großmann, J. Sauer, C. Schulz, P. Steil, and S. Witt. "FLASH-TB: Integrating Arc-Flags and Trip-Based Public Transit Routing". In: *Transportation*

Science 60 (2 Mar. 2026), pages 242–263. ISSN: 0041-1655. DOI: 10.1287/trsc.2023.0481 (cited on page 95).

[Gut04] R. J. Gutman. "Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks". In: *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics, New Orleans, LA, USA, January 10, 2004*. Edited by L. Arge, G. F. Italiano, and R. Sedgewick. SIAM, 2004, pages 100–111 (cited on page 14).

[GYB21] F. Golbabaei, T. Yigitcanlar, and J. Bunker. "The role of shared autonomous vehicle systems in delivering smart urban mobility: A systematic review of the literature". In: *International Journal of Sustainable Transportation* 15 (10 2021), pages 731–748. ISSN: 15568334. DOI: 10.1080/15568318.2020.1798571 (cited on pages 1, 6, 62).

[Häm13] L. Häme. "Demand-Responsive Transport: Models and Algorithms". PhD thesis. Aalto University, 2013, page 70. ISBN: 978-952-60-5163-5. URL: http://lib.tkk.fi/Diss/2013/isbn9789526051635/isbn9789526051635.pdf (cited on page 8).

[HHL12] C. H. Häll, M. Högberg, and J. T. Lundgren. "A Modeling System for Simulation of Dial-a-Ride Services". In: *Public Transport* 4 (1 2012), pages 17–37. ISSN: 1866-749X. DOI: 10.1007/s12469-012-0052-6 (cited on pages 11, 21).

[Hil+09] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. "Fast Point-to-Point Shortest Path Computations with Arc-Flags". In: *The Shortest Path Problem: 9th DIMACS Implementation Challenge* (2009) (cited on pages 14, 26).

[HNA14] H. Hosni, J. Naoum-Sawaya, and H. Artail. "The Shared-Taxi Problem: Formulation and Solution Methods". In: *Transportation Research Part B: Methodological* 70 (2014), pages 303–318. ISSN: 01912615. DOI: 10.1016/j.trb.2014.09.011 (cited on page 9).

[HNA16] A. Horni, K. Nagel, and K. W. Axhausen, editors. *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press, 2016, page 618. ISBN: 9781909188754. DOI: 10.5334/baw (cited on pages 22, 42, 53, 62, 64, 127).

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Trans. Syst. Sci. Cybern.* 4.2 (1968), pages 100–107. DOI: 10.1109/TSSC.1968.300136 (cited on page 14).

[Ho+18] S. C. Ho, W. Szeto, Y.-H. Kuo, J. M. Leung, M. Petering, and T. W. Tou. "A Survey of Dial-a-Ride Problems: Literature Review and Recent Developments". In: *Transportation Research Part B: Methodological* 111 (2018), pages 395–421. ISSN: 01912615. DOI: 10.1016/j.trb.2018.02.001 (cited on page 7).

[Hor02] M. E. Horn. "Fleet Scheduling and Dispatching for Demand-Responsive Passenger Services". In: *Transportation Research Part C: Emerging Technologies* 10 (1 2002), pages 35–63. ISSN: 0968090X. DOI: 10.1016/S0968-090X(01)00003-1 (cited on pages 11, 21).

[Hou+16] Y. Hou, W. Zhong, L. Su, K. F. Hulme, A. W. Sadek, and C. Qiao. "TASeT: Improving the Efficiency of Electric Taxis With Transfer-Allowed Rideshare". In: *Trans. Veh. Technol.* 65.12 (2016), pages 9518–9528. DOI: 10.1109/TVT.2016.2592983 (cited on page 80).

[HS02] B. Hunsaker and M. Savelsbergh. "Efficient Feasibility Testing for Dial-a-Ride Problems". In: *Operations Research Letters* 30 (3 2002), pages 169–173. ISSN:

01676377. DOI: 10.1016/S0167-6377(02)00120-7 (cited on pages 10, 21, 53).

[Hua+14] Y. Huang, F. Bastani, R. Jin, and X. S. Wang. "Large Scale Realtime Ridesharing with Service Guarantee on Road Networks". In: *Proceedings of the VLDB Endowment*. Volume 7. Association for Computing Machinery, 2014, pages 2017–2028. DOI: 10.14778/2733085.2733106 (cited on pages 10, 22, 53).

[HW12] W. Herbawi and M. Weber. "A Genetic and Insertion Heuristic Algorithm for Solving the Dynamic Ridematching Problem with Time Windows". In: *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation*. 2012, pages 385–392. ISBN: 9781450311779. DOI: 10.1145/2330163.2330219 (cited on pages 8, 13).

[Ibr+20] A. Ibraeva, G. H. de Almeida Correia, C. Silva, and A. P. Antunes. "Transit-oriented development: A review of research achievements and challenges". In: *Transportation Research Part A: Policy and Practice* 132 (Feb. 2020), pages 110–130. ISSN: 0965-8564. DOI: 10.1016/J.TRA.2019.10.018 (cited on page 1).

[ITV14] S. Irnich, P. Toth, and D. Vigo. "Chapter 1: The Family of Vehicle Routing Problems". In: *Vehicle Routing: Problems, Methods, and Applications*. Edited by P. Toth and D. Vigo. 2nd edition. Society for Industrial and Applied Mathematics, Nov. 2014, pages 1–33. DOI: 10.1137/1.9781611973594.ch1 (cited on pages 6, 7).

[Jaw+86] J.-J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. Wilson. "A Heuristic Algorithm for the Multi-Vehicle Advance Request Dial-a-Ride Problem with Time Windows". In: *Transportation Research Part B: Methodological* 20 (3 1986), pages 243–257. ISSN: 01912615. DOI: 10.1016/0191-2615(86)90020-2 (cited on pages 10, 21, 53).

[JJP16] J. Jung, R. Jayakrishnan, and J. Y. Park. "Dynamic Shared-Taxi Dispatch Algorithm with Hybrid-Simulated Annealing". In: *Computer-Aided Civil and Infrastructure Engineering* 31 (4 2016), pages 275–291. ISSN: 10939687. DOI: 10.1111/mice.12157 (cited on pages 10, 11, 62).

[JM25] K. Jamie and K. Musilek. "Gig Economy". In: *The Blackwell Encyclopedia of Sociology* (Feb. 2025), pages 1–4. DOI: 10.1002/9781405165518.WBEOS1463.PUB2. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/9781405165518.wbeos1463.pub2%20https://onlinelibrary.wiley.com/doi/abs/10.1002/9781405165518.wbeos1463.pub2%20https://onlinelibrary.wiley.com/doi/10.1002/9781405165518.wbeos1463.pub2 (cited on page 7).

[JSM19] J.-P. Jokinen, T. Sihvola, and M. N. Mladenovic. "Policy Lessons from the Flexible Transport Service Pilot Kutsuplus in the Helsinki Capital Region". In: *Transport Policy* 76 (2019), pages 123–133. ISSN: 0967070X. DOI: 10.1016/j.tranpol.2017.12.004 (cited on pages 6, 20, 79).

[KFK21] N. Kostorz, E. Fraedrich, and M. Kagerbauer. "Usage and User Characteristics—Insights from MOIA, Europe's Largest Ridepooling Service". In: *Sustainability* 13 (2 2021), page 958. ISSN: 2071-1050. DOI: 10.3390/su13020958 (cited on pages 6, 20, 79).

[Kno+07] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner. "Computing Many-to-Many Shortest Paths using Highway Hierarchies". In: *SIAM Workshop*

on Algorithm Engineering and Experiments (ALENEX). 2007. DOI: 10.1137/1.9781611972870.4 (cited on pages 16, 21, 22).

[KO13] L. Kaan and E. V. Olinick. "The Vanpool Assignment Problem: Optimization models and solution algorithms". In: *Computers and Industrial Engineering* 66 (1 2013), pages 24–40. ISSN: 03608352. DOI: 10.1016/j.cie.2013.05.020 (cited on pages 8, 21).

[Kof04] D. Koffman. "Operational experiences with flexible transit services". In: *Transportation Research Board* (January 2004), pages 1–69. URL: http://onlinepubs.trb.org/onlinepubs/tcrp/tcrp_syn_53.pdf (cited on pages 8, 94, 96).

[Kos+24] N. Kostorz-Weiss, R. Engelhardt, G. Wilkes, F. Dandl, M. Heilig, F. Zwick, E. Fraedrich, K. Bogenberger, P. Vortisch, and M. Kagerbauer. *Assessing the Role of Ride-Pooling in Urban Transportation Systems Using a Large-Scale Integrated Supply and Demand Model*. Technical report. Manuscript submitted for publication. Karlsruhe Institute of Technology, Technical University of Munich, May 2024. DOI: 10.2139/ssrn.4849663 (cited on pages 6, 96, 108).

[KP12] S. N. Kumar and R. Panneerselvam. "A Survey on the Vehicle Routing Problem and Its Variants". In: *Intelligent Information Management* 4 (2012), pages 66–74. DOI: 10.4236/iim.2012.43010 (cited on page 7).

[Kue+23] N. Kuehnel, H. Rewald, S. Axer, F. Zwick, and R. Findeisen. "Flow-Inflated Selective Sampling for Efficient Agent-Based Dynamic Ride-Pooling Simulations". In: *Transportation Research Record: Journal of the Transportation Research Board* (2023), page 036119812311706. ISSN: 0361-1981. DOI: 10.1177/03611981231170624 (cited on pages 6, 20, 79).

[Lap92] G. Laporte. "The vehicle routing problem: An overview of exact and approximate algorithms". In: *European Journal of Operational Research* 59 (3 June 1992), pages 345–358. ISSN: 0377-2217. DOI: 10.1016/0377-2217(92)90192-C (cited on page 6).

[Lau+26] M. Laupichler, R. Andre, K. Kandler, P. Sanders, and P. Vortisch. *Advancing Dynamic Ride-Pooling Simulation – A Highly Scalable Dispatcher*. Technical report. Karlsruhe Institute of Technology, May 2026. DOI: 10.48550/arXiv.2605.11798 (cited on pages 2, 5, 14, 52, 61, 144).

[LB19] G. Leich and J. Bischoff. "Should autonomous shared taxis replace buses? A simulation study". In: *Transportation Research Procedia* 41 (Jan. 2019), pages 450–460. ISSN: 2352-1465. DOI: 10.1016/J.TRPRO.2019.09.076 (cited on page 62).

[LHN25] J. Laudan, P. Heinrich, and K. Nagel. "High-Performance Mobility Simulation: Implementation of a Parallel Distributed Message-Passing Algorithm for MATSim". In: *Inf.* 16.2 (2025), page 116. DOI: 10.3390/INFO16020116 (cited on pages 42, 63).

[Li+21] X. Li, T. Wang, W. Xu, and J. Hu. "A Novel Model for Designing a Demand-Responsive Connector (DRC) Transit System with Consideration of Users' Preferred Time Windows". In: *IEEE Transactions on Intelligent Transportation Systems* 22 (4 Apr. 2021), pages 2442–2451. ISSN: 15580016. DOI: 10.1109/TITS.2020.3031060 (cited on page 96).

[Li+26] Y. Li, H. Sun, X. Chang, Y. Lv, and K. Gao. "Towards smarter on-demand ride-sharing: Leveraging spatiotemporal flexibility to improve efficiency via delayed matching and walking". In: *Transportation Research Part A: Policy*

and Practice 209 (July 2026), page 105029. ISSN: 09658564. DOI: 10.1016/j.tra.2026.105029 (cited on page 22).

[Lin+12] Y. Lin, W. Li, F. Qiu, and H. Xu. "Research on Optimization of Vehicle Routing Problem for Ride-sharing Taxi". In: *Procedia - Social and Behavioral Sciences* 43 (2012), pages 494–502. ISSN: 18770428. DOI: 10.1016/j.sbspro.2012.04.122 (cited on page 9).

[Lit21] T. Litman. *Evaluating Public Transit as an Energy Conservation and Emission Reduction Strategy*. Technical report. Victoria Transport Policy Institute, Apr. 2021. URL: www.vtpi.org (cited on page 93).

[LLW05] K.-T. Lee, D.-J. Lin, and P.-J. Wu. "Planning and Design of a Taxipooling Dispatching System". In: *Transportation Research Record: Journal of the Transportation Research Board* 1903 (1 Jan. 2005), pages 86–95. ISSN: 0361-1981. DOI: 10.1177/0361198105190300110 (cited on page 96).

[Lor+23] E. Lorente, E. Codina, J. Barceló, and K. Nökel. "An Approach Based on Simulation and Optimisation for the Intermodal Dispatching of Public Transport and Ride-Pooling Services". In: *Applied Sciences* 13 (6 Mar. 2023), page 3803. ISSN: 2076-3417. DOI: 10.3390/app13063803 (cited on pages 96–98, 100, 104, 107, 123, 124).

[Lot+22] C. Lotze, P. Marszal, M. Schröder, and M. Timme. "Dynamic stop pooling for flexible and sustainable ride sharing". In: *New Journal of Physics* 24 (2 2022), page 023034. ISSN: 1367-2630. DOI: 10.1088/1367-2630/ac47c9 (cited on pages 21, 22).

[LQ10] X. Li and L. Quadrifoglio. "Feeder transit services: Choosing between fixed and demand responsive policy". In: *Transportation Research Part C: Emerging Technologies* 18 (5 Oct. 2010), pages 770–780. ISSN: 0968090X. DOI: 10.1016/j.trc.2009.05.015 (cited on pages 8, 96, 100).

[LRV14] G. Laporte, S. Ropke, and T. Vidal. "Chapter 4: Heuristics for the Vehicle Routing Problem". In: *MOS-SIAM Series on Optimization* (Nov. 2014), pages 87–116. DOI: 10.1137/1.9781611973594.CH4. URL: https://epubs.siam.org/doi/10.1137/1.9781611973594.ch4 (cited on page 9).

[LS17] A. Lee and M. Savelsbergh. "An extended demand responsive connector". In: *EURO Journal on Transportation and Logistics* 6 (2017), pages 25–50. DOI: 10.1007/s13676-014-0060-6 (cited on pages 96, 97).

[LS23] M. Laupichler and P. Sanders. "Fast Many-to-Many Routing for Dynamic Taxi Sharing with Meeting Points". In: (Nov. 2023) (cited on pages 5, 20).

[LS24] M. Laupichler and P. Sanders. "Fast Many-to-Many Routing for Dynamic Taxi Sharing with Meeting Points". In: *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2024, pages 74–90. DOI: 10.1137/1.9781611977929.6 (cited on pages 2, 5, 14, 18, 20, 25, 49, 53, 144).

[LWB96] C.-F. Liaw, C. White, and J. Bander. "A decision support system for the bimodal dial-a-ride problem". In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 26 (5 1996), pages 552–565. ISSN: 10834427. DOI: 10.1109/3468.531903 (cited on pages 96, 97, 100, 107).

[MAT20] MATSim. *The MATSim Open Los Angeles Scenario*. 2020. URL: https://github.com/matsim-scenarios/matsim-los-angeles (cited on page 64).

[MKV13] N. Mallig, M. Kagerbauer, and P. Vortisch. "mobiTopp – A Modular Agent-based Travel Demand Modelling Framework". In: *Procedia Computer Science* 19 (2013), pages 854–859. ISSN: 1877-0509. DOI: 10.1016/j.procs.2013.06.114 (cited on pages 63, 65, 77, 127).

[Moh72] H. Mohring. "Optimization and Scale Economies in Urban Bus Transportation". In: *The American Economic Review* 62 (4 1972), pages 591–604. ISSN: 00028282. URL: http://www.jstor.org/stable/1806101 (cited on page 1).

[MOI26a] MOIA GmbH. *Let's get moving.* https://www.moia.io/en/why-moia. Accessed 2026-05-12. 2026 (cited on page 6).

[MOI26b] MOIA GmbH. *Moving cities, moving you.* https://www.moia.io/en. Accessed 2026-05-12. 2026 (cited on page 6).

[MOI26c] MOIA GmbH. *Self-driving technology, safety-first by design.* https://www.moia.io/en/innovation/autonomous-driving. Accessed 2026-05-12. 2026 (cited on page 6).

[Mou+21] M. Mounesan, V. Jayawardana, Y. Wu, S. Samaranayake, and H. T. Vo. "Fleet management for ride-pooling with meeting points at scale: a case study in the five boroughs of New York City". In: (2021). DOI: 10.48550/arXiv.2105.00994 (cited on pages 2, 22, 23, 48).

[MPC19] A. Mourad, J. Puchinger, and C. Chu. "A survey of models and algorithms for optimizing shared mobility". In: *Transportation Research Part B* 123 (2019), pages 323–346. DOI: 10.1016/j.trb.2019.02.003. URL: https://doi.org/10.1016/j.trb.2019.02.003 (cited on pages 7, 9).

[MRR95] O. B. G. Madsen, H. F. Ravn, and J. M. Rygaard. "A Heuristic Algorithm for a Dial-a-Ride Problem with Time Windows, Multiple Capacities, and Multiple Objectives". In: *Annals of Operations Research* 60 (1 1995), pages 193–208. ISSN: 0254-5330. DOI: 10.1007/BF02031946 (cited on pages 10, 21, 53).

[MS03] U. Meyer and P. Sanders. "Δ-stepping: a parallelizable shortest path algorithm". In: *Journal of Algorithms* 49.1 (2003). 1998 European Symposium on Algorithms, pages 114–152. ISSN: 0196-6774. DOI: 10.1016/S0196-6774(03)00076-2 (cited on page 84).

[MS07] M. Müller-Hannemann and M. Schnee. "Finding All Attractive Train Connections by Multi-criteria Pareto Search". In: *Algorithmic Methods for Railway Optimization*. Springer, 2007, pages 246–263. DOI: 10.1007/978-3-540-74247-0_13 (cited on page 95).

[Mül+07] M. Müller-Hannemann, F. Schulz, D. Wagner, and C. Zaroliagis. "Timetable Information: Models and Algorithms". In: *Algorithmic Methods for Railway Optimization*. Springer, 2007, pages 67–90. DOI: 10.1007/978-3-540-74247-0_3 (cited on page 95).

[MvAvW17] D. Milakis, B. van Arem, and B. van Wee. "Policy and Society Related Implications of Automated Driving: A Review of Literature and Directions for Future Research". In: *Journal of Intelligent Transportation Systems* 21 (4 2017), pages 324–348. ISSN: 1547-2450. DOI: 10.1080/15472450.2017.1291351 (cited on pages 1, 6, 20, 62).

[MZW13] S. Ma, Y. Zheng, and O. Wolfson. "T-Share: A Large-Scale Dynamic Taxi Ridesharing Service". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pages 410–421. ISBN: 978-1-4673-4910-9. DOI: 10.1109/ICDE.2013.6544843 (cited on pages 8–10, 12, 22, 53, 62, 79).

[MZW15] S. Ma, Y. Zheng, and O. Wolfson. “Real-Time City-Scale Taxi Ridesharing”. In: *IEEE Transactions on Knowledge and Data Engineering* 27 (7 2015), pages 1782–1795. ISSN: 1041-4347. DOI: 10.1109/TKDE.2014.2334313 (cited on pages 10, 22, 53).

[Ngu25] H. L. Nguyen. *Algorithm Engineering for Software-Defined Public Transportation (PTaxi: Combining Taxi-sharing and Public Transit)*. Technical report. Karlsruhe Institute of Technology, 2025. DOI: 10.5445/IR/1000193123 (cited on page 93).

[NK18] C. Nobis and T. Kuhnimhof. *Mobilität in Deutschland- MiD: Ergebnisbericht*. Technical report. Bundesministerium für Verkehr und digitale Infrastruktur (BMVI), 2018 (cited on page 64).

[NKK16] P. Newman, L. Kosonen, and J. Kenworthy. “Theory of urban fabrics: Planning the walking, transit/public transport and automobile/motor car cities for reduced car dependency”. In: *Town Planning Review* 87 (4 June 2016), pages 429–458. ISSN: 1478341X. DOI: 10.3828/TPR.2016.28. URL: https://www.liverpooluniversitypress.co.uk/doi/10.3828/tpr.2016.28 (cited on page 1).

[Ota+15] M. Ota, H. Vo, C. Silva, and J. Freire. “A scalable approach for data-driven taxi ride-sharing simulation”. In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, Oct. 2015, pages 888–897. ISBN: 978-1-4799-9926-2. DOI: 10.1109/BigData.2015.7363837 (cited on pages 1, 10, 11, 22, 53).

[Ota+17] M. Ota, H. Vo, C. Silva, and J. Freire. “STaRS: Simulating Taxi Ride Sharing at Scale”. In: *IEEE Transactions on Big Data* 3 (3 2017), pages 349–361. ISSN: 2332-7790. DOI: 10.1109/TBDATA.2016.2627223 (cited on pages 10, 11, 22, 53).

[Pel+15] D. Pelzer, J. Xiao, D. Zehe, M. H. Lees, A. C. Knoll, and H. Aydt. “A Partition-Based Match Making Algorithm for Dynamic Ridesharing”. In: *IEEE Transactions on Intelligent Transportation Systems* 16 (5 2015), pages 2587–2598. ISSN: 1524-9050. DOI: 10.1109/TITS.2015.2413453 (cited on page 13).

[PS13] S. N. Parragh and V. Schmid. “Hybrid column generation and large neighborhood search for the dial-a-ride problem”. In: *Computers & Operations Research* 40 (1 Jan. 2013), pages 490–497. ISSN: 0305-0548. DOI: 10.1016/J.COR.2012.08.004 (cited on page 9).

[PS22] M. Potthoff and J. Sauer. “Fast Multimodal Journey Planning for Three Criteria”. In: *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Jan. 2022, pages 145–157. DOI: 10.1137/1.9781611977042.12 (cited on pages 64, 95).

[Psa80] H. N. Psaraftis. “A Dynamic Programming Solution to the Single Vehicle Many-to-Many Immediate Request Dial-a-Ride Problem”. In: *Transportation Science* 14 (2 1980), pages 130–154. ISSN: 0041-1655. DOI: 10.1287/trsc.14.2.130 (cited on pages 9–11, 21, 126).

[PV19] D.-M. Phan and L. Viennot. “Fast Public Transit Routing with Unrestricted Walking Through Hub Labeling”. In: *Analysis of Experimental Algorithms*. Springer, 2019, pages 237–247. DOI: 10.1007/978-3-030-34029-2_16 (cited on pages 95, 96).

[Pyr+08] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. “Efficient models for timetable information in public transportation systems”. In: *ACM Journal of Experimental Algorithmics* 12 (June 2008), pages 1–39. ISSN: 1084-6654. DOI: 10.1145/1227161.1227166 (cited on page 95).

[RCL07] S. Ropke, J. F. Cordeau, and G. Laporte. “Models and branch-and-cut algorithms for pickup and delivery problems with time windows”. In: *Networks* 49 (4 July 2007), pages 258–272. ISSN: 00283045. DOI: 10.1002/NET.20177 (cited on page 9).

[Rod+18] C. Rodier, M. Jaller, E. Pourrahmani, J. Bischoff, J. Freedman, and A. Pahwa. *Automated Vehicle Scenarios: Simulation of System-Level Travel Effects Using Agent-Based Demand and Supply Models in the San Francisco Bay Area*. Technical report. UC Davis, Sept. 2018. URL: https://escholarship.org/uc/item/4dk3n531 (cited on pages 1, 6).

[Ruc+21] C. Ruch, C. Lu, L. Sieber, and E. Frazzoli. “Quantifying the Efficiency of Ride Sharing”. In: *IEEE Transactions on Intelligent Transportation Systems* 22 (9 Sept. 2021), pages 5811–5816. ISSN: 15580016. DOI: 10.1109/TITS.2020.2990202 (cited on pages 12, 62).

[San+14] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti. “Quantifying the benefits of vehicle pooling with shareability networks”. In: *Proceedings of the National Academy of Sciences* 111 (37 Sept. 2014), pages 13290–13294. ISSN: 0027-8424. DOI: 10.1073/pnas.1403657111 (cited on pages 1, 22).

[Sav85] M. Savelsbergh. “Local Search in Routing Problems with Time Windows”. In: *Annals of Operations Research* 4 (1 1985), pages 285–305. ISSN: 0254-5330. DOI: 10.1007/BF02022044 (cited on pages 8, 9, 79).

[SC19] S. Shaheen and A. Cohen. “Shared Ride Services in North America: Definitions, Impacts, and the Future of Pooling”. In: *Transport Reviews* 39 (4 2019), pages 427–442. ISSN: 0144-1647. DOI: 10.1080/01441647.2018.1497728 (cited on pages 1, 7, 8).

[SCB24] S. Shaheen, A. Cohen, and A. Bayen. *The Benefits of Carpooling*. Technical report. UC Berkeley, Oct. 2024. DOI: 10.7922/G2DZ06GF. URL: https://escholarship.org/uc/item/7jx6z631 (cited on page 8).

[Son+21] C. Song, J. Monteil, J.-L. Ygnace, and D. Rey. “Incentives for Ridesharing: A Case Study of Welfare and Traffic Congestion”. In: *Journal of Advanced Transportation* (2021). ISSN: 0197-6729. DOI: 10.1155/2021/6627660 (cited on page 20).

[Spi25] Spiegel Wirtschaft. *Moia stellt Betrieb in Hannover ein – Hamburg bleibt*. https://spiegel.de/wirtschaft/unternehmen/moia-mobilitaetsdienstleister-beendet-betrieb-in-hannover-a-bb5e0196-8dc9-4e91-8c6e-7798b6fe7837. Accessed 2026-05-12. July 2025 (cited on page 6).

[Ste25] P. Steil. “A Multilevel Design for Scalable Pareto-Optimal Public Transit Queries”. Master Thesis. Karlsruhe Institute of Technology, Nov. 2025. DOI: 10.5445/IR/1000190843 (cited on page 95).

[Sti+15] M. Stiglic, N. Agatz, M. Savelsbergh, and M. Gradisar. “The Benefits of Meeting Points in Ride-Sharing Systems”. In: *Transportation Research Part B: Methodological* 82 (2015), pages 36–53. ISSN: 01912615. DOI: 10.1016/j.trb.2015.07.025 (cited on pages 13, 21, 24, 97).

[Sti+18] M. Stiglic, N. Agatz, M. Savelsbergh, and M. Gradisar. "Enhancing urban mobility: Integrating ride-sharing and public transit". In: *Computers & Operations Research* 90 (Feb. 2018), pages 12–21. ISSN: 0305-0548. DOI: 10.1016/J.COR.2017.08.016 (cited on page 97).

[SWW00] F. Schulz, D. Wagner, and K. Weihe. "Dijkstra's algorithm on-line: An empirical case study from public railroad transport". In: *ACM Journal of Experimental Algorithmics* 5 (Dec. 2000), page 12. ISSN: 1084-6654. DOI: 10.1145/351827.384254 (cited on page 95).

[SX13] D. O. Santos and E. C. Xavier. "Dynamic Taxi and Ridesharing: A Framework and Heuristics for the Optimization Problem". In: *IJCAI International Joint Conference on Artificial Intelligence*. 2013. URL: https://www.ijcai.org/Proceedings/13/Papers/424.pdf (cited on page 12).

[SX15] D. O. Santos and E. C. Xavier. "Taxi and Ride Sharing: A Dynamic Dial-a-Ride Problem with Money as an Incentive". In: *Expert Systems with Applications* 42 (19 2015), pages 6728–6737. ISSN: 09574174. DOI: 10.1016/j.eswa.2015.04.060 (cited on pages 11, 62).

[Tie22a] L. Tiede. "Integrating Ridesharing and Public Transit Routing". Bachelor Thesis. Karlsruhe Institute of Technology, 2022 (cited on page 93).

[Tra03] K. E. Train, editor. *Discrete Choice Methods with Simulation*. Cambridge University Press, 2003, page 334. ISBN: 0521816963 (cited on page 64).

[TV14] P. Toth and D. Vigo, editors. *Vehicle Routing: Problems, Methods, and Applications*. 2nd edition. Society for Industrial and Applied Mathematics, Nov. 2014. ISBN: 978-1-61197-358-7. DOI: 10.1137/1.9781611973594 (cited on pages 7, 96).

[TW08] C. Tao and C. Wu. "Behavioral Responses to Dynamic Ridesharing Services - The Case of Taxi-Sharing Project in Taipei". In: *2008 IEEE International Conference on Service Operations and Logistics, and Informatics*. IEEE, 2008, pages 1576–1581. ISBN: 978-1-4244-2012-4. DOI: 10.1109/SOLI.2008.4682777 (cited on pages 6, 20, 79).

[Ube26] Uber Technologies Inc. *When you want to Uber and save, there's Share*. https://www.uber.com/us/en/ride/uberx-share/. Accessed 2026-05-12. 2026 (cited on page 6).

[UC 24] UC Davis and KIT. *Car Dependence in Los Angeles*. Technical report. Study supported by BMW group. Accessed 2026-02-25. University of California, Davis and Karlsruhe Institute of Technology, Apr. 2024. URL: https://3rev.ucdavis.edu/sites/g/files/dgvnsk14786/files/media/documents/LA_Survey_Exec_Summary_Final_Version.pdf (cited on page 66).

[Uni15] United Nations. *Sustainabile Development Goal 11: Make cities and human settlements inclusive, safe, resilient and sustainable*. https://sdgs.un.org/goals/goal11. 2015 (cited on page 1).

[Van+23] N. Vandycke, G. S. Sehmi, J. N. Irungu, and M. N. Fabian. *Global Mobility Report 2022*. Technical report. Sustainable Mobility for All, 2023 (cited on page 1).

[VB19] V. Verbavatz and M. Barthelemy. "Critical factors for mitigating car traffic in cities". In: *PLOS ONE* 14 (7 July 2019), e0219559. ISSN: 1932-6203. DOI: 10.1371/JOURNAL.PONE.0219559 (cited on page 94).

[Wal23] J. Walker. *Lyft: The End of Shared Rides*. Personal blog, https://humantransit.org/2023/05/lyft-the-end-of-shared-rides.html. Accessed 2026-05-12. 2023 (cited on page 6).

[Wan+23] D. Wang, Q. Wang, Y. Yin, and T. Cheng. "Optimization of ride-sharing with passenger transfer via deep reinforcement learning". In: *Transportation Research Part E: Logistics and Transportation Review* 172 (2023), page 103080. DOI: https://doi.org/10.1016/j.tre.2023.103080 (cited on page 80).

[Wec+18] C. Weckström, M. N. Mladenović, W. Ullah, J. D. Nelson, M. Givoni, and S. Bussman. "User Perspectives on Emerging Mobility Services: Ex Post Analysis of Kutsuplus Pilot". In: *Research in Transportation Business & Management* 27 (2018), pages 84–97. ISSN: 22105395. DOI: 10.1016/j.rtbm.2018.06.003 (cited on pages 6, 20, 79).

[Wil+21] G. Wilkes, R. Engelhardt, L. Briem, F. Dandl, P. Vortisch, K. Bogenberger, and M. Kagerbauer. "Self-Regulating Demand and Supply Equilibrium in Joint Simulation of Travel Demand and a Ride-Pooling Service". In: *Transportation Research Record: Journal of the Transportation Research Board* 2675 (8 2021), pages 226–239. ISSN: 0361-1981. DOI: 10.1177/0361198121997140 (cited on pages 6, 20, 63, 79, 127).

[Wil23a] M. Willich. "Improving Vehicle Detour in Dynamic Ridesharing using Transfer Stops". Master's thesis. Karlsruher Institut für Technologie (KIT), 2023. 70 pages. DOI: 10.5445/IR/1000165197 (cited on page 80).

[Wit15] S. Witt. "Trip-Based Public Transit Routing". In: *European Symposium on Algorithms*. Edited by N. Bansal and I. Finocchi. Springer, Sept. 2015, pages 1025–1036. DOI: 10.1007/978-3-662-48350-3_85 (cited on page 95).

[Wör+21] T. Wörle, L. Briem, M. Heilig, M. Kagerbauer, and P. Vortisch. "Modeling intermodal travel behavior in an agent-based travel demand model". In: *Procedia Computer Science* 184 (2021), pages 202–209 (cited on page 64).

[Wör+25] M. Wörlen, S. Wist, A. Reiffer, T. Hallensleben, R. Kuhn, M. Kagerbauer, P. Bofinger, and K. Kandler. *Verkehrsentlastung durch neue Arbeitsformen und Mobilitätstechnologien: Abschlussbericht VENAMO Projekt*. Technical report. ETH Zurich, 2025 (cited on page 65).

[WZ17] D. Wagner and T. Zündorf. "Public Transit Routing with Unrestricted Walking". In: *Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*. Schloss Dagstuhl, 2017. DOI: 10.4230/OASIcs.ATMOS.2017.7 (cited on page 96).

[Xu+25] W. Xu, T. Gu, H. Chung, Z. Jiang, H. Li, K. Huang, and W. Zhu. "The multimodal dynamics of "ride-pooling" and metro: Spatial-temporal patterns from East Asia". In: *Journal of Transport Geography* 127 (July 2025), page 104295. ISSN: 0966-6923. DOI: 10.1016/J.JTRANGEO.2025.104295 (cited on page 96).

[Yan10] H. Yanagisawa. "A Multi-Source Label-Correcting Algorithm for the All-Pairs Shortest Paths Problem". In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pages 1–10. ISBN: 978-1-4244-6442-5. DOI: 10.1109/IPDPS.2010.5470362 (cited on page 26).

[Yia93] P. N. Yianilos. "Data structures and algorithms for nearest neighbor search in general metric spaces". In: *Symposium on Discrete Algorithms (SODA)*. Austin, Texas, USA: SIAM, 1993, pages 311–321. ISBN: 0898713137 (cited on page 133).

[Yu+17] B. Yu, Y. Ma, M. Xue, B. Tang, B. Wang, J. Yan, and Y.-M. Wei. "Environmental Benefits From Ridesharing: A Case of Beijing". In: *Applied Energy* 191 (2017), pages 141–152. ISSN: 03062619. DOI: 10.1016/j.apenergy.2017.01.052 (cited on pages 6, 20, 79).

[ZG26] L. Zhen and W. Gu. "Optimal demand-responsive connector design: Comparing fully-flexible routing and semi-flexible routing strategies". In: *Transportation Research Part B: Methodological* 206 (2026). DOI: 10.1016/j.trb.2026.103431. URL: https://doi.org/10.1016/j.trb.2026.103431 (cited on page 96).

[Zha+21] X. Zhan, W. Y. Szeto, C. S. Shui, and X. (Chen. "A modified artificial bee colony algorithm for the dynamic ride-hailing sharing problem". In: *Transportation Research Part E: Logistics and Transportation Review* 150 (June 2021), page 102124. ISSN: 1366-5545. DOI: 10.1016/J.TRE.2020.102124 (cited on page 12).

[Zhu21] D. Zhu. "The Limits of Money in Daily Ridesharing: Evidence from a Field Experiment in Rural France". In: *Revue d'économie industrielle* (173 2021), pages 161–202. ISSN: 0154-3229. DOI: 10.4000/rei.9984 (cited on pages 6, 20, 79).

[ZKN19] D. Ziemke, I. Kaddoura, and K. Nagel. "The MATSim Open Berlin Scenario: A Multimodal Agent-Based Transport Simulation Scenario Based on Synthetic Demand Modeling and Open Data". In: *Procedia Computer Science* 151 (2019). ISSN: 1877-0509. DOI: doi.org/10.1016/j.procs.2019.04.120 (cited on page 42).

[Zwi+22] F. Zwick, G. Wilkes, R. Engelhardt, S. Axer, F. Dandl, H. Rewald, N. Kostorz, E. Fraedrich, M. Kagerbauer, and K. W. Axhausen. "Mode choice and ride-pooling simulation: A comparison of mobiTopp, Fleetpy, and MATSim". In: *Procedia Computer Science* 201 (2022), pages 608–613. ISSN: 18770509. DOI: 10.1016/j.procs.2022.03.079 (cited on pages 6, 20, 62, 79).