Karlsruher Institut für Technologie

Bachelor Thesis

Optimizing Meeting Points in Dynamic Taxi Sharing

Arne Kuchenbecker

Date: 31. August 2024

First Examiner: Prof. Dr. Peter Sanders
Second Examiner: Prof. Dr. Thomas Bläsius
Advisor: M.Sc. Moritz Laupichler

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

The taxi sharing problem models a dispatcher for a fleet of shared taxis which has to assign vehicles to transportation requests issued by passengers (as used by UberPool). By including the possibility of the passenger being picked up and dropped off at a meeting point that is not their current location, the dispatching quality (measured for example in average trip time or vehicle operation time) can be improved considerably. However, this introduces a large performance overhead as the running time of the dispatcher scales with the number of meeting points that are evaluated.

We introduce the possibility to filter meeting points before finding assignments in the dynamic taxi sharing problem. We speed up an existing dispatcher for the problem with meeting points by filtering the meeting points with various heuristics and evaluate those on large and realistic test instances. We find that by selecting a suitable subset of the potential meeting points, our filter strategies can speed up the dispatcher by a factor of three while largely retaining solution quality according to multiple parameters such as passenger trip time and vehicle operation time.

Deutsche Zusammenfassung

Beim Taxi Sharing Problem wird ein Dispatcher für eine Flotte von Shared Taxis modelliert, der Fahrzeuge Passagieren zuweist (wie etwa bei UberPool). Durch die Möglichkeit, dass die Passagiere an Treffpunkten, die nicht ihr aktueller Standort sind, in das Fahrzeug ein- bzw. aus dem Fahrzeug aussteigen, kann die Qualität der Zuweisung (gemessen zum Beispiel an der Gesamtreisezeit von Passagieren oder der Zeit, die die Fahrzeuge im Betrieb sind) deutlich verbessert werden. Dies führt allerdings zu einem großen Performance Overhead, da die Laufzeit solcher Dispatcher mit der Anzahl der Meeting Points, die in Erwägung gezogen werden, skaliert.

Wir führen die Möglichkeit ein, Treffpunkte im Dynamic Taxi Sharing Problem zu filtern, bevor der Dispatcher eine optimale Zuweisung findet. Wir beschleunigen einen existierenden Algorithmus, indem wir verschiedene Heuristiken implementieren und auf großen, realistischen Szenarien testen. Unsere Filter Strategien können durch geschickte Auswahl von potentiellen Treffpunkten den Dispatcher um den Faktor drei beschleunigen, wobei die Lösungsqualität gemessen an der Gesamtreisezeit und der Zeit, die die Fahrzeuge im Betrieb sind, sich gegenüber dem Dispatcher ohne Filter nur wenig verschlechtert.

Acknowledgments

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 31. August 2024

Contents

1 Introduction

Transportation systems, especially in large cities, rely heavily on both individual transport and public transportation by bus or train. Individual transport often relies on a lot of cars which consume a lot of energy, produce pollution and take up space for parking and streets [5], while public transportation is often slow and inflexible [7]. Taxi sharing systems try to combine the advantages of both of these modes of transport by having not everyone taking their own car but instead having a fleet of taxis that can serve transportation requests dynamically. Then, passengers going into the same direction can share such a taxi, reducing traffic.

Taxi sharing has many advantages over both individual transport by car and public transport as it can reduce car traffic and is more flexible than traditional public transport [5], however current systems are often limited by their potential of sharing rides: Picking up another passenger often results in unreasonably large detours for passengers already in the vehicle. Introducing the possibility for passengers to walk a short distance for the vehicle to pick them up or drop them off at a more convenient spot alleviates this issue but instead introduces a performance problem as many shortest path distances have to be computed when increasing the maximum walking time permissible for the passengers [17, 10]. Our goal in this thesis is to reduce that performance overhead by reducing the number of meeting points that have to be evaluated by the dispatcher.

1.1 Contribution

Our starting point is the KaRRi dispatcher introduced by Laupichler and Sanders [10]. The number of shortest path queries required to assign a vehicle to each request (and with it the running time of the dispatcher) increases with the number of meeting points evaluated. To reduce the running time we introduce strategies to filter the potential meeting points identified by KaRRi. We test these filter strategies on large realistic data sets to evaluate the how the different strategies impact the average trip time and the time it takes for the vehicles to fulfill all requests. Also, we try to find the minimum number of meeting points necessary to maintain the solution quality for each of our filter strategies. We show that our filter strategies manage to outperform a baseline of choosing random meeting points in terms of solution quality with minimal runtime overhead. We find that we can speed up the running time of KaRRi by up to a factor of three while largely retaining the solution quality (to within one percent of the solution produced when selecting all meeting points).

1.2 Structure of Thesis

In Chapter 2 we introduce the taxi sharing problem and refer to related work before introducing the problem definition, notation and some techniques used throughout this thesis in Chapter 3. In Chapter 4 we then describe our filter strategies in detail and evaluate them in Chapter 5. Finally, we give a conclusion as well as an overview of future work in Chapter 6.

2 Problem Description and Related Work

Taxi sharing (often also referred to as ride sharing) is a well researched topic. In this chapter, we will give an overview over the different variants of the problem and proposed solutions to these variants.

2.1 Taxi Sharing

The taxi sharing problem models the task of dispatching a fleet of vehicles used for taxi sharing. In this model, passengers issue requests to be transported from their current position to a destination. The dispatcher then has to assign a vehicle to pick up the passenger and drop them off at their destination while respecting some time constraints such as the total trip time or the latest time the passenger can be picked up. Multiple passengers can share a vehicle even if they are not traveling from or to the same location. The goal of the dispatcher is to assign riders to vehicles in such a way that an objective function or cost function is optimized. The nature of this cost function varies between works but usually is some linear combination of the total operation time of the fleet (i.e. the sum of the operation times of each vehicle) and a measurement for customer satisfaction such as their total trip time [10, 14, 2].

2.2 Dynamic and Static Taxi Sharing

Two types of the taxi sharing problem can be distinguished depending on the dispatcher's knowledge of the problem instance: In the static taxi sharing problem, the number of riders as well as their routes are known in advance, in theory allowing the dispatcher to find an optimal solution. Static taxi sharing can be considered a special case of the static Dial-a-Ride problem (DARP, [2]), which even in its simplest version of only one vehicle serving all passengers is NP-hard: In that case the problem essentially constitutes a Traveling Salesman problem [14].

In the dynamic taxi sharing problem (which is what we consider in this thesis) however, passengers issue requests over time and the dispatcher has to process each of them as they

arrive. Due to an increasing amount of data in human mobility being available and advancements in the modeling of mobility, predictions of future rider demand are made possible, which Manna and Prestwich [12] use to enable the dispatcher to adjust assignments to those predictions. In this thesis however, we will stick to the traditional dispatcher model that does not know anything about future request, not even this probabilistic information [14, 10]. Due to this limited information, the dispatcher has to employ an online algorithm and thus rely on local rules and heuristics to try to find the best assignment, usually by minimizing a cost function which assigns a cost to the assignment by using the information that is available to the online algorithm. The goal is to find a cost function that translates into results which have good global solution quality (measured for example in total operation time of the vehicles or average trip time of the passengers). To do so, cost functions usually consider the additional detour for the assigned vehicle (and thus any passengers already in that vehicle) [11] or a linear combination of the resulting walking time and vehicle driving time [10, 17]. Note that this means that a globally optimal solution can not always be found.

2.3 Vehicle Detours

Incorporating a new passenger into the current schedule usually induces some sort of detour for the vehicle picking up and dropping of the passenger [7]. These detours then have an impact on the total cost of the assignment. Note that a vehicle making a detour also means that the trip time of all passengers in that vehicle increases, possibly violating their time constraints. This means that in order for the dispatcher to evaluate an assignment, it has to know the extent of these detours for each vehicle. While some works assume the detours to be known [9], calculating the shortest paths contributing to the detours in the road network can take a lot of time and greatly reduce the performance of the dispatcher. Especially if assignments have to be computed in real time, e.g. for interactive applications, there is a need to speed up these shortest path searches.

One option is to employ some filtering heuristic to find a smaller set of vertices for which shortest paths distances have to be calculated, for example by combining direct line distances and the time constraints to rule out vertices far away from the vehicle [13] or by using a spatial index for the taxis and pre-computed shortest distances for representatives of the grid cells used for this index [11]. Another approach used by the dispatcher we will consider in this paper was introduced by Buchhold et al. [1], where Bucket Contraction Hierarchies, a modification of Contraction Hierarchies (CHs), are used together with the time constraints to prune the search space of the shortest path query and compute a much smaller set of candidate vehicles. Combining this approach with Customizable Contraction Hierarchies [3] allows up-to-date traffic information to be incorporated quickly.

2.4 Meeting Points

Meeting Points introduce the option for vehicles to pick up and drop off passengers not necessarily at their origin and destination but at any nearby location. This requires the passenger to walk a short distance but may pay off by reducing the total cost of an assignment or even enabling an assignment that would otherwise be impossible due to other passengers' time constraints by reducing the trip time of other passengers to now lie within their time constraints. However, this comes with a substantial increase in problem complexity as the dispatcher has to evaluate the assignment cost for each pair of pickup and dropoff points. By describing the static taxi sharing problem as an Integer Linear Programming instance, Fielbaum et al. [6] find that introducing meeting points increases the running time of their algorithm by a factor of ten even on a relatively small road network (circa 10000 edges) while reducing the number of rejected requests as well as the average time it takes passengers to reach their destination. By expanding LOUD [1] to be able to handle meeting points, Laupichler and Sanders [10] introduce KaRRi, an algorithm for the dynamic taxi sharing problem. They find that the running times of KaRRi increase by a factor of three for each five minutes of maximum walking time but considerably reduce both the operation time of the vehicles (by approximately twelve percent for a maximum walking distance of ten minutes) and the average trip time of the passengers (by approximately five percent for the same maximum walking distance). However, these running times are still too large to be acceptable on larger scales. Wang et al. [17] compute a set of potential meeting points during a preprocessing phase of their algorithm in order to reduce the number of assignments their dispatcher has to evaluate.

2.5 Taxi Sharing and Ride Matching

In the taxi sharing problem, the only purpose of the vehicles (and their drivers) is to service requests issued by passengers. In the closely related ride matching problem [7], however, both passengers and drivers have origins, destinations and time constraints. This means that compared to taxi sharing, the vehicles are much more limited both in terms of the locations and the time windows they are available at as drivers perform rides that are of personal interest to them. Otherwise, the ride matching problem shares many difficulties with the taxi sharing problem, including the need of calculating shortest paths for vehicle detours when given the option of using meeting points [16].

3 Fundamentals

This chapter introduces the notation and definitions used in this thesis. As the taxi sharing problem is based on road networks, some important shortest path algorithms will be introduced after the basic definitions.

In this thesis, we consider the dynamic taxi sharing problem with meeting points, i.e. the dispatcher assigns a vehicle to each request immediately after receiving it and it can ask passengers to walk short distances both from their origin to the pickup point and from the dropoff point to their destination (see Chapter 2 for a more detailed comparison of the different variants of taxi sharing).

Note that as the dispatcher has to assign a vehicle to requests immediately after receiving the request, this process has to be modeled as an online algorithm: The dispatcher has no knowledge of any potential future requests, nor can it infer any information about future requests from past requests. This means that an optimal solution can not always be found but the dispatcher instead has to rely on some local metric to figure out which vehicle to assign to a request. The same goes for choosing the meeting points of the passenger and the vehicle.

3.1 Basic Definitions

This section describes the formal definitions used throughout this thesis. The definitions closely follow the definitions of [10] and [1].

Definition 3.1.1 (Road Network)**.** A *road network* is a weighted directed graph $G = (V, E, \ell)$ with vertex set V, edge set $E \subseteq V \times V$ and a map $\ell : E \to \mathbb{N}$ associating a *travel time* $\ell(e) = \ell(v, w)$ with every edge $e = (v, w) \in E$. Edges represent road segments and vertices represent intersections. The shortest path distance according to ℓ (or travel time) from a vertex v to a vertex w is denoted by $\delta(v, w)$.

In the scenario considered by the dynamic taxi sharing problem there is a fleet of vehicles. Formally we consider the following:

Definition 3.1.2 (Vehicles, Routes, and Stops)**.** Let F be a fleet of vehicles. Each *vehicle* $\nu \in F$ is associated a current *route* $R(\nu) = \langle s_0(\nu), \ldots, s_{k(\nu)}(\nu) \rangle$, a sequence of *stops* $s_i(\nu)$ currently 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 $s_1(\nu)$.

Once a vehicle arrives at a stop, its route is updated to keep this invariant, i.e. the stop it arrived from is deleted and the stop it is now at is the first stop in the route.

Note that $k(\nu) = |R(\nu)| - 1$ is the number of stops the vehicle is scheduled to visit but has not yet visited.
As vehicles are operating in the road network, we need a way to assign a location in the graph to each stop in order to compute shortest paths:

Definition 3.1.3 (Locations of Stops)**.** If $s_i(\nu)$ is a stop of a vehicle ν then we define $loc(s_i(\nu)) \in V$ as the *location* of the stop in the graph. If it is clear from the context, we write s_i instead of $s_i(\nu)$ and s_i instead of $loc(s_i)$.

In the scenario, the dispatcher receives requests from passengers and immediately assigns them to vehicles (more on the dispatcher in Chapter 2 and Section 3.4).

Definition 3.1.4 (Request)**.** A *request* $r = (orig, dest, t_{req})$ has an origin location $orig \in V$, a destination location $dest \in V$ and a time t_{req} at which the request is issued. In the scenario considered in this thesis pre-booking is not allowed. This means that the earliest departure time is the time at which the request is made.

In this thesis we focus on finding meeting points for vehicles and passengers efficiently: Passengers do not have to be picked up at their origin location. Instead, the dispatcher can ask them to walk or cycle a short while to a location in their vicinity where it is more opportune for the vehicle to pick them up.

Definition 3.1.5 (Passenger Graph)**.** We represent the paths available to passengers by walking or cycling in a road network $G_{psg} = (V_{psg}, E_{psg}, \ell_{psg})$. We denote the rider shortest path distance in G_{psg} between vertices $u, v \in V_{psg}$ by $\delta_{psg}(u, v)$.

G_{psg} represents the same geographical area as the road network for vehicles, however both graphs may contain edges that are not present in the other - pedestrians are not allowed to walk on the highway, for example. We assume that the intersection of both graphs is not empty, i.e. there are locations that are accessible both on foot and by car.
When deciding where the passenger and the vehicle should meet and where the vehicle should drop off the passenger, the dispatcher considers locations that the passenger can reach within a certain time limit.

Definition 3.1.6 (Pickup and Dropoff Locations)**.** For a request r and a model parameter ρ denoting a time radius in which to search for pickup and dropoff locations, the set of eligible *pickup locations* $P_\rho(r)$ contains all vertices that both the passenger and the vehicle can reach and that are within a time radius of ρ of $orig(r)$, i.e. $P_\rho(r) := \{p \in V_{psg} \cap V \mid \delta_{psg}(orig(r), p) \leq \rho\}$. Similarly, the set of eligible *dropoff locations* $D_\rho(r)$ is defined as $D_\rho(r) := \{d \in V_{psg} \cap V \mid \delta_{psg}(d, dest(r)) \leq \rho\}$.
We denote with $N^p_\rho(r) := |P_\rho(r)|$ the number of pickup locations and with $N^d_\rho(r) := |D_\rho(r)|$ the number of dropoff locations of a request r.

We collectively refer to the pickup and dropoff locations of a request r as the meeting points of r. Later we will see that the number of meeting points per request plays an important role in the running time of the algorithm.

For each request r the dispatcher finds a way of assigning a the request to a vehicle and inserting a pickup location and a dropoff location of r in the vehicles route. We call this assignment an insertion. The dispatcher does this in a way such that the cost of the insertion according to a cost function is minimized.

Definition 3.1.7 (Insertion)**.** An *insertion* is a 6-tuple (r, p, d, ν, i, j), where

(i) ν is the vehicle assigned to request r

(ii) $0 \leq i \leq j \leq k(\nu)$

(iii) ν picks up the passenger at pickup location $p \in P_\rho(r)$ immediately after stop s_i

(iv) ν drops of the passenger at dropoff location $d \in D_\rho(r)$ immediately after stop s_j

3.2 Cost Function and Constraints

This section describes the cost function used by the dispatcher to assign vehicles to requests. While KaRRi allows for configuration of the cost function in the form of model parameters, we will focus on one specific cost function described below.

The cost $c(\iota)$ of an insertion $\iota = (r, p, d, \nu, i, j)$ is dependent on multiple factors:

First, the trip time of the newly added passenger. Passengers want to arrive at their destination as fast as possible, so a shorter trip time is better. We denote this time as $t_{trip}(\iota)$. The trip time is the time that passes between the issuing the of the request and the arrival of the passenger at the destination $dest(r)$.

Second, the additional time that the vehicle assigned to the request has to operate. Operators of vehicle fleets want to minimize their maintenance, fuel and driver salary costs, so shorter operation times of vehicles are better. We denote this additional operation time as $t_{detour}(\iota)$.

Third, the combined additional trip time of any passengers already assigned to the vehicle (whether they are already sitting in it or still waiting to be picked up). Requests can not be assigned to a vehicles route without incurring delays to the passengers of the vehicle due to stop times even if the newly added stops are on the path the vehicle would have taken anyway. As before, shorter trip times are better. We denote this time as $t^+_{trip}(\iota)$.

Fourth, additional cost is added if the insertion violates any of the following constraints: KaRRi considers a maximum wait time, i.e. a maximum time between any passenger issuing a request and being picked up by a vehicle, and a maximum trip time, i.e. a maximum time between any passenger issuing a request and arriving at their destination. The maximum trip time is dependent on the shortest path distance $\delta(\mathit{orig}, \mathit{dest})$ from *orig* to *dest*: If a trip has to cover a longer distance, its maximum trip time will be longer. These are hard

constraints for any request already assigned to ν (thus if any of these constraints are broken by ι we set $c(\iota) = \infty$). For r however, these are soft constraints, i.e. violating them incurs a cost penalty in form of $c^{vio}_{wait}(\iota)$ in the case of the maximum wait time and $c^{vio}_{trip}(\iota)$ in the case of the maximum trip time. Both of those parameters scale with the amount of time the maxima are exceeded by.

Fifth, the walking time of the passenger. Walking might not be desirable from a passenger point of view (especially with luggage or similar), so a shorter walking time from their origin to the pickup location and from the dropoff location to the destination is better. We denote the walking time with $t_{walk}(\iota)$.

In general, KaRRi combines these factors in a linear combination, allowing some configuration via model parameters. The general formula for the cost of an insertion is

$$c(\iota) = t_{detour}(\iota) + \tau \cdot \left(t_{trip}(\iota) + t^{+}_{trip}(\iota)\right) + \omega \cdot t_{walk}(\iota) + c^{vio}_{wait}(\iota) + c^{vio}_{trip}(\iota) \tag{3.2.1}$$

However, in this thesis we will only consider the case of $\tau = 1$, $\omega = 0$, i.e. the walking time is not considered for the cost function. This is done in order to not overly penalize picking someone up at a location further away from their origin and ensures that total trip time is prioritized.

3.3 Shortest Path Algorithms

In this section we describe shortest path algorithms relevant to this thesis.

As described above, the cost of an insertion depends on the resulting trip time for the passengers. As all combinations of meeting points and possibilities of inserting them into vehicles' routes need to be evaluated, KaRRi needs to perform a lot of shortest path queries in order to find an optimal insertion. To achieve this, a speedup technique called Contraction Hierarchies that are based on Dijkstra's Shortest Path Algorithm is used.

3.3.1 Dijkstra's Shortest Path Algorithm

Given a connected, weighted graph $G = (V, E, c)$ with vertex set V, edge set E and cost function $c : E \to \mathbb{N}_0$, Dijkstra's Shortest Path Algorithm [4] is a single-source shortest path algorithm, i.e. it computes the shortest paths from an input vertex $s \in V$ to all other vertices in the graph.

The algorithm operates by iteratively scanning vertices and then updating tentative shortest path distances $\delta(v)$: It uses a priority queue Q that contains all non-scanned vertices for which the shortest path distance to s is already known. Q is initialized to contain only s and $\delta(s) = 0$. In each iteration the algorithm extracts the vertex v with the smallest tentative shortest path distance out of the queue and relaxes all incident edges (v, u): It computes a

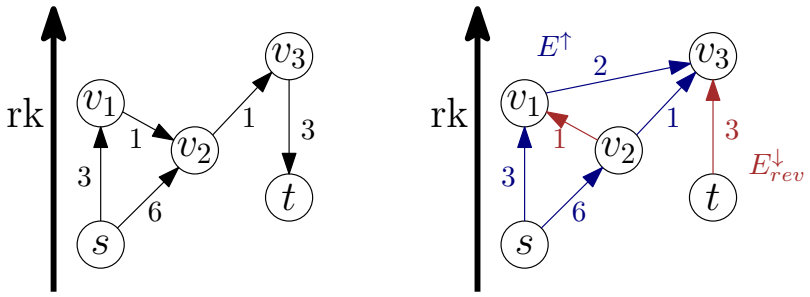


Figure 3.1: A graph (on the left) and its corresponding CH (on the right). Note that the edges in $E^{\downarrow}_{rev}$ are pointing in the opposite direction as in the original graph to facilitate shortest path queries. The edge (v_1, v_3) is inserted when contracting v_2 as v_2 lies on the shortest path from v_1 to v_3.

new distance to u by $d_{new}(u) = \delta(v) + c(v, u)$ and updates $\delta(u)$ if $d_{new}(u) < \delta(u)$. Once a vertex is scanned, its preemptive shortest path distance is also the actual shortest path distance from s. The algorithm stops once Q is empty. By storing a parent pointer, one can also reconstruct the shortest paths afterwards.

If only the shortest path distance from a vertex s to a specific vertex t is required, Dijkstra's Algorithm can be sped up by performing a bidirectional search. This means that a search in G starting from s (the forward search) and a search in the reverse graph $G_{rev} = (V, E_{rev}, c_{rev})$ with $E_{rev} = \{(v, u) \mid (u, v) \in E\}$ and $c_{rev}(v, u) = c(u, v)$ starting from t (the backward search) are performed simultaneously (e.g. by scanning vertices in either search in an alternating fashion). Once one search scans a vertex that has already been scanned by the other search, the shortest path from s to t has been found.

3.3.2 Contraction Hierarchies

When computing shortest path distances in road networks, one can exploit the fact that road networks are usually highly hierarchical. CHs [8] do so by contracting vertices in order of some metric measuring the importance of a vertex:

Assume we are given a weighted graph $G = (V, E, c)$ with vertex set V, edge set E and cost function $c : E \to \mathbb{N}_0$ with the vertices labeled 1 through $n = |V|$ in ascending importance. We call this labeling the rank of a vertex, denoted by $\mathrm{rk}(v)$. Then the CH is constructed by contracting the vertices in this order. To contract a vertex, we remove it from the graph making sure to preserve the shortest paths in the remaining overlay graph. This is done as follows: When contracting vertex v, consider each pair consisting of a source vertex $u \in V$ with $(u, v) \in E$ and a target vertex $w \in V$ with $(v, w) \in E$. If the only shortest path from u to w was $\langle u, v, w \rangle$, we insert a shortcut edge (u, w). Otherwise, there exists a shortest path from u to w that does not use v and no shortcut edge is necessary. As computing shortest paths may be costly, the range of the shortest path searches from u to w may be limited (see [8], chapter 2 for more details). If we do so, we need to make sure to insert a shortcut

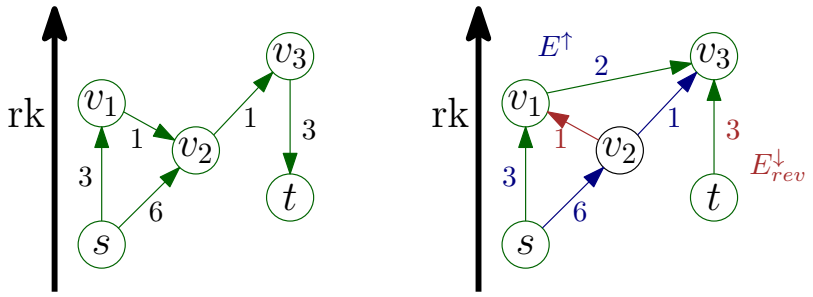


Figure 3.2: Search spaces of a bidirectional Dijkstra search (left) and a shortest path query in the CH (right). Scanned vertices and relaxated edges are marked in green. Note that in the CH only vertices on the shortest path were scanned by exploiting that v_2 has low rank and thus a shortcut was created between v_1 and v_3.

edge, whenever we do not find a path not using v that is shorter than $\langle u, v, w \rangle$. Figure 3.1 shows an example of a graph and its corresponding CH.

While finding an optimal ordering as required above is quite difficult, very useful CHs can be constructed even by just using local heuristics. Buchhold et al. [1] describe multiple such heuristics, mainly focusing on the amount of shortcut edges that need to be created.

To find shortest paths, the original graph is combined with the added shortcut edges to obtain $G' = (V, E')$ where E' is the union of E and all added shortcut edges and then split into an upward graph $G^{\uparrow} := (V, E^{\uparrow})$ with $E^{\uparrow} := \{(u, v) \in E' \mid u < v\}$ and a downward graph $G^{\downarrow} := (V, E^{\downarrow})$ with $E^{\downarrow} := \{(u, v) \in E' \mid u > v\}$. Finding a shortest path from $s \in V$ to $t \in V$ now just requires a modified bidirectional Dijkstra search: Instead of performing the forward search in G and the backward search in G_{rev}, we perform the forward search in $G^{\uparrow}$ and the backward search in $G^{\downarrow}$ (or rather, as described above, in its reverse graph). Both searches will then meet at a vertex v with the highest rank of all nodes in a shortest s-t-path (note that this vertex does not have to be the only vertex at which both searches meet) [8, 15]. This can also be expanded for Many-To-Many-Routing. Figure 3.2 shows the search space of both a bidirectional Dijkstra search and the query in the CH in the graph of Figure 3.1.

Note that even if the CH is constructed using local heuristics, the rank of a vertex in the CH is still a useful metric for the importance of a vertex when trying to find shortest paths. We will exploit this in some of the heuristics introduced in Chapter 4. Also, shortest paths in a CH always consist of a first part going upwards in rank and then a second part going downwards in rank. Thus, if one is trying to find a shortest s-t-path and s is not the vertex with the highest rank in that path, some upward neighbor of s has to be on the path (and similarly for t and its neighbors in the reverse downward graph).

3.4 LOUD and KaRRi

This thesis builds upon an existing dispatcher for the dynamic taxi sharing problem with meeting points, KaRRi [10], which is in turn built upon LOUD [1], a dispatcher for the dynamic taxi sharing problem without meeing points.

LOUD is an online algorithm assigning incoming requests to vehicles by minimizing the vehicle detour while still obtaining a feasible insertion (i.e. no hard constraints are violated). Even without meeting points, many shortest path distances have to be computed to do so: The shortest path from the origin to the destination of the request and for each vehicle $\nu \in F$ the shortest path from each stop $s_i(\nu)$ to the origin and from the destination to $s_i(\nu)$. LOUD does this by utilizing a speedup technique called elliptic pruning: For each pair of consecutive stops $s_i, s_{i+1} \in R(\nu)$, the time constraints of passenger already assigned to ν define a maximum detour the vehicle can make in order to pick up or drop off the new passenger. Any insertion exceeding this maximum detour is automatically infeasible as it would violate one or more hard constraints. Thus, the search space of the shortest path query can be constrained to the ellipse around s_i and s_{i+1} defined by that maximum detour, considerably speeding up the queries.

KaRRi expands LOUD by introducing meeting points: When processing a request, $P_\rho(r)$ and $D_\rho(r)$ are calculated and afterwards, the cost of the insertion is calculated for each pair of pickup and dropoff locations, again choosing the one with minimal cost that still results in a feasible insertion. KaRRi uses many-to-many routing deploying similar pruning techniques as described for LOUD above to find and discard infeasible insertions as early as possible to reduce the running time. As KaRRi does not use any heuristics to discard insertions, it always produces locally optimal insertions. However, on large problem instances and with large parameter ρ, the running time of KaRRi is still very long.

4 Filtering Meeting Points

In this chapter we describe the different strategies we developed to filter the potential meeting points identified by KaRRi in order to improve its running time.

4.1 Number of Meeting Points and Running Time of KaRRi

As discussed in Chapter 2, introducing meeting points to the dynamic taxi sharing problem increases the solution quality the dispatcher is able to reach. However, the number of potential meeting points evaluated for a request plays an important role in the running time per request of KaRRi. This is mainly due to the fact that for each combination of meeting points and vehicles a number of different shortest path distances have to be computed. While the use of the speedup techniques described in Chapter 3 does decrease the time each such shortest path query takes, the sheer amount needed still results in unacceptable long running times.

If we consider a request r and a fleet of vehicles F, we find that each triple $(\nu, p, d) \in F \times P_\rho(r) \times D_\rho(r)$ results in

$$\sum_{i=0}^{k(\nu)+1} (k(\nu) + 1 - i)$$

possible insertions $\iota = (r, p, d, \nu, i, j)$, i.e. the number of possible insertions per such triple is in $\mathcal{O}(k(\nu)^2)$. To calculate the cost of each of these insertions, KaRRi needs to compute four shortest path distances in G: One from $s_i(\nu)$ to p, one from p to s_{i+1}, one from s_{j-1} to d and one from d to $s_j(\nu)$ (in the case of $i = j$, only three shortest paths have to be computed as the shortest paths from p to s_{i+1} and from s_{j-1} to d are replaced by the shortest path from p to d).

However, as the number of potential meeting points increases roughly quadratically with the maximum walking time ρ, this quickly results in large running times: On larger instances even values for ρ as small as five minutes result in average times for processing a request of almost two milliseconds, increasing to seven milliseconds for a radius of ten minutes (on the setup described in Chapter 5, Laupichler and Sanders [10] report running times of 3.5 and just above thirteen milliseconds respectively). When combining taxi sharing with the possibility of the passenger using a bicycle or an e-scooter, even larger radii could be considered, leading to even larger running times.

4.2 Filter Strategies

When considering meeting points, KaRRi simply calculates the cost of each possible insertion for every meeting point. However, when looking at real road networks, we can see that some locations in these networks are more suitable as meeting points than others. If we can identify these better locations, we can limit the cost calculation to the interesting meeting points and reduce the time it takes KaRRi to make an assignment.

Thus, to combat the large running times that come with increasing ρ, we introduce the possibility to filter the potential meeting points identified by KaRRi. Before calculating the cost of every possible insertion, some potential meeting points are ruled out in order to decrease the number of meeting points for which shortest paths need to be computed. We call algorithms that perform this process filter strategies. Formally, we consider the following:

Definition 4.2.1 (Potential Meeting Points)**.** For a vertex $v \in V \cap V_{psg}$ and a maximum walking distance $\rho \in \mathbb{R}_{\geq 0}$ we denote the set of all potential meeting points reachable from v by

$$M_\rho(v) := \{v' \in V \cap V_{psg} \mid \delta_{psg}(v, v') \leq \rho\}$$

Note that for any request r, we have $M_\rho(\mathit{orig}(r)) = P_\rho(r)$ and $M_\rho(\mathit{dest}(r)) = D_\rho(r)$.

Definition 4.2.2 (Filter Strategy)**.** Let $\mathcal{M} := \mathcal{P}(V \cap V_{psg}) \setminus \{\emptyset\}$ be the set of all sets of possible meeting points where $\mathcal{P}(M)$ denotes the power set of a set M.

A *filter strategy* f is a map

$$f : V \times \mathbb{R}_{\geq 0} \to \mathcal{M}$$

such that the following holds:

(i) $f(v, \rho) \subseteq M_\rho(v)$

(ii) $v \in f(v, \rho)$

As condition (i) ensures a maximum distance from v for any selected meeting point, we refer to v as the *center* of the meeting points.

We write $\hat{N}^p_{f,\rho}(r) := |f(\mathit{orig}(r), \rho)|$ and $\hat{N}^d_{f,\rho}(r) := |f(\mathit{dest}(r), \rho)|$. If the filter strategy used is clear from context, we write $\hat{N}^p_\rho(r)$ instead of $\hat{N}^p_{f,\rho}(r)$ and $\hat{N}^d_\rho(r)$ instead of $\hat{N}^d_{f,\rho}(r)$.

We will apply these filter strategies independently on *orig* and *dest* of each request with the model parameter ρ. Note that for any request r, condition (i) ensures $f(\mathit{orig}(r), \rho) \subseteq P_\rho(r)$ and $f(\mathit{dest}(r), \rho) \subseteq D_\rho(r)$.

As the filter strategies are part of the dispatcher, they have to be implemented by an online algorithm, i.e. without any knowledge of future requests. Additionally, while we allow filter strategies to depend on the general topology of the road network (e.g. the graph or the CH), we do not consider the current state of the vehicle fleet when applying them. Given

this limited information, filter strategies may not always include the optimal meeting point in the filtered set of meeting points, resulting in the loss of solution quality (see Chapter 5). Moreover, depending on the metric used to determine solution quality, different strategies can be suited better to find good solutions (e.g. one might reduce the operation time of the vehicles compared to another which in turn produces assignments with lower average trip time).

In contrast to Wang et al. [17], who compute a set of eligible meeting points in a preprocessing phase and then find their meeting point candidates for each request from that set, we apply the filter strategies whenever a new request arrives at the dispatcher. While this does increase the running time of processing a request, this increase is small relative to the overall running time per request, as KaRRi can very quickly compute $P_\rho(r)$ and $D_\rho(r)$ and our filter strategies are fast when these sets are known. As the selection algorithm by Wang et al. [17] relies on edge weights, the preprocessing step would need to be run again if edge weights change (for example as a result of including current traffic information) which is not necessary for our filter strategies.

We implement multiple filter strategies to try to find a good selection of meeting points. A good filter strategy should select only few meeting points (i.e. $|f(v, \rho)|$ should be small) but still allow good insertion quality. To achieve this, most of the filter strategies we use rely in some way on the CH of the vehicle network prioritizing pickup locations with a higher rank. Vertices with high rank are considered more important by the heuristic used to build the CH. As the assumption behind this heuristic is that more important vertices are used by more shortest paths, we can assume that selecting meeting points with high rank increases the likelihood of these meeting points being situated on or near the route of a vehicle.

In the following sections we will describe the filter strategies both in words and by providing a formal definition. Keep in mind that we apply our filter strategies for $v \in \{\mathit{orig}, \mathit{dest}\}$ and the model parameter ρ entered when starting up KaRRi. We will use the following notation:

Definition 4.2.3 (Top k elements)**.** For a finite set L and an injective property $p : L \to \mathbb{N}$, $k \in \mathbb{N}$, let

$$\operatorname*{argmax_k}_{\ell \in L} p(\ell)$$

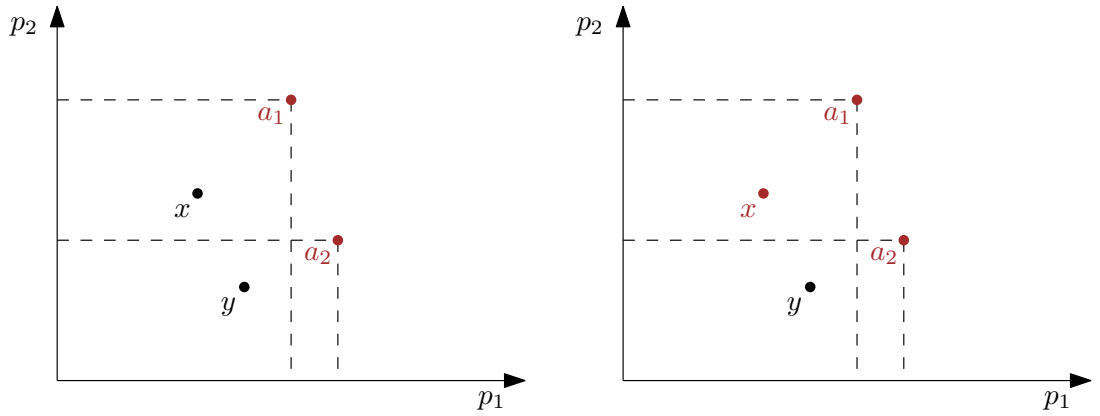
be the subset of L with the k largest elements of L according to p, i.e. for

$$\mathrm{gt}_L^p(\ell) := |\,\{\ell' \in L \mid p(\ell) < p(\ell')\}\,|$$

we define

$$\operatorname*{argmax_k}_{\ell \in L} p(\ell) := \{\ell \in L \mid \mathrm{gt}_L^p(\ell) < k\}$$

We will mainly be applying $\mathrm{argmax_k}$ to the rank of vertices. Note that $\mathrm{rk} : V \to \{1, ..., |V|\}$ mapping a vertex to its rank according to the CH is injective.



(a) Colored red are 0-Pareto optima: x is dominated by a_1 and y is dominated by both a_1 and a_2, thus they are both not 0-Pareto optima. a_1 is a 0-Pareto optimum although $p_1(a_1) < p_1(a_2)$ since $p_2(a_1) > p_2(a_2)$.

(b) Colored red are 1-pareto-optima: x is only dominated by a_1 and thus is 1-pareto-optimal while y is dominated by both a_1 and a_2 and thus is not 1-pareto-optimal

Figure 4.1: Examples for c-Pareto optima

Definition 4.2.4 (c-Pareto optima)**.** For a finite set L, properties $p_1, ..., p_k : L \to \mathbb{N}, k \in \mathbb{N}$, and $c \in \mathbb{N}$, we define the c-Pareto optima of L with respect to $p_1, ..., p_k$ as follows:

$$\underset{p_1,...,p_k}{\text{pareto}_c}\, L = \{\ell \in L \mid |dom(\ell)| \leq c\}$$

with

$$dom(\ell) := \{\ell' \in L \mid \forall i \in \{1, ..., k\} : p_i(\ell') \geq p_i(\ell) \wedge \exists i \in \{1, ..., k\} : p_i(\ell') > p_i(\ell)\}$$

Note that an element $\ell \in L$ is a c-Pareto optimum if and only if there is no more than c other elements of L that are better than or equal to ℓ in all parameters and better in at least one parameter. Figure 4.1 shows example of c-Pareto optima.

4.2.1 `All`

We define the filter strategy `All` by

$$\texttt{All} : V \times \mathbb{R}_+ \to \mathcal{M},\ (v, \rho) \mapsto M_\rho(v).$$

This filter strategy simply selects all potential meeting points. While this obviously does not reduce the running time of KaRRi, it does provide an upper bound on the solution quality as all combinations of meeting points are considered. In Chapter 5 we will use this filter strategy to evaluate the quality loss produced by the other filter strategies.

4.2.2 `Random(k)`

For a randomly chosen map $n : M_\rho(v) \to \{1, ..., |M_\rho(v)|\}$ and a model parameter $k \in \mathbb{N}$, we define the filter strategy `Random(k)` by

$$\begin{aligned} &\texttt{Random(k)} : V \times \mathbb{R}_{\geq 0} \to \mathcal{M}, \\ &(v, \rho) \mapsto \underset{\ell \in M_\rho(v) \setminus \{v\}}{\text{argmax}_{\text{k-1}}} \, n(\ell) \cup \{v\} \end{aligned}$$

This filter strategy selects up to $k - 1$ meeting points that are not the center at random and then adds the center to the resulting set. The parameter k can be chosen on startup of KaRRi. Smaller values of k will result in fewer meeting points being selected which reduces the running time but also the solution quality.

While this strategy reduces the number of meeting points, it does not take any information about the road network into account. As it is easy to compute, however, it can be used as a lower bound on the solution quality for other filter strategies (ones that produce similar or worse solutions in the same time as this heuristic may be disregarded).

Note that $\texttt{Random(1)}\,(v, \rho) = \{v\}$ regardless of ρ, meaning that no meeting points are used. However, the potential meeting points are still computed (which involves a Dijkstra search from the origin), incurring a performance overhead compared to just running KaRRi with $\rho = 0s$.

4.2.3 $\texttt{CH}_{\texttt{fix}}\texttt{(k)}$

This (and the following) filter strategy make use of the graph and the CH to select meeting points.

For a model paramter $k \in \mathbb{N}$, we define $\texttt{CH}_{\texttt{fix}}\texttt{(k)}$ by

$$\begin{aligned} &\texttt{CH}_{\texttt{fix}}\texttt{(k)} : V \times \mathbb{R}_{\geq 0} \to \mathcal{M}, \\ &(v, \rho) \mapsto \underset{\ell \in M_\rho(v) \setminus \{v\}}{\text{argmax}_{\text{k-1}}} \, \text{rk}(\ell) \cup \{v\} \end{aligned}$$

As before, we pick up to $k-1$ meeting points and then add the center, however the locations are not chosen at random. Instead, we use the CH of the vehicle graph to choose the $k - 1$ locations with the highest rank in the CH.

By including information about the road network, we can find pickup locations that produce smaller detours by avoiding sending vehicles into smaller streets of a neighborhood. Instead the vehicle will pick up the passenger at some larger road. To do so, we exploit the fact that the rank of a vertex in the CH is a useful metric of the importance of the vertex (see Chapter 3). As before, smaller values of k will result in a reduced running time and reduced solution quality. Also, as with `Random(k)`, $\texttt{CH}_{\texttt{fix}}\texttt{(1)}$ selects only v, but incurs a performance overhead compared to running KaRRi with $\rho = 0s$.

As we include a fixed number of vertices, we run the risk of ignoring meeting points even if they have a relatively high rank. Also, the vertices with the highest rank might all be part of the same road, leading to very clustered potential meeting points that are all in one cardinal direction from the origin or they might be very close to the origin thus eliminating most of the advantage of using meeting points in the first place.

4.2.4 $\texttt{CH}_{\texttt{dyn}}\texttt{(p)}$

Instead of picking a fixed number of meeting points as the previous two filter strategies, $\texttt{CH}_{\texttt{dyn}}\texttt{(p)}$ selects all meeting points whose rank is higher than p times the rank of the meeting point in $M_\rho(v)$ with the highest rank, i.e. for a model parameter $p \in [0, 1)$ we have

$$\begin{aligned}\texttt{CH}_{\texttt{dyn}}\texttt{(p)} : V \times \mathbb{R}_{\geq 0} &\to \mathcal{M}, \\ (v, \rho) &\mapsto \{\ell \in M_\rho(v) \mid \mathrm{rk}(\ell) \geq p\,\mathrm{rk}(\ell^*)\} \cup \{v\}\end{aligned}$$

for

$$\ell^* = \operatorname*{argmax}_{\ell \in M_\rho(v)} \mathrm{rk}(\ell)$$

This somewhat alleviates the issue of excluding some vertices with a rank very close to the best rank in $M_\rho(v)$, however it introduces a much higher variance in the number of vertices picked for different requests, especially for values of p close to 1. In particular, if v is the vertex with the highest rank and the next best vertex is below the threshold, we are performing an insertion without considering any meeting points (which reintroduces the problem of having large detours for people in the vehicles that KaRRi attempts to solve with meeting points). Also, if $M_\rho(v)$ contains a lot of vertices with very similar rank, this strategy might not reduce the number of potential meeting points by any relevant margin.

4.2.5 Pareto(c)

The previous strategies only took one parameter (the rank of the vertices in the CH) into account. This strategy (and the next one) introduce more parameters in order to produce better results. As different parameters are usually not easily comparable, we instead consider c-Pareto optima: These filter strategies select all meeting points that are not dominated by more than c other meeting points in all parameters. By increasing c we can increase the number of meeting points a meeting point can be dominated by and still be selected, increasing the total number of meeting points that will be selected by this strategy.

Here, we consider the rank of the vertex in the CH and the walking distance δ_{psg} from v to the vertex. We consider a higher rank and a longer walking distance to be better. This is due to vertices in very close proximity to v not providing significantly shorter paths for

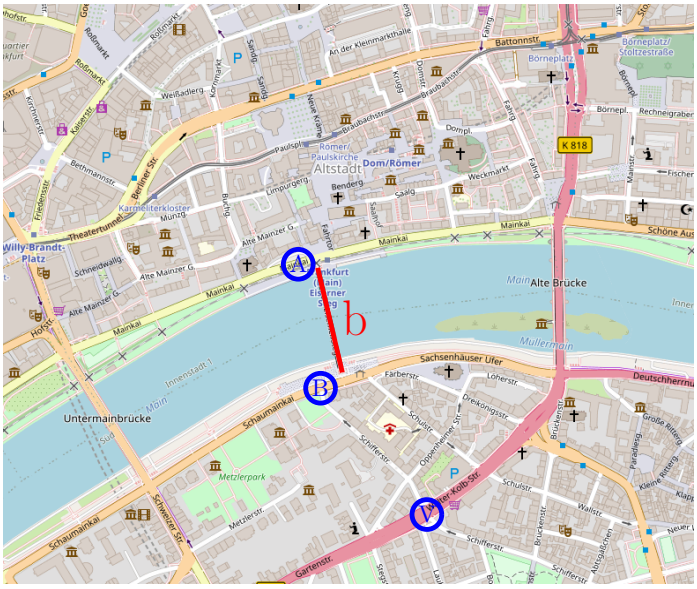


Figure 4.2: Example of how using a pedestrian bridge can reduce vehicle detours by a lot: The bridge `b` marked in red is a pedestrian only bridge across the river Main. If for a passenger at `A` only pickup locations north of the river were selected, a vehicle at `V` would have to drive a long way around on one of the vehicle bridges. $\texttt{Pareto}_{\texttt{dir}}$ ensures that also locations south of the river are consider (e.g. `B`) enabling routing the passenger across the pedestrian bridge `b`, significantly reducing the vehicle's detour.

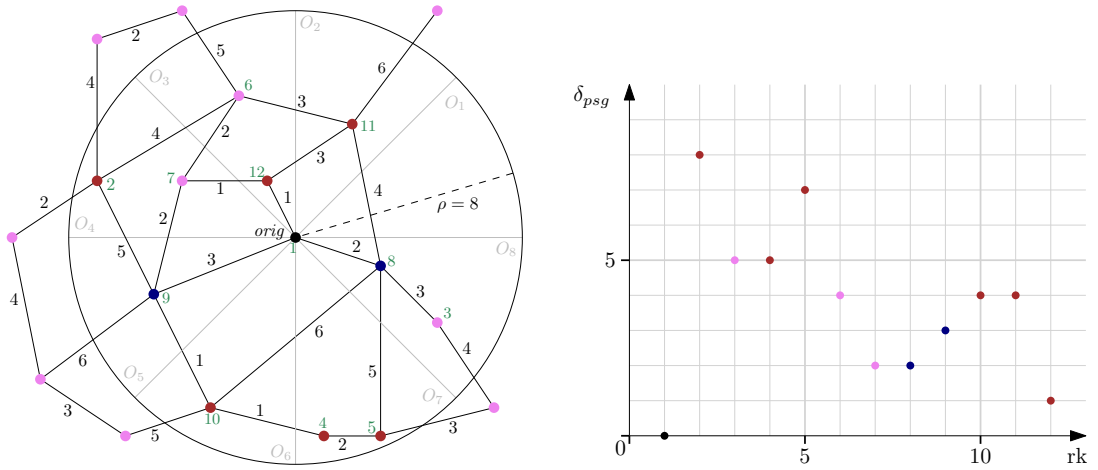
the vehicles, although they would produce shorter walking time. However, reducing the vehicles' detours means more assignments are possible in the first place (since the time constraints of the passenger already assigned to the vehicle are less likely to be violated) which provides a larger benefit than the shorter walking distances to closer vertices.

We can define this filter strategy formally by

$$\begin{aligned} \texttt{Pareto(c)} : V \times \mathbb{R}_{\geq 0} &\to \mathcal{M}, \\ (v, \rho) &\mapsto \underset{\text{rk},\delta_{psg}}{\text{pareto}_\text{c}}\, M_\rho(v) \cup \{v\} \end{aligned}$$

4.2.6 $\texttt{Pareto}_{\texttt{dir}}\texttt{(c)}$

So far, we considered information about the road network obtained from the CH but disregarded any geographical information. However, choosing meeting points based only on graph metrics of the road network may cause all meeting points to lie in the same cardinal direction from the origin or destination of the passenger. Especially in cases where the passenger and the vehicle network are different (e.g. pedestrian zones in the inner city, pedestrian only bridges across rivers) it may be beneficial to cover all cardinal directions to avoid large vehicle detours (see Figure 4.2 for an example). Even if that is not the case, as



(a) Example graph with non-pink vertices being selected by $\texttt{Pareto}_{\texttt{dir}}\texttt{(1)}$. Edges and black numbers are road links and their respective travel times in the passenger network. Green numbers represent the vertices' rank.

(b) Vertices' rank and distance in the passenger network. Notice how only the red vertices are 1-Pareto optima with regard to these parameters

Figure 4.3: Result of applying $\texttt{Pareto}_{\texttt{dir}}\texttt{(1)}$ on a graph with $\rho = 8$. Pink vertices are not selected. Red vertices are selected because they are 1-Pareto optima with regards to rk and δ_{psg}. These are the vertices that would be selected by `Pareto(1)` (in addition to *orig*). Blue vertices are additionally selected by $\texttt{Pareto}_{\texttt{dir}}\texttt{(1)}$ to fulfill the condition that from every octant there is at least one vertex. Note that O_1 does not contain any vertices so no vertex from O_1 is selected.

we do not consider the current state of the vehicle fleet, if all meeting points are in the same cardinal direction but all of the nearest vehicles are in the opposite direction, unnecessarily large detours may be created anyway.

To combat this, we improve the previous heuristic by dividing the area around the origin of the request into octants and making sure at least c locations from each octant are selected (or all locations from an octant that contains fewer than c locations). We do so by additionally selecting unselected vertices with the highest rank from each octant. Again, increasing c results in more meeting points being selected. Figure 4.3 shows an example of the result of $\texttt{Pareto}_{\texttt{dir}}$.

Formally, let $O_i \subseteq M_\rho(v)$ be the set containing all potential meeting points in the i-th octant and

$$O_i^c := O_i \setminus \operatorname*{pareto_c}_{\mathrm{rk},\delta_{psg}} M_\rho(v)$$

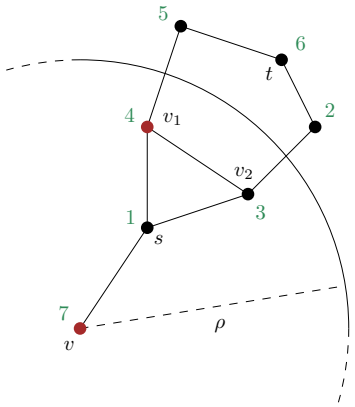


Figure 4.4: Result of applying `RankCover`. Red vertices are selected. Note how any shortest path from s to t has to use v_1 or v (otherwise there would have to be a shortcut edge from v_2 to t). Thus, v and v_1 are likely to be better meeting point candidates as the shortest path from s contains them anyway and they would keep the vehicles to more important streets.

Then $\texttt{Pareto}_{\texttt{dir}}\texttt{(c)}$ selects the following meeting points:

$$\texttt{Pareto}_{\texttt{dir}}\texttt{(c)} : V \times \mathbb{R}_{\geq 0} \to \mathcal{P}_v,$$
$$(v, \rho) \mapsto \underset{\text{rk},\delta_{psg}}{\text{pareto}_\text{c}}\, M_\rho(v) \cup \bigcup_{i=1}^{8} \text{fillup}(O_i) \cup \{v\}$$

with

$$\text{fillup}(O_i) := \begin{cases} \emptyset, & |O_i \cap \underset{\text{rk},\delta_{psg}}{\text{pareto}_\text{c}}\, M_\rho(v)| \geq c \\ \underset{\ell \in O_i^c}{\text{argmax}_\text{k}}\, \text{rk}(\ell), & |O_i \cap \underset{\text{rk},\delta_{psg}}{\text{pareto}_\text{c}}\, M_\rho(v)| =: k < c \end{cases}$$

4.2.7 `RankCover`

For this filter strategy we exploit the fact that shortest paths in the CH are up-down-paths, i.e. when following the path, the rank of the vertices first increases monotonously until the vertex with the highest rank on the path is reached after which it decreases monotonously. As removing the first or last few vertices from a shortest path again results in a shortest path (from a different start vertex to a different destination vertex), we can keep vehicles to more important roads by excluding any vertices whose neighbors in both the upward graph and the reverse downward graph are also in the set of potential meeting points.

Although both the upward and the reverse downward graph contain the shortcut edges created when building the CH, the upwards neighbors of a vertex u close to the center tend to also fall within the walking radius, making u less likely to be the best meeting

point. Thus, this strategy mainly selects meeting points that are further away from the center, resulting in longer walking distances. Figure 4.4 shows an example of the result of `RankCover`.

If we denote the upward neighbors of a vertex $v \in V$ as

$$N^{\uparrow}(v) := \{v' \in V \mid (v, v') \in E_{\uparrow}\}$$

and the neighbors in the reverse downward graph of v as

$$N^{\downarrow}(v) := \{v' \in V \mid (v', v) \in E_{\downarrow}\}$$

we can define RankCover by

$$\texttt{RankCover} : V \times \mathbb{R}_{\geq 0} \to \mathcal{P}_v,$$
$$(v, \rho) \mapsto \left\{v' \in M_\rho(v) \mid N^{\uparrow}(v') \not\subseteq M \vee N^{\downarrow}(v') \not\subseteq M\right\} \cup \{v\}$$

5 Experimental Evaluation

In this chapter, we first describe shortly how we implemented the filter strategies and our experimental setup and then evaluate the filter strategies introduced in Chapter 4.

5.1 Implementation

In order to easily be able to change which filter strategy is used, we use dependency injection and C++ templates to choose the filter strategy we want to use at compile time. For each filter strategy, we created a class that implements the method `filter(pdLocs)`, where `pdLocs` is a reference to a `std::vector` containing all potential meeting points identified by KaRRi. In that method, we implement the filter strategy by manipulating `pdLocs`. In the following, we describe how we implemented the different filter strategies.

`All` does not perform any work and immediately returns.

If the parameter k for `Random(k)` is less than the number of possible meeting points, no work is done. Otherwise, a random permutation is created and applied to `pdLocs`. Afterwards, v is swapped to index 0 and `pdLocs` is resized to fit k elements.

$\texttt{CH}_{\texttt{fix}}\texttt{(k)}$ is implemented similarly to `Random(k)`: No work is performed if k is too large, otherwise we sort `pdLocs` by the rank of the meeting points, swap in v and resize `pdLocs` to k elements.

$\texttt{CH}_{\texttt{dyn}}\texttt{(p)}$ does two runs through `pdLocs`: In the first run, the maximum rank is determined. We then select v. In the second run, the number of elements with high enough rank that have been found so far is maintained and whenever we find a new element that should be selected we swap that element with the first element not selected.

To calculate the c-Pareto optima for `Pareto(c)` and $\texttt{Pareto}_{\texttt{dir}}\texttt{(c)}$, we start by sorting `pdLocs` by rank, keeping v at index 0. Then, for each vertex at an index greater than $c+1$ (i.e. every vertex except the c meeting points with the highest rank and v), we check in order of descending rank whether they are dominated with regard to δ_{psg} by more than c meeting points selected so far. If that is not the case, we swap them with the first vertex not selected. In the case of $\texttt{Pareto}_{\texttt{dir}}\texttt{(c)}$, we then add meeting points with the highest rank that have not yet been selected from each octant until we have c vertices from each octant (or the respective octant does not contain any more vertices). Afterwards, we resize `pdLocs` to only fit the selected elements.

For `RankCover`, we again first select v and then iterate over `pdLocs` and check for each meeting point, whether their upwards neighbors are in `pdLocs` as well (finding these neighbors takes $\mathcal{O}(1)$ as the graphs created by the CH are using adjacency arrays, allowing efficient iteration of a vertex's outgoing edges). If any one of them is not in `pdLocs`, the meeting point is again swapped with the first not selected meeting point and at the end, `pdLocs` is resized to the number of elements selected.

5.2 Experimental Setup and Methodology

In this section we describe our experimental setup as well as how we evaluated our filter strategies.

Our source code[1] is written in C++17 and compiled with GCC 11.4.0 using `-O3`. We run our experiments on a machine with `Ubuntu 22.04.4 LTS`, 768 GB of Memory and 4 20-core Intel Xeon Gold 6138 processors.

5.2.1 Tuning Parameters

As our filter strategies use different parameters and reduce the number of potential meeting points by different amounts, comparing them directly is somewhat difficult. To still make useful comparisons, we run all of our filter strategies with a range of different parameters and then compare the solution quality depending on the average number of meeting points selected per request: `All` and `RankCover` do not have any additional parameters so we run them only once, while we run `Random` and $\texttt{CH}_{\texttt{fix}}$ for $k \in \{1, 2, 3, 4, 5, 10, 15, 20, 25\}$, $\texttt{CH}_{\texttt{dyn}}$ for $p \in \{0.99, 0.98, 0.97, 0.96, 0.95, 0.94, 0.92, 0.9, 0.88, 0.85, 0.8\}$ and both `Pareto` and $\texttt{Pareto}_{\texttt{dir}}$ for $c \in \{0, 1, 2, 3, 4, 5, 6\}$. These parameters were chosen to evaluate both what happens when very few meeting points are chosen and how the running time and solution quality develop when the number of meeting points selected increases. Also, they ensure that the number of selected meeting points is comparable between the strategies.

5.2.2 Instances

We evaluate the filter strategies by running KaRRi on the `Berlin-1pct (B-1%)` and `Berlin-10pct (B-10%)` instances [1] that represent the respective percentage of the taxi sharing demand in the Berlin metropolitan area on a weekday.

The Berlin instances are based on the Open Berlin Scenario [18], from which requests are generated by simulating the scenario in the transport simulation MATSim with modified

[1] Source code for KaRRi with our filter strategies can be found here: https://github.com/arnekuchenbecker/karri

instance	$\|V\|$	$\|E\|$	$\|F\|$	requests
Berlin-1pct	73 689	159 039	1 000	16 569
Berlin-10pct	73 689	159 039	10 000	149 185

Table 5.1: Key figures of our test instances

parameters making taxi sharing cheaper and private cars more expensive, effectively replacing private car traffic with taxi sharing. MATSim iteratively generates travel patterns for each individual by simulating the movement of a given population (in this case one or ten percent of the adults living in the Berlin metropolitan area). Over the course of the iterations, the travel patterns become very realistic, making these instances a good benchmark for the use of dispatchers in a real life scenario.

We run KaRRi for different values for the maximum walking distance $\rho \in \{300s, 600s\}$ and a decreased number of vehicles (20 percent on the `-1pct` instances and 15 percent on the `-10pct` instances) to enforce a higher occupancy of the vehicles. As KaRRi in its default configuration allows passengers to simply walk from their origin to their destination (which would not increase utilization of the vehicles), we force vehicle usage when running KaRRi with reduced vehicle numbers. This disables the possibility of a so-called pseudo-insertion where no vehicle is assigned to a request as the total cost of the passenger walking to their destination is lower than the cost of them waiting for any vehicle. Table 5.1 shows key figures for the baseline of the different instances.

The instances' road networks were obtained from OpenStreetMap data and the known speed limit of each road is used to determine the travel time along each of the edges in the vehicle network. In the passenger network, a walking speed of 4.5 km/h is assumed.

5.2.3 Solution Quality

Depending on the viewpoint we take to look at the dynamic taxi sharing problem, different metrics can be used to judge the global solution quality. Remember that while the dispatcher assigns vehicles to requests by minimizing a cost function, this cost function is only a local heuristic and just adding up the costs of each insertion may not be a good measure for global solution quality.

From a passenger's point of view, the most important metric to judge a taxi sharing system by (apart from the price) is the time it takes them to get from where they are to where they want to be (i.e. the total trip time). For this reason, one of the metrics we are using is the average trip time.

From a taxi sharing company's point of view, however, it is more important to have the operating vehicles serve the requests as fast as possible (and in turn reducing the number of hours they are operating and drivers have to be paid) and to maximize the occupancy of each moving vehicle to improve efficiency and profits. Thus, we also consider the average operation time of the vehicles and the average time spent driving without passengers.

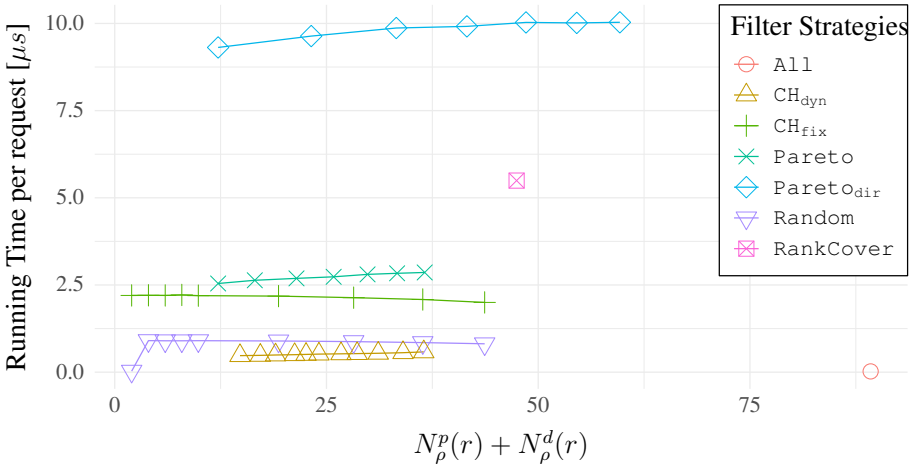


Figure 5.1: Average Running Times of the filtering algorithm of the different filter strategies for their respective paramters per request on the `B-1%` instance with $\rho = 300s$

5.2.4 Running Time

While our main objective is to reduce the overall running time of KaRRi, in order to understand how the different filter strategies perform, we also measured the time it takes to perform the filtering itself. As can be seen in Figure 5.1, the c-Pareto optima based strategies impose the highest overhead. However, even the overhead for `Pareto`$_{\text{dir}}$ only represents at most around two percent of the total running time per request for $\rho = 300s$ (see Appendix B for total running times of KaRRi on our setup), making the overhead for applying any of our filtering strategies almost negligible. We will later show that the speedups reached by filtering meeting points are much larger than this overhead.

5.3 Evaluation

In the following section, we analyze the effect the different filter strategies have on both running times and solution quality. For our test runs, we ran five iterations of every filter strategy per instance and report average running times.

To compare how the filter strategies perform it is important to keep in mind the goal we are trying to achieve: We want to reduce the running time of KaRRi as much as possible while still keeping the solution quality close to the optimum. Thus, if two filter strategies reduce the running time of KaRRi by close to the same amount but one of them maintains a better solution quality, that one is preferable over the other. However, if a filter strategy reduces the running time of KaRRi more than another, but also reduces the solution quality by more, there is no clearly superior strategy between the two.

k	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{CH}_{\texttt{fix}}}[\%]$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{CH}_{\texttt{fix}}}[\%]$
1	1.0	98	1.0	98
2	1.9	96	1.9	96
3	2.9	94	2.9	93
4	3.9	91	3.9	91
5	4.9	89	4.9	89
10	9.6	79	9.6	78
15	14.1	68	14.0	68
20	18.3	59	18.0	59
25	22.0	51	21.6	51

(a) $\texttt{CH}_{\texttt{fix}}$. Note that `Random` selects the same number of meeting points as both strategies select up to k meeting points, but prioritize different ones.

c	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{Pareto}_{\texttt{dir}}}[\%]$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{Pareto}_{\texttt{dir}}}[\%]$
0	4.5	90	7.6	83
1	10.9	75	12.2	72
2	16.2	64	16.9	62
3	20.6	54	20.9	53
4	24.3	46	24.2	45
5	27.4	39	27.1	39
6	30.0	33	29.6	33

(b) $\texttt{Pareto}_{\texttt{dir}}$

	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{RankCover}}[\%]$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{RankCover}}[\%]$
`RankCover`	22.7	49	24.66	44

(c) `RankCover`

Table 5.2: Average number of meeting points selected by different filter strategies on the `Berlin` instance with $\rho = 300s$. The φ columns show the relative reduction compared to selecting all meeting points in percent.

5.3.1 Number of Meeting Points

Different filter strategies with different parameters select different amounts of meeting points. As we saw in Chapter 4, selecting fewer meeting points means faster running times for KaRRi. Thus it is important to look at the amount of meeting points selected by each strategy.

As both instances rely on the same road network, both will result in the same potential meeting points if run with the same maximum walking distance ρ. For $\rho = 300s$ there are on average 44.7 potential pickup locations and 44.4 potential dropoff locations per request. Table 5.2 shows the average number of pickup and dropoff locations for the `Random(k)`, $\texttt{CH}_{\texttt{fix}}$`(k)`, $\texttt{Pareto}_{\texttt{dir}}$`(c)` and `RankCover` filter strategies with different values of k and c with $\rho = 300s$. We give both the absolute value (rounded down to one decimal digit) and the relative reduction compared to selecting all meeting points in percent (rounded to the nearest integer). See Table A.1 for the other filter strategies.

Note that even though `Random` and $\texttt{CH}_{\texttt{fix}}$ try to pick a fixed number of vertices (e.g. 25) the actual number of selected vertices is often lower than that target. This is due to locations not having enough potential meeting points in their proximity. The filter strategies then select all available meeting points (but still less than their target number). Also, in most cases the average number of selected pickup and dropoff locations is very close to

k	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{CH}_{\texttt{fix}}}[\%]$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{CH}_{\texttt{fix}}}[\%]$
1	1.0	99	1.0	99
2	2.0	99	2.0	99
3	3.0	98	3.0	98
4	4.0	98	4.0	98
5	4.9	97	4.9	97
10	9.9	94	9.9	94
15	14.8	92	14.8	92
20	19.8	89	19.8	89
25	24.6	86	24.6	86

(a) $\texttt{CH}_{\texttt{fix}}$. Again, note that `Random` selects the same number of meeting points.

c	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{Pareto}_{\texttt{dir}}}[\%]$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{Pareto}_{\texttt{dir}}}[\%]$
0	6.0	97	10.8	94
1	15.0	92	17.5	90
2	23.1	87	25.3	86
3	30.8	83	32.5	82
4	38.0	79	39.5	78
5	44.9	75	46.1	74
6	51.5	71	52.5	71

(b) $\texttt{Pareto}_{\texttt{dir}}$

	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{RankCover}}[\%]$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{RankCover}}[\%]$
`RankCover`	62.9	65	68.3	62

(c) `RankCover`

Table 5.3: Average number of meeting points selected by different filter strategies on the `Berlin` instance with $\rho = 600s$.

each other or even identical. This probably is due to the fact that our scenario consider requests produced over an entire day which often include travel in both directions (e.g. from your home to work), leading to origins and destinations being distributed similarly. As we always apply the same filter strategy for pickup locations and dropoff locations, the average number of meeting points then is also similar.

Increasing the maximum walking time leads to an approximately quadratic increase in the number of meeting points: For $\rho = 600s$, there are on average 178.1 potential pickup locations and dropoff locations each. Table 5.3 shows the number of meeting points selected by `Random(k)`, $\texttt{CH}_{\texttt{fix}}$ and $\texttt{Paretor}_{\texttt{dir}}$. Again, see Table A.2 for the other filter strategies.

5.3.2 Speedup and Solution Quality

As our filter strategies reduce the running time of KaRRi by reducing the number of meeting points for which the cost of insertions have to be computed, we evaluated our filter strategies by comparing their solution quality depending on the number of meeting points they select. Figure 5.2 shows how the number of meeting points selected by the different filter strategies impacts the running time as compared to selecting all meeting points.

In the following, we show how the different filter strategies perform with regards to the solution quality metrics we identified previously. As the running time of KaRRi correlates strongly with the number of potential meeting points, we compare the solution quality

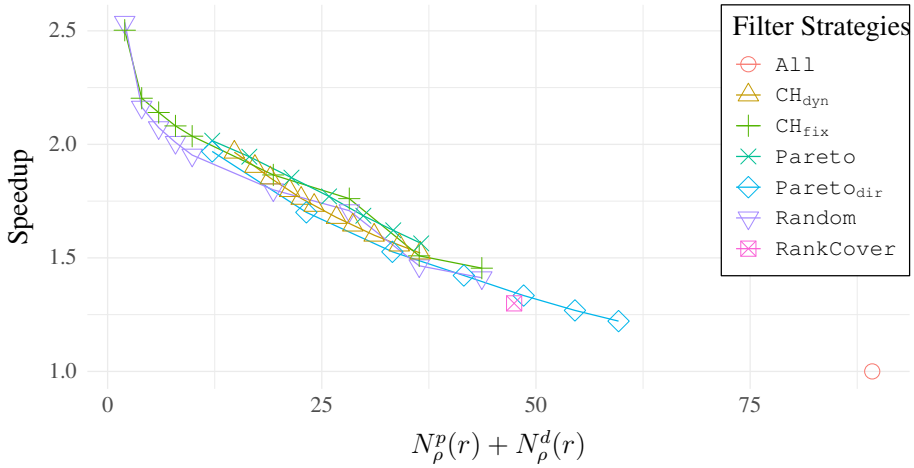


Figure 5.2: Speedup versus selecting all meeting points gained by the different filter strategies for their respective parameters. Runs were done on the `B-1%` instance with $\rho = 300s$ and $|F| = 1000$.

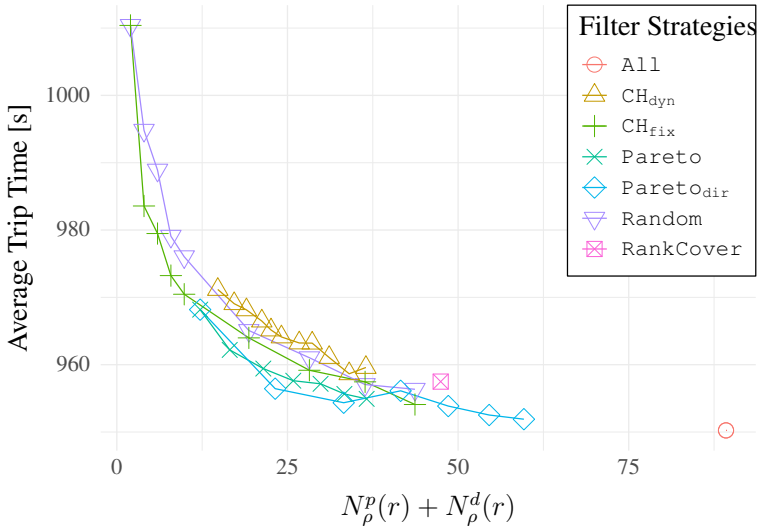


Figure 5.3: Average trip time for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies. Runs were done on the `B-1%` instance with $\rho = 300$ and $|F| = 1000$.

depending on the number of meeting points selected by our strategies. We put some plots into Appendix C to make this section more readable.

Figure 5.3 shows the average trip time over the number of selected meeting points on the `B-1%` instance with $\rho = 300$. Note reducing the amount of meeting points always increases the average trip time, but that every filter strategy except CH$_{dyn}$ and `RankCover` is an improvement over randomly selecting the meeting points. We find that depending on how

much we can afford for the trip time to rise, different filter strategies are best suited: For very low numbers of meeting points, $\texttt{CH}_{\texttt{fix}}$ produces the least increase in trip time, but is later outperformed by $\texttt{Pareto}_{\texttt{dir}}$ which even for $c = 0$ includes around ten total meeting points. Especially by including the comparison with `Pareto`, we can see that including geographical information does help producing better results as $\texttt{Pareto}_{\texttt{dir}}$ often produces better solution quality with the same number of meeting points than `Pareto`.

Notably, $\texttt{CH}_{\texttt{dyn}}$ performs worse than `Random`, something that we will see again in the other variations of our testing. This is mainly due to the large variance in the number of meeting points selected: Most of the insertions that lead to a high trip time or increase operation and empty time are done with very few or even no meeting points other than the origin and the destination of the request.

We also see that we can filter out quite a few meeting points while still largely retaining the average trip time: While selecting only the origin and destination of a request (i.e. `Random(1)` or $\texttt{CH}_{\texttt{fix}}\texttt{(1)}$) increases the average trip time from 950 seconds to 1010 seconds (an increase of one minute or just above six percent), selecting between 25 and 50 meeting points (i.e. 12 to 25 pickup locations and dropoff locations each) only increases the average trip time by around six to ten seconds (less than one percent) for both the $\texttt{CH}_{\texttt{fix}}$ and $\texttt{Pareto}_{\texttt{dir}}$ strategies.

Note that increasing the number of meeting points does not always lead to better solution quality (for example, consider the end of the line of $\texttt{CH}_{\texttt{dyn}}$ or the point of $\texttt{Pareto}_{\texttt{dir}}\texttt{(3)}$ at 41 meeting points), even though intuitively one would expect the larger set of meeting points (which in both of these cases is a superset of the smaller set) to increase solution quality. This is due to the fact that assignments are done based on a local heuristic which may produce sub-optimal results when evaluating afterwards on a global scale.

When looking at the average operation time of the vehicles (Figure 5.4) and the average time vehicles spend driving empty (Figure C.1), we find that our filter strategies perform very similar according to these parameters. Notably, `RankCover` is almost competitve according to these two parameters while it produced a higher increase in trip time than `Random` with similar running times. This can be explained by looking at which vertices `RankCover` selects: As it is based on the idea of removing the vertices with low rank from the shortest paths that would be computed, it increases the walking distance (and thus often the trip time) for the passengers. However, it is also able to keep the vehicles to important vertices, thus reducing their detours and becoming more competitve in the vehicle based metrics.

5.3.3 Reduced Fleet Size

The values above were obtained while the average occupation of the vehicles was around 0.9, i.e. on average there was less than one passenger in each vehicle at any given time. Since one of the goals of taxi sharing is to reduce the amount of traffic, we also ran our experiments on instances with fewer vehicles and enforcing the use of the vehicles to try to

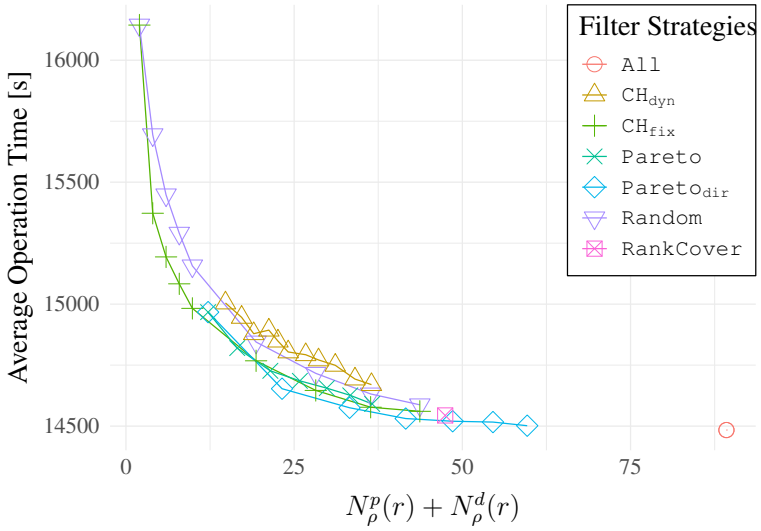


Figure 5.4: Average vehicle operation time for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies. Runs were done on the `B-1%` instance with $\rho = 300$ and $|F| = 1000$.

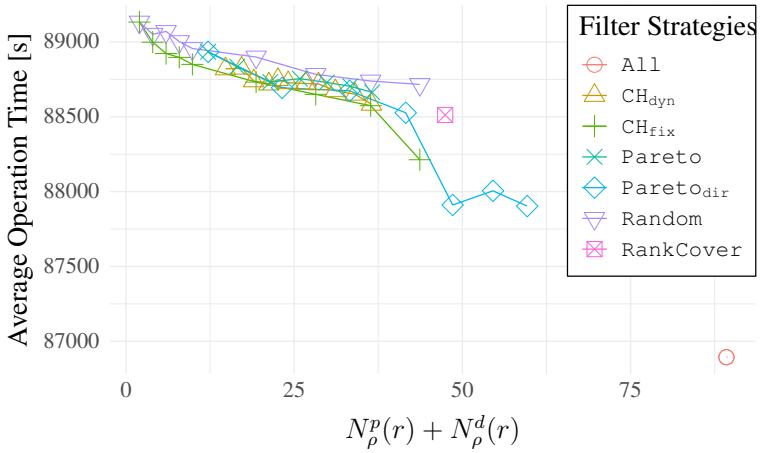


Figure 5.5: Average vehicle operation time for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies. Runs were done on the `B-1%` instance with $\rho = 300$ and $|F| = 200$.

increase the occupation. We find that the reduction in solution quality with regard to trip time (Figure C.3) and time vehicles spend driving without passenger (Figure C.4) is similar to the reduction we had for $|F| = 1000$. However, when looking at the vehicle operation time (see Figure 5.5), we see a significant increase in operation time for all of our strategies, even when selecting a large amount of meeting points. This can be explained by looking at the cost function used to evaluate insertions: Remember that the penalties for violating

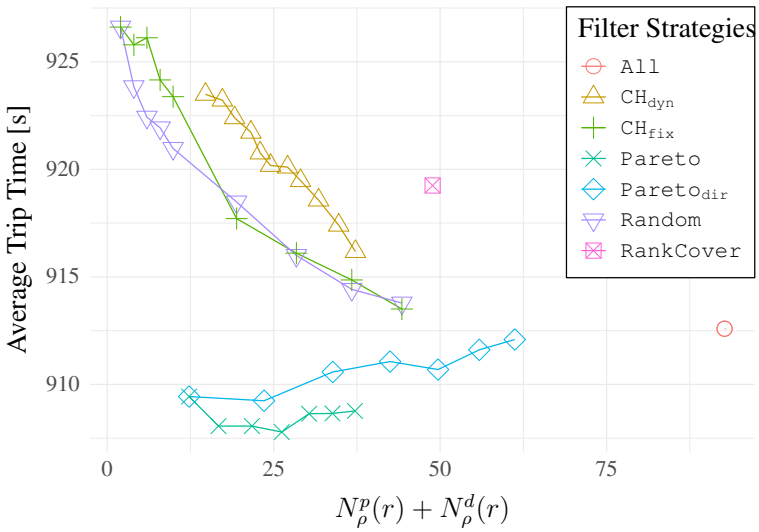


Figure 5.6: Average trip time for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies. Runs were done on the `B-10%` instance with $\rho = 300$ and $|F| = 10000$.

the time constraints of the new passenger scale with the amount of time the maximum is exceeded by. As the average trip time (and the average wait time) is very large, the penalties for violating the time constraints of the new passenger outweigh the vehicle detour cost. Thus, the cost function mostly only reduces wait and trip time, leading to the dispatcher prioritizing getting passengers into vehicles and to their destination quickly over reducing the vehicle's detours.

Note that in this scenario, the average trip time is excessively large, mainly due to the passengers often having to wait more than an hour before they can be picked up. Thus, reducing the amount of vehicles in this way does not reflect a realistic scenario, as passengers would just not use the taxi sharing service and walk or take their own car instead.

5.3.4 Increased Request Density

Another way of looking at how our strategies behave when the vehicles are utilized more is to increase the request density. That way, more people want to be transported at the same time, increasing the chance that people want to go from the same area to destinations that also either are on the way or close to each other. The `Berlin-10pct` instance offers precisely that (while also increasing the number of available vehicles by the same factor): On the `B-10%` instance, the average vehicle occupation is around 1.3, a lot higher than on the smaller instance. As the number of potential meeting points stays the same, the speedup in relation to the amount of meeting points selected also stays largely the same as on the smaller instance (Figure C.6).

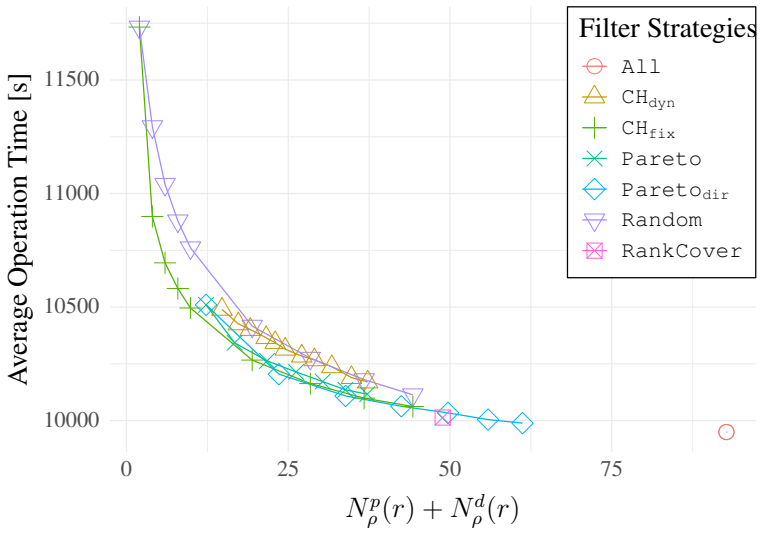


Figure 5.7: Average time the vehicles spend driving empty for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies. Runs were done on the `B-10%` instance with $\rho = 300$ and $|F| = 10000$.

We find that on this larger instance, the overall service quality (i.e. average trip time) is better than on the smaller instance, probably due to there being more vehicles for the same area being serviced. Thus, the likelihood of a vehicle headed for the same direction being nearby when a request is issued is a lot higher. Note that on this instance, introducing meeting points does not impact the average trip time by a lot: Selecting only the origin and the destination of a request results in an average trip time only 14 seconds longer than when selecting all potential meeting points (compared to an increase of one minute on the smaller instance). Interestingly, as can be seen in Figure 5.6, both the c-Pareto optima based filter strategies actually decrease the average trip time (although only by a few seconds). This could be due to them finding a good middle ground between having the passengers walk some distance and keeping vehicles on important roads.

While the service quality for the passengers does not change much by introducing meeting points, the same can not be said for vehicle operation and empty times: The average vehicle operation time when only selecting the origin and destination of a request is a lot higher than when selecting all potential meeting points (note that 1500 seconds or approximately half an hour of additional operation time across 10000 vehicles means a lot of additional cost for the operator). Thus, it can be very beneficial from an operators perspective to implement meeting points anyway. As can be seen in Figure 5.7, the filter strategies $\texttt{CH}_{\texttt{fix}}$, $\texttt{Pareto}_{\texttt{dir}}$ and `Pareto` are all capable of producing results very close to the optimum while still reducing the number of meeting points and thus the runtime of KaRRi by a significant margin to between 25 and 50 total meeting points. The same is true when looking at the time vehicles spend driving without passengers (see Figure C.7 for this plot).

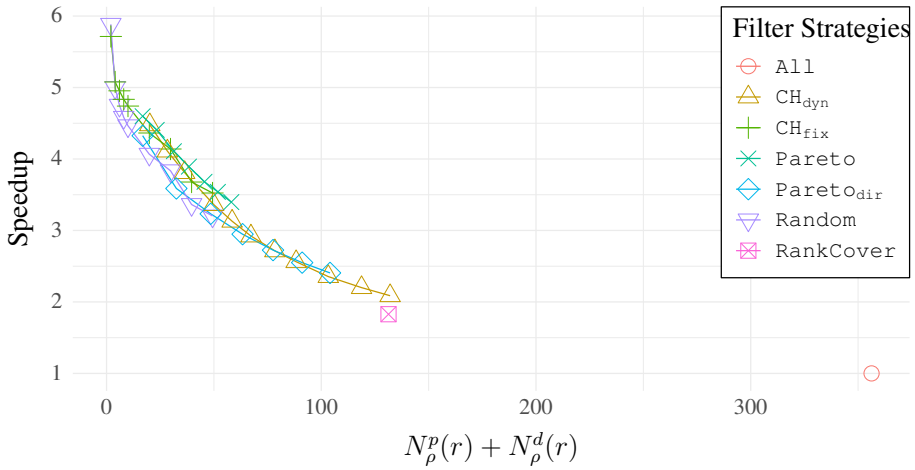


Figure 5.8: Speedup gained by the different filter strategies. Runs were done on the `B-1%` instance with $\rho = 600$ and $|F| = 1000$.

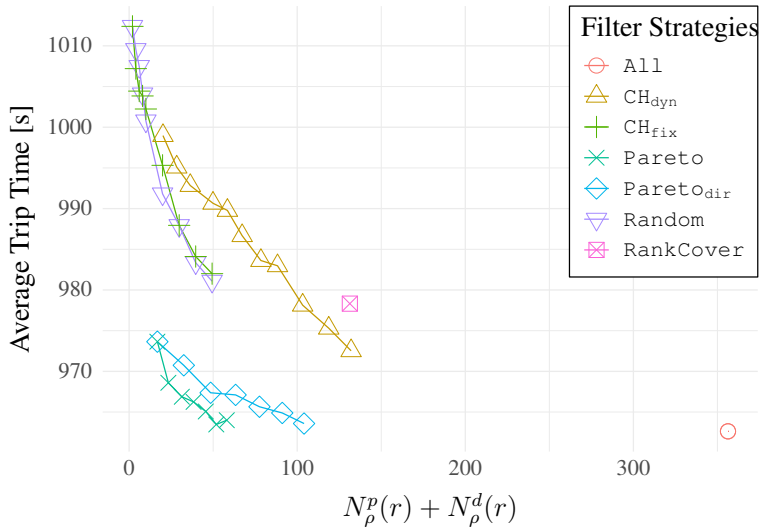


Figure 5.9: Average trip time for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies. Runs were done on the `B-1%` instance with $\rho = 600$ and $|F| = 1000$.

5.3.5 Increased Maximum Walking Time

The last variation we evaluated was increasing the maximum walking time ρ. As we saw above, this increases the number of potential meeting points roughly by a factor of 4, which also means that our filter strategies have the potential to produce a higher speedup. Figure 5.8 shows the speedup versus the number of selected meeting points as we will again be comparing solution quality against the number of selected meeting points for each of

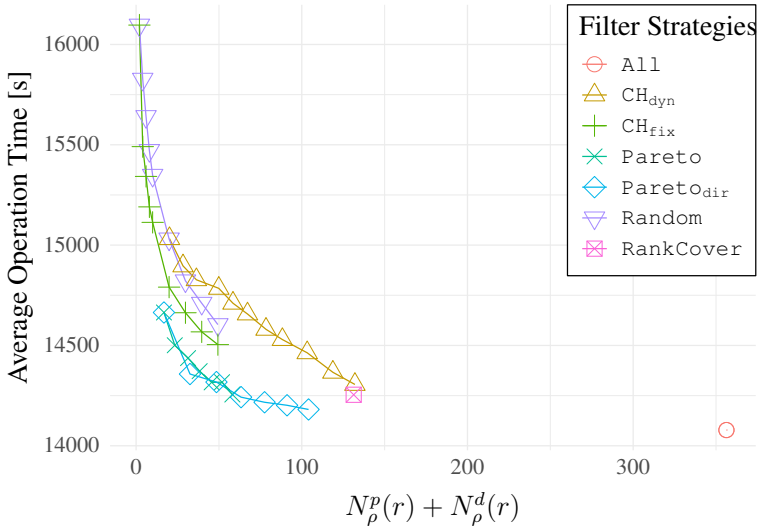


Figure 5.10: Average vehicle operation time for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies on the `B-1%` instance with $\rho = 600$ and $|F| = 1000$.

the strategies. As can be seen, while the speedup for $\rho = 300s$ ranged between 1.25 and 2.5, for $\rho = 600s$ the speedups increase to a factor between 2 and 6.

Considering the solution quality produced by our filter strategies, we find that while both c-Pareto optima based strategies continue to produce quite good results, barely increasing both average trip time (see Figure 5.9), vehicle operation time (Figure 5.10) and the time vehicles spend driving empty (Figure C.5), all other filter strategies are performing barely any better than `Random(k)`, if at all. This is likely due to there now being so many potential meeting points that just looking at the rank of vertices is not enough to ensure a good selection. Also, the c-Pareto optima based strategies have the opportunity to start the search by having the passenger walking a fairly long distance to meet the vehicle at a spot close to its current position or planned route. Only if that is not possible it is necessary to consider larger detours, which then are confined to important vertices, keeping the extra time to a minimum. Note that these strategies are the only ones that beat randomly selecting meeting points by a large margin, especially in terms of the average operation time.

We did not run the strategies tuned for selecting more meeting points as that would defeat the point of reducing running times (even so, for $\texttt{Pareto}_{\texttt{dir}}\texttt{(6)}$, processing a request takes KaRRi 1.4 milliseconds on our system). Still, we can see that we are able to speedup KaRRi by a factor of 2.5 to 3 while increasing the average trip by only between 0.8 and 5 seconds (less than one percent).

Increasing the walking radius on the larger instance again has the c-Pareto optima based filter strategies produce better results than all of the others (for the plots, see Figure C.9 and Figure C.10). Notably however, when looking at the average trip time, we find that for most

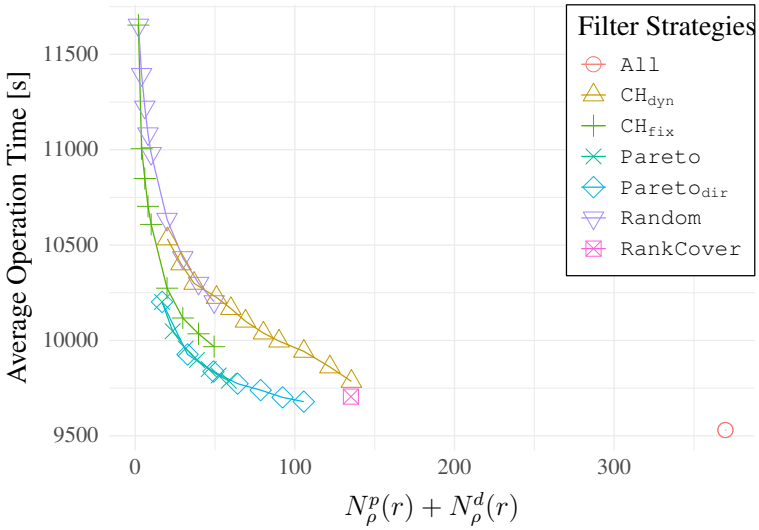


Figure 5.11: Average vehicle operation time for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies on the `B-10%` instance with $\rho = 600$ and $|F| = 10000$.

filter strategies, selecting fewer meeting points actually produces better results (although the same can not be said for the operation time and the time vehicles spend driving empty). Again, the increases and reductions in trip time are within a few percent of the solution found by selecting all meeting points, while meaningful differences can be observed for the other two parameters. Note that the c-Pareto based strategies again outperform the other strategies by a large margin (as can be seen in Figure 5.11). On this instance, we can see that for larger k, $\texttt{CH}_{\texttt{fix}}$ does produce better results than randomly selecting meeting points, although not as good results as the c-Pareto optima based strategies, which again have the advantage of being able to find a middle ground between walking distance of the passengers and keeping vehicles to important roads.

6 Conclusion

In this chapter, we draw a conclusion from our experiments. Also, we want to mention some points in which future work might use or improve on the results of this thesis.

In Chapter 5 we saw how our filter strategies impacted both running time and solution quality of KaRRi. Depending on the scenario and the selected model parameters, different strategies were performing better than others, however some general conclusions can be drawn:

First, for $\rho = 300s$ all filter strategies that allow tuning (i.e. all except `RankCover` and `All`) can be tuned in such a way that by selecting approximately 25 total meeting points (i.e. about 12 pickup locations and dropoff locations each), a speedup of around 1.7 can be achieved while keeping the solution quality within one percent of the solution found by selecting all meeting points.

For $\rho = 600$, we find that utilizing the `Pareto` and $\texttt{Pareto}_{\texttt{dir}}$ filter strategies produce very good result (again, within one percent of the solution found by selecting all meeting points) while being able to reach speedups of up to 3 by selecting between 50 and 100 meeting points.

Second, the $\texttt{Pareto}_{\texttt{dir}}$ and `Pareto` (and, for $\rho = 300$, $\texttt{CH}_{\texttt{fix}}$) filter strategies consistently outperform the other strategies with `RankCover` and $\texttt{CH}_{\texttt{dyn}}$ often producing worse results than `Random` while selecting the same number of meeting points. While this is not a major problem on the `B-10%` instance with regards to the trip time, the increase in vehicle operation time and time the vehicles spend empty is significant. Note that an increase in average operation time of four minutes (which is the difference between the strategies at around 25 selected meeting points) over the 10000 vehicles in use in that scenario means a total of more than 600 hours of increased operation time, which means more costs for the operator, more traffic in the city and more pollution into the environment.

Third, while `RankCover` does produce relatively good results with regards to the vehicle operation time on the `B-10%` instance, the improvement over the c-Pareto optima based strategies is only small, even when comparing to runs where the other strategies select a lot less meeting points.

To conclude, we find that the `Pareto` and $\texttt{Pareto}_{\texttt{dir}}$ filter strategies are a good way of speeding up the dispatcher for the dynamic shared taxi problem as utilizing them with we are able to reach speedups of up to a factor of 3 while still keeping the solution quality within one percent of the solution produced by selecting all potential meeting points.

While we did extensive testing of our own filter strategies, this thesis does not provide a comparison to the filtering algorithm of Wang et al. [17]. In particular, a comparison of a filter strategy based on k-skip covers could be compared to the ones introduced in this thesis. Otherwise, just as mentioned in the paper introducing KaRRi [10], not only the dispatcher itself, but also the filter strategies could benefit from parallelization, increasing running times even further.

Also note that we did not find an explanation as to makes a location a good meeting point. This warrants further investigation, especially in the static taxi sharing problem where more global metric could be employed.

A Amount of Selected Meeting Points

p	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{CH}_{\texttt{dyn}}}$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{CH}_{\texttt{dyn}}}$
0.99	7.1	0.84	7.6	0.83
0.98	8.3	0.81	8.8	0.80
0.97	9.2	0.79	9.7	0.78
0.96	10.4	0.77	10.8	0.76
0.95	11.1	0.75	11.4	0.74
0.94	11.9	0.73	12.1	0.73
0.92	13.2	0.70	13.4	0.70
0.90	14.2	0.68	14.3	0.68
0.88	15.5	0.65	15.5	0.65
0.85	17.0	0.62	16.9	0.62
0.80	18.2	0.59	18.2	0.59

(a) $\texttt{CH}_{\texttt{dyn}}$

c	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{Pareto}}$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{Pareto}}$
0	4.5	0.90	7.6	0.83
1	7.5	0.83	8.9	0.80
2	10.0	0.77	11.3	0.74
3	12.3	0.72	13.4	0.70
4	14.4	0.68	15.4	0.65
5	16.3	0.64	17.0	0.62
6	17.9	0.60	18.6	0.58

(b) `Pareto`

Table A.1: Average number of meeting points selected by different filter strategies on the `Berlin` instance with $\rho = 300$.

p	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{CH}_\texttt{dyn}}$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{CH}_\texttt{dyn}}$
0.99	9.8	0.94	10.2	0.94
0.98	13.8	0.92	14.4	0.92
0.97	17.7	0.90	18.6	0.90
0.96	24.6	0.86	25.2	0.86
0.95	29.0	0.84	29.4	0.83
0.94	33.5	0.81	33.7	0.81
0.92	39.2	0.78	39.1	0.78
0.90	44.2	0.75	44.0	0.75
0.88	51.5	0.71	51.7	0.71
0.85	59.3	0.67	59.3	0.67
0.80	66.0	0.63	65.9	0.63

(a) $\texttt{CH}_\texttt{dyn}$

c	$\hat{N}^p_\rho(r)$	$\varphi^p_{\texttt{Pareto}}$	$\hat{N}^d_\rho(r)$	$\varphi^d_{\texttt{Pareto}}$
0	6.0	0.97	10.8	0.94
1	10.4	0.94	12.8	0.93
2	14.3	0.92	16.9	0.91
3	18.0	0.90	20.2	0.89
4	21.6	0.88	24.0	0.87
5	24.9	0.86	26.9	0.85
6	28.0	0.84	30.0	0.83

(b) `Pareto`

Table A.2: Average number of meeting points selected by different filter strategies on the `Berlin` instance with $\rho = 600$.

B Running Time of KaRRi

The following tables contains the average running time per request t_r of KaRRi (rounded down to nanoseconds) with our filter strategies on the setup described in Chapter 5 for both the `B-1%` and the `B-10%` instance.

	$t_r(1\%)$	$t_r(10\%)$
`All`	904 416	1 866 283

Table B.1: Average running time of KaRRi per request for $\rho = 300s$ in nanoseconds with the filter strategy `All`.

	$t_r(1\%)$	$t_r(10\%)$
`RankCover`	695 846	1 374 094

Table B.2: Average running time of KaRRi per request for $\rho = 300s$ in nanoseconds with the filter strategy `RankCover`.

k	$t_r(1\%)$	$t_r(10\%)$
1	356 406	787 275
2	417 950	856 427
3	436 178	870 504
4	450 251	890 274
5	463 142	905 716
10	502 532	962 266
15	529 552	1 009 005
20	617 277	1 205 108
25	640 327	1 250 173

Table B.3: Average running time of KaRRi per request for $\rho = 300s$ in nanoseconds with the filter strategy `Random`.

k	$t_r(1\%)$	$t_r(10\%)$
1	361 380	790 694
2	410 494	830 348
3	422 555	843 221
4	434 561	859 745
5	444 200	873 023
10	484 863	934 839
15	513 590	982 216
20	599 337	1 174 209
25	621 761	1 220 053

Table B.4: Average running time of KaRRi per request for $\rho = 300s$ in nanoseconds with the filter strategy $\texttt{CH}_{\texttt{fix}}$.

p	$t_r(1\%)$	$t_r(10\%)$
0.99	460 162	908 655
0.98	475 323	941 851
0.97	488 358	969 217
0.96	503 520	1 002 334
0.95	513 255	1 023 272
0.94	522 440	1 042 334
0.92	538 997	1 075 384
0.90	550 054	1 096 778
0.88	565 316	1 126 762
0.85	581 110	1 157 537
0.80	594 487	1 181 518

Table B.5: Average running time of KaRRi per request for $\rho = 300s$ in nanoseconds with the filter strategy $\texttt{CH}_{\texttt{dyn}}$.

c	$t_r(1\%)$	$t_r(10\%)$
0	448 572	895 682
1	464 829	920 833
2	488 007	963 263
3	510 756	1 010 820
4	536 566	1 059 678
5	557 820	1 106 222
6	578 322	1 151 131

Table B.6: Average running time of KaRRi per request for $\rho = 300s$ in nanoseconds with the filter strategy `Pareto`.

c	$t_r(1\%)$	$t_r(10\%)$
0	459 361	907 363
1	531 837	1 025 146
2	592 340	1 152 165
3	635 828	1 244 945
4	677 931	1 330 727
5	712 919	1 406 539
6	740 473	1 465 860

Table B.7: Average running time of KaRRi per request for $\rho = 300s$ in nanoseconds with the filter strategy $\texttt{Pareto}_{\texttt{dir}}$.

C Further Plots

This appendix contains plots for the test instances that were not displayed in the thesis above.

`B-1%` with $\rho = 300$ and $|F| = 1000$

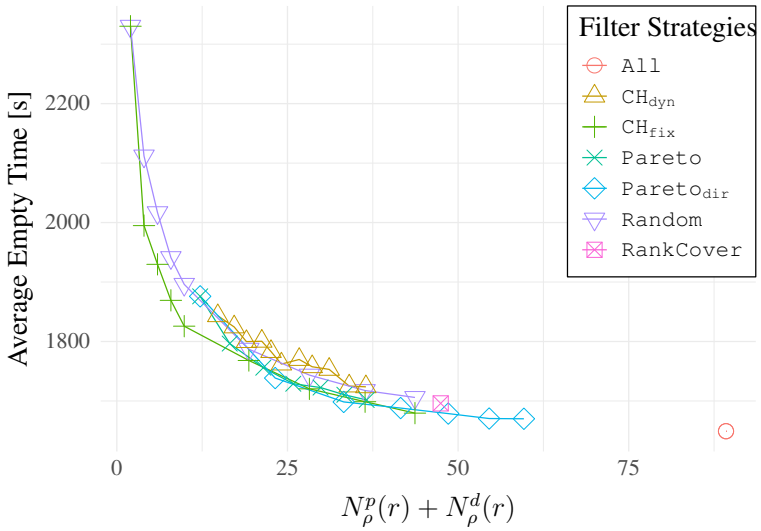


Figure C.1: Average time the vehicles spend driving without passengers for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies. Runs were done on the `B-1%` instance with $\rho = 300$ and $|F| = 1000$.

B-1% with $\rho = 300$ and $|F| = 200$

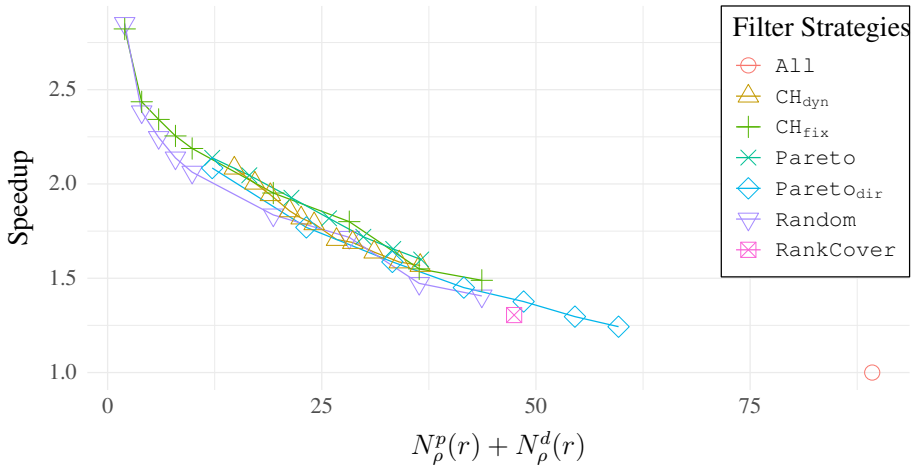


Figure C.2: Speedup gained by the different filter strategies on the B-1% instance with $\rho = 300$ and $|F| = 200$.

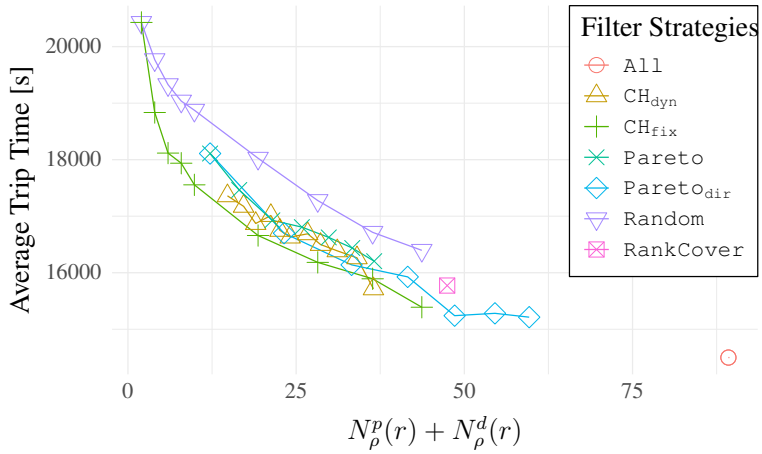


Figure C.3: Average trip time for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies on the B-1% instance with $\rho = 300$ and $|F| = 200$.

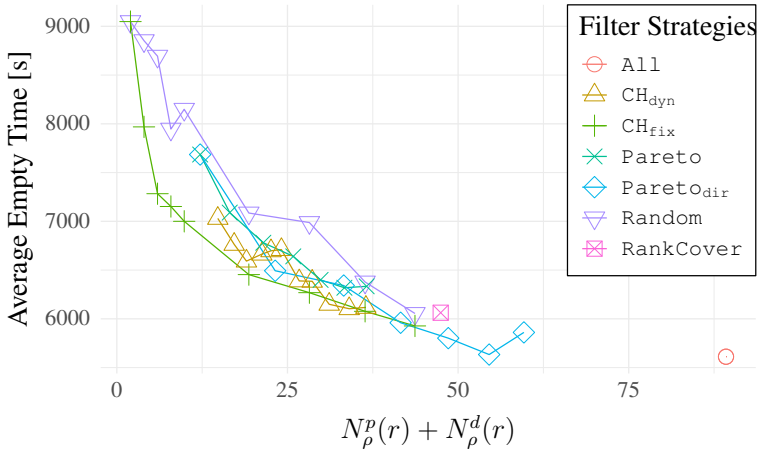


Figure C.4: Average time the vehicles spend driving without passengers for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies on the `B-1%` instance with $\rho = 300$ and $|F| = 200$.

`B-1%` with $\rho = 600$ and $|F| = 1000$

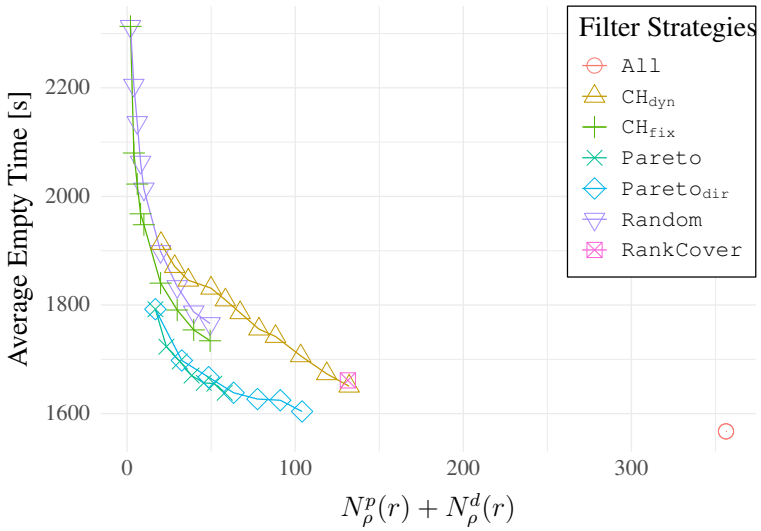


Figure C.5: Average time the vehicles spend driving without passengers for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies on the `B-1%` instance with $\rho = 600$ and $|F| = 1000$.

B-10% with $\rho = 300$ and $|F| = 10000$

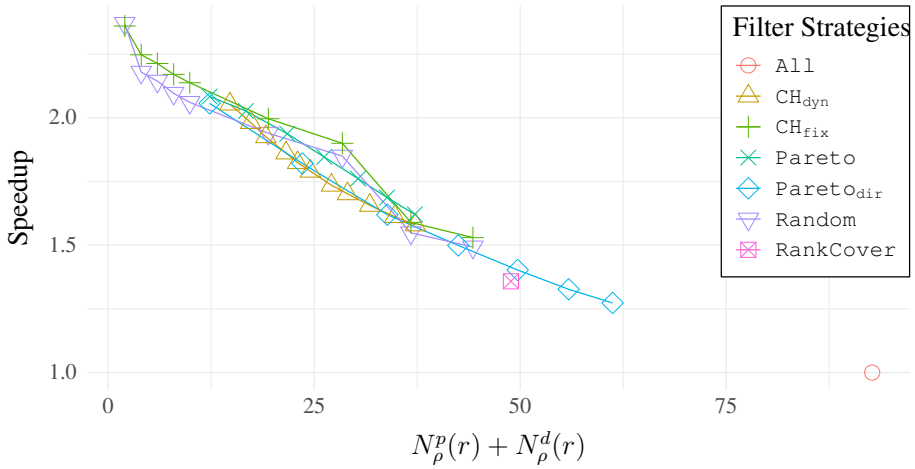


Figure C.6: Speedup gained by the different filter strategies on the B-10% instance with $\rho = 300$ and $|F| = 10000$.

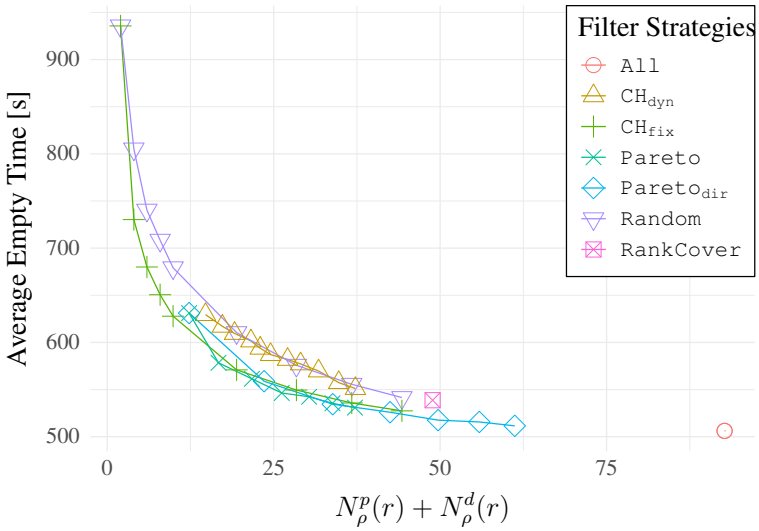


Figure C.7: Average time the vehicles spend driving without passengers for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies on the B-10% instance with $\rho = 300$ and $|F| = 10000$.

B-10% with $\rho = 600$ and $|F| = 10000$

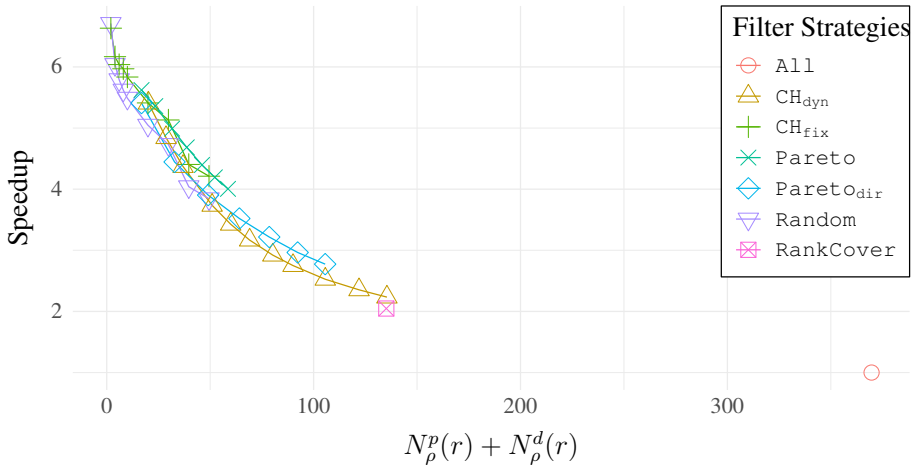


Figure C.8: Speedup gained by the different filter strategies on the B-10% instance with $\rho = 600$ and $|F| = 10000$.

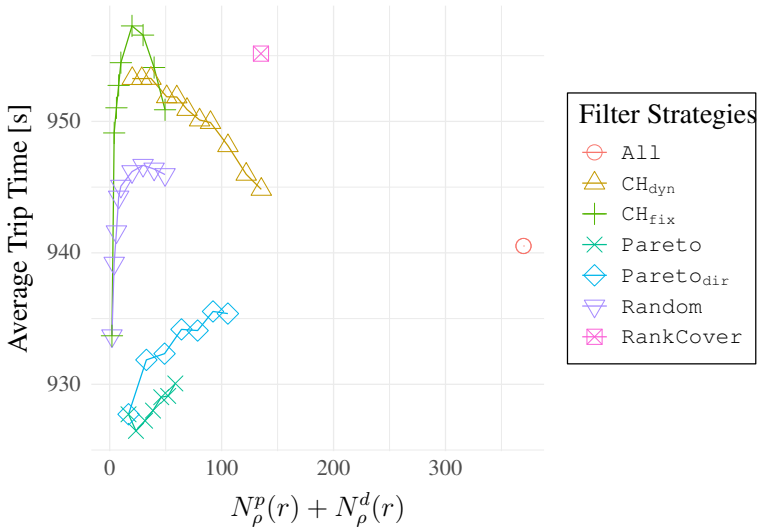


Figure C.9: Average trip time for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies on the B-10% instance with $\rho = 600$ and $|F| = 10000$.

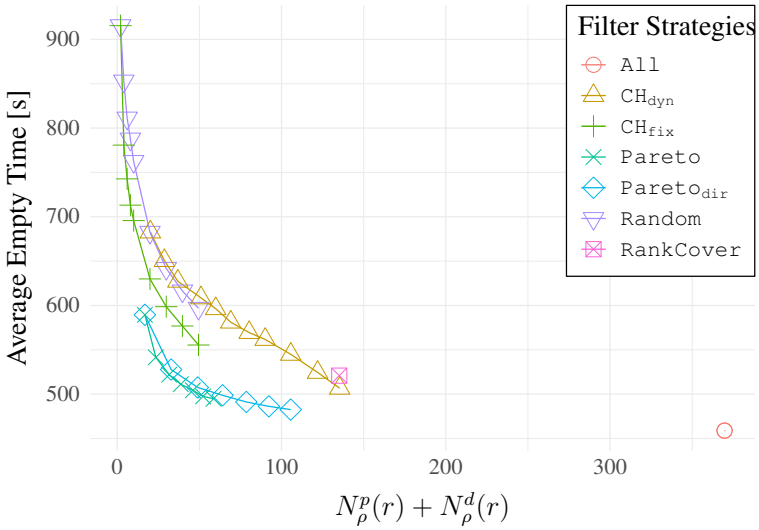


Figure C.10: Average time the vehicles spend driving without passengers for the different filter strategies against the number of selected meeting points for the different parameters of the filter strategies on the `B-10%` instance with $\rho = 600$ and $|F| = 10000$.

Bibliography

[1] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Fast, Exact and Scalable Dynamic Ridesharing. In *2021 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, 2021. doi:10.1137/1.9781611976472.8.

[2] Jean-François Cordeau and Gilbert Laporte. The Dial-a-Ride Problem (DARP): Models and Algorithms. *Annals of Operations Research*, 2007. doi:10.1007/s10479-007-0170-8.

[3] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies. *ACM J. Exp. Algorithmics*, 2016. doi:10.1145/2886843.

[4] E. W. Dijkstra. A Note on two Problems in Connexion with Graphs. *Numerische Mathematik 1*, 1959.

[5] Pedro d'Orey and Michel Ferreira. Reducing the Environmental Impact of Taxi Operation: The Taxi-Sharing Use Case. In *2012 12th International Conference on ITS Telecommunications, ITST*, 2012. doi:10.1109/ITST.2012.6425191.

[6] Andres Fielbaum, Xiaoshan Bai, and Javier Alonso-Mora. On-Demand Ridesharing with Optimized Pick-Up and Drop-Off Walking Locations. *Transportation Research Part C: Emerging Technologies*, 2021. doi:10.1016/j.trc.2021.103061.

[7] Masabumi Furuhata, Maged Dessouky, Fernando Ordóñez, Marc-Etienne Brunet, Xiaoqing Wang, and Sven Koenig. Ridesharing: The State-of-the-Art and Future Directions. *Transportation Research Part B: Methodological*, 2013. doi:https://doi.org/10.1016/j.trb.2013.08.012.

[8] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Experimental Algorithms*. Springer, 2008. ISBN 978-3-540-68552-4.

[9] Brady Hunsaker and Martin Savelsbergh. Efficient Feasibility Testing for Dial-a-Ride Problems. *Operations Research Letters*, 2002. doi:10.1016/S0167-6377(02)00120-7.

[10] Moritz Laupichler and Peter 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)*, 2024. doi:10.1137/1.9781611977929.6.

[11] Shuo Ma, Yu Zheng, and Ouri Wolfson. T-Share: A Large-Scale Dynamic Taxi Ridesharing Service. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013. doi:10.1109/ICDE.2013.6544843.

[12] Carlo Manna and Steve Prestwich. Online Stochastic Planning for Taxi and Ridesharing. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, 2014. doi:10.1109/ICTAI.2014.138.

[13] Neda Masoud and R. Jayakrishnan. A Real-Time Algorithm to Solve the Peer-to-Peer Ride-Matching Problem in a Flexible Ridesharing System. *Transportation Research Part B: Methodological*, 2017. doi:https://doi.org/10.1016/j.trb.2017.10.006.

[14] Harilaos N Psaraftis. A Dynamic Programming Solution to the Single Vehicle Many-to-Many Immediate Request Dial-a-Ride Problem. *Transportation Science*, 1980. doi:10.1287/trsc.14.2.130.

[15] Dominik Schultes. Route Planning in Road Networks. *PhD thesis, Universität Karlsruhe (TH)*, 2008. doi:10.5445/IR/1000007755.

[16] Mitja Stiglic, Niels Agatz, Martin Savelsbergh, and Mirko Gradisar. The Benefits of Meeting Points in Ride-Sharing Systems. *Transportation Research Part B: Methodological*, 2015. doi:https://doi.org/10.1016/j.trb.2015.07.025.

[17] Jiachuan Wang, Peng Cheng, Libin Zheng, Lei Chen, and Wenjie Zhang. Online Ridesharing with Meeting Points. *Proceedings of the VLDB Endowment*, 2022. doi:10.14778/3565838.3565849.

[18] Dominik Ziemke, Ihab Kaddoura, and Kai Nagel. The MATSim Open Berlin Scenario: A Multimodal Agent-Based Transport Simulation Scenario Based on Synthetic Demand Modeling and Open Data. *Procedia Computer Science*, 2019. doi:https://doi.org/10.1016/j.procs.2019.04.120.