Master Thesis

Public Transportation in Rural Areas: The Clustered Dial-a-Ride Problem

Fabian Feitsch

Date of Submission: October 5, 2018
Advisors: Prof. Dr. Sabine Storandt
Prof. Dr. Alexander Wolff

Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen, Komplexität und wissensbasierte Systeme

Abstract

This thesis introduces the *Clustered* Dial-a-Ride problem, which is NP-hard. In practice, the Clustered Dial-a-Ride problem occurs in the domain of public transportation in rural areas. Every village is represented by one cluster. A restriction of the structure of allowed solutions yields a fixed parameter algorithm whose parameter represents the size of the clusters. This makes the Clustered Dial-a-Ride problem solvable faster in practice than the original Dial-a-Ride problem. It turns out that the restriction on the set of solutions does in many realistic cases not play a role. In order to decide whether a specific instance allows the restriction, a classifier is presented. Experiments show that the classifier becomes more accurate the wider the villages lie apart. The classifier's recall is above 80 percent for instances with six passengers and a mean distance of eight kilometers between the villages.

Zusammenfassung

Diese Arbeit führt das *Clustered* Dial-a-Ride Problem ein, welches NP-schwer ist. In der Praxis taucht das Clustered Dial-a-Ride Problem bei der Umsetzung von öffentlichem Nahverkehr im ländlichen Raum auf, wobei Ortschaften durch Cluster abgebildet werden. Durch eine Einschränkung der erlaubten Lösungen wird ein Festparameter-Algorithmus möglich, dessen Parameter die Größe der Cluster ist. Damit ist das Problem in der Praxis schneller lösbar als das traditionelle Dial-a-Ride Problem. Experimente zeigen, dass die Einschränkung der Lösungsmenge in vielen realistischen Fällen keine relevante Rolle spielt. Um zu entscheiden, ob eine gegebene Instanz eine solche Einschränkung erlaubt, wird ein Klassifikator präsentiert. Der Klassifikator wird akkurater, je weiter die Ortschaften auseinanderliegen und erreicht eine Trefferquote von über 80 Prozent für Instanzen mit sechs Passagieren und mittleren Abständen von acht Kilometern zwischen den Ortschaften.

Acknowledgement

Working on this thesis was real fun and I am grateful for the support I received by the members of the Chair, especially by Sabine Storandt. I thank also my partner Sonja Hohlfeld for the solace in times the programs made what I said they should do instead of doing what I meant them to do. Thanks also go to my father, Dieter Feitsch, who proofread the thesis and with whom I had the one or other fruitful discussion.

Contents

1 Introduction

Bus stops are lousy. Often, they must be reached by foot. Sometimes, one feels insecure waiting at night for the bus. Always, standing at a bus stop is wasted time, which becomes even more inconvenient if the weather becomes unfriendly. In the city, these aspects are not as severe as in villages: The density of bus stops is higher, therefore walking distance is reduced. There are lights and other people around. And bus stops in the city are highly frequented, so the waiting time is not that long. Nowadays, villages are often connected to the cities with traditional bus lines using normal bus stops, even though the thoughts above indicate that standard bus lines are not well suited to be used in less densely populated areas.

This thesis suggests a different mode of public transportation in rural areas. As mentioned above, bus stops are often only located on the main street of the village and the walking distance to the bus stop can be very long. Figure 1.1 shows such a situation. The bus line goes from south to north and the colored dots indicate source and destination of the customers. The red lines show the distance they have to walk to and from the bus stops. However, it is not helpful to increase the density of stops inside the villages. The buses had to drive detours through the village to serve those stops even if no customer is waiting there. Due to the small number of customers this is supposed to happen often.

The complementary means of transport solves these problems: By using taxi cabs, the customers can be fetched at their doorstep and brought exactly to their destination. No time is wasted at the bus stop and there is no need to walk through raining weather. But taxi cabs are nor as cheap as public buses neither are they very environment-friendly because they carry only one passenger at once.

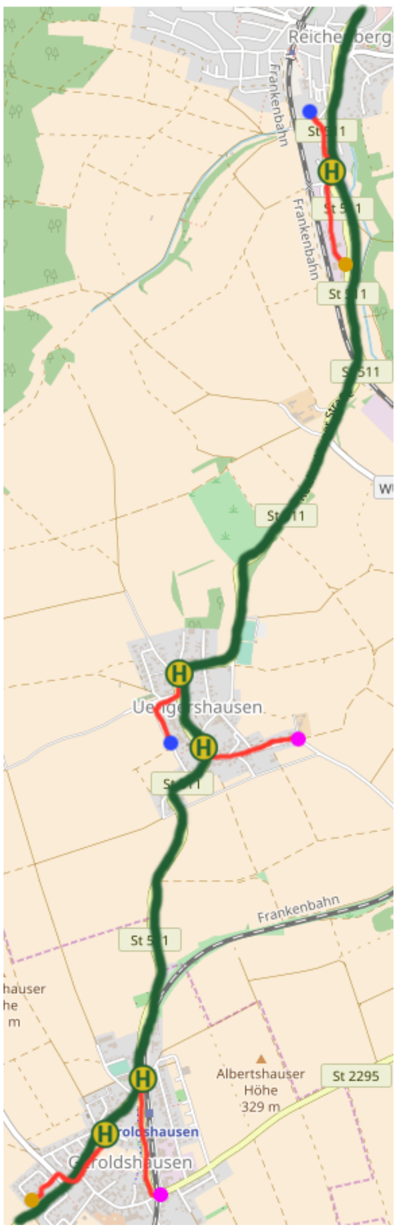


Fig. 1.1: A traditional bus line through villages. The red lines indicate walking distances of customers.

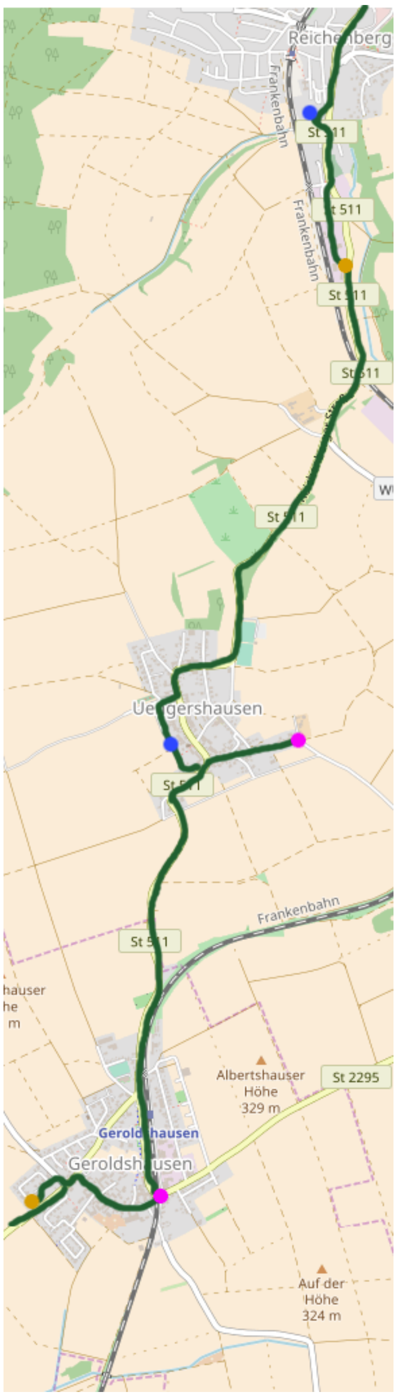


Fig. 1.2: The optimal route to serve the customers from the doorstep. It is only slightly longer than the traditional bus line.

It turns out that a combination of both means of transport is sensible: The customers are served not at static bus stops but rather at their precise venues, yet, unlike taxi cabs, other passengers may board or unboard during the journey. Naturally, the customers do not want to experience big detours to fetch or deliver foreign people, but they may accept little deviations of the shortest path. An instance is shown in Figure 1.2. The customers are identical to Figure 1.1, but the vehicle serves the customers directly at their venues. The detours experienced by the the customers are small: The yellow customer is on board while fetching the violet customer in the southern village. The extra journey for the blue dot in the northern village is still more neglectable.

The task of finding a route that minimizes the sum of all detours to serve a set of customers is called the *Dial-a-Ride* problem. Unfortunately, this problem is known to be computationally hard because it is a close relative to the well known Traveling Salesperson Problem. This and other kinsmen are glanced onto in Section 2. There exist slow exponential time algorithms to solve the Dial-a-Ride problem which are described in Chapter 3. However, there is one observation which makes the case at hand special: All customers are located inside villages. Intuitively, the cheapest tour is supposed to serve the villages along the main roads. This property would make the problem easier than the general variant. But the intuition is wrong. There are cases in which the optimal tour does not use the main road connecting the villages unidirectionally. Instead, it goes back and forth between the villages. In order to grasp and describe such instances the Clustered Dial-a-Ride problem is introduced in Chapter 4. In the same chapter, the central contribution of the thesis can be found: A classifier which decides for a given set of customers and villages if the optimal tour is unidirectional. Chapter 5 explains how the algorithms presented in the two proceeding chapters were implemented technically and Chapter 6 examines realistic examples and evaluates the classifier. The last section of the thesis concludes the findings made in the preceding sections and gives proposals to future work.

2 Related Work

The introduction used the term *Dial-a-Ride* problem, which is the central algorithmic topic of this thesis. This chapter locates the Dial-a-Ride problem in its surrounding literature.

When talking about the Dial-a-Ride problem, one must take care because there are many different variants discussed in literature. The most basic of these variants resembles the Traveling Salesperson Problem (TSP) very strongly [31]. In a (euclidean) TSP instance, there are several points in the plane. The task is to find a roundtrip through all points that is as short as possible. This roundtrip must not contain any junctions or branches. The problem is very hard to solve and it remains hard even if only a simple path (not a roundtrip) is wanted. A small modification of the TSP is to organize the points in ordered pairs. For every pair, an order is defined in which both points of the pair must be visited. This variant of the TSP is the simplest representative of the Dial-a-Ride problem. It is also called *TSP with Pickup and Delivery* (TSPPD). The goal remains to find the shortest path containing all points, but the visiting order must be maintained. This problem can be generalized to the *TSP with Precedence Constraints*, where the order of some points (not necessarily pairs) must be obeyed [2]. Figure 2.1 shows a normal TSP instance and a TSPPD instance. It is easy to see that the route solving the TSP instance does not solve the TSPPD instance. In 2008, Irina Dumitrescu et. al. tackled this variant of the Dial-a-Ride problem with an ILP based branch-and-cut algorithm [13]. Their algorithm is able to solve instances of up to 35 pairs in several minutes, more than doubling previous results [23]. But this TSPPD setting neither does punish detours of customers nor obeys the number of seats in the vehicle. These two additional constraints, however, occur relatively early in the history of the TSP. In 1959, the father of linear programing, Georg Dantzig, formulated the Vehicle Routing Problem (VRP) [8]. The task is to deliver a certain amount of goods from a fixed depot to customers. He introduced a capacity constraint for the vehicles and solves the problem with ILP, too. The paper also mentions that other constraints and target functions can be incorporated easily into the VRP. For example, one can add a last-in-first-out requirement on the tour. In this setting, every unloading must occur in reverse order of the loading. Jean-François Cordeau solves this kind of problem with a branch-and-bound algorithm [4]. On the other hand, it is also possible to ask for the first-in-first-out property where the first customer to be fetched must also be the first customer to be delivered. Ideas to solve this problem can be found in work by Carrabs et al. [3].

Most of the publications mentioned above focus on the transportation of goods. They mainly optimize and restrict the feasible routes in a way that allows efficient cargo delivery. If paying customers are to be transported, then other aspects become important and

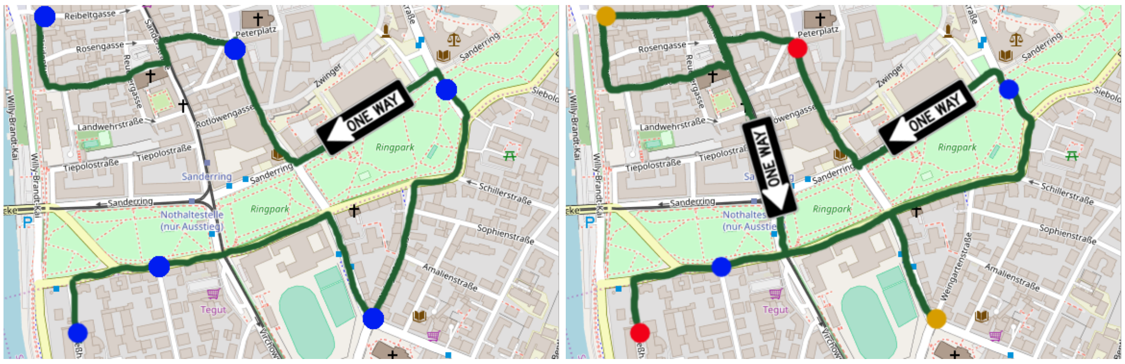


Fig. 2.1: The left image shows a TSP instance and a route visiting all points. In the right image, the points are paired. For every pair, the left point has to be visited before the right point. The tour of the TSP can not be used to visit the points in the correct oder.

the problem is refered to as Dial-a-Ride problem. In 1980, Harilaos Psaraftis augmented the well known Held-Karp algorithm [18] for solving the TSP problem so that it is able to compute an optimal route for the Dial-a-Ride problem with sophisticated constraints and target functions [29]. Originally, his implementation obeys a capacity constraint and punishes detours as well as waiting times of the customers. It also incorporates an order in which the customers requested the service. This ensures that no customer waits forever if other customers at more convenient locations keep posting requests. However, it is easy to change constraints and target function in Psaraftis' algorithm without making it more complex. In Section 3.2, this algorithm is explained in detail. It is the fundamental algorithm of this thesis.

The aforementioned variants of the TSP problem, including the Dial-a-Ride problem, share the property that they are NP-hard. A reduction from TSP (which is known to be NP-hard [15]) to the TSPPD works as follows: Take a TSP instance and duplicate every vertex so that the original vertex and its copy reside in the same location. For every such pair any feasible route must visit the original vertex first, then its copy. Every tour in the TSPPD instance can be transformed into a TSP tour by deleting the copy vertices. In the optimal TSPPD tour no detours are made because every pair of vertices is visited consecutively. Consequently, the reduction can also be used to show the NP-hardness of the Dial-a-Ride problem, which incorporates the minimization of detours in its objective function.

The Dial-a-Ride problem is part of the algorithmic scope of the ride sharing problem. This problem is characterized by drivers owning vehicles and offering tours between two locations and customers without vehicles who search a suitable offer to travel to their destination. Examples for ride sharing platforms are BlaBlaCar [27] and Uber [22]. However, their applications are different from the Dial-a-Ride or TSPPD problem described above. The main task of those platforms is to match requests to offers such that the detours of the offering drivers are not too big. A method to find a good matching is presented by Geisberger et. al. [16]. Their algorithm is meant to be applied by ride sharing platforms mainly addressing private drivers wanting to reduce the costs of single journeys. On the other side, there are companies like Uber, as well as traditional taxi companies, which have to assign requests to their vast fleet of vehicles in real time. Often, these companies operate in metropolitan areas where it is desirable that the customer is fetched only minutes after posting the request. For these high-demand environments Alonso-Mora et. al. propose a heuristic matching method [1].

After the matching of offers and requests is done, the actual route is either computed with an exhaustive search (if the capacity of the vehicle is small), or heuristics are used [1]. Another way to find a route after the matching has been done is to agree on one single destination [26]. This use-case is motivated by touristic points of interests. Several tourists traveling to a city together agree on one venue in the city where they all start their sightseeing tour individually.

As can been seen, the Dial-a-Ride problem is only a subproblem of the bigger setting of ride sharing. However, *exact* algorithms to find the optimal route once the requests have been matched to offers are rarely used. In the rural setting of this work, the matching of requests is done naturally because the village is served by one bus line and for every customer inside a village it is clear which line to use. If the bus line is replaced by a more dynamic vehicle, it is still clear which customer has to take which vehicle. This thesis does not cover changing bus lines during the journey. However, it can be useful to aggregate customers with many small vehicles to a assembly point where they transfer to a bigger vehicle. Drews and Luxen [12] examine ride sharing with hops. However, they pose two limitations on their approach: The hopping stations are pre-defined and the capacity of the vehicle being regarded is two. It is not clear whether their approach can be incorporated in the Dial-a-Ride instance considered in this thesis.

As mentioned in the introduction, the goal of this thesis is not only to solve the Dial-a-Ride problem, but to exploit the fact that all customers are located inside villages. In other words, the instances to be solved are clustered. The author is not aware of any existing literature about the Clustered Dial-a-Ride problem. However, there is work on the clustered version of the traditional TSP problem (CTSP). In CTSP, the points are partitioned in predefined clusters and all points inside one cluster must be visited consecutively. Ding et. al. present a genetic algorithm that gives good heuristic results on CTSP instances [10]. Unfortunately, the optimal TSP tour through a CTSP instance often does not obey the clusters, so the lengths of the shortest routes of TSP and CTSP differ in general. Another approach is to find the clusters *while* computing a good TSP tour. Schneider et. al allow small deteriorations of the optimal route [30] while computing natural clusters. However, despite their similar nature to the Dial-a-Ride problem in rural areas, both approaches are not useful to find exact solutions.

The next section formulates the Dial-a-Ride variant used in this thesis in detail. Then, three different approaches to solve the Dial-a-Ride problem are presented. The clustered variant of the Dial-a-Ride problem is covered in Section 4.

3 The Dial-a-Ride Problem

As shown in the last chapter, the name Dial-a-Ride problem is not unique and there are many different ways to optimize a route. There are also various ways to limit the number of feasible routes, for example by defining ordering constraints or introducing vehicle capacities. This chapter describes the precise nature of the Dial-a-Ride problem concerned in this thesis. Then, the algorithm developed by Psaraftis [29] is explained in detail. This algorithm is already very powerful, but its recursive structure has some disadvantages, especially in storage management. A non-recursive variant is presented in this chapter, too, which admits a more economic usage of space. The last section of the chapter consists of a excursion to an ILP formulation of the Dial-a-Ride problem.

3.1 Problem Definition

The central part of all variants of the TSPPD, respectively. Dial-a-Ride problems are the customers. In literature, there are often called *riders*. Apart from the riders, a special person is introduced: the *driver* conducts the vehicle. A Dial-a-Ride instance is a triple $I = (n, S, [d_{i,j}])$. The variable n is the number of riders, S is the number of seats in the vehicle being used. Every person contributes two points to the instance, her pickup point and her dropoff point. The exact coordinates of these points are not relevant but rather the distances between all pairs of points (i, j) play an important role. They are stored in the distance matrix $[d_{i,j}]$. Since there are n riders plus one driver, there are $2n+2$ different points. Let $m = n+1$ be the number of persons in the instance. Hence, the distance matrix has the size $2m \times 2m$. Technically, the number n is encoded in the size of the matrix $[d_{i,j}]$ but stating n explicitly in I makes its relevance clearer. Similar as in Psaraftis' work, all points induced by I are ordered in a special way: The location with index 0 always refers to the starting point of the driver, and the location with index m is his target point. The pickup point of the first rider has index 1 and the dropoff point of the first rider has index $m + 1$. Generally, the pickup point of the ith rider is found at index i and the corresponding dropoff point at index $m+i$. Notice that the riders are counted starting from 1, since 0 is reserved for the driver. The following example makes this indexation clear.

Example. Let $n = 5$, i. e. there are five riders. Then there are 12 points in total and $[d_{i,j}]$ has size 12×12. The entry $d[4, 7]$ refers to the distance between the pickup point of the forth rider and the dropoff point of the first rider (since $1 = 7-n+1 = 7-m$). On the other hand, the entry $d[4, 9]$ is the distance between the pickup and dropoff location of the forth rider.

A *location* (of I) is an element from the interval $[0, \dots, 2m-1]$. A *tour* or *route* T is a permutation of all locations. Let $p_T(i)$ be an indicator variable which is 1 if the ith step of a tour is a pickup point; steps being counted starting from 1. Analogously, $d_T(i)$ indicates whether the ith step is a dropoff point. Both values can be computed easily:

$$p_T(j) = \begin{cases} 1 \text{ if } T[j] < m \\ 0 \text{ else} \end{cases} \qquad d_T(j) = \begin{cases} 1 \text{ if } T[j] \geq m \\ 0 \text{ else} \end{cases} \qquad = 1 - p_T(j)$$

Then $k_T(i) = \sum_{j=1}^{i} p_T(j) - d_T(j)$ counts the number of persons inside the vehicle after the ith step of a route T. The route T is *feasible* if three conditions are met. First, it begins with location 0 and ends with location m. Second, for dropoff point d in T, the corresponding pickup point $p = d - m$ must precede d in T. Third, for every possible step $1 \leq i \leq 2m$ the inequality $k_T(i) \leq S$ must hold. These constraints result in the following properties which all feasible routes share (given $n > 0$): $k_T(1) = 1$ since the driver is boarded first, $k_T(2) = 2$ because the second step must always be a pickup, $k_T(2m-1) = 1$ follows from the last rider to be dropped and $k_T(2m) = 0$ marks the end of the journey.

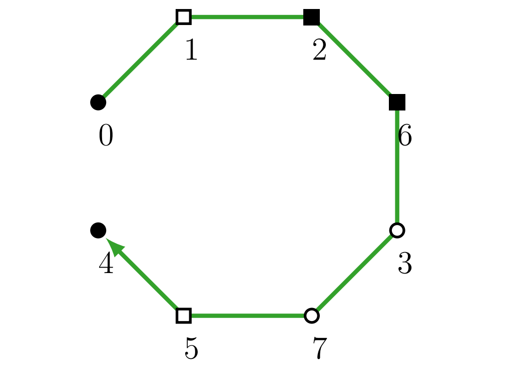
Example. Let $T = [0, 1, 2, 6, 3, 7, 5, 4]$ be a route. Since T consists of 8 steps, $m = 4$. After picking the first rider (1), the second rider is picked (2). Then the second rider is dropped ($2 + 4 = 6$) and the third rider is picked (3). In the sixth step, the third rider is delivered (7). On all these journeys rider 1 was in the vehicle and is not dropped until the last-but-one step (5). Consequently, this route is only feasible if $S \geq 3$ since at most three persons are sitting in the vehicle at once. The route $T' = [0, 7, 2, 6, 3, 1, 5, 4]$ is not feasible because rider 3 is dropped before she was picked.

Finding such a feasible route is easy. The permutation $T = [0, 1, m+1, 2, m+2, \dots, m]$ is always allowed. It means that every rider is fetched and delivered one after another, similar to taxi cabs. But this tour might not be satisfying. Naturally, one wants to minimize the costs of a tour T. There are three intuitive possibilities to define tour costs.

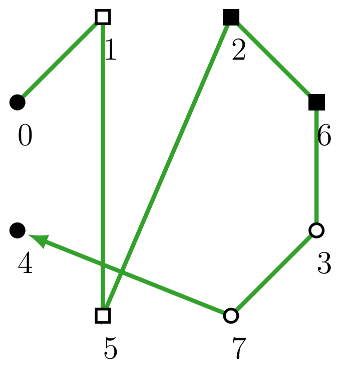
Distance being driven by the Driver Given a tour T, the distance driven by the driver is given by the following summation. Figure 3.1a shows an instance and a tour which optimizes this costs function.

$$c(T) = \sum_{i=2}^{2m} d[T[i-1], T[i]]$$

Finding the tour minimizing these costs may be useful if the entities being transported are goods. Humans, on the other side, do not want to travel along huge detours, especially to serve foreigners. Thus, this criterion is not used in this thesis.



(a) This route minimizes the distance driven by the driver. But the first rider is not happy



(b) This route minimizes the total distance that is driven. No rider is exposed to detours.

Fig. 3.1: A simple Dial-a-Ride instance with two different routes. The two locations belonging to the same rider are drawn with identical shapes. The numbers of the locations reflect the order of the locations inside the distance matrix $[d_{i,j}]$. In this example, all distances are euclidean.

Total Person distance driven Unlike the previous costs, the costs based on total distance contain the sum of all distances experienced by any person inside the vehicle. The resulting optimal tour can differ compared to the tour optimizing the previous costs, as depicted in Figure 3.1b. The costs function is realized by multiplying the distance with the number of persons inside the vehicle after every step:

$$c(T) = \sum_{i=2}^{2m} k_T(i-1) \cdot d[T[i-1], T[i]]$$

This costs function is sensible in the setting of this thesis because it does only punish detours, but not waiting times of the passengers. Waiting times need not to be taken into account because the aim of this thesis is to improve bus lines, which naturally have fixed departure times. Therefore, riders are willing to wait at home until they are fetched, even if the vehicle sometimes comes several minutes later than normal.

Total distance driven and waiting time The last costs function takes waiting times into account. The numbers of passengers waiting after step i of a route T is given by $k'_T(i) = \sum_{j=i+1}^{2m} p_T(j)$. Distance and waiting time are regarded to be equal. This is not quite accurate because the vehicle does not drive constant speed, but it simplifies the model without making totally insensible assumptions.

$$c(T) = \sum_{i=2}^{2m} \left(k_T(i-1) + k'_T(i-1)\right) \cdot d[T[i-1], T[i]]$$

Psaraftis uses a deviation of this objective function in his original paper, which also incorporates a weighting between travel time and waiting time as well as a parameter describing customers' preferences. As mentioned above, the second objective function suits better to the use case at hand. Thus, the total person distance driven objective is used in the thesis, except stated differently. All three objective functions evolve naturally from Psaraftis' original objective function by setting selected parameters to 1 or 0, so no major modifications need to be made.

The first attempt to find an optimal route is to enumerate all feasible routes. As with its close relative TSP, this approach is not bearable for the Dial-a-Ride problem. Low values for the capacity S reduce the number of feasible routes, but it is easy to see that $n!$ is a lower bound on the number of feasible routes if $S = 2$: Serving the riders one after another yields $n!$ possible routes. If $S > m$, then the exact number of feasible routes is given by the following theorem:

Theorem 3.1. *Given n riders, there are $\Pi_{i=1}^{n}(2i^2 - i)$ permutations in which every pickup occurs before the corresponding drop-off.*

Proof. The theorem can be proven by induction. For $n = 1$ there is only one allowed permutation and $2 \cdot 1^2 - 1 = 1$. The statement is proven for an arbitrary n under the assumption that it is correct for $n - 1$. Let N be the number of permutations for $n - 1$ riders. Since the statement is correct for $n - 1$ the value for N is $N = \Pi_{i=1}^{n-1}(2i^2 - i)$. Select one arbitrary permutation P of these N permutations. Then there are $\sum_{i=1}^{2(m-1)-1} i$ possibilities to incorporate an additional rider into P. To see this, fix the pickup of the new rider to be after the first step of P, which is picking up the driver. Then there are $2(m - 1) - 1$ ways to locate the dropoff in P because the last step cannot be a dropoff of a rider. In general, if the pickup is fixed to happen after the ith step in P, then there are $2(m - 1) - i$ ways to arrange the dropoff. The sum accumulates the possibilities for every legal choice of i. Applying Gauss yields

$$\sum_{i=1}^{2(m-1)-1} i = \frac{2(m-1)(2(m-1)-1)}{2} = \frac{(2m-2)(2m-3)}{2}$$

$$= \frac{4m^2 - 10m + 6}{2} = 2m^2 - 5m + 3 = 2(m-1)^2 - (m-1).$$

There are N ways to select P, therefore, there are $N \cdot \big(2(m-1)^2 - (m-1)\big)$ ways to serve an additional rider. Substituting $m - 1$ with n yields the term $N \cdot (2n^2 - n)$. Since $N = \Pi_{i=1}^{n-1}(2i^2 - i)$, the number of feasible routes for n riders is $\Pi_{i=1}^{n}(2i^2 - i)$. □

It holds that $\Pi_{i=1}^{n}(2i^2 - i) > n!$ and $n! \in 2^{O(n \log n)}$ for sufficiently large n. Thus, every exponential time algorithm has a better performance than this brute force approach.

Before such an exponential time algorithm is introduced, the definition of the Dial-a-Ride problem is extended. This extension makes it easier to understand the algorithms presented in the next section and is necessary in Section 4 where an algorithm is needed to solve Dial-a-Ride instances *partially*.

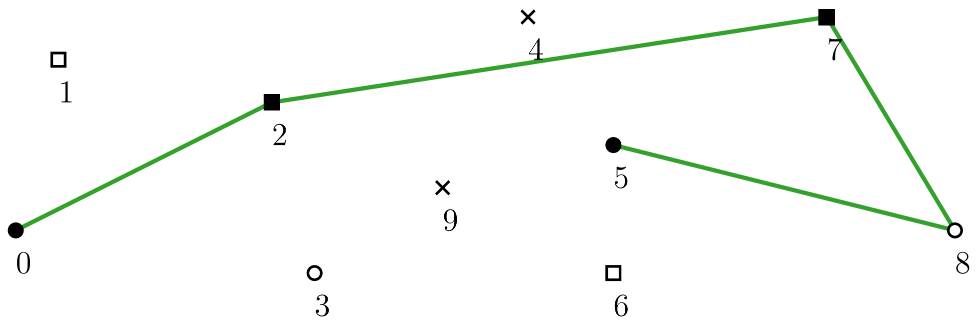
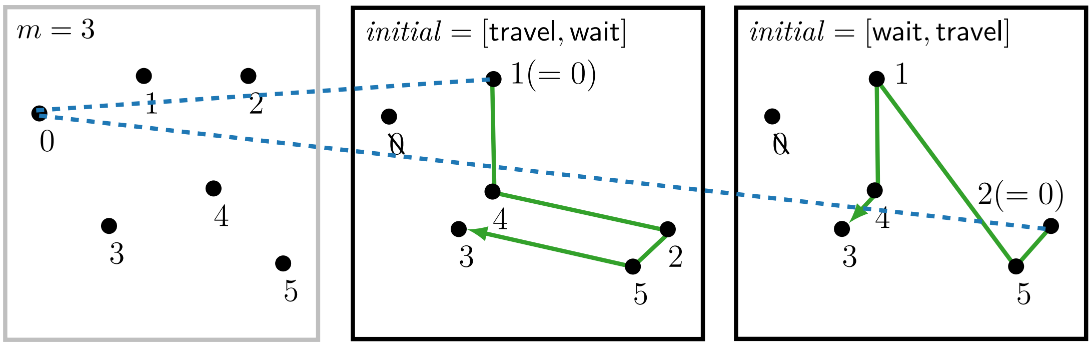


Fig. 3.2: A feasible (possibly not the best) tour visiting only necessary locations of the partial Dial-a-Ride instance $I' = (4, S, [\mathsf{finish}, \mathsf{wait}, \mathsf{travel}, \mathsf{wait}], [\mathsf{finish}, \mathsf{finish}, \mathsf{finish}, \mathsf{wait}])$.

Therefore, the term *partial Dial-a-Ride problem* is defined. A partial Dial-a-Ride instance $I' = (n, S, [d_{i,j}], \mathit{initial}, \mathit{final})$ is a quintuple. The first three properties are the same as in the normal Dial-a-Ride instance I. The new properties *initial* and *final* are both arrays of length n, indexed at 1. Every cell of these two arrays contains one element of $\{\mathsf{wait}, \mathsf{travel}, \mathsf{finish}\}$. These elements encode states of the riders. A wait at location i means that the rider i is waiting at his pickup point. When the driver i is in the vehicle, then the ith entry contains a travel. The element finish means that the rider is delivered. Such arrays are referred to as *state vectors* or *state arrays* in this thesis. Given a partial Dial-a-Ride instance I', the goal is to find a cost optimal route through the locations such that when starting with *inital*, the states in *final* evolve. The route still has to start at location 0 and end at location m.

Example. Let $I' = (4, S, [d_{i,j}], [\mathsf{finish}, \mathsf{wait}, \mathsf{travel}, \mathsf{wait}], [\mathsf{finish}, \mathsf{finish}, \mathsf{finish}, \mathsf{wait}]$ be a partial Dial-a-Ride instance. The *inital* array states that the first rider is already delivered, the second and the forth rider are still waiting and the third rider is in the vehicle. The aim is to find a tour T such that after traveling T the first three riders are delivered and the forth rider is still waiting. All tours satisfying this property are permutations of the locations $[0, 2, 5, 7, 8]$ because the other locations must not be touched. A sample tour is depicted in Figure 3.2.

Of course, the *inital* and *final* states must be compatible, otherwise there is no solution. If $\mathsf{wait} > \mathsf{travel} > \mathsf{finish}$, this condition can be expressed by $\mathit{initial}[i] \geq \mathit{final}[i]$ for every $1 \leq i \leq n$. Given a normal instance $I = (n, S, [d_{i,j}])$, its corresponding partial version is $I' = (n, S, [d_{i,j}], [\mathsf{wait}]^n, [\mathsf{finish}]^n)$. Although the ordering of the locations and the state vectors were introduced by Psaraftis, his work did not mention partial Dial-a-Ride instances. Despite that, his algorithm can natively compute partial Dial-a-Ride instances without major modifications. Because his algorithm plays an important role in this thesis, the entire next section is dedicated to his algorithm. However, Psaraftis did restrict the set of feasible routes by obeying the order in which the riders posted their requests. For example, the fourth rider that posted a request must be picked up between steps $4 - c$ and $4 + c$ where c is a constant. This restriction does not take place in this thesis because it does not fit to the use case addressed in Chapter 4.



$m = 3$
0
1
2
3
4
5
$initial = [\mathsf{travel}, \mathsf{wait}]$
$1(=0)$
0
4
3
2
5
$initial = [\mathsf{wait}, \mathsf{travel}]$
1
0
4
$2(=0)$
3
5

Fig. 3.3: The left box shows a Dial-a-Ride instance I, the other two boxes show partial instanes I' in which the location 0 was moved accordingly. The optimal route for I can be obtained by comparing the sum of costs of connecting location 0 to the subroutes and the costs of the subroutes themselves.

3.2 Psaraftis' Algorithm

When thinking about the partial Dial-a-Ride problem, one can recognize the recursive nature of the problem. Let $I = (n, S, [d_{i,j}])$ be a Dial-a-Ride instance. Suppose that the first i steps of the optimal tour T^* are known and the remaining steps are still unknown. The goal is to find an optimal tour through the remaining locations only. This task can be represented as a partial Dial-a-Ride instance whose *initial* state is defined by the already visited locations. This thought results in an approach to solve I: Guess the second step of T^* (the first is always visiting location 0) and solve the remaining partial instance recursively. There are n guesses in total, of which the best one is chosen.

An example with two riders is shown in Figure 3.3. The leftmost box shows the original instance, the two other boxes show the optimal tour with the second step being guessed. Since partial Dial-a-Ride instances have to start at location 0, too, the location 0 was artificially moved to location 1 and 2 in the recursion. Every recursive call reduces the complexity of the problem by one location. As soon as the there is only one location left, the sub-problem can be solved trivially.

This structure invites to use the *dynamic programing* technique, where a problem is reduced to several smaller problems of the same kind, which are computed and than incorporated into the main result [5]. Psaraftis' algorithm essentially works as sketched above, however, it uses a more elegant way to reduce the problem instead of moving locations around.

The algorithm of Psaraftis is a recursive algorithm. It employs a global associative map $V\colon [0, \dots, 2m-1] \times \{\mathsf{finish}, \mathsf{travel}, \mathsf{wait}\}^n \to \mathbb{R}$ which assigns a rational number to a combination of a location index and a state array. The number $V[l, s]$ represents the minimal costs for reaching the *final* state and then traveling to location m when starting with the location l and state s.

Algorithm 1: Psaraftis' Algorithm to solve the Dial-a-Ride problem

```
Input: A partial Dial-a-Ride instance $I'$
Output: The costs of the cheapest route to solve $I'$
1 $(n, S, [d_{i,j}], initial, final) = I'$            // Extract partial instance $I'$
2 $V$ = new Map() with all entries being initially $\infty$  // Initialize global map $V$
3 PsaraftisRecursive($I'$, 0 , $initial$)               // Call the recursive method
4 return $V[0, initial]$
```

Example. If $\mathit{final} = [\mathsf{finish}]^n$ then $V[6, [\mathsf{travel}] + [\mathsf{finish}]^{n-1}]$ represents the optimal costs when starting at location 6, then delivering the first rider and than driving to m. Notice that $m < 6$ in this instance because all pickups have clearly been visited. The exact value of $V[6, [\mathsf{travel}] + [\mathsf{wait}]^{n-1}]$ is $2d[6, 1 + m] + d[1 + m, m]$, because on the second last leg of the route, two persons are inside the vehicle and the last leg is only driven by the driver.

Psaraftis' algorithm populates V with the correct values such that the properties above are fulfilled after the completion of the algorithm. At that moment, the costs of the optimal route for an instance $I = (n, S, [d_{i,j}])$ is found in the cell $V[0, [\mathsf{wait}]^n]$. For partial instances $I' = (n, S, [d_{i,j}], \mathit{initial}, \mathit{final})$ this value is located in the cell $V[0, \mathit{initial}]$. The algorithm consists of two methods. The first method is a wrapper method that starts a recursion. The second method does the main work and is recursive. These two methods are shown in detail in Algorithm 1 and Algorithm 2. The two pseudocodes do not repeat the original algorithm but rather present a modified version which suits better to the Dial-a-Ride problem investigated in this thesis. The basic idea however was formulated by Psaraftis [29].

Algorithm 2 works as follows. The first if-statement checks that the current state does not violate the capacity constraint. The second if-statement abbreviates the calculation in case the value has already been computed. The third if-statement cancels the recursive algorithm in case the final state is reached. In this case, the cheapest value is only the distance of the current location i to the destination of the driver m, which is saved in V. Otherwise, the computation starts from scratch in the for-loop. The idea is to generate all states which may follow from i and *state*, compute their optimal values and select the successor which yields the minimal costs in combination with i and *state*. This is done by filtering all state entries which are not yet finished. For each of those entries a new location i' a new state state' are constructed. The new state advances the respective rider to the next stage. To this end, the $^{\downarrow}$-operator is utilized. It is a shorthand for a conditional statement and has the following semantic: $\mathsf{wait}^{\downarrow} = \mathsf{travel}$ and $\mathsf{travel}^{\downarrow} = \mathsf{finish}$. Then the recursive calculation is started. At the end of the loop, the best value is saved in V.

The algorithm always terminates because every recursive call represents a smaller instance, until the instances are so small that they are trivial to solve. The running time of the algorithm is $O(n^2 3^{n-1})$. There are $2n3^{n-1} + 2$ entries in V because there are $2n+2$ locations. Except for locations 0 and m, there are 3^{n-1} allowed states for every

Algorithm 2: PsaraftisRecursive(I',i,$state$)

```
Input: A partial Dial-a-Ride instance $I'$, a location index $i$ and a state array $state$
Output: Costs of cheapest route starting at $i$ with state $state$, ending at $final$ of $I'$
1  $(n, S, [d_{i,j}], initial, final) = I'$              // Extract the partial instance $I'$
2  $k$ = Count occurrences of waits in $state$        // Does not contain the driver.
3  if $k + 1 > S$ then return
4  if $V[i, state] < \infty$ then return                       // Value is already cached.
5  if $final == state$ then
6      $V[i, state] = d[i, m]$
7      return
8  $v = \infty$
9  foreach $index, entry \in state$ do
10     if $entry == final[index]$ or $entry ==$ finish then
11         continue
       // The ":"  inside arrays is Phython's slice notation.
12     $state' = state[: index] + [entry^{\downarrow}] + state[index + 1 :]$
13     $i' = \begin{cases} index & \text{if } entry == \text{wait} \\ index + m & \text{if } entry == \text{travel} \end{cases}$     // Computing new location.
14     PsaraftisRecursive($I'$, $i'$, $state'$)
15     if $(k + 1) \cdot d[i, i'] + V[i', state'] < v$ then
16         $v = (k + 1) \cdot d[i, i'] + V[i', state']$
17 $V[i, state] = v$
```

location because the location fixes the state for exactly one rider. Computing the value of a location i and a state $state$ needs $O(n)$ steps. The claimed running time follows.

The alert reader may have noticed that the algorithm described as above does not return the optimal tour, but only the value of the optimal tour. This is typical for dynamic programs. To obtain the actual optimal tour, a second associative map P is introduced. This map stores parents of V's key values. Every time a value v is stored in V, the location and state that caused v are saved in P. Thus at the end of the algorithm P contains pointers. Starting with $P[0, [\text{wait}]^n]$, traversing the pointers and thereby collecting the locations until the location m is reached yields the actual tour.

The algorithm of Psaraftis' is the fundamental algorithm of this thesis. However, the experiments that were conducted are based on another variant of Psaraftis' algorithm than described above. This variant and the reasons for using it instead of the original algorithm are discussed in the next section.

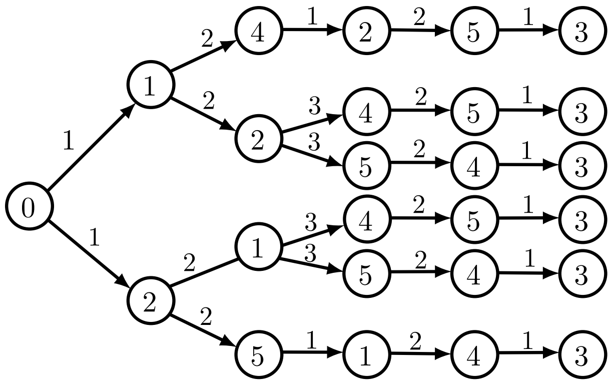


Fig. 3.4: A complete decision tree for two riders. The numbers on the edges indicate the number of persons in the vehicle at the time the edge is taken. If $S = 2$, then the four inner branches would be pruned and there were only two feasible routes.

3.3 Incremental Variant of Psaraftis' Algorithm

A natural way to approach the Dial-a-Ride problem is to illustrate the feasible routes as a decision tree. An example of such a tree for two riders is shown in Figure 3.4. The capacity S of this instance is ∞. In the first step, the driver can choose which rider to fetch first. Then he can either pick an additional rider or drop the recently boarded rider first. Every path in the tree represents a feasible tour. Remember that the locations are referenced by their indices. Thus, the leaves of such a decision tree must always contain the driver's dropoff, which is m. In the example figure $m = 3$ because there are two riders. An algorithm that builds and evaluates such a tree would essentially be a brute force algorithm that enumerates all feasible routes. The number of leaves is predicted by Theorem 3.1. However, the tree can be compressed since many parts of the tree are redundant. Consider two vertices with the same depth that share the same set of predecssors, including themselves. For example, in Figure 3.4 both vertices of location 4 in the penultimate step have the same set of predecessors, but in different order. Everything that happens after these two vertices (i.e. to the right of them) is identical. Therefore, such vertices are merged. The compressed graph of Figure 3.4 is shown in Figure 3.5. Using the compressed directed acyclic graph, Psaraftis' algorithm can be illustrated. It proceeds in a depth-first-search manor through the graph and computes the values from right to left. Every vertex in the graph is a recursive call. The parameter i can be found inside the vertex itself and the parameter *state* can easily be deduced from the predecessors of the vertex. Every edge is traversed exactly one time because cached values are reused.

Another possibility is to compute the values from left to right in a breadth-first-search manor. Starting with the leftmost vertex having value 0, the values of the successors are computed level-wise until the rightmost vertex is reached. The pseudocode is shown in Algorithm 3. A huge share of the algorithm is very similar to the original variant. The main difference is the utilization of a queue to organize the commutated values instead of using recursion, which results in a different interpretation of the map V. The

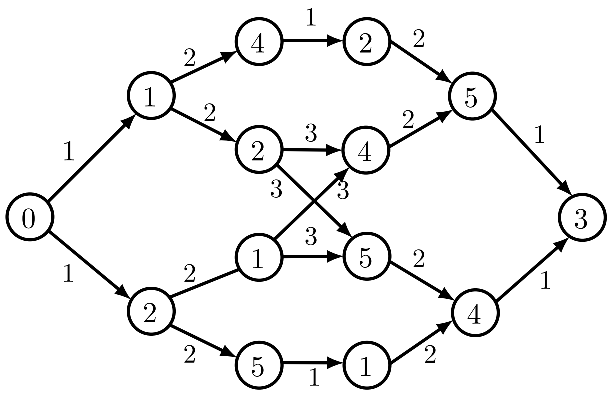


Fig. 3.5: The compressed variant of Figure 3.4. Again, the current vehicle load is shown on the edges. If $S = 2$, the four paths through the center would not exist.

cell $V[l, \mathit{state}]$ contains the costs that are at most necessary to reach location l and state vector *state*. The numbers in V decrease while the algorithm is running and are optimal by the end of the algorithm. It starts by setting $V[0, \mathit{initial}]$ to 0 and storing it in a queue. The main for-loop extracts the states from the queue in first-in-first-out order. For every extracted key all possible succeeding states are generated in the same way as the recursive variant does. However, infeasible states are discarded at once and only feasible states enter the queue. The if-condition in line 19 ensures that no location/state-pair can enter the queue twice. All entries that are removed from Q have their optimal value already set. The overall optimal value is found in $V[m, \mathit{final}]$. Notice that the queue never contains more vertices than the number of maximal vertices in one vertical layer of Figure 3.4.

The reason why this algorithm is preferred over the algorithm from literature is that the author finds it more intuitive and better debugable than the original algorithm. Also, the incremental algorithm seemed to be faster in reality, even if the asymptotic running times are, of course, identical. This observation is not based on thorough benchmarking but rather on some quick checks. Another advantage of the incremental algorithm is that it offers a possibility to save space. While in the original recursive algorithm of Psaraftis every state must be kept because it may be needed later on, the incremental version need only keep two layers in the memory at once. If not only the value of the optimal solution has to be found, than the parents must be saved too, in the same way as they must be stored in the recursive variant. However, some of the states eventually can not be reached any more. These states can also be deleted. Thus, when the algorithm is implemented in a language that supports effective garbage collection, then the advantage of needing less memory than the recursive variant remains.

Despite these fundamental changes to the original algorithm, the idea stays the same. Therefore, the term Psaraftis' Algorithm refers to either of both variants in the remainder of the thesis. If the exact version is important, it will be stated at that point. Before turning to the Dial-a-Ride problem in rural areas, a quick excursion is done. The next section presents an ILP model for the Dial-a-Ride problem.

Algorithm 3: A variant of Psaraftis' algorithm. It proceeds in breadth-first-manor.

```
Input: A partial Dial-a-Ride instance $I'$
Output: The value of the best tour to deliver all riders.
1  $(n, S, [d_{i,j}], \mathit{initial}, \mathit{final}) = I'$
2  $Q$ = new Queue()
3  $V$ = new Map() with all entries being initially $\infty$
4  $V[0, \mathit{initial}] = 0$
5  $Q$.enqueue[$(0, \mathit{initial})$]
6  while $Q \neq \emptyset$ do
7      $i, \mathit{state} = Q$.dequeue()
8      if $\mathit{state} == \mathit{final}$ then
9          $V[m, \mathit{final}] = \min\{V[m, \mathit{final}], V[i, \mathit{state}] + d[i, m]\}$
10         continue
11     foreach $\mathit{index}, \mathit{entry} \in \mathit{state}$ do
12         if $\mathit{entry} == \mathit{final}[\mathit{index}]$ or $\mathit{entry} ==$ finish then
13             continue
14         $\mathit{state}' = \mathit{state}[: \mathit{index}] + [\mathit{entry}^{\downarrow}] + \mathit{state}[\mathit{index} + \mathit{1} :]$
15         $k$ = Count occurrences of waits in $\mathit{state}'$
16         if $k + 1 > S$ then
17             continue
18         $i' = \begin{cases} \mathit{index} & \text{if } \mathit{entry} == \text{wait} \\ \mathit{index} + m & \text{if } \mathit{entry} == \text{travel} \end{cases}$
19         if $V[i', \mathit{state}] == \infty$ then
20             $Q$.enqueue[$i', \mathit{state}'$]
21         $V[i', \mathit{state}]' = \min\{V[i', \mathit{state}'], (k + 1) \cdot d[i, i'] + V[i, \mathit{state}]\}$
22 return $V[m, \mathit{final}]$
```

3.4 Excursion: An ILP for Dial-a-Ride Instances

Linear Programs are a popular technique to tackle optimization problems. If one is able the formulate the problem using linear inequalities and a linear objective function, an algorithm to solve the problem comes for free. The first algorithm to solve linear programs is Dantzig's Simplex algorithm [7] which runs in polynomial time in most cases. Every Dial-a-Ride instance I can be described as a linear program and the variable values computed by the solving algorithm can be translated into the optimal tour. Unfortunately, the Simplex algorithm assigns fractional values to variables so that their interpretation becomes ambiguous. The solution is to force all variables to be integer values. However, if a NP-hard problem is modeled as such an integer linear program (ILP), then solving the ILP becomes NP-hard, too. Thus there is little hope to solve the Dial-a-Ride problem in polynomial time using ILP (assuming $P \neq NP$). Even so, the Dial-a-Ride problem may be one of the problems for which solving the corresponding ILP is faster or easier than developing and running a combinatorial algorithm.

To check this conjecture and to verify the results of the other algorithms, a program was written to translate Dial-a-Ride instances I into integer linear models. The program is written in the Optimization Programming Language [21], a language that creates models for the IBM ILOG CPLEX Optimization Studio [20]. However, it turned out that CPLEX needs much more time for solving an instance than the other two algorithms: While Psaraftis' algorithm needs only a couple of seconds to solve instances of size $n = 8$, CPLEX did not finish after fifteen minutes. Both tests were run on usual desktop computers. Thus, the ILP implementation can be regarded as pilot study with little relevance for the remainder of the thesis. The author is aware that sophisticated methods to improve CPLEX's (and similar algorithms') performance exist, but this is not the main topic of the thesis. Ideas of utilizations of these advanced techniques for the Dial-a-Ride problem can be found in the work by Dumitrescu et. al. [13] and Cordeau [4].

The remainder of the section presents the integer linear model describing a Dial-a-Ride problem. It is important to note that in this linear model the driver is considered to be a rider which means that the rider $r = 0$ is the driver. Given a Dial-a-Ride instance $I = (n, S, [d_{i,j}])$, a linear model can be constructed as follows. Introduce a set of variables $x_{i,r}^{\text{pick}}$ and a set of variables $x_{i,r}^{\text{drop}}$ for $1 \leq i \leq 2m$ and $0 \leq r < m$ and allow only values 0 or 1 for every variable. These variables are used to model the optimal tour. If the solver assigns $x_{i,r}^{\text{pick}} = 0$, then rider r is picked sometime after the ith step in the tour. If, on the other hand, $x_{i,r}^{\text{pick}} = 1$, then the rider is picked at or before step i. Analogously, $x_{i,r}^{\text{drop}}$ is interpreted. The consequence is that the position of rider r can be determined for every fixed step i. If $x_{i,r}^{\text{pick}} = 0$ and $x_{i,r}^{\text{drop}} = 0$ then the rider was not yet fetched at step i of the tour, if $x_{i,r}^{\text{pick}} = 1$ and $x_{i,r}^{\text{drop}} = 0$, then the rider is in the vehicle at step i and when both variables are 1, then the rider is already delivered. It is obvious that this distinction lacks one case: Logically, the case $x_{i,r}^{\text{pick}} = 0$ and $x_{i,r}^{\text{drop}} = 1$ cannot be interpreted because it contradicts the definition of the variables. This asks for a set of constraints that guarantee a feasible route. The following inequalities accomplish this goal. Be reminded that persons are 0-indexed and steps are 1-indexed.

$$x_{i,r}^{\text{pick}} \in \{0,1\} \text{ and } x_{i,r}^{\text{drop}} \in \{0,1\} \qquad \text{for all } i \in [1,2m], r \in [0,m-1] \tag{3.1}$$

$$x_{i,r}^{\text{pick}} \leq x_{i+1,r}^{\text{pick}} \text{ and } x_{i,r}^{\text{drop}} \leq x_{i+1,r}^{\text{drop}} \qquad \text{for all } i \in [1,2m-1], r \in [0,m-1] \tag{3.2}$$

$$x_{i,r}^{\text{drop}} \leq x_{i,r}^{\text{pick}} \qquad \text{for all } i \in [1,2m], r \in [0,m-1] \tag{3.3}$$

$$\sum_{r=0}^{m-1} x_{i,r}^{\text{pick}} + \sum_{r=0}^{m-1} x_{i,r}^{\text{drop}} = i \qquad \text{for all } i \in [1,2m] \tag{3.4}$$

$$\sum_{r=0}^{m-1} x_{i,r}^{\text{pick}} - \sum_{r=0}^{m-1} x_{i,r}^{\text{drop}} \leq S \qquad \text{for all } i \in [1,2m] \tag{3.5}$$

$$x_{1,0}^{\text{pick}} = 1 \text{ and } x_{2m-1,0}^{\text{drop}} = 0 \tag{3.6}$$

The first line advises the solver of the linear program that all variables must be either 0 or 1. The second line ensures that once a variable for rider r at a fixed step i is 1, then it stays 1 for the rest of the tour. In other words, the variables are monotonous with regard to the step i for fixed r. Equation 3.3 forbids that a rider is dropped without being picked before. Technically, this would be the case if $x_{i,r}^{\text{pick}} = 0$ and $x_{i,r}^{\text{drop}} = 1$. Therefore, it is required that the pick variable is always at least as big as the corresponding drop variable. In Equation 3.4 every step is forced to carry out exactly one action. This can be imagined as follows. Every time a person is picked or dropped in a feasible tour, exactly one variable becomes 1. For a certain step i, there must have been i actions carried out so far. Adding up all variables for *this* step has to be equal to i, otherwise too few or too many actions happened so far, which is not allowed. It must also be guaranteed that the capacity of the vehicle is never violated during the tour. Again, this task can be regarded step-wise. For a fixed step, the difference of pick-variabes and drop-variables denote the number of persons in the vehicle at step i, which must not be greater than S. Equation 3.5 realizes this requirement. The last two inequalities (Equation 3.6) distinguish the driver from the riders by forcing the driver to be the first person to be fetched and the last person to be delivered. To illustrate the interaction of these equations, an example with two riders and one driver is given in Figure 3.6. It can be seen that the assignment of the variables fulfill the equations and induce a feasible route. The complexity of the model so far is $O(m^2)$ variables and $O(m^2)$ constraints.

As mentioned earlier, it is easy to find a feasible route without an objective function. Indeed, CPLEX produces feasible routes very fast if the objective function is not stated. Unfortunately, the variables introduced above are not sufficient to state the desired objective function. Therefore, more variables are defined that are only needed to formulate the objective function. The long term aim is to define a variable $x_{(u,v),r}$ being 1 iff person r was in the vehicle at the time the vehicle drove from location u to location v. Using this variable, the objective function can be stated with Equation 3.7.

Person	Variable	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
Driver	$x_{i,0}^{\text{pick}}$	1	1	1	1	1	1
	$x_{i,0}^{\text{drop}}$	0	0	0	0	0	1
Rider 1	$x_{i,1}^{\text{pick}}$	0	1	1	1	1	1
	$x_{i,1}^{\text{drop}}$	0	0	0	1	1	1
Rider 2	$x_{i,2}^{\text{pick}}$	0	0	1	1	1	1
	$x_{i,2}^{\text{drop}}$	0	0	0	0	1	1
Location of Vehicle		0	1	2	4	5	3
Equation 3.4		1	2	3	4	5	6
Equation 3.5		1	2	3	2	1	0

Fig. 3.6: A feasible assignment of the $x_{i,r}^{\text{pick}}$ and $x_{i,r}^{\text{drop}}$ variables. The route induced by this assignment is: pick 0, pick 1, pick 2, drop 1, drop 2, drop 0. The last two lines illustrate the referenced constraints.

$$\min \sum_{(u,v)\in[0,2m-1]^2} \sum_{r=0}^{m-1} x_{(u,v),r} \cdot d[u,v] \tag{3.7}$$

The matrix $[d_{i,j}]$ is that of the instance I. When a connection (u,v) is never used in a tour, the variable $x_{(u,v),r}$ is 0 for every rider and the edge is not included in the objective value. The tuple (u,v) will be abbreviated as edge e in the remainder of the section. This remainder explains how $x_{(u,v),r}$ is realized.

Let $e \in [0, 2m-1]^2$ be an edge between two locations. Then the value 1 should be assigned to the binary variable $x_{i,e}$ if the edge e is used directly after the ith step. In Figure 3.6 the edge (2,4) is taken between steps 3 and 4, so $x_{3,(2,4)} = 1$. Location 2 is the pickup location for the second rider and 4 is the dropoff location of the first rider. In order to assign the correct value to $x_{i,e}$ it is important to distinguish four edge types: An edge is either a connection between two pickup locations or two dropoff locations or a combination of them. This thesis explains how the value $x_{i,e}$ is obtained for edges connecting two pickup locations, the other types work with the same argumentation. Let $e = (u,v)$ be an edge with u and v both being pickup locations of riders r and s. With other words, $u < m$ and $v < m$, $u = r$ and $v = s$. There is only one possible constellation which allows the vehicle to use edge e at step i. Figure 3.7 shows that situation. The values for the pick-variables must look like those depicted in the table. The trick to formulate a constraint for $x_{i,e}$ is to add up all variables that ought to 1 and to subtract all variables that must not be 1. If this calculation yields the value 3, then $x_{i,e}$ must be 1. The values for the drop-variables can be ignored because they are forced to be 0 by the Constraints 3.2 and 3.3. Constraint 3.8 attempts to realize $x_{i,e}$.

Person	Variable	Step $i-1$	Step i	Step $i+1$
		$\vdots$		
Rider r	$x_{i,r}^{\text{pick}}$	0	1	1
	$x_{i,r}^{\text{drop}}$	0	0	0
Rider s	$x_{i,s}^{\text{pick}}$	0	0	1
	$x_{i,s}^{\text{drop}}$	0	0	0
		$\vdots$		

Fig. 3.7: In this situation, the vehicle uses the edge (u, v) at step i. There is no other way to assign the variables and keep the edge used in step i.

$$x_{i,e} \geq -x_{i-1,r}^{\text{pick}} - x_{i-1,s}^{\text{pick}} - x_{i,s}^{\text{pick}} + x_{i,r}^{\text{pick}} + x_{i+1,r}^{\text{pick}} + x_{i+1,s}^{\text{pick}} - 2 \tag{3.8}$$

$$\text{for all } i \in [2, 2m-1], e \in [0, 2m-1]^2$$

The constraints for the other three types of edges have the same structure, the only difference is that x^{pick} must be replaced by x^{drop} accordingly. The variables of the form $x_{1,e}$ must be handled specially to avoid index out of bounds errors. It is granted that the first edge to be used connects location 0 and another pickup location (otherwise the instance could be solved trivially). Thus the variables $x_{1,(0,u)}$ for arbitrary pickup locations u must be at least as great as $x_{2,u}^{\text{pick}}$. There must be one u for which $x_{2,u}^{\text{pick}} = 1$ and the corresponding edge variable is forced to the correct value with Constraint 3.9.

$$x_{1,(0,u)} \geq x_{2,u}^{\text{pick}} \text{ for all } u \in [1, m-1] \tag{3.9}$$

These edge constraints introduce $O(m^3)$ new variables and $O(m^3)$ new constraints. The alert reader may have noticed that there is a problem with the constraints stated above: The solver will set $x_{i,(r,r+m)} = 1$, independently from the assignment of the domain variables $x_{i,r}^{\text{pick}}$ and $x_{i,r}^{\text{drop}}$. The reason for this is that then all riders are delivered without any detours which clearly minimizes the objective function. The core problem is that in this case more edges are used than allowed. For m persons there can only be $2m-1$ used edges in any feasible tour. The following constraint asserts this property and prevents the solver from using too many edges:

$$2m - 1 = \sum_{e \in [0,2m-1]^2} \sum_{i=1}^{2m-1} x_{i,e} \tag{3.10}$$

Notice that the edge variables also intentionally contain trivial edges $e = (p, p)$. These variables are never assigned 1 and thus are not causing any harm. Moreover, it simplifies the notation.

Let t_r^{pick}, t_r^{drop} be variables describing the step in which rider r was picked or dropped. The variable t_e represents the step *after* which e was used, respectively. Be aware that these three kinds of variables are natural, not binary. The values of all three variable types are defined straight forwardly in the following three constraints. Since the steps are counted starting with 1, the variables t_r^{pick} and t_r^{drop} are incremented by 1.

$$t_r^{\text{pick}} = 2m - \sum_{i=1}^{2m} x_{i,r}^{\text{pick}} + 1 \qquad \text{for all } r \in [0, m-1] \tag{3.11}$$

$$t_r^{\text{drop}} = 2m - \sum_{i=1}^{2m} x_{i,r}^{\text{drop}} + 1 \qquad \text{for all } r \in [0, m-1] \tag{3.12}$$

$$t_e = \sum_{i=1}^{2m-1} x_{i,e} \cdot i \qquad \text{for all } e \in [0, 2m-1]^2 \tag{3.13}$$

These three kinds of variables are the only variables that are not binary. This is unfortunate because these variables increase the space of possible solutions enormously. It is possible to replace these real number variables with binary variables, but this means $O(m^4)$ new variables instead of $O(m)$ new variables and a lot more constraints. Tests showed that the running times of the binary-only linear programs were not better than the running time of the model presented here. The author refrains from explaining the binary variant in this thesis.

Naturally, $x_{e,r} = 1$ iff rider r was picked before the edge e was used and dropped after it was used, i.e $x_{e,r} = 1 \Leftrightarrow t_r^{\text{pick}} \leq t_e < t_r^{\text{drop}}$. In order to express this logical rule with linear inequalities two more auxiliary variables are needed. These variables reflect whether the two relation-signs in the latter term are fulfilled or not. Let $x_{e,r}^{\text{pick}}$ be 1 iff person r was picked any time before the vehicle used edge e. Analogously, $x_{e,r}^{\text{drop}}$ should be 1 iff person r was dropped any time after the vehicle used edge e. The correct values for these types of variables are implemented by the those constraints:

$$x_{e,r}^{\text{pick}} > \frac{t_e - t_r^{\text{pick}}}{2m-1} \qquad \text{for all } e \in [0, 2m-1]^2, r \in [0, m-1] \tag{3.14}$$

$$x_{e,r}^{\text{drop}} \geq \frac{t_r^{\text{drop}} - t_e}{2m-1} \qquad \text{for all } e \in [0, 2m-1]^2, r \in [0, m-1] \tag{3.15}$$

Even though these terms include division it is a legal linear expression because $2m-1$ is a constant in the realm of the model. Per definition, the result of Equation 3.14 lies in the interval $]-1, 1[$. The variable t_r^{pick} is at most $2m-2$ because the last two actions must be dropoffs, and t_e is at least 0. Thus, the minimal value of the division is $-(2m-2)/(2m-1) > -1$. The maximal value is $(2m-1-1)/(2m-1) < 1$. The result is negative if e was used before r was picked or e was not used at all, otherwise it is greater or equal to 0. Because $x_{e,r}^{\text{pick}}$ must be strictly greater than the result of the devision and be binary, the correct value of either 0 or 1 is assigned to $x_{e,r}^{\text{pick}}$. In the same way the correctness of Equation 3.15 can be shown, which is omitted here. There are $O(m^3)$ variables of the type $x_{e,r}^{\text{pick}}$ and $x_{e,r}^{\text{drop}}$, which contribute $O(m^3)$ new constraints.

With these auxiliary variables, the value of $x_{e,r}$ can be determined. Its value is given by the following term. If both values on the right side are 1 then $x_{e,r}$ can only be 1, if at least one variable is 0, the value $x_{e,r}$ can either be set to 0 or 1 but since the objective function is minimized the solver sets only those $x_{e,r} = 1$ which are absolutely necessary.

$$x_{e,r} \geq x_{e,r}^{\text{pick}} + x_{e,r}^{\text{drop}} - 1 \text{ for all } e \in [0, 2m-1]^2, r \in [0, m-1] \tag{3.16}$$

The overall complexity of the linear model is $O(m^3)$ variables and $O(m^3)$ constraints. Thus, even for small numbers of riders, the linear program becomes very big. For an instance with three riders CPLEX generated 1 850 constraints and 1 352 variables. An instance with eight riders asks for 20 630 constraints and 14 922 variables. Advanced ILP techniques may be used to handle such instances, for example column generation [9], but the author did not follow this path. Instead, the combinatorial algorithms from the previous sections are used to solve the use case described in the introduction. The next section defines the precise structure of the Dial-a-Ride problem in rural areas and shows approaches to accelerate the computation of optimal tours in these environments.

4 The Dial-a-Ride Problem in Rural Areas

As mentioned in the introduction, the goal of this thesis is to establish a means of public transport for rural and less frequented areas. The algorithms described in the last section can indeed solve the Dial-a-Ride instances arising from rural areas, but their running time is long. This chapter introduces the notion of clustered Dial-a-Ride instances which model provincial public transportation scenarios. As mentioned in the introduction, the intuitive optimal tour handles the villages unidirectionally. To this end, the $\overrightarrow{T^*}$-algorithm is described. It computes unidirectional optimal route and runs faster than the algorithms from Section 3. As it turns out, the intuition is unfortunately wrong: There are instances whose optimal tours serve the villages in an arbitrary order. When the $\overrightarrow{T^*}$-algorithm is used on these instances, the resulting tour is not globally optimal. The second part of this chapter formulates a classifier that decides if the new $\overrightarrow{T^*}$-algorithm gives an optimal result on a given instance. Based on that decision, the appropriate algorithm can be chosen to compute the optimal tour.

4.1 The Clustered Dial-a-Ride Problem

Suppose a regional bus service line is to be replaced by a more flexible Dial-a-Ride system which fetches the passengers at their doorstep. The resulting Dial-a-Ride instances have a special structure: All requests occur only inside villages, the villages are often connected by a big main road and the order in which the villages are visited is predefined by the former bus line. Figure 4.1 shows an example of four villages aligned along one road. It *seems* to be clear that there is only one sensible order in which the villages are visited.

The *clustered* Dial-a-Ride problem extends the basic Dial-a-Ride problem with a cluster quadruple Q: A clustered Dial-a-Ride instance I is a quadruple $(n, C, [d_{i,j}], Q)$, with $Q = ([C_j], [a_j], [e_j], [d_i])$. Q carries the information about the villages, which are from now on referred to as clusters. The list $[C_j]$ assigns every location $0 \leq j \leq 2m-1$ the cluster in which location j lies. Every cluster (village) has two border points: an access point and an exit point. These are the points through which the village is entered and exited when all villages are visited one after another, shown as black dots in Figure 4.1. The lists $[a_j]$ and $[e_j]$ carry for every location j the distances to its border points. Thus $a[j]$ is the distance from location j to the access point of cluster $C[j]$, and $e[j]$ denotes the distance to the exit point of cluster $C[j]$. The last entry in Q is $[d_i]$. It carries information about the distances between consecutive border points. If i is even, then $d[i]$ refers to the distance between access and exit point of the $(i/2)$th cluster. If i is uneven, than $d[i]$ refers to the distance between cluster $(i-1)/2$ and cluster $(i+1)/2$. Notice that this formulation simplifies reality because it assumes that all distances in Q

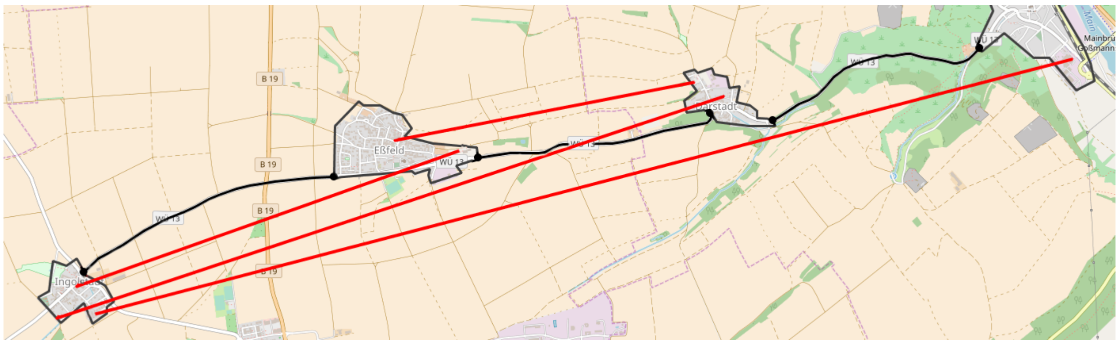


Fig. 4.1: This figure shows some clusters with their portals as well as the source and destination of three riders and the driver. All persons are heading to the right

are symmetric. This is not necessarily the case in street networks. However, asymmetric distances can be incorporated into the model by using three additional lists $[a'_j]$, $[e'_j]$ and $[d'_i]$ which store the distances in counter-direction. For the sake of simplicity, it is assumed that the differences between the plain lists and the primed lists are small enough to not affecting the structure of the solutions. Notice also that the naming of the border points only reflects the predefined order of the villages. The optimal tour is allowed to enter villages from their exit points and leave them through the access points.

Since the driver always starts in the first cluster $(i = 0)$ and finishes in the last cluster $(i = \max\{[C_j]\})$, these two clusters are special. For $i = 0$, the access point is set to coincide with the driver's source point and for $i = \max\{[C_j]\}$, the exit point is the same as the driver's destination. The notation C_i henceforth is used to identify the cluster with index i. It can be imaged as a subset of the location set $\{0, \ldots, 2m - 1\}$ together with the access point as well as the exit point. Be aware not to confuse C_i with the list $[C_j]$ of Q. The former is the natural mental image of a village with all its properties, the latter is an assignment of locations *to* clusters.

As already mentioned, this chapter focuses heavily on tours that visit every village at most once. A necessary (but not sufficient) requirement is that the capacity of the vehicle is great enough to support an unidirectional tour. This does not mean that the capacity S must be at least as high as the number of persons m in the instance, but rather that the maximal number of passengers that intersect an imaginary line between two subsequent clusters does not exceed S. In Figure 4.1 this number is three which means that three seats are sufficient, even though there are four persons. It is easy to test this property for a given instance. Therefore, it is from now on silently assumed that S allows a unidirectional tour. Formally, $S = \infty$ in the following discourse.

Figure 4.1 shows three requests of riders and provides additional observations that make working with clustered Dial-a-Ride instances easier. First, the remaining part of the thesis assumes that the clusters are located from left to right, i.e. cluster $i = 0$ is always the leftmost cluster. This is a comfortable definition to allow for natural terms such as "leave the cluster to the right" or "come from the left".

The access point of a cluster always lies on the left boundary of the cluster and the exit point on the right boundary. Second, all riders and the driver aim to the right, i.e. the pickup location is always located to the left of the corresponding dropoff location (or lies in the same cluster). Third, the distances encoded in Q are conform with to those in $[d_{i,j}]$: For two locations j and j', with $j < j'$, the terms $d[j][j']$ and $e[j] + \sum_{i=i_1+1}^{i_2-1} d[i] + a[j']$ are equal.

In real world, this equation is not always fulfilled because the path through several villages might not be the cheapest way to connect some of the villages. However, when dealing with bus lines, small detours to serve additional villages are frequently met. Figure 4.2 shows such a case. In 2018, there exists a regular bus route serving these villages from north to south, even if this means a detour for some passengers: Traveling from the northern to the southern municipality is both faster and cheaper using the federal highway than the country road. In the evaluation in Section 6 it is shown that the "equality" required above allows a certain slack. As a side note, the work by Welch et. al. [34] gives directions how to examine whether a detour from the optimal path is sensible to increase the earnings from a bus route. Their analysis is based on estimating the expected number of new passengers in the detour segment and comparing it to the other passengers' loss of time caused by the detour. From now on it is assumed that the equation above does hold and the riders are ready to accept detours implied by the ordering of the clusters.

Both, Figure 4.1 and Figure 4.2 suggest that the optimal tour only leaves clusters through their exit points and enters clusters only from their access points. If a route has this property it will receive a little arrow above its name, like $\overrightarrow{T}$. The optimal unidirectional route is called $\overrightarrow{T^*}$, in contrast to $\overleftrightarrow{T}$ which denotes a non-unidirectional tour. The optimal non-unidirectional tour is referred to with the symbol $\overleftrightarrow{T^*}$. If there is no arrow over the name, it can be either unidirectional or not. Intuitively, instances with clusters being wide apart should have the property that $T^* = \overrightarrow{T^*}$. Although there is no guarantee for that intuition, the next theorem is a strong indicator that the inter-cluster distances pay an important role in defining the optimal tour.

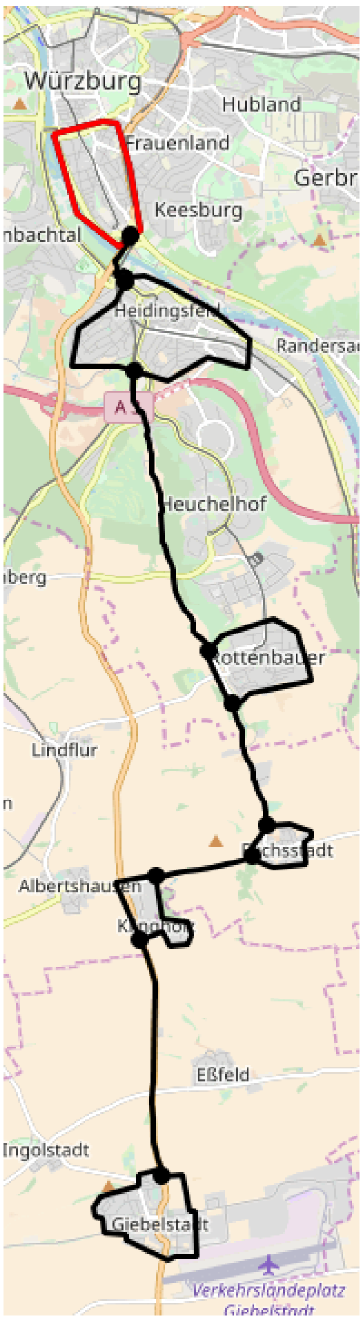


Fig. 4.2: A bus line starting from the red city to the south. The highway is shorter, but loses customers in the smaller villages.

Theorem 4.1. *Let I be a clustered Dial-a-Ride instance with $\overrightarrow{T^*} = T^*$. Add $x \in \mathbb{R}^+$ to every entry with uneven index in $[d_j]$. Modify $[d_{i,j}]$ accordingly and obtain instance J. For J, it holds that $\overrightarrow{T^*} = T^*$.*

Proof. The first observation is that any route remains feasible if only the inter-cluster distances are changed because routes are only permutations of locations. Let T be the optimal route for I. According to the premise, T is unidirectional. Let U be the optimal route of J and assume, for the sake of contradiction, that U is not unidirectional. T's costs in J are $c_J(T) = c_I(T) + x(q-1)$ with q being the amount of clusters and it holds that $c_J(T) > c_J(U)$. U uses more than $q-1$ inter cluster connections (otherwise it would be unidirectional). Therefore the costs of U in I are $c_I(U) \leq c_J(U) - xq$. Combining the previous results yields that $c_I(U) < c_I(T) + x(q-1) - xq = c_I(T) - x$. Consequently, T was not an optimal tour for I. Contradiction. $\square$

Remark. The theorem stays correct even if the x added to the inter-cluster distances is different for every pair of subsequent clusters, i.e. a vector $\boldsymbol{X}$ is added. The proof must be altered such that every occurrence $x(q-1)$ is replaced by $\sum \boldsymbol{X}$ and every occurrence of xq is replaced by $\sum \boldsymbol{X} + \min\{\boldsymbol{X}\}$.

Remark. The wording of the discussion assumes that the optimal tour is unambiguous. In deed it could happen that $c(\overrightarrow{T^*}) = c(\overleftrightarrow{T^*})$. Though, the statements and findings of this thesis remain correct, but their description and proofs would require a more distinct and cumbersome text. Thus, it is implied that *either* $c(T^*) = c(\overrightarrow{T^*})$ *or* $c(T^*) = c(\overleftrightarrow{T^*})$.

Of course, the practical applicability of the theorem is questionable, but it indicates that instances with a unidirectional tour will not lose this property if the inter cluster distances are increased.

The main difference of computing $\overrightarrow{T^*}$ compared to computing T^* is obvious: At every location, the former computation needs only glance at the locations in the same cluster and the next cluster at most. The latter computation can be carried out by Psaraftis' algorithm. As already seen in Figure 3.5, this computation considers all unvisited locations in every step. Consequently, the computation tree for $\overrightarrow{T^*}$ is much smaller than T^*'s. The optimal unidirectional tour $\overrightarrow{T^*}$ can be obtained by applying Psaraftis' algorithm subsequently to the individual clusters. The key trick is to divide the clustered Dial-a-Ride instance into several partial Dial-a-Ride instances. The following observation shows how this can be established:

Observation. Any unidirectional tour $\overrightarrow{T}$ can use every connection between clusters at most once, and this can only happen from the left to the right. Thus the only way two tours $\overrightarrow{T}$ and $\overrightarrow{T'}$ can differ is how they operate inside the clusters.

The idea is to regard the clusters as partial Dial-a-Ride instances and solve them with Psaraftis' algorithm. These partial instances are smaller than the original instance and since Psaraftis' algorithm has exponential running time, it is faster to run it several times with small input sizes than once with a big input size. The next observation reveals how the *initial* and *final* state for partial instances can be generated:

Observation. Given a cluster C_i with index i and a unidirectional tour $\overrightarrow{T}$. When the vehicle enters C_i through its access point, the status of all other riders are known: All riders with both their pick up and drop off points lying left of C_i must have been delivered, all riders with pick up left of C_i and drop off in or right of C_i must be inside the vehicle and all other passengers are still waiting. These states are known because the algorithm is not allowed to drive to the clusters left of C_i again. Thus, all locations in those clusters must already be visited. For the same reason, the locations in C_i and in the clusters right of C_i cannot be visited at the moment C_i is entered through the access point.

The same observation can be applied in the case when the vehicle leaves C_i through the exit point. Let $\mathsf{stateAccess}(I,i)$ and $\mathsf{stateExit}(I,i)$ be algorithms computing these states for every rider. Both algorithms can be implemented such that their running time is linear in the number or riders; their exact implementation is omitted here. Both methods are used in Algorithm 4, which computes an optimal unidirectional tour $\overrightarrow{T^*}$. The pseudocode of this $\overrightarrow{T^*}$-algorithm is straight forward. In a loop, the optimal tour for every cluster is computed and then merged with the already existing tour.

Algorithm 4: An algorithm computing $\overrightarrow{T^*}$

```
Input: A clustered Dial-a-Ride instance $I = (n, \infty, [d_{i,j}], Q)$
Output: An optimal unidirectional tour $\overrightarrow{T^*}$
1  $\overrightarrow{T} = [0]$
2  for $i \in \{1, \dots, q\}$ do
3      $initial$ = stateAccess($I$, $i$)
4      $final$ = stateExit($I$, $i$)
5      $[d'_{i,j}]$ = copy of $[d_{i,j}]$
6      modify $[d'_{i,j}]$ such that location 0 is $C_i$'s entry and location $m$ is $C_i$'s exit
7      $I' = (n, \infty, [d'_{i,j}], initial, final)$
8      $T$ = tour found by Psaraftis on $I'$
9      $\overrightarrow{T} = \overrightarrow{T} + T[1:-1]$            // Omit first and last entry of $T$
10 return $\overrightarrow{T} + [m]$
```

The running time of the algorithm is $O\left(q \cdot (n + t_{\text{Psaraftis}}(I')\right)$. The complexity of I' is only determined by the maximum number k of locations inside any cluster. This makes the $\overrightarrow{T^*}$-algorithm a fixed parameter algorithm with k being the parameter [6]. If k is small then the $\overrightarrow{T^*}$-algorithm can compute $\overrightarrow{T^*}$ for instances with a great number of riders. The parameter k can be regarded to be small in villages. Thus, if $\overrightarrow{T^*} = T^*$, then the algorithm above offers the ability to find optimal routes a lot faster than Psaraftis' algorithm. The question is how to decide if $\overrightarrow{T^*} = T^*$ without computing both tours. The answer lies in a classifier. In the remainder of this chapter, such a classifier is introduced step by step.

4.2 Classifying Clustered Dial-a-Ride Instances

The classifier to be established in the next three sections receives a clustered Dial-a-Ride instance I and responds with either “yes” or “no”. If the answer is “yes”, then $\vec{T^*} = T^*$ and the faster algorithm can be used. If the answer is “no”, then there is no information about the relation of $\vec{T^*}$ and T^*. In this case Psaraftis’ algorithm must be applied, even if this may be unnecessary. As a first step, induced costs $\Upsilon(C_i, T)$ are introduced.

Definition 4.2. For an instance I let $\mathcal{C}$ be the set of all C_i and $\mathcal{T}$ the set of all feasible tours in I. A relation $\Upsilon\colon \mathcal{C}\times\mathcal{T} \to \mathbb{R}^+$ which has the property that $\sum_{C_i\in\mathcal{C}} \Upsilon(C_i, T) = c(T)$ for all tours T represents the *induced costs* $\Upsilon(C_i, T)$ of cluster C_i in T.

When T is clear from the context, the notation is abbreviated by $\Upsilon(C_i)$. An important point is that the induced costs are defined abstractly in this section. Two sensible and concrete definitions are discussed in the next section. For now it suffices that $\Upsilon(C_i)$ fulfills the summation above, independently of how it is realized. The classifier works by computing two values $\Phi(C_i)$ and $\Psi(C_i)$ for every cluster C_i. These values are compared and the result of this comparison defines the answer.

Let $\Phi(C_i) \leq \Upsilon(C_i, T^*)$ be a lower bound on the induced costs of C_i in the optimal tour. In contrast, let $\Psi(C_i) = \Upsilon(C_i, \vec{T^*})$ be the *exact* induced costs of C_i in the best unidirectional tour $\vec{T^*}$. The following theorem forms the basis of the classifier:

Theorem 4.3. *If $\forall C_0, \ldots, C_q \in \mathcal{C}\colon \Psi(C_i) = \Phi(C_i)$, then $\vec{T^*} = T^*$.*

Proof. Suppose $T^* \neq \vec{T^*}$, then there is an non-unidirectional route $\overleftrightarrow{T^*}$ with $\overleftrightarrow{T^*} = T^*$. In $\overleftrightarrow{T^*}$ there must be a sequence $K = [C_i, \ldots C_{i'}]$ of contiguous clusters which are all at least visited twice. It is always possible to select $|K| \geq 2$, otherwise $\overleftrightarrow{T^*}$ was unidirectional. The costs $c(K)$ of K are related to $\Psi(C_i)$ and $\Phi(C_i)$:

$$c(K) = \sum_{j=i}^{i'} \Upsilon(C_j, T^*) = \sum_{j=i}^{i'} \Upsilon(C_j, \overleftrightarrow{T^*}) \geq \sum_{j=i}^{i'} \Phi(C_j)$$

By definition of $\Psi(C_i)$, the clusters in K can be handled unidirectionally with costs c'_K defined be the summation:

$$c'(K) = \sum_{j=i}^{i'} \Psi(C_j) = \sum_{j=i}^{i'} \Phi(C_j) \leq \sum_{j=i}^{i'} \Upsilon(C_j, T^*) = c(K)$$

Therefore, the costs of $c'(K)$ are at most $c(K)$ and K can be handled unidirectionally for at most the same costs. Contradiction to the assumption that $c(\overleftrightarrow{T^*}) < c(\vec{T^*})$.

□

The classifier basically tests the theorem for every cluster. If there is a cluster for which the condition of the theorem does not hold, then the answer is “no”, otherwise it is “yes”. The next section introduces a sensible implementation of induced costs $\Upsilon(C_i, T)$. In Section 4.4, a method to compute a good lower bound $\Phi(C_i)$ is presented. The performance of the classification with these realizations is evaluated in Chapter 6.

4.3 Classifier Step 1: Distribute Costs To Clusters

The last section introduced a powerful theorem which helps to determine if the result of the $\overrightarrow{T^*}$-algorithm computes an globally optimal tour T^*. Remember that if the answer is "no", then the $\overrightarrow{T^*}$-algorithm may still provide an optimal tour. Therefore, the aim is to find induced costs $\Upsilon(C_i)$ and lower bounds $\Phi(C_i)$ that maximize the number of correct answers. This section covers the choice of induced costs $\Upsilon(C_i)$. Remember that for a tour T the sum $\sum \Upsilon(C_i, T)$ of the induced costs for every cluster C_i must add up to the cost of the tour $c(T)$.

A natural cost distribution is obtained by looking directly at the individual clusters. For a cluster C_i, let $\Upsilon(C_i, T)$ be the costs of all subtours happening inside C_i plus the costs of all edges leaving the cluster C_i. This way $\sum_{C_i \in \mathcal{C}} \Upsilon(C_i, T) = c(T)$ is guaranteed. Unfortunately, this approach is not powerful enough. The problem is that any lower bound on $\Upsilon(C_i)$ cannot incorporate riders coming into C_i from the wrong direction, for example, riders with dropoffs left of or in C_i that enter C_i from the right. These riders are counted in the induced costs of one of the neighboring clusters. But neither the lower bound on the neighbors nor the bound on C_i can include those costs because the lower bound must assume the best case, which is in this case that the riders are already delivered when reaching C_{i+1}. Figure 4.3 illustrates the problem. Cluster C_i is handled in two parts. The rider 5 with destination 15 was picked in C_{i-1}, travels through C_i, then to C_{i+1} and then back to C_i. Nominally, the last external edge of rider 5 is charged by the induced costs of C_{i+1} but the lower bound for C_{i+1} assumes the best case: that rider 5 was already dropped and does not visit C_{i+1} at all. Therefore, the edge from C_{i+1} to C_i gets lost. This is bad because the greater the lower bounds are, the more likely is it that Theorem 4.3 can be applied.

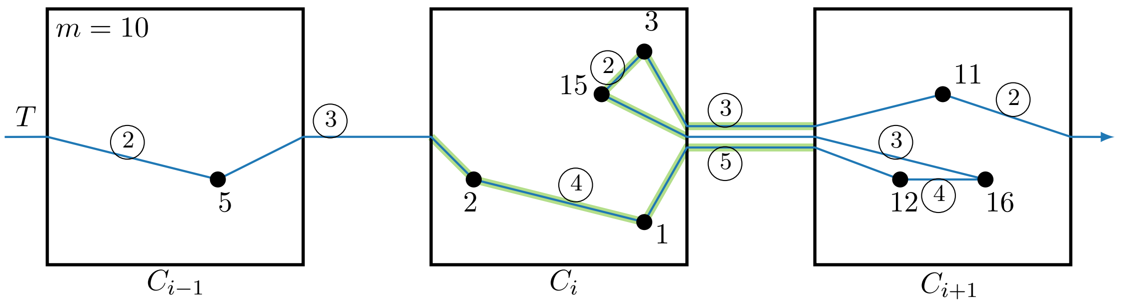


Fig. 4.3: A sample tour T and edges counted in simple realization of $\Upsilon(C_i)$ (fat). Riders 1 and 5 visit C_i twice. Notice that the edges are multiplied with the number of persons inside the vehicle when it is on that edge, written as encircled numbers. Before entering C_{i-1} only rider 6 and the driver are on board, when leaving C_{i+1} rider 3 is on board.

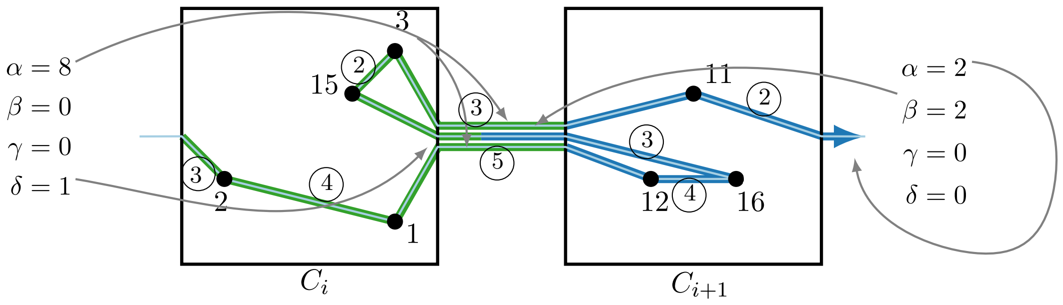


Fig. 4.4: The same instance as in Figure 4.3. The modified cost distribution shares the center edge between C_i and C_{i+1}. The values for the counters for both clusters are given left and right.

Therefore, another cost distribution $\Upsilon(\cdot)$ is used. Let r be a rider, p_r the index of his pickup cluster and d_r the index of his dropoff cluster. For a feasible tour T and a cluster C_i, the following counters are defined: α counts the events that a rider r with $p_r \leq i$ or the driver exits to the right, β the events that a rider r with $d_r \geq i$ or the driver exits to the left. In the same spirit, the counter γ and δ are used: γ counts how often a rider r with $p_r \geq i$ enters from the left, and last, δ counts the occurrences of riders with $d_r \leq i$, that enter from the right.

Then $\Upsilon(C_i, T)$ is given by:

$$\Upsilon(C_i, T) = \alpha d[2i + 1] + \beta d[2i - 1] + \gamma d[2i - 1] + \delta d[2i + 1] + \mathsf{inside}(C_i) \tag{4.1}$$

The list $[d_i]$ denotes the list of distances between clusters which is found in Q. Figure 4.4 shows an example of the induced costs of cluster C_i in the feasible tour T. In this instance, one can easily check that the sum of the induced costs of all clusters is exactly the cost of the tour T. This property is formally stated for all feasible tours in the next theorem. However, this theorem requires a brief lemma:

Lemma. *Consider a journey from C_i to C_{i+1}. Then the journey can* either *be counted by C_i's α-counter* or *by C_{i+1}'s γ-counter, but not both. The same applies for a left-bound journey between C_{i+1} and C_i.*

Proof. If the right-bound journey is covered by C_i's α-counter, then $p_r \leq i$. Yet, C_{i+1}'s γ-counter catches the journey only if the condition $p_r \geq i + 1$ is true. That cannot happen since the first condition prohibits it. Thus, the γ-counter can only be used if the α-counter of the cluster to the left does not fire. Symmetrically, the same argumentation shows that either C_i's δ-counter or C_{i+1}'s β-counter is responsible for a journey from the cluster C_{i+1} to the cluster C_i. □

With the help of this lemma the validity of the induced costs can be shown:

Theorem 4.4. *For every feasible tour T, the sum of induced costs according to Equation 4.1 equals the total cost of T: $\sum_{C_i \in \mathcal{C}} \Upsilon(C_i, T) = c(T)$.*

Proof. All costs generated inside the clusters are covered once by the last term of Equation 4.1, so the main task is to show that the costs that are generated between the clusters are only counted once. Pick an arbitrary journey of one person between the two clusters C_i and C_{i+1}. If this person is the driver and the journey goes from cluster C_i to cluster C_{i+1}, then the α-counter of C_i carries this journey and no counter of C_{i+1} counts the same movement. This applies for a journey of C_{i+1} to C_i symmetrically.

If the journey's person is a rider r and the journey goes from C_i to C_{i+1}, then it can only be counted by C_i's α counter or C_{i+1}'s γ counter. It suffices to show that at least one of these counters catches the journey. The previous lemma ensures that no other counter catches the same journey. The two locations of r can lie in four different ways relative to C_i:

Both locations are left of C_i. This case occurs when $p_r < d_r \leq i$, i. e. the rider travels further than he needed. This journey is counted by C_i's α because $p_r < i$.

Pickup left of C_i and dropoff in or right of C_i That is the case if $p_r < i \leq d_r$. Since $p_r \leq i$, it is counted by C_i's α counter.

Both locations lie in C_i Then $p_r = i = d_r$ and C_i's α counter is responsible for counting the journey.

Pickup left of or in C_i and dropoff right of C_i This condition can be expressed mathematically as $p_r \leq i < d_r$. It is counted by C_i's α counter because $p_r \leq i$.

Both locations are right of C_i In formal terms, $i < p_i \leq d_i$ and thus, only C_{i+1}'s γ counter cares about this journey.

The same steps can be applied to a right-to-left journey and C_i's δ counter and C_{i+1}'s β counter. Therefore, every journey is counted exactly once in $\sum_{C_i \in \mathcal{C}} \Upsilon(C_i, T)$. □

Interestingly, the induced costs $\Psi(C_i)$ in a unidirectional tour $\overrightarrow{T}$ are the same for both the simple induced costs depicted in Figure 4.3 and the counter-based induced costs. In such a tour all clusters are visited only once and it cannot happen that any inter-cluster edge is used leftwards. Thus, all counters except α are zero and the costs generated by leaving C_{i+1} are covered by α. A nice consequence is that $\Psi(C_i) = \Upsilon(C_i, \overrightarrow{T^*})$ can be computed easily for all C_i. A simple modification of the $\overrightarrow{T^*}$-Algorithm collects the costs generated by Psaraftis' algorithm in cluster C_i and adds them to the costs of connecting C_i to C_{i+1}. This sum is saved as $\Psi(C_i)$ and can be accessed afterwards. The last ingredient needed for a working classifier is a computable lower bound $\Phi(C_i)$ on the induced costs $\Upsilon(C_i, T^*)$.

The next section introduces a non-trivial lower bound on $\Upsilon(C_i, T^*)$ which can be computed in bearable time for small cluster sizes.

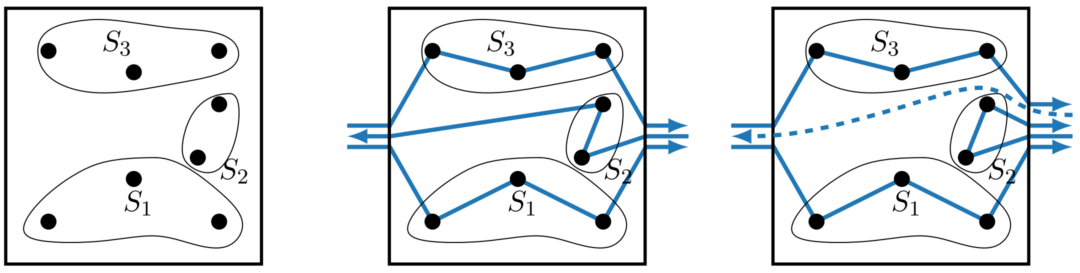
4.4 Classifier Step 2: Estimate Distributed Costs

The last section introduced a definition for induced costs $\Upsilon(\cdot)$ which can be combined with Theorem 4.3 in order to decide whether a faster algorithm than Psaraftis' algorithm can be used to compute the optimal tour T^*. Until now, the classifier lacks a manageable and sensible lower bound $\Phi(C_i)$ on the induced costs of C_i in $\overleftrightarrow{T^*}$. This section presents a method to compute such a lower bound $\Phi(C_i)$ in bearable time if the number of locations inside C_i is small. The idea is based on the observation that a tour that handles C_i in one or more subtours partitions the cluster into several subsets. Thus the approach is to enumerate all possible partitions and compute a lower bound for all of them. The smallest of these lower bounds is the wanted value.

Let C_i be an arbitrary cluster and $\mathcal{S}$ an ordered partition of C_i into several subsets $S_1, \dots, S_k$. Then $\Phi_{\mathcal{S}}(C_i)$ represents the lower bound of serving cluster C_i in the subsets defined by $\mathcal{S}$. Notice that for $\mathcal{S} = [\{3,5\},\{10,4\},\{1\}]$ and $\mathcal{S}' = [\{1\},\{3,5\},\{10,4\}]$ the bounds $\Phi_{\mathcal{S}}(C_i)$ and $\Phi_{\mathcal{S}'}(C_i)$ may differ because of the order of the subsets. The overall lower bound $\Phi(C_i)$ is given by Equation 4.2.

$$\Phi(C_i) = \min_{\mathcal{S} \text{ ordered partition of } C_i} \Phi_{\mathcal{S}}(C_i) \tag{4.2}$$

For a given partition $\mathcal{S}$ of C_i there are several ways to connect the subsets inside S because every subset $S \in \mathcal{S}$ can be entered from the left or from the right of the cluster. Analogously, the vehicle can leave the cluster through the access point or through the exit point. Therefore, for $k = |\mathcal{S}|$ there are 2^{2k} possibilities to serve $\mathcal{S}$. Let P be a list with length k of 2-tuples. Every tuple is an element of $\{\mathsf{a},\mathsf{e}\} \times \{\mathsf{a},\mathsf{e}\}$. The i-th tuple in P determines through which border point the i-th set of $\mathcal{S}$ is entered. For some choices of P the vehicle must traverse a cluster without touching a point in it. This happens when the individual entries in P do not match. An example is given in Figure 4.5. The path entries of the instance in the center match a finished domino game while the path of the right instance contains non-matching entries.



$C_i, \mathcal{S} = [S_1, S_2, S_3]$ $\qquad$ $P = [[\mathsf{a},\mathsf{e}],[\mathsf{e},\mathsf{a}],[\mathsf{a},\mathsf{e}]]$ $\qquad$ $P = [[\mathsf{a},\mathsf{e}],[\mathsf{e},\mathsf{e}],[\mathsf{a},\mathsf{e}]]$

Fig. 4.5: The left picture shows a partition $\mathcal{S}$ of the cluster and the two right pictures show different paths to serve the same partition. In order to handle the rightmost partition, the vehicle has to traverse C_i without visiting a point inside C_i. This journey is indicated as dashed line.

Let $\Phi_{\mathcal{S},P}(C_i)$ be a lower bound to handle the partition $\mathcal{S}$ with respect to the path P. The value of $\Phi_{\mathcal{S}}(C_i)$ can thus be calculated with Equation 4.3.

$$\Phi_{\mathcal{S}}(C_i) = \min_{P \in (\{\mathsf{a},\mathsf{e}\}^2)^k} \Phi_{\mathcal{S},P}(C_i) \tag{4.3}$$

Summarizing the last paragraph, the algorithm computing a lower bound $\Phi(C_i)$ first generates all ordered partitions of C_i. For each ordered partition $\mathcal{S}$ the set of paths going through $\mathcal{S}$ is computed. Then for every partition $\mathcal{S}$ and every path belonging to $\mathcal{S}$ the lower bound $\Phi_{\mathcal{S},P}$ is calculated. The smallest of these bounds over all partitions and paths represents the lower bound $\Phi(C_i)$ on the induced costs $\Upsilon(C_i, T^*)$. Before exploring how $\Phi_{\mathcal{S},P}(C_i)$ is determined, a quick excursion into combinatorics is done in order to get a feeling for the magnitude of required computations.

Let C_i be a cluster and n the number of locations inside the cluster. Then there are $a(n)$ ordered partitions $\mathcal{S}$ of C_i. The term $a(n)$ refers to the ordered Bell number which counts the number of possible weak orderings of a set having the size n. In OEIS, $a(n)$ has the sequence number A670 [32]. There is a simple recursive formula to determine $a(n)$, which is $a(n) = \sum_{i=1}^{n} \binom{n}{i} a(n-i)$ [17]. The ordered Bell number alone does not represent the computational effort because it does not cover the number of possible paths P through the subsets of the partitions. The number of paths can be easily incorporated into the formula:

$$a'(n) = \sum_{i=1}^{n} 4 \cdot \binom{n}{i} a'(n-i) \qquad\qquad a'(0) = 1 \tag{4.4}$$

The only difference is the factor with which every summand is multiplied. This factor is 4 because every set in the partition can be entered and exited in four possible ways. Table 4.1 compares the vanilla Bell number $a(n)$ with the augmented Bell number $a'(n)$ containing paths. Suppose that the computation of $\Phi_{\mathcal{S},P}(C_i)$ takes about 40 microseconds for $n = 5$. Then roughly 129 millisconds are needed to compute the lower bound for a cluster size of $n = 5$. Assume that 40 microseconds are also sufficient to compute $\Phi_{\mathcal{S},P}(C_i)$ for clusters of size $n = 6$. Then approximately four seconds are needed to find the minimum of all 5 227 236 configurations. However, in reality the time needed to solve $\Phi_{\mathcal{S},P}(C_i)$ for $n = 6$ is higher because the underlying algorithm of Psaraftis has exponential running time. Therefore the estimation above is rather optimistic. Using a parallelized implementation, the running time for $n = 6$ can be reduced to be roughly

n	1	2	3	4	5	6	7	8
$a(n)$	1	3	13	75	541	4 683	47 293	545 835
$a'(n)$	4	36	484	8 676	194 404	5 227 236	163 978 084	5 878 837 476

Tab. 4.1: A comparision between the ordered Bell number $a(n)$ and the Bell numbers augmented with path according to Equation 4.4.

three seconds. Thus, clusters of size $n = 6$ pose the practical limit for acceptable in-time computations. The description of the classifier is continued by the next paragraph, which gives an algorithm to compute $\Phi_{\mathcal{S},P}(C_i)$.

Given a partition $\mathcal{S}$ of a cluster C_i and a path P through the partition, the task is to find a lower bound $\Phi_{\mathcal{S},P}(C_i)$ of $\Upsilon(C_i, T)$ for all tours T handling C_i in the order defined by $\mathcal{S}$ and P. Algorithm 5 sketches a realization of $\Phi_{\mathcal{S},P}(C_i)$.

The algorithm is divided in four parts. Every part computes the costs of different aspects in C_i. Refer to Figure 4.6 for an example of these aspects. Part One, Two and Three only consider costs generated by the driver or riders that have either their pickup or dropoff location inside C_i. Part Four aggregates the costs of riders traversing C_i.

Part One collects the costs at the beginning and at the end of $\mathcal{S}$. It calls two additional methods $\mathsf{start}(\cdot)$ and $\mathsf{end}(\cdot)$. They contribute the costs generated by the first and the last leg of handling C_i. In order to do that both methods compute the number of riders that must inevitable sit inside the vehicle when $S[0]$ is entered and $S[|\mathcal{S}| - 1]$ is exited. For example, in the centered instance of Figure 4.5 the vehicle must contain all riders dropped in S_2 as soon as it enters S_1. In the same instance, the riders picked up in S_2 must be still on board if S_3 is accessed. Let these both numbers be named a and b. The number a can be computed by traversing $\mathcal{S}$, starting with $\mathcal{S}[0]$ and follow the way through $\mathcal{S}$ according to P. In every subset $S \in \mathcal{S}$, the amount of dropoffs is counted. The traversion stops as soon as C_i is exited to the left or entered from the left. The riders of all dropoffs encountered so far must sit inside the vehicle when $S[0]$ is entered. Similarily, b can be computed by traversing $\mathcal{S}$ backwards, counting the pickups and stopping when the vehicle comes from the right or drives to the right. The method $\mathsf{start}(\cdot)$ is implemented as follows:

$$\mathsf{start}(\mathcal{S}, P, C_i, I) = \begin{cases} 0 & \text{if } S[0] \text{ is entered via access point} \\ (a+1)d[2i] + (2a+1)d[2i+1] & \text{else} \end{cases}$$

If the first subset is entered through the access point then according to the definition of $\Upsilon(\cdot)$ no costs are generated. If, however, the first subset is entered through the exit point of C_i, than all riders and the driver must have been traveled *directly* through C_i to the next cluster C_{i+1} (see α-counter of previous section). After doing business there, they returned and entered from right. This time, the driver is not counted (see δ-counter). Analogously, $\mathsf{end}(\cdot)$ is realized:

$$\mathsf{end}(\mathcal{S}, P, C_i, I) = (b+1)d[2i+1] + \begin{cases} 0 & \text{if } S[0] \text{ is left} \\ & \quad \text{via exit point} \\ (2b+1)d[2i-1] + (b+1)d[2i] & \text{else} \end{cases}$$

In any case, the picked riders and the driver must leave the cluster to the right, so these costs (covered by the α-counter) are always included. If, however, the vehicle leaves the cluster through its access point, then the vehicle must return at some point with all riders still on board. The costs of driving to C_{i-1} and coming from it are represented

Algorithm 5: A realization of $\Phi_{\mathcal{S},P}(C_i)$

```
   Input: Partition $\mathcal{S}$, path $P$, cluster $C_i$ and a clustered Dial-a-Ride instance $I$
   Output: A lower bound $\Psi_{\mathcal{S},P}(C_i)$
   // Part One
1  $costs = \mathsf{start}(\mathcal{S}, P, C_i, I) + \mathsf{end}(P, \mathcal{S}, C_i, I)$
   // Part Two
2  foreach $S \in \mathcal{S}$ do
3      $initial = \mathsf{initialStateVector}(S)$
4      $final = \mathsf{finalStateVector}(S)$
5      $I' = (n, \infty, [d_{i,j}], initial, final)$
6      $costs = costs$ + cost of optimal tour $T^*$ in $I'$
7      $traversals = \mathsf{countTraversals}(\mathcal{S}, S)$
8      $costs = costs + traversals \cdot \mathsf{length}(T^*)$
   // Part Three
9  for $i \in \{2, \dots, |P|\}$ do
10     $lastExit = P[i-1][1]$
11     $nextEntry = P[i][0]$
12     $costs = costs + \mathsf{handleTransition}(lastExit, nextEntry, S_{i-1}, S_i, I)$
   // Part Four
13 $costs = costs + \mathsf{handleTraversingRiders}(\mathcal{S}, P, C_i, I)$
14 return $costs$
```

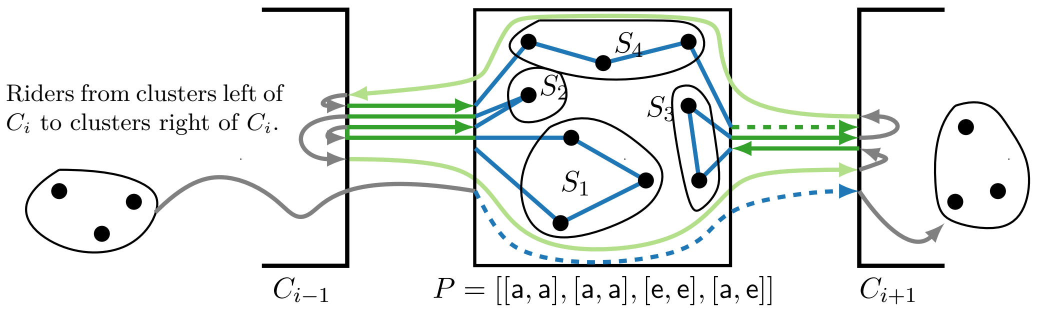

Riders from clusters left of C_i to clusters right of C_i.
S_4
S_2
S_3
S_1
C_{i-1}
$P = [[\mathsf{a}, \mathsf{a}], [\mathsf{a}, \mathsf{a}], [\mathsf{e}, \mathsf{e}], [\mathsf{a}, \mathsf{e}]]$
C_{i+1}

Fig. 4.6: An partition S and a path P through the partition. The dashed green line shows the costs of which Part One takes care, Part Two is responsible for the costs depicted in solid blue. Part Three's costs are drawn in solid green and Part Four is represented in dashed blue. The gray lines are not part of the costs of cluster C_i but rather assist the reader to trace the route.

by the first summand in the case distinction. These costs are covered by the β- and γ-counters, respectively. This first part is drawn in dashed green in Figure 4.6. In that instance Part One only consists of the last edge outgoing from C_i. The explanation of the algorithm's first part is finished.

The second part covers all costs generated inside the cluster C_i, which are drawn in solid blue in Figure 4.6. To this end, for every $S \in \mathcal{S}$ the initial state vector *initial* is computed. This vector represents the states of all riders before entering S. Symmetrically, the vector *final* is created. Both vectors have length n, where n is the number of riders. The values for the entries is given by the assignment below:

$$\mathit{initial}[j] = \begin{cases} \mathsf{finish} \text{ if the dropoff of rider } j \text{ lies left of } C_i \\ \mathsf{travel} \text{ if the dropoff of rider } j \text{ is inside } S \\ \mathsf{wait} \text{ else} \end{cases}$$

$$\mathit{final}[j] = \begin{cases} \mathsf{finish} \text{ if the dropoff of rider } j \text{ lies left of } C_i\text{or in } S \\ \mathsf{travel} \text{ if the pickup of rider } j \text{ is inside } S \\ \mathsf{wait} \text{ else} \end{cases}$$

The emerging vectors are used to solve partial Dial-a-Ride instances with Psaraftis' algorithm, whose costs are summed up. But the bound of the internal costs of C_i is not sharp enough yet. It does not take riders boarded before visiting S or dropped after leaving S into account. These riders may sit in the vehicle for the complete tour through S and their contribution to the costs should also be included in $\Phi(C_i)$. For example, a rider picked in $S[i-1]$ is still on board if neither $S[i-1]$ is exited to the right nor $S[i]$ is entered from the right. These riders could be considered by altering the assignments of the state vectors above, but this would overly complicate the generation of the states. Instead, these traversals of S are counted manually (like a and b for the first and last subset in Part One). The number of traversals is then multiplied with the length (not costs) of the optimal partial tour and added to the total costs. After accumulating all these inside costs of C_i, two more types of costs are missing.

The third part consists of aggregating the costs happening outside of C_i, but caused by riders with at least one location in C_i. These are the costs which are covered by the counters α, β, γ and δ. The function $\mathsf{handleTransition}(\cdot)$ is similar to the already presented $\mathsf{start}(\cdot)$ and $\mathsf{end}(\cdot)$ methods. It consists mainly of case distinctions on the *lastExit* and *nextEntry* variables and counting riders inside the vehicle, similar to the a and b in $\mathsf{start}(\cdot)$ and $\mathsf{end}(\cdot)$. The exact implementation of $\mathsf{handleTransition}(\cdot)$ is left as exercise for the reader. The costs of the third part are shown in Figure 4.6 as solid green lines. The dashed lines show tours that go directly through C_i without touching a location inside. This intermediate crossings of C_i are necessary because the path P contains non-matching entries. This is probably the right place to mention that $\mathcal{S}$ and P as depicted in Figure 4.6 are most likely not the best way to solve C_i.

The last part deals with the costs generated by riders passing through C_i. In other words, it deals with those riders whose pickup cluster is left of C_i and whose dropoff cluster is right of C_i, as indicated in Figure 4.6 with dashed the blue line. Those riders are called *hoppers*. The number of these hoppers h can be determined easily using the list $[C_i]$. There are three ways to transport the hoppers through C_i of which the cheapest one is chosen. The first possibility is to fetch them in an extra tour after C_i was handled. The second possibility is only available if there exists a $S \in \mathcal{S}$ which is entered from the left and exited to the right. The hoppers can be transported through C_i using the tour handling S. The costs are h times the length of the shortest left-right-tour through any such S, if there is one. The last possibility exists if there are two consecutive tuples $t = (\cdot, \mathsf{a})$ and $t' = (\mathsf{e}, \cdot)$ in P. This constellation means that the vehicle must drive directly through the cluster from its access point to its exit point. In Figure 4.6 this situation is met. However, in that example it is also possible to carry the hoppers while serving S_4. The cheapest available way of these three possibilities to transport the hoppers is chosen.

It remains to discuss if the method above indeed computes a lower bound for $\Upsilon(C_i, T^*)$. In other words, the question is whether the induced costs $\Upsilon(C_i, T)$ of cluster C_i in every tour T that handles the locations in C_i according to $\mathcal{S}$ and P are always greater or equal to the result returned by Algorithm 5. Part One, Two and Three only consider costs contributed by riders with either their pick up or drop off located inside C_i. Every one of these parts assumes the shortest possible tour to reach the locations without violating the order and traversing rules implied by $\mathcal{S}$ and P is used.

However, there is one pitfall, or so it seems. It is assumed that between two matching entries of P there is no additional traverse through C_i. For example, between $(\cdot, \mathsf{e})$ and $(\mathsf{e}, \cdot)$ *could* be an even number of journeys through C_i in T^* that do not touch any location. Since those journeys are never assumed to happen in Part Three of Algorithm 5 this sounds like the algorithm, and therefore the classifier, too, are broken. Yet, if argued a bit more carefully it becomes clear that there is a solution to this predicament. First, Algorithm 5 is by definition correct because it cares only about serving locations inside C_i and two successive intermediate tours are not part of C_i's service. Second, the classifier is also correct because in a non-unidirectional optimal tour $\overset{\leftrightarrow}{T^*}$ there must be at least one cluster without such intermediate tours, otherwise the tour would trivially not be optimal. This cluster is identified by Theorem 4.3 as the causing cluster for the non-unidirectionality and the classifier responds correctly.

Only Part Four is occupied with foreign riders. For these riders, it is supposed that all of them take the shortest route through C_i. In a real tour T, all these assumptions must not be necessarily true, but in this case, the induced costs only increase. This makes the result of Algorithm 5 a valid lower bound $\Phi_{\mathcal{S},P}(C_i)$.

This finishes the implementation of the classifier and closes the chapter. In the previous sections and paragraphs the Clustered Dial-a-Ride problem was introduced, as well as a classifier. Using this classifier can shorten the total time needed to compute an optimal route. The expected amount of saved time and other properties of Clustered Dial-a-Ride instances are evaluated in the next-but-one chapter. Before that, the implementation of all participating algorithms and programs is described. This includes the pseudocodes depicted so far as well as instance generators for both artificial euclidean instances and realistic geographical instances.

5 Implementation

In order to evaluate the performance of the classifier and to gain knowledge about the clustered Dial-a-Ride problem, the algorithms presented so far were implemented, along with auxiliary programs. The auxiliary programs include two instance generators, a visualizer for the optimal routes and some other tools to process the acquired data. This chapter explains the important details of these components.

The implementation of the algorithms can be divided into two phases. The first phase consisted of implementing Psaraftis' original algorithm as well as the modified incremental algorithm in the Java programming language. These prototypes were mainly tested in respect to maintenance, performance and readability, i.e. they were a proof of concept. The incremental variant fared better in all three categories, so the decision was to use the incremental algorithm. The ILP was also implemented in the first phase, but due to its poor performance not evaluated further. This phase only considered the vanilla Dial-a-Ride problem, not the clustered variant. Sample instances could either be generated by defining every rider manually or by selecting a rectangular section of a geographical map, in which a specified number of riders was generated randomly.

After all the algorithms and the data needed to describe the problem were understood properly, the second phase started, in which the complete workbench was implemented. Unfortunately, the Java programming language has some properties which complicates the development of complex algorithms. This includes cumbersome generic variable declarations, boilerplate code to realize multi-type return statements and the inconvenient way of using functions as first-order objects. Therefore, the main components of the workbench like all algorithms and some parts of the instance generators were (re)written in the Go Programming Language [11], version 1.10.3. Figure 5.1 shows the workbench after the second phase was finished. The modules in shaded boxes are implemented in Java, the rest is written in Go. It can be seen that Java is only used for geographical purposes and visualizations. Beginning at the left side, the journey of a (clustered) Dial-a-Ride instance can be traced through the workbench.

The first step consists of creating the Dial-a-Ride instance. This thesis differentiates between two types of instances: *Euclidean instances* are guaranteed to fulfill the assumptions of Chapter 4, *geographical instances* reflect the problem domain better. Thus, it is sensible to examine both kinds. The euclidean instances reside in the two-dimensional plane and its generator is realized as command line tool written in Go. The geographical instances are based on real street networks. The corresponding generator offers a graphical interface in which the user can define clusters by selecting polygonal shapes on a OpenStreetMap map. He can also pick the border points of the clusters and define the number of riders as well as the maximal number of locations inside the cluster. This visualization is done with JXMapViewer2, a component that displays map tiles [33]

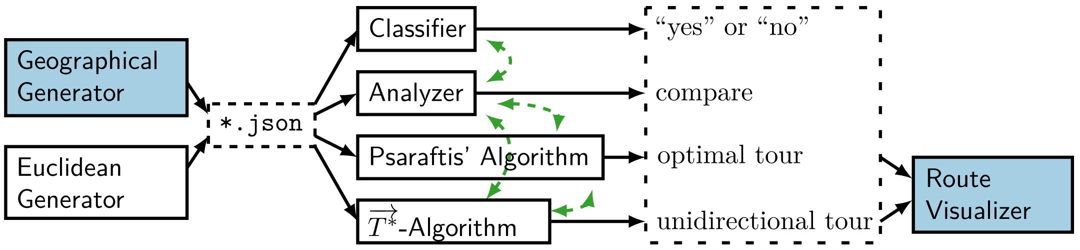


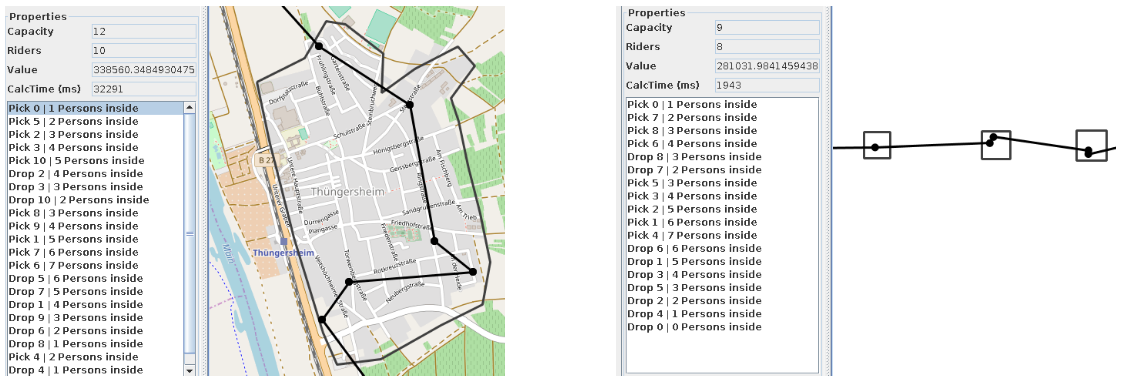
Fig. 5.1: The complete workbench. All shaded components are implemented using Java, the white components are implemented in the Go Programming Language. Dashed arrows indicate communication between the components.

in Swing. The distances between the randomly chosen locations are computed by the GraphHopper Java library [24]. The data of the underlying street network and the tile images are provided by OpenStreetMap data [28]. However, the data was not used directly, but rather downloaded from the Geofabrik site, which offers data for individual regions, thus avoiding the download of the complete Earth data set [25].

No matter whether the generated scenarios are euclidean or geographical, they are saved in a json-file with identical structure for both types. These files can be fed into four different commands. The first component represents the classifier introduced in Section 4.2. For a given instance, it replies with "yes" or "no", as explained in Section 4.2. Psaraftis' algorithm is the implementation of the incremental variant of Algorithm 2, its pseudocode can be found in Algorithm 3. The output is an optimal tour. Between both of the previous components lies the Analyzer. The command runs the classifier and Psaraftis' Algorithm and records whether the prediction of the classifier was correct by checking the tour given by Psaraftis' Algorithm. It also runs the $\overrightarrow{T}^*$-algorithm and gathers additional data which is evaluated in Chapter 6. The last component is the aforesaid $\overrightarrow{T}^*$-Algorithm which gives an optimal uni-directional route. If the classifier's response is "yes", this route is also optimal under all feasible routes. Routes are stored as json-file (omitted in the figure) and then visualized, which is done by the Route Visualizer. If a route is based on a geographical instance, the tour can be drawn either directly onto a map, or in the euclidean plane. All other instances can only be drawn in the euclidean plane. A screenshot of the Route Visualizer is depicted in Figure 5.2.

While most of the implementation described so far is not difficult to realize, some points should be emphasized because they play an important role in the forthcoming evaluation of Chapter 6. The implementation of Psaraftis' algorithm generalizes the pseudocode so that all three target functions from Section 3.1 can be used easily without altering the code of the algorithm. The vertices of the graph are realized as Go-structs which were garbage collected after they could not be reached any more, therefore, storage capacities could be saved. The most complicated component is that of the classifier. As already sketched in Section 4.4, there are several stages.

The first stage consists of generating all ordered partitions, the second stage augments these partitions with paths through the subsets of the partitions. Stage three computes the lower bound and the forth stage collects the minimal value of these lower bounds.



```
Properties
Capacity        12
Riders          10
Value           338560.3484930475
CalcTime (ms)   32291

Pick 0 | 1 Persons inside
Pick 5 | 2 Persons inside
Pick 2 | 3 Persons inside
Pick 3 | 4 Persons inside
Pick 10 | 5 Persons inside
Drop 2 | 4 Persons inside
Drop 3 | 3 Persons inside
Drop 10 | 2 Persons inside
Pick 8 | 3 Persons inside
Pick 9 | 4 Persons inside
Pick 1 | 5 Persons inside
Pick 7 | 6 Persons inside
Pick 6 | 7 Persons inside
Drop 5 | 6 Persons inside
Drop 7 | 5 Persons inside
Drop 1 | 4 Persons inside
Drop 9 | 3 Persons inside
Drop 6 | 2 Persons inside
Drop 8 | 1 Persons inside
Pick 4 | 2 Persons inside
Drop 4 | 1 Persons inside
```

```
Properties
Capacity        9
Riders          8
Value           281031.9841459438
CalcTime (ms)   1943

Pick 0 | 1 Persons inside
Pick 7 | 2 Persons inside
Pick 8 | 3 Persons inside
Pick 6 | 4 Persons inside
Drop 8 | 3 Persons inside
Drop 7 | 2 Persons inside
Pick 5 | 3 Persons inside
Pick 3 | 4 Persons inside
Pick 2 | 5 Persons inside
Pick 1 | 6 Persons inside
Pick 4 | 7 Persons inside
Drop 6 | 6 Persons inside
Drop 1 | 5 Persons inside
Drop 3 | 4 Persons inside
Drop 5 | 3 Persons inside
Drop 2 | 2 Persons inside
Drop 4 | 1 Persons inside
Drop 0 | 0 Persons inside
```

Fig. 5.2: The visualization of a geographical tour and an euclidean tour in Java. Geographical Routes can be shown in two ways: Fix to the road or like the crow flies.

Figure 5.3 shows these stages. All of these stages can be imagined as independent workers which receive their input and send their output to other workers. Because of that, the four workers can run in parallel. Pieces of code that run synchronously are called *goroutines* in the Go Programming Language. The communication between several goroutines happens via channels. When a goroutine sends some data onto a channel, the goroutine gets blocked until the data is processed by another goroutine. Analgousley, a goroutine reading from a channel is delayed until there is some data on that channel that can be received. The idea of this type of synchronous communication originated in a 1978 paper by Hoare [19]. The advantage of using native channels is that the communication between synchronously running pieces of code is handled by the Go runtime itself and need not to be reimplemented. It should be mentioned that besides the channels described so far there exist also *buffered* channels. These offer the possibility to place a limited amount of data into the channel even if there is no goroutine ready to read it. The implementation of this thesis did not make use of buffered channels. All communications between workers took place through unbuffered channels.

Unsurprisingly, the four workers are not equally fast. Tests on a normal desktop computer revealed that the partition generator supplies on average every 7 microseconds a new partition. The path generator needs 21 microseconds to handle a partition. Thus, the partition has to idle 14 microseconds. The cost calculator computes $\Phi_{\mathcal{S},P}(C_i)$ in approximately 27 microseconds and the minimum selector is ready to receive a new value every 81 nanoseconds. These values are far from a solid analysis because the

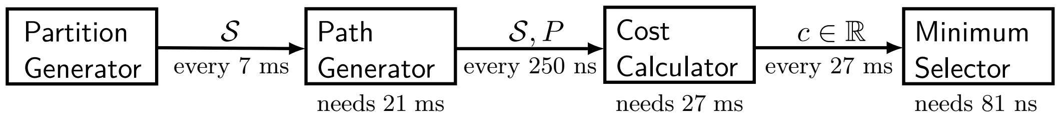


Fig. 5.3: The chain of workers to selected the smallest lower bound over all partitions and paths. The four workers work in parallel and communicate with channels. These channels are depicted as unidirectional arrows.

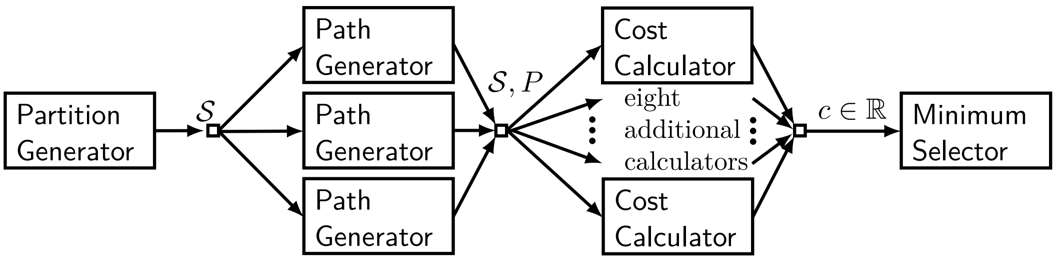


Fig. 5.4: This variant of workers incorporates 15 parallelized tasks and finishes after a sixth of the time the variant with four parallel workers needs (see Figure 5.3).

running times of the path generator and the cost calculator depend on the size of the partition. The values above are gained by evaluating 15 different clusters of which one with size 0, three with size 1, one with size 2, three with size 3, five with size four and two with size 5. It can be seen in a few moments that this quick analysis already accelerates the computations to approximately a sixth of the original times.

The idea is to increase the capacity of the workers. Since they are independent, the slower workers can be duplicated in order to keep in step with the faster workers. Therefore, three path generators are introduced so that the partition generator usually finds a free path generator to which the partition can be sent. The cost generator is also fairly slow, compared to the interval in which a path generator is able to produce paths: A path generator is able to send a new path every 250 nanoseconds on average. Since there are three of them, a new path is ready every 80 nanoseconds. This means that approximately 34 cost calculators are needed so that no path generator does ever have to wait. However, physical resources limit the theoretical thoughts because the overhead of context switching outperforms the advantages of parallel computation. The number of ten cost calculators yielded the fastest worker chain on both desktop computers and an above-average computer (see next chapter for details). Figure 5.4 shows the enhanced chain of workers. There are still only three channels, which are illustrated as white rectangles.

If a worker wants to put something onto a channel, it has to wait until a consumer is ready to receive the data. Analogously, a consumer blocks until there is data in its incoming channel. The Go runtime takes care that every dataset is only put to exactly one receiving worker. The minimum selector is fast enough to deal with ten cost calculators. The complete analysis can be carried out in more detail, of course, but the enhanced version already accelerated the computation: Without duplicate workers computing $\Phi(C_i)$ with C_i having size five needs 18 seconds, the enhanced version finishes under three seconds.

The next chapter evaluates the tool chain presented in this chapter. It measures the time the implementations needed as well as it finds other interesting properties of Clustered Dial-a-Ride instances. For example, the prevalence of instances allowing unidirectional routes is examined and the error rate of the classifier is determined for different parameters of the instance generation.

6 Evaluation

The last chapters mainly focused on the theoretical aspects of the Dial-a-Ride problem. In contrast, the current chapter deals with the practical properties of the algorithms and problems discussed before. This includes running times of the programs described in Chapter 5 as well as the general structure of Clustered Dial-a-Ride instances in reality. As already pointed out in Theorem 4.1, the distances between clusters play an important role in the question whether $T^* = \overrightarrow{T^*}$ or not. These inter-cluster distances may also affect the performance of Chapter 4's classifier. Thus, an evaluation that takes this distances into account seems sensible. In real world instances, it is not easy to vary these distances without violating the relations of shortest paths in the underlying street network. To this end, the evaluation of real world examples is postponed to Section 6.2. First, Section 6.1 introduces euclidean Clustered Dial-a-Ride instances with whose help the evaluation can take place in dependence on the inter-cluster distances.

All experiments were carried out on a computer with above-average performance running Ubuntu Xenial 16.04.3 LTS with Linux 4.13.0-36. It possessed an AMD Ryzen Threadripper 1950X 16-Core Processor (3.4 GHz) with hyperthreading enabled, so there are 32 virtual and real cores in total. The algorithms had access to 125 gigabytes of random access memory and 41 gigabytes of swap. However, the algorithms did not make use of this space because they all are written to use as few storage as possible. As described in the last chapter, the classifier consists of 15 parallel tasks, so half of the cores are used at once. It turned out that while running the classifier none of cores was used to full capacity. This suggests that the cores waited a significant share of their time for new input or that their output was processed. As described in the last chapter, the number of needed parallel workers is fixed for any cluster size. An idea for future work is to balance the number of used cores in accordance with the size of the cluster to be examined. On the other hand, the classifier's running time with the current implementation is already rather neglectable, as the next section shows.

6.1 Euclidean Setting

As discussed in the introduction of the chapter, the euclidean setting simplifies analyzing the algorithms. The meaning of the simplification gets clearer after the structure of euclidean instances is explained. In euclidean instances, all clusters are represented by squares which are vertically centered on the x-axis, as shown in Figure 6.1. Consequently, every cluster has two intersections with the x-axis. The access point of a cluster is its left intersection with the x-axis and the exit point is the right intersection point with the x-axis. For two locations in the plane, their distance is given by the following rule: If both locations are in the same cluster, then their distance is the euclidean length between

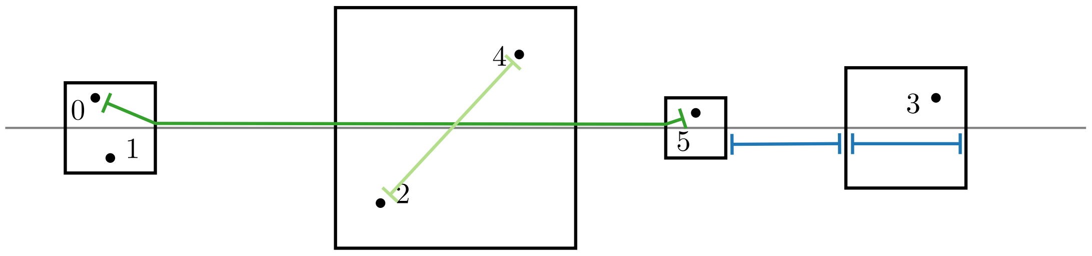


Fig. 6.1: A sketch of an euclidean example. The two green lines indicate the distances between to pairs of points. The blue lines show the intermediate distance between two clusters and the width of a cluster, respectively.

them. If they are in different clusters, their distance is the sum of three components: The distance of the left location to the exit point of its cluster, the distance of the right location to the access point of its cluster and the distance between both border points. An example for these lengths is shown by the greenish lines in the example figure. The blue lines show the definition of the inter-cluster distance and the width of a cluster. Distances are notated without any units in this section, however they can be interpreted to be meters, kilometers or other arbitrary length measures.

The tests described in this section were carried out using randomly generated euclidean instances. To control the variety of test cases, the following properties were fixed throughout the complete test session, except stated differently.

Gaussian Distribution of Cluster Widths The cluster widths where constituted by making use of the Gaussian distribution. The standard deviation was set to 1 000 and the expected mean to 3 000. This seems reasonable because villages normally have a diameter of this magnitude. The minimal cluster width is supposed to be 500 at least, avoiding negative numbers generated by the Gaussian Distribution.

Standard Deviation of Inter-Cluster Distance Although the actual distance between two subsequent clusters was changed for different tests, its standard deviation remained the same. It was set to 2 000, which is small compared to the standard deviation of cluster sizes. The reason is that the performance of the algorithms should be tested in dependence of the inter-cluster distances. Allowing a bigger standard deviation blurs the results obtained from these test. The value of 2 000 is small enough to get sensible results and big enough to allow variety to a certain degree. The minimum distance between to clusters is limited to be 0.

Number Of Clusters The number of clusters in the instances was set to eight. Normally, there are not more than eight villages on a bus route and less villages limit the freedom of the riders too much. For example, suppose there were only six clusters. In order to place 12 riders in the six clusters, the 26 locations must be distributed among the clusters. Since the maximal cluster size is six (see Section 4.4), all clusters would be nearly full, no matter how the locations are distributed.

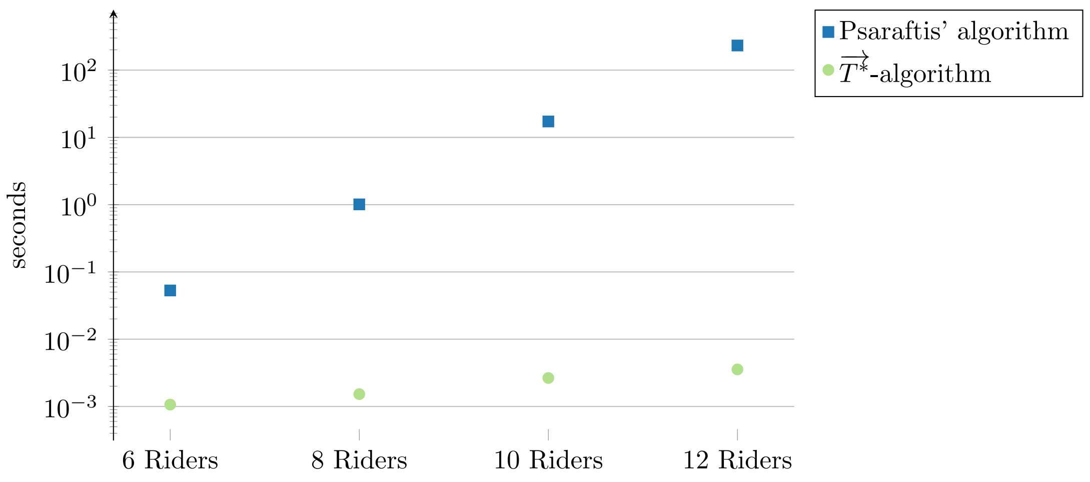


Fig. 6.2: The diagrams show the average running time to solve a Dial-a-Ride instance optimally for different numbers of riders. The squares represent the running time for the $\overrightarrow{T^*}$-algorithm. Notice that the y-axis is drawn in logarithmic scale.

Distribution Of Riders Into Clusters A rider was generated by picking two different clusters and promote the cluster with smaller index to be her pickup cluster, while the other cluster is her dropoff cluster. Then two random points are chosen inside both clusters which realize the actual locations. However, picking two clusters in the first place was not implemented by randomly drawing two cluster indices, but rather in a way that gives advantage to clusters with higher indices. The reason for this is that on bus lines the bus stops tend to be busier the nearer the bus is at the final city. A cluster is chosen by traversing the clusters from right to left and stopping with a probability of 30% at every cluster. This means that one of the two rightmost clusters are chosen with a probability of approximately 50%. As already pointed out in Section 4.4 the bearable cluster size is limited. Therefore, the maximal size of the rightmost cluster was set to 6, and declines to the left. The rate of declining depends on the number or riders being tested. All clusters that are already full are not considered during the traverse.

There are three main algorithms in this thesis: The incremental variant of Psaraftis' algorithm, the $\overrightarrow{T^*}$-algorithm and the classifier. Figure 6.2 shows with squares the running time of Psaraftis' algorithm for different numbers of riders. Since the computational effort to solve an instance does not depend on the distances found in $[d_{i,j}]$ and $[d_i]$, the variations on the inter-cluster distance are not found in the figure. The dots indicate how long the $\overrightarrow{T^*}$-algorithm took in average to solve the same instances. The difference in both running times is very plain. While solving an instance with 12 riders optimally takes approximately two minutes, an unidirectional optimal route $\overrightarrow{T^*}$ can be found in less than 20 milliseconds. It should be noted that the $\overrightarrow{T^*}$-algorithm can be parallelized easily by computing the partial tour for the clusters synchronously. This was *not* done in these tests in order to compare it without bias to the single threaded algorithm of Psaraftis.

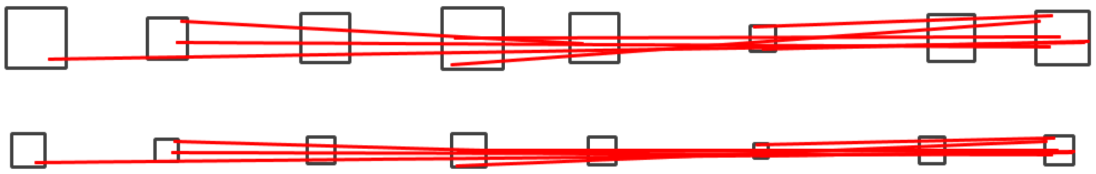
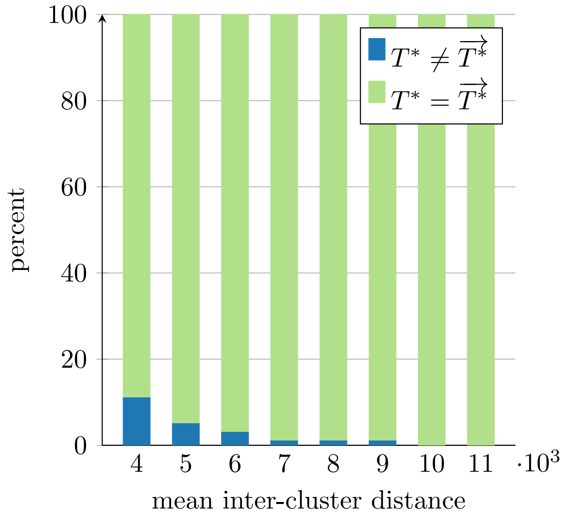


Fig. 6.3: Two times same instance with six riders, but with different inter-cluster distances. It can be seen that the number of riders increases the more the clusters are to the right.

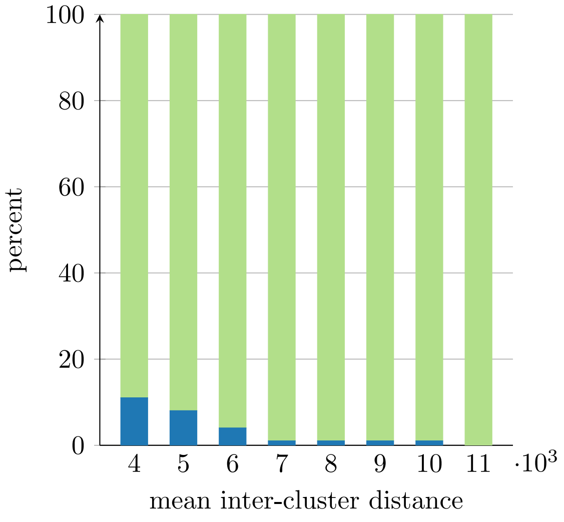
To describe the following results some terminology has to be introduced. A *stack* $S(n,d)$ is a set of 100 randomly created euclidean Dial-a-Ride instances. The number n denotes the number of riders every instance has and d is the mean inter-cluster distance. Stacks $S(n,\cdot)$ with identical values for n form an aggregation $A(n) = \bigcup_{d=4\,000}^{24\,000} S(n,d)$. The union sign increments d in steps of 1 000. An important aspect is that the ith instance of $S(n,d)$ differs *only* in the inter-cluster distances from the ith instance of $S(n,d+1000)$. Figure 6.3 shows an element of $S(6,4\,000)$ above the same element of $S(6,10\,000)$. The similarity between two instances is obvious. Consequently, $A(n)$ contains the same instance 11 times, but with different inter-cluster distances. The stacks $S(12,\cdot)$ are an exception because they contain only 50 instances. The reason for this is that solving 100 instances with 12 riders would have not been practical. Running times are stated as average per instance, so the smaller size of $A(12)$ does not matter.

As discussed above, the tour $\overrightarrow{T^*}$ needs not to be the globally optimal route, but can be. Thus it is worth studying the ratio of instances where $T^* = \overrightarrow{T^*}$. The data belonging to that aspect can be found in Figure 6.4. The four diagrams show clearly that the probability for the optimal tour being an unidirectional tour is higher the wider the clusters lie apart. In all four aggregations it suffices that the mean inter-cluster distance is 18 000 to make all instances in the stack $A(\cdot, 18\,000)$ have unidirectional optimal tours. By comparing the stacks of fixed mean inter-cluster distances between the different aggregations one can also see that the more riders are present the more likely it is that $T^* \neq \overrightarrow{T^*}$: While for $S(12,4\,000)$ the ratio of such instances is greater than 60%, it is approximately 10% for $S(8,4\,000)$ and $S(6,4\,000)$. Depending on the instances at hand, one can obtain an optimal route by using the $\overrightarrow{T^*}$-algorithm in at least 40% of the cases. If the inter-cluster distances are at least 6 000 or the number of riders is at most 10, then $\overrightarrow{T^*}$ is an optimally global tour in at least 50% of the cases.

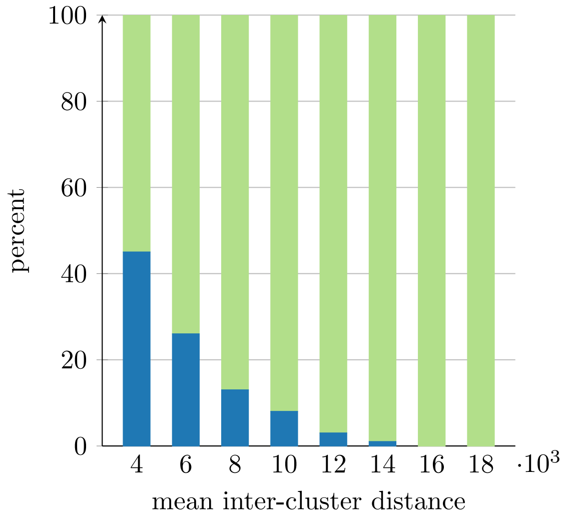
Section 4.2 provides a classifier which decides if for a clustered Dial-a-Ride instance I the equation $T^* = \overrightarrow{T^*}$ holds. Let instances for which $T^* = \overrightarrow{T^*}$ indeed holds be called *positives* and all other instances *negatives*. Positive instances for which the classifier's response was "yes" are *true positives*. On the other hand, negative instances that were recognized by the classifier as such are referred to as *true negatives*. In between are the *false positives* and the *false negatives*. The former instances were erroneously answered with "yes" and the latter ones were erroneously answered with "no" by the classifier. The *precision* of a classifier is the ratio of occurrences of true positives to the occurrences of



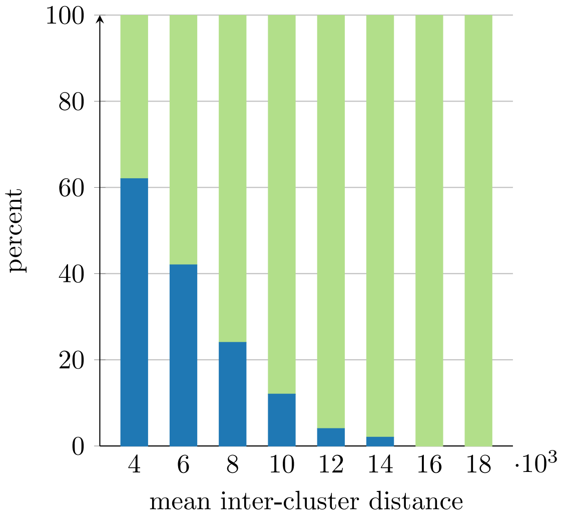
(a) Distribution for 6 riders.



(b) Distribution for 8 riders.

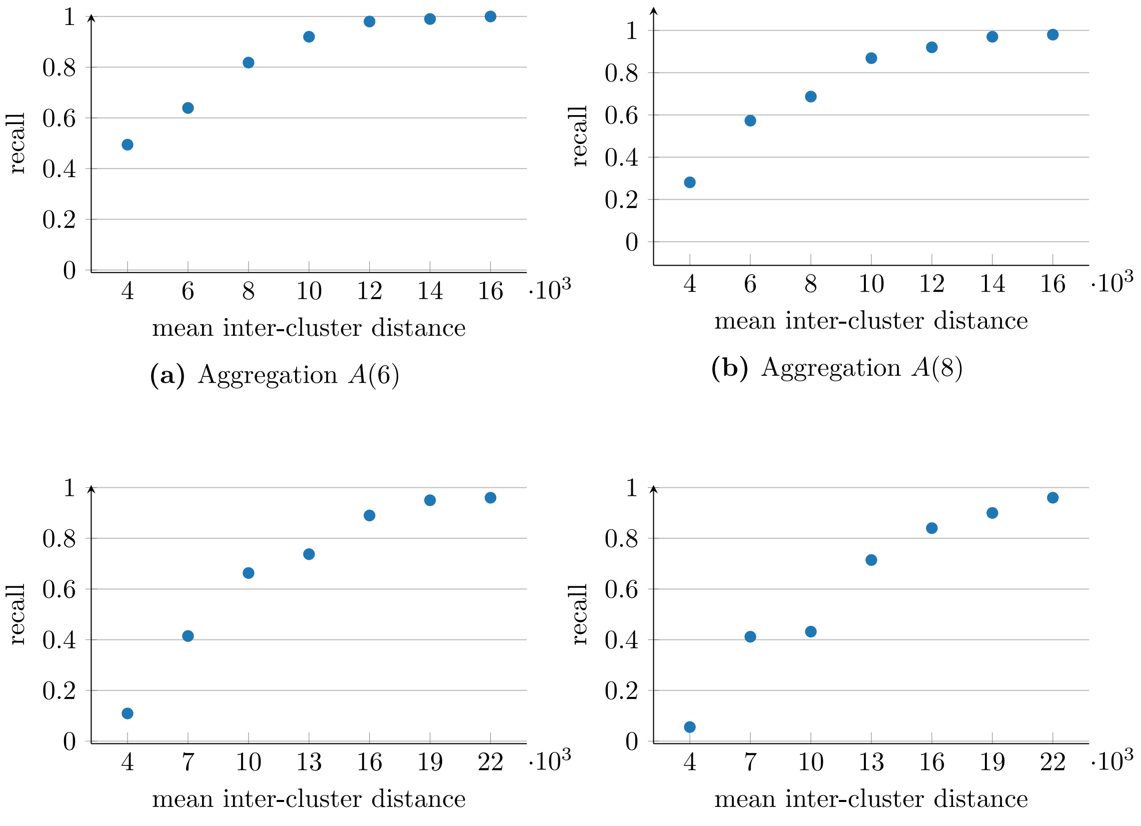


(c) Distribution for 10 riders.



(d) Distribution for 12 riders.

Fig. 6.4: The ratio of instances where $T^* = \overrightarrow{T^*}$. Notice that the upper two diagrams have a different x-scale than the two lower diagrams.



(a) Aggregation $A(6)$ **(b)** Aggregation $A(8)$

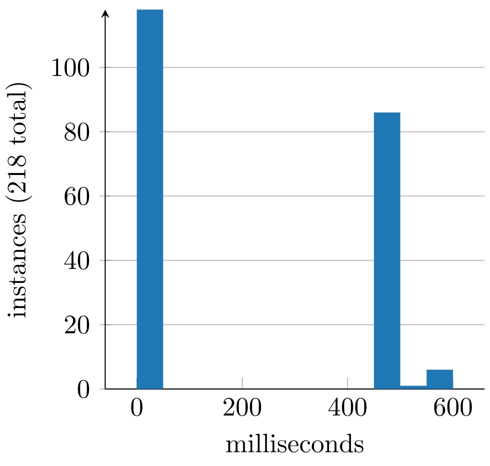
(c) Aggregation $A(10)$ **(d)** Aggregation $A(12)$

Fig. 6.5: The classifier's recalls for the different stacks and aggregations. Notice that the upper two diagrams have a different x-scale than the two lower diagrams.

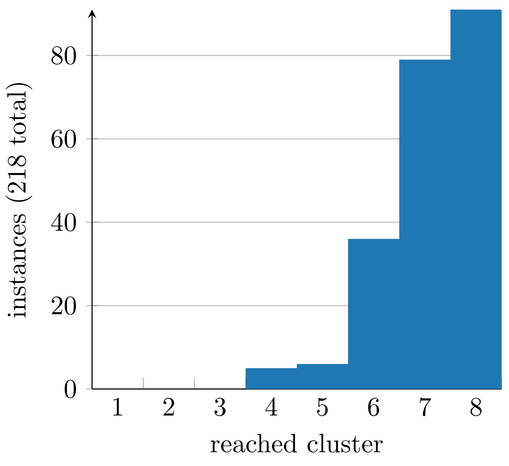
true positives and false positives. Since the classifier of Section 4.2 does, per definition, never produce false positives (if the assumptions of Section 4.1 are met), the precision of this specific classifier is always 1. Consequently, the classifier recognizes correctly that all instances in the lower parts of the stacks in Figure 6.4 fulfill $T^* \neq \overrightarrow{T^*}$.

The *recall* denotes the ratio of occurrences of true positives to the occurrences of true positives and false negatives. The recall for the four aggregations is shown in Figure 6.5. Spoken vividly, a recall of 0.5 means that half of the instances for which $T^* = \overrightarrow{T^*}$ holds were actually recognized by the classifier. The figures make it clear that the errors made by the classifier become the smaller the higher the inter-cluster distance is. Therefore the stacks of Figure 6.4 consisting of only one part are nearly all classified correctly.

Following from Figure 6.5, the classifier has a decent error rate for inter-cluster distance below 10 000 and a very good error rate for higher inter-cluster distances. Thus, from the point of view of accuracy it is worth using the classifier. The other important aspect is the time it takes to compute a response. The classifier was implemented so that it checks one cluster after an other from left to right if it might be cheaper to handle the cluster piecewise versus handling it in one tour. If a cluster was found that was cheaper



(a) Histogram of the running times in the 218 “no”-instances of $A(6)$



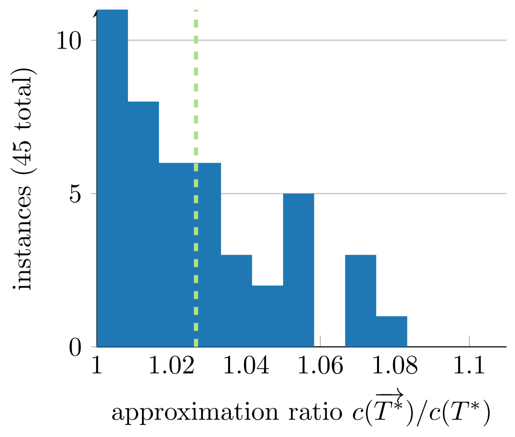
(b) In about 45% of the cases, the classifier reached the last cluster.

Fig. 6.6: The share of instances in which the last cluster was reached corresponds to the share of instances in which the running time of the classifier took longest.

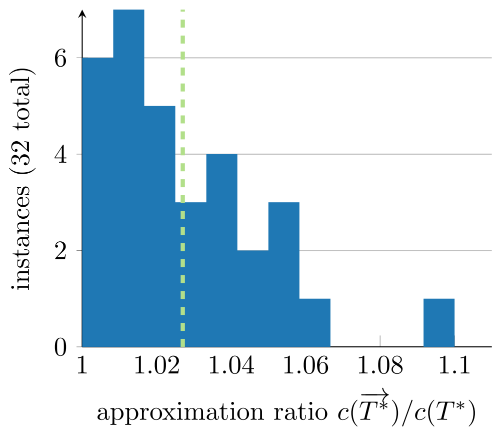
to handle piecewise then the answer can only be “no”. In this case the calculation was canceled immediately. Thus it is sensible to distinguish the running time of the classifier into two cases.

The first case concerns the instances in which the classifiers answer was “yes”. In these instances, the classifier handled *every* cluster, maximizing the computational effort it could possibly have. The average running time of the classifier in positive $A(12)$ instances was approximately 4.2 seconds and did not vary greatly.

The second case is more interesting. It examines the classifier’s running time in negative-claimed instances. The running times of these classifications varies greatly because it depends on how far the classifier gets until it finds the cluster aborting the calculation. The size of the clusters play also an important role because the Dial-a-Ride problem must be solved internally for every cluster. Figure 6.6a shows a histogram of the running times of all instances in $A(6)$ that were answered with “no”. There are two ranges of running times. The first range is from 0 milliseconds to 50 milliseconds and the second range is from 450 milliseconds to 600 milliseconds. The lower range contains 104, the higher range 114 of the 218 instances. The reason for this clear split lies in the predefined cluster sizes. As described above, the generation of the artificial instances preferred clusters lying further to the right to simulate traffic towards a bigger city. In $A(6)$, the rightmost cluster was allowed to have six locations in it and the other clusters contained three riders at most. Thus, all instances lying in the first range could be classified before reaching the last cluster. The instances lying in the second range, on the other hand, were decided with the rightmost cluster, for which the internal Dial-a-Ride problem took longer. To cross-check this conjecture, the indices of the clusters at which the classifier exited were recorded. A histogram which counts the indices of these clusters is presented in Figure 6.6b. These records support the conjecture that the last



(a) Approximation ratio in $S(10, 4\,000)$, there are 45 relevant instances (of 100).



(b) Approximation ratio in $S(12, 4\,000)$, there are 32 relevant instances (of 50).

Fig. 6.7: The histograms show the empiric approximation ratios in $S(10, 4\,000)$ and $S(12, 4\,000)$. The dashed lines indicate the mean.

cluster was responsible for the long running times in the second range. Interestingly, the shape of the diagram in Figure 6.6a is nearly identical for all stacks $S(6, \cdot)$.

The same analysis was carried out for $A(12)$, with similar results. A notable difference between $A(12)$ and $A(6)$ is that in $A(12)$ the right range is not from 450 ms to 600 ms, but from 3 000 ms to 4 000 ms. The reason for this range shift is that in $A(12)$ the clusters were more occupied than in $A(6)$. In $A(12)$, the last cluster had a maximal size of six and the last-but-one cluster had a maximal size of five. Combined with the exponential running time of Psaraftis' algorithm this leads to a shift of the maximal running times.

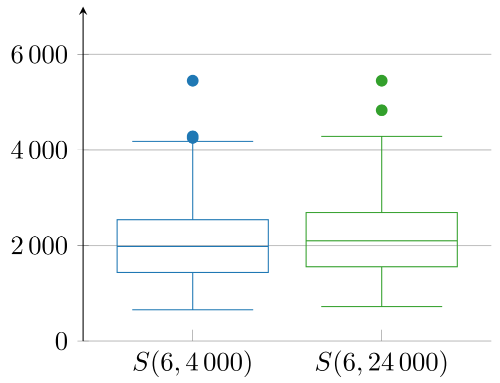
The key point of this analysis is that the running time of the classifier depends on the instance at hand. For a similar distribution of riders into clusters like the examined distributions, the classifier can be expected to finish after several seconds, compared to a couple of minutes for running Psaraftis' algorithm. Using the classifier is therefore always worthwhile because the potential time loss is neglectable.

The time to solve an instance of 12 riders to optimality is rather high compared to the running time of the $\overrightarrow{T^*}$-algorithm. As seen in Figure 6.4 the probability that $\overrightarrow{T^*} \neq T^*$ is rather small when the inter-cluster distances grow. An understandable idea is to always use the $\overrightarrow{T^*}$-algorithm and accept non-optimal routes. The question is how bad $\overrightarrow{T^*}$ is compared to the globally optimal route T^*. For a given instance I, the *approximation ratio* is defined by $c(\overrightarrow{T^*})/c(T^*)$. It turns out that the worst approximation ratio that was experienced in the test data is approximately 1.1. This occurs in the stack $S(12, 4\,000)$. However, the mean ratio is 1.03 in $S(12, 4\,000)$, as well as in $S(10, 4\,000)$. Figure 6.7 shows the two histograms that illustrate the magnitude of approximation ratios. It is important to note that both diagrams only consider instances for which $\overrightarrow{T^*} \neq T^*$. In other stacks than $A(12, 4\,000)$ and $A(10, 4\,000)$ the share of these instances is lesser and illustrating their errors is not sensible for these small number of instances. To conclude, in most cases the $\overrightarrow{T^*}$-algorithm will yield an optimal tour and if it does not, then the tour is nearly optimal. However, there is no theorem that limits the error that can be made, so using the $\overrightarrow{T^*}$-algorithm without the classifier should be done with care.

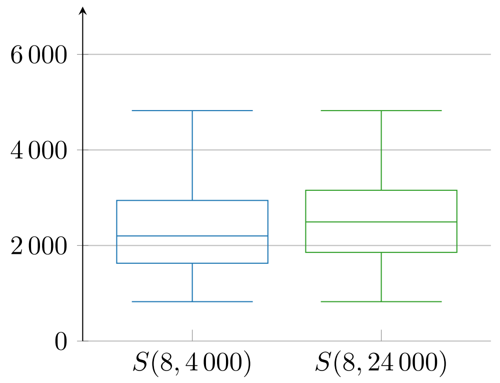
The next paragraph examines the absolute difference in the objective function of $\overrightarrow{T^*}$ and T^*. However, interpreting the absolute values of the objective function is really difficult. For example, an instance in $S(12, 4\,000)$ has $c(T^*) = 334\,545$ and $c(\overrightarrow{T^*}) = 337\,686$. Both values carry not much information about the routes themselves. The really interesting topic is not the objective value itself but rather the detours that are experienced by the riders. The detour of a rider in a tour T is the difference of distance he really travels in T and the direct distance between is pickup and dropoff location. The former value can be computed from T in linear time, the latter value is stored in the distance matrix $[d_{i,j}]$. It seems plausible that there is a psychological bound on the detours the passengers accept. If a bus tour induces detours greater than this limit, the passengers will chose other means of transport. The author is not aware of any studies on such accepted detours, so the analysis of detours can only be made without any judging.

Figure 6.8 shows the average detours experienced by the riders of the dedicated stacks in the optimal tour. For example, a randomly selected rider from any instance of $S(6, 4\,000)$ has to accept a detour in the range from $1\,750$ to $2\,250$ with probability 50%. In $S(10, 4\,000)$ there was one rider who had to drive a detour of $6\,100$. In general, the detours induced by the optimal tour increase with a growing number of riders and rising inter-cluster distances. Logic tells that the detours induced by a non-unidirectional route grows by at least $1\,000$ if the inter-cluster distances are increased by $1\,000$. Deducing from the Figure 6.8, the detours for $S(\cdot, 12\,000)$ could be in the range around $26\,000$. However, they are not, because at some d the optimal route becomes unidirectional. Once the optimal route is unidirectional, the detours do not increase any more. Yet, the detours in $S(\cdot, 12\,000)$ are longer than in $S(\cdot, 4\,000)$ because in the optimal unidirectional tour the riders are obliged to witness all riders boarding and unboarding during the journey. At low inter-cluster distances there are fewer unidirectional optimal tours and some riders can be served in a more direct way than in the unidirectional tour. As the four stacks show, this effect is the more obvious the more riders are present. The figures only show the detours for the optimal tour T^*. If $\overrightarrow{T^*}$ is more expensive than T^*, then the difference of the objective costs distributes to the riders, but it was not examined how this distribution looks like. One extreme is that the difference is shared equally by all drivers, the other extreme is that one rider receives the entire difference alone. Most likely, the truth lies somewhere in between.

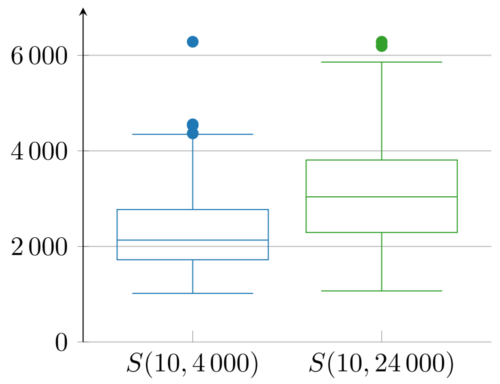
The next aspect of the artificial instances' evaluation consists of the potential improvements and modifications for the classifier. Since the classifier's heart consists of the estimation of the lower bound $\Phi(C_i)$ it is sensible to improve this lower bound. If an instance I was incorrectly declared as negative then for a cluster C_i in I the inequality $\Phi(C_i) < \Upsilon(C_i, \overrightarrow{T^*})$ was fulfilled, i.e. there was a cluster for which the lower bound on serving it in several parts was cheaper than serving it in one part. This is no contradiction to the optimal route being unidirectional since the classifier does not take pairs of clusters into account.



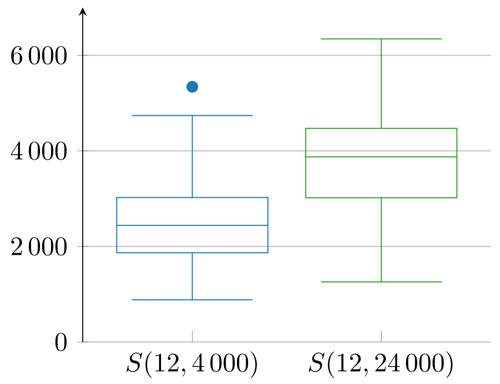
(a) Detours for 6 riders.



(b) Detours for 8 riders.



(c) Detours for 10 riders.



(d) Detours for 12 riders.

Fig. 6.8: The average detours of the riders (without the driver) in the dedicated stacks shown in Tukey box plots [14]. The outliers are defined by the interquartile range $k = 1.5$.

The best solution to solve C_i in several partitions may cause immense costs in the neighboring clusters, but since the classifier works cluster-wise, these potential costs are not considered in the lower bound. An open problem for future research is to combine the lower bound of several clusters to make the prediction more robust. In order to turn a false negative into a true positive, an improvement of the estimation must span the difference between both numbers so that the inequality sign is flipped. Figure 6.9 shows that the lower bound is relative close to $\Upsilon(C_i, \vec{T^*})$, in nearly all cases the difference was below the mean inter-cluster distance. This means that if only one additional (and unnecessary) journey to one of the neighboring cluster could be identified by the classifier, then the chances are high that the recall of the classifier rises significantly. On the other hand, an improvement of the cluster's internal tours may likely yield an improvement, too, because there are many instances with a difference in the range under 1 000.

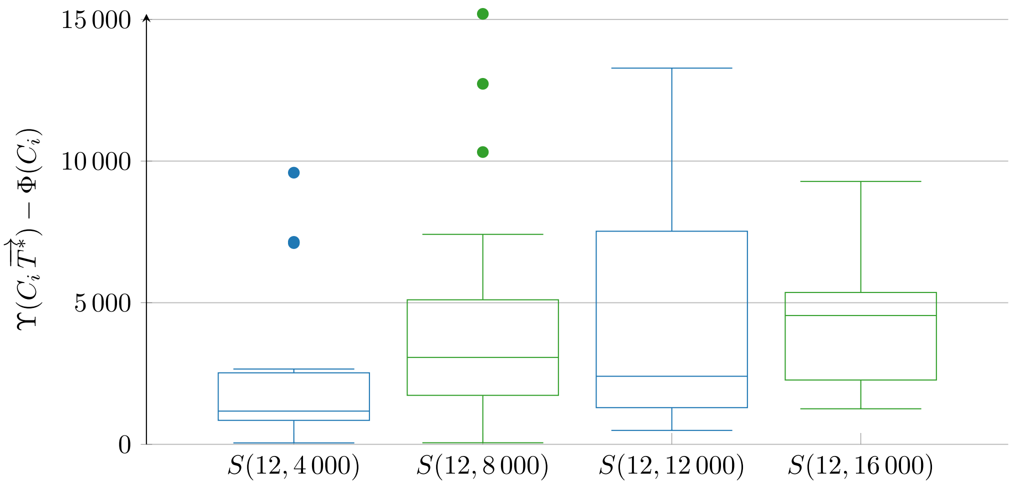


Fig. 6.9: The difference between $\Upsilon(C_i, \overrightarrow{T^*})$ and $\Phi(C_i)$ in instances for which the classifier incorrectly stated that $T^* \neq \overrightarrow{T^*}$.

An interesting modification of the classifier consists of altering it so that it can cope with other objective functions. Section 3.1 introduced the objective to minimize travel distance *and* waiting time. To simplify matters, waiting time is regarded to be equivalent to travel distance. The optimal tour of all instances of $A(6), A(8)$ and $A(10)$ were computed with the modified target function. Of these instances, only five had non-unidirectional tours. These instances had a mean inter-cluster distance of less than 6 000 and at most eight riders. The reason why there are so few non-unidirectional routes is simple. Let T be a tour and e be one edge in the tour. Then the costs of e are multiplied with the number of riders that have not yet been delivered. Consequently, the long distances between the clusters weight heavier than in the former objective function. It is advisable to use as few of them as possible which results in more unidirectional optimal routes. The classifier can also handle the new objective function because the lower bound remains valid. However, the accuracy sinks enormously: For $S(6, 4\,000)$, the recall is only about 0.07 which is significantly worse than the recall depicted in Figure 6.5 for the same stack. After increasing the inter-cluster distances to 24 000, the recall for $S(6, 24\,000)$ is again 1. In general, the recall gets better for increasing inter-cluster distances and worse for more riders. In principle, it should be easy to modify the classifier to incorporate the new objective function adequately by adjusting the weights of Algorithm 5. In contrast, the prevalence of instances with non-unidirectional routes is very small, so just using the $\overrightarrow{T^*}$-algorithm as heuristic should be acceptable in most cases.

This completes the picture of the artificial euclidean instances which had the nice property that the distances between the locations were easy to understand. The main point of the analysis was to discover the impact of inter-cluster distances on the structure of the optimal tour. The next section turns to more realistic examples in which the distances and locations are based on authentic geographical data.

6.2 Geographical Setting

While the last section dealt with artificial instances and simple distance matrices, this section addresses real world applications. The main goal is to investigate how often unidirectional tours occur in practice and how good the optimal unidirectional tour $\overrightarrow{T^*}$ is compared to T^* . Three scenarios are investigated, each of them with a different use case. All distances are based on a street network. Thus it might occur that the distance between two locations implied by the border points of the cluster is greater than the direct connection between the locations.

Rural Bus Line This scenario reflects the ideas from the previous section. Its model is a rural bus line connecting six small villages and with the german city Würzburg. The mean inter-cluster distance is 1.2 km, the minimum distance is 808 m and the maximum distance is 3.4 km. The villages have a diameter ranging from 610 m to 2.6 km. The average diameter is 1.3 km. The capacity of the last cluster was set to six, all other villages can contain at most four locations. The riders are distributed randomly inside the clusters without simulating a higher density towards the last clusters like in the last section.

Regional Bus Line The regional bus line connects six smaller towns in the region of Lower Franconia but does not serve small villages. The towns are located along a federal highway. The average distance between two towns is 7.9 km, ranging from 2.4 km to 11 km. The diameters of the villages range from 1.3 km to 6.4 km. All clusters have a capacity of six and are chosen equally likely. The interesting part of this scenario is that the federal highway follows a river, but the shortest distance between two towns is shorter because the river makes a turn. The distance matrix $[d_{i,j}]$ encodes the shorter distance, but the cluster quadruple Q obeys the ordering of the clusters.

Intercity Bus The intercity bus connects six major german cities: Munich – Ingolstadt – Nuremberg – Erfurt – Magdeburg – Berlin. The smallest distance between two cities is 67 km, the longest 218 km, with a mean of 129 km. The diameters are also greater than in the other two scenarios: Ranging from 1.5 km to 20 km. Both values express two extreme situations: Ingolstadt is located left of an autobahn. To enter the city, the driver re-enters the autobahn via the same junction as he left it. Thus, the distance between Ingolstadt's border points is rather small. Magdeburg, on the other hand is very stretched and the shortest path through the city is 20 km long.

All three scenarios were tested with 100 randomly generated instances, each of them containing 10 riders. These numbers form a compromise to generate a great variety of instances while keeping the running time to solve them in an acceptable time frame.

Figure 6.10 illustrates the performance of the classifier. Of all rural instances 77% have a unidirectional optimal tour. However, the classifier would only state 5% of them correctly. It makes sense to compare the rural stack to $S(10, 4\,000)$ from the previous

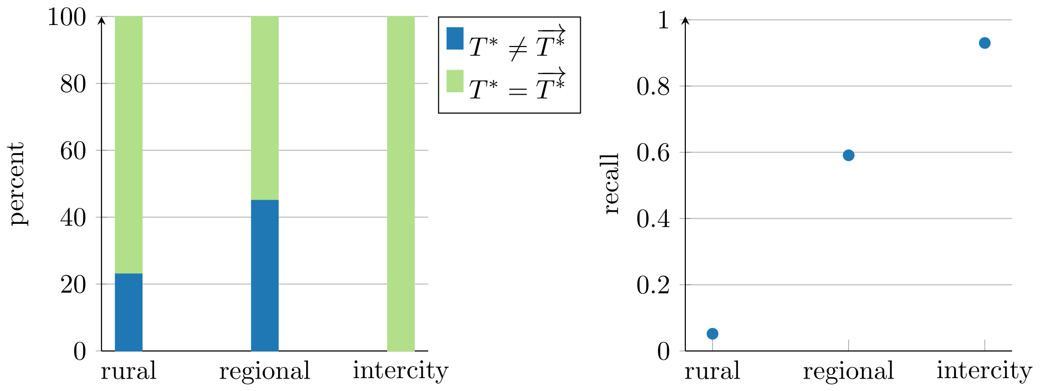


Fig. 6.10: The results of the three geographical stacks. The effect of the difficult regional instance can easily be recognized in the left picture.

section. It does not surprise that the recall of the rural stack is smaller than that for $S(10, 4\,000)$ because the mean inter-cluster distances are significantly smaller. It does surprise that the number of instances with unidirectional optimal tours is nevertheless higher than in the artificial case. The reason for this likely is the special metric of the underlying street network but the exact cause could not be determined. An important topic to stress is that the classifier did not produce false positives in the rural stack. Remember that per definition, if the classifier responds $T^* = \overrightarrow{T^*}$, then this is correct, i.e. false positives are impossible. Yet, there is a restriction on this strong statement because it is only correct when all the assumptions made in Section 4 are fulfilled. However, street networks often violate these assumptions and irritate the classifier. One of these assumptions is that is never advisable to *bypass* a cluster. In the evaluation of the last section this was achieved by using the euclidean distances between clusters.

In reality, this condition is not met, as Figure 6.11 demonstrates. On the left is the expected order of clusters, but one instance used the order on the right instead. Other similar cases could be found in the rural stack. In all these instances not one false positive was generated. This suggests that the classifier is robust to some degree if the idealistic assumptions under which it was developed and proven correctly are not met.

The regional bus line was designed to ignore all assumptions completely. This results not only in fewer instances in which $T^* = \overrightarrow{T^*}$ does hold (Figure 6.10) but also in several false positives. In these cases the classifier said that $T^* = \overrightarrow{T^*}$ but that was not true. There were 16 false positives in the regional stack, thus the classifier's precision in this stack was 0.61 which is rather small compared to the precision of 1 in all stacks encountered so far. As already mentioned, the regional bus line is a difficult scenario for the classifier. Figure 6.12 shows why this scenario is so difficult as well as one tour which was classified incorrectly. The reason for the classifier failing on these kind of instances is that there is a huge discrepancy between the distances in $[d_{i,j}]$ and the distances in $[d_i]$. Psaraftis' algorithm uses only the former distances while the classifier's response is mainly based on the latter distances. If these distances state that two locations are wider apart than they are in reality, then the classifier likely reaches a wrong conclusion. In the case

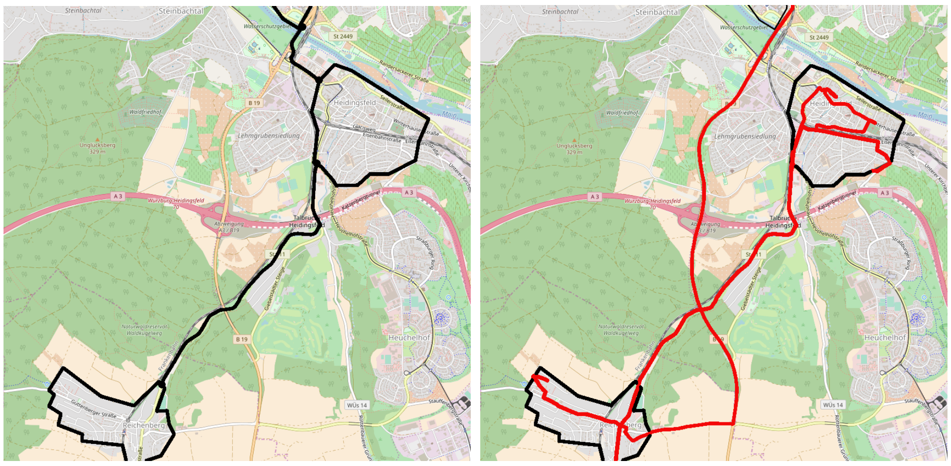


Fig. 6.11: The left image shows the intended order of the clusters, the right image shows one Dial-a-Ride instance with an optimal tour that does not obey the intended order.

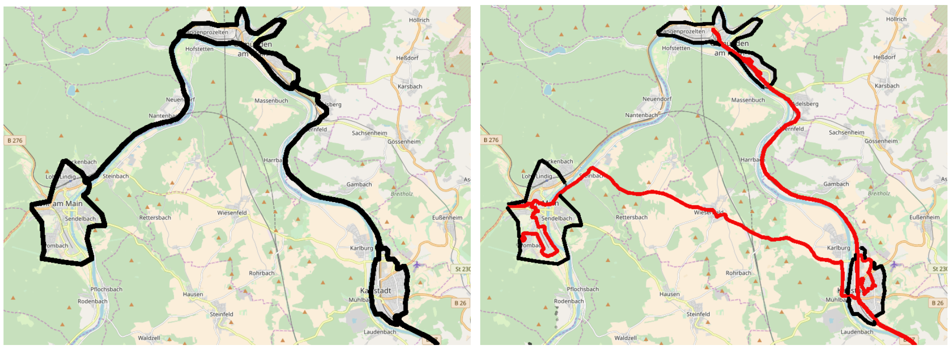


Fig. 6.12: The supposed tour follows the river while many of the optimal tours bypass the nothern city or visit it later to avoid detours for riders boarded in the leftmost town.

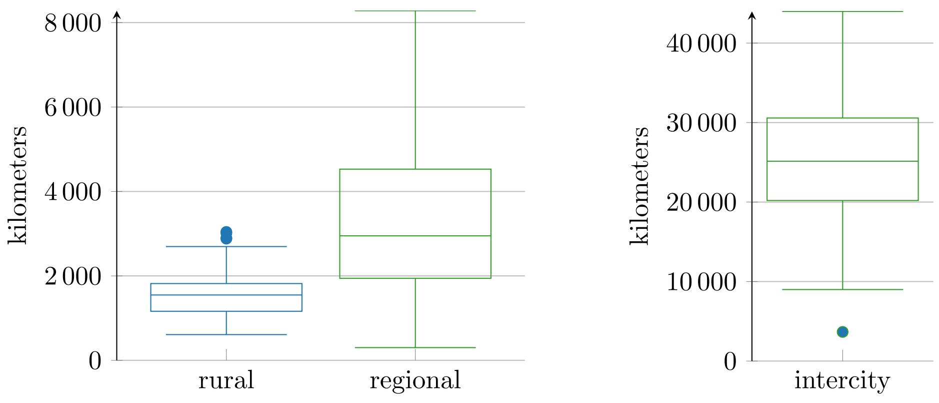


Fig. 6.13: The average detour a rider experiences in the optimal route. A different scale had to be used for the intercity scenario because the detours are very big in this setting.

at hand, the classifier assumes that the shortest path from the left to the right town goes through the northern town because the clusters are defined in that way. When the clusters are ignored, then there is a shorter path which bypasses the northern village. However, the regional cases examined here is very extreme and the rural scenario shows that *some* discrepancy is allowed between the two distance matrices.

To avoid false positives both distance matrices can be harmonized manually so that they encode the same distance between two locations. Essentially this means pretending that the shorter of the two alternative connections between the two locations is not existent. The downside of this solution is that it forces the riders to take detours in *all* instances, pretty much like a bus line induces detours for some riders. These detours occur independently from the customers present at a specific day.

Unsurprisingly, all instances in the intercity scenario have an unidirectional optimal tour, of which the classifier found 93%. In all but one of the seven false negative instances the classifier reached only Nuremberg. The distance between Nuremberg and Ingolstadt is the smallest inter-cluster distance in the scenario, so it is plausible that the classifier fails there. In one false negative Ingolstadt was reached which is understandable for the same reason.

Again, the acceptance of a service such as Dial-a-Ride depends on the detours made. To this end, the average detours by all riders are examined. This examination meets the expectations: In the rural setting the detours implied by the optimal route were relatively small, and in the intercity scenario riders have to accept detours of tens of kilometers. Figure 6.13 uses box plots to illustrate the detours for the different scenarios.

Especially in scenarios similar to the intercity setting, the temptation exists to use the $\vec{T}^*$-algorithm as a heuristic. It turned out that the approximation ratio in all three scenarios was very similar to those of Figure 6.7 for the euclidean instances. The average approximation ratio for negatives was around 1.02 for the rural scenario and 1.07 in the regional scenario. However there are a few outliers, the worst of them was an approximation ratio of 1.28. In all outliers a similar situation as in Figures 6.11 and 6.12 was met. A bad approximation ratio in these instances is acceptable, more important is that the approximation is indeed quite good for sensible instances.

This concludes the evaluation of the classifier. It became clear that the classifier is a powerful tool that can shorten the time to compute an optimal route significantly in many cases because if the classifier declares an instance to be a “yes”-instance, then a faster algorithm can be used. If the classifier returns “no”, then only the time expensive algorithm by Psaraftis’ guarantees an optimal solution. If an instance was erroneously classified with “no”, than the slower algorithm is used unnecessarily, causing a loss of time. Fortunately, these cases are, depending on the instances at hand, rather rare. If an instance really does not admit a unidirectional optimal route than the loss of time by using the classifier is neglectable. The next and last chapter summarizes the findings of the thesis and proposes directions for future work.

7 Conclusion and Future Work

In order to set up a system for rural public transportation which combines the cheapness of traditional bus routes with the convenience of door-to-door deliveries offered by taxi cabs many aspects must be considered. Besides economical studies one has also to focus on routes that are ecologically sensible and do not discourage potential customers from using the service. This thesis dealt with approaches to compute tours that minimize a certain objective function, for example total distance driven or waiting times. In Chapter 3 the Dial-a-Ride problem was introduced formally and a basic algorithm to solve it was presented. However, the Dial-a-Ride problem and the existing algorithms did not portray the structure of instances evolving in rural environments. To this end, the problem definition was extended to the Clustered Dial-a-Ride problem in Chapter 4.

This extended problem variant allowed for the $\overrightarrow{T^*}$-algorithm which computes the best route serving the villages one-after-another. The evaluation showed that there are many cases in which this optimal unidirectional route is also the globally optimal route. In order to distinguish such cases, Section 4.2 presented the idea of classifying instances. This led to a concrete classifier which decides if for a given instance T^* equals $\overrightarrow{T^*}$. Per definition, the classifier can never produce false positives if the instance at hand satisfies some natural restrictions. Although not obeying them may lead the classifier to false conclusions, tests showed that the classifier remains robust even if some of these restrictions are violated.

The classifier was evaluated in Chapter 6. The main result was that utilizing the classifier in order to check if a specific instance admits the $\overrightarrow{T^*}$-algorithm is worth waiting a few seconds for the answer. Tests on real geographical data showed that the classifier is best used in regional settings connecting several towns with wide inter-cluster distances. In settings involving many small villages with low distance between them, the classifier is not that accurate. However, one can use the $\overrightarrow{T^*}$-algorithm and accept non-optimal routes in less than a quarter of the cases, especially given the good empiric approximation ratio of the $\overrightarrow{T^*}$-algorithm.

The prospect to future work on the computational side is threefold. The first point addresses the estimation at the heart of the classifier: It may be possible to increase the lower bound so that more instances are recognized correctly. Additionally, it may be also worth to study the interaction of several clusters in the estimation process. The second suggestion is more technical. The classifier's implementation is parallelized, but without a profound study on the impact of the number of processes. Chances are high that some time can be gained by dynamically spawning parallel processes. The last topic concerns the objective function. The classifier of this thesis can deal adequately with one objective function only, namely minimizing the total distance driven. It is desirable to generalize it so that other objective functions can be used easily.

Bibliography

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

[2] Norbert Ascheuer, Michael Jünger, and Gerhard Reinelt. A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Computational Optimization and Applications*, 17(1):61–84, 2000.

[3] Francesco Carrabs, Raffaele Cerulli, and Jean-François Cordeau. An additive branch-and-bound algorithm for the pickup and delivery traveling salesman problem with lifo or fifo loading. *INFOR: Information Systems and Operational Research*, 45(4):223–238, 2007.

[4] Jean-François Cordeau, Manuel Iori, Gilbert Laporte, and Juan José Salazar González. A branch-and-cut algorithm for the pickup and delivery traveling salesman problem with lifo loading. *Networks*, 55(1):46–59, 2010.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 3 edition, 2009.

[6] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 3. Springer, 2015.

[7] George Dantzig. *Linear programming and extensions.* Princeton University Press, 1966.

[8] George Dantzig and John Ramser. The truck dispatching problem. *Management Science*, 6(1):80–91, 1959.

[9] Jacques Desrosiers and Marco E Lübbecke. A primer in column generation. In *Column Generation*, pages 1–32. Springer, 2005.

[10] Chao Ding, Ye Cheng, and Miao He. Two-level genetic algorithm for clustered traveling salesman problem with application in large-scale tsps. *Tsinghua Science and Technology*, 12(4):459–465, 2007.

[11] Alan AA Donovan and Brian W Kernighan. *The Go programming language.* Addison-Wesley Professional, 2015.

[12] Florian Drews and Dennis Luxen. Multi-hop ride sharing. In *Sixth Annual Symposium on Combinatorial Search*, 2013.

[13] Irina Dumitrescu, Stefan Ropke, Jean-François Cordeau, and Gilbert Laporte. The traveling salesman problem with pickup and delivery: polyhedral results and a branch-and-cut algorithm. *Mathematical Programming*, 121(2):269, Jul 2008.

[14] Michael Frigge, David C Hoaglin, and Boris Iglewicz. Some implementations of the boxplot. *The American Statistician*, 43(1):50–54, 1989.

[15] Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

[16] Robert Geisberger, Dennis Luxen, Sabine Neubauer, Peter Sanders, and Lars Volker. Fast detour computation for ride sharing. *arXiv preprint arXiv:0907.5269*, 2009.

[17] Oliver A Gross. Preferential arrangements. *The American Mathematical Monthly*, 69(1):4–8, 1962.

[18] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[19] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[20] IBM ILOG. IBM ILOG CPLEX Optimization Studio. `https://www.ibm.com/analytics/cplex-optimizer`. Accessed on August 25th, 2018.

[21] IBM ILOG. Optimization Programming Language. `https://www.ibm.com/analytics/optimization-modeling`. Accessed on August 25th, 2018.

[22] Travis Kalanick and Garret Camp. Uber. `https://www.uber.com/`. Accessed on August 8th, 2018.

[23] Bahman Kalantari, Arthur V. Hill, and Sant R. Arora. An algorithm for the traveling salesman problem with pickup and delivery customers. *European Journal of Operational Research*, 22(3):377 – 386, 1985.

[24] Peter Karich, Stefan Schröder, and Michael Zilske. GraphHopper. `https://github.com/graphhopper/graphhopper`, 2018.

[25] Geofabrik GmbH Karlsruhe. Geofrabik Downloadserver. `http://download.geofabrik.de/`. Accessed on August 30th, 2018.

[26] AKM Khan, Oscar Correa, Egemen Tanin, Lars Kulik, and Kotagiri Ramamohanarao. Ride-sharing is about agreeing on a destination. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 6. ACM, 2017.

[27] Frédéric Mazzella. BlaBlaCar. `http://www.blablacar.de/`. Accessed on August 8th, 2018.

[28] OpenStreetMap contributors. Planet dump retrieved from `https://planet.osm.org`. `https://www.openstreetmap.org`, 2018.

[29] Harilaos N Psaraftis. A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. *Transportation Science*, 14(2):130–154, 1980.

[30] Johannes J Schneider, Thomas Bukur, and Antje Krause. Traveling salesman problem with clustering. *Journal of Statistical Physics*, 141(5):767–784, 2010.

[31] Alexander Schrijver. On the history of combinatorial optimization (till 1960). *Handbooks in Operations Research and Management Science*, 12:1–68, 2005.

[32] Neil JA Sloane et al. The on-line encyclopedia of integer sequences, 2003.

[33] Martin Steiger. JXMapViewer2. `https://github.com/msteiger/jxmapviewer2`, 2018.

[34] William Welch, Russel Chisholm, David Schumacher, and Subhash R Mundle. Methodology for evaluating out-of-direction bus route segments. *Transportation Research Record*, 1308:43–50, 1991.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 5. Oktober 2018

. . . .
Fabian Feitsch