Intermodal Tour Planning for Delivery and Pickup Activities for CEP Service Providers

Algorithm Engineering Research Group
Institute of Theoretical Informatics and
Institute for Transport Studies
Department of Civil Engineering, Geo and Environmental Sciences
Karlsruhe Institute of Technology

Bachelor's Thesis

of

Jordan Körte

Reviewers:
Prof. Dr. rer. nat. Peter Sanders
Prof. Dr.-Ing. Peter Vortisch

Advisors:
M.Sc. Robin Andre
M.Sc. Jelle Kübler
M.Sc. Moritz Laupichler

Date of Submission: 20.03.2024
Karlsruhe

Acknowledgment

The author is a scholarship holder of the Heinrich Böll Foundation.

Abstract

The demand for parcel deliveries in city centers has been rising steadily for years. This development is also problematic due to the climate-damaging exhaust gases and the large amount of space required by delivery vehicles. Scientists are therefore developing new concepts for deliveries in city centers that use trams and cargo bikes in particular as transporting vehicles for parcels. However, there is a lack of algorithms customized for these concepts that take intermodal delivery into account. This thesis therefore introduces an algorithm for intermodal tour planning that focuses on delivery using a combination of tram and cargo bike. The algorithm is based on the Ant Colony Optimization metaheuristic and supports deliveries and pickups as well as time windows and limited capacities. The thesis demonstrates the practicability of the introduced algorithm using street networks of existing cities and synthetically generated parcel demand.

Zusammenfassung

Die Nachfrage nach Paketlieferungen in Innenstädten steigt seit Jahren kontinuierlich an. Problematisch ist diese Entwicklung auch wegen der klimaschädlichen Abgase und des hohen Platzbedarfs der Zustellfahrzeuge. In der Wissenschaft werden daher aktuell neue Konzepte für Zustellungen in Innenstädten entwickelt, die insbesondere Trams und Lastenfahrräder als Transportmittel für Pakete nutzen. Diesen Konzepten mangelt es jedoch an geeigneten Algorithmen, die intermodale Zustellung berücksichtigen. Daher führt diese Arbeit einen Algorithmus zur intermodalen Tourenplanung ein, der den Fokus auf die Zustellung mit der Kombination aus Tram und Lastenfahrrad legt. Der Algorithmus basiert auf der Ameisenalgorithmus-Metaheuristik und unterstützt Zustellungen und Abholungen sowie Zeitfenster und begrenzte Kapazitäten. Die Arbeit demonstriert die Praxistauglichkeit des eingeführten Algorithmus anhand von Straßennetzen existierender Städte und synthetisch generiertem Paketaufkommen.

Contents

List of Figures

List of Tables

1 Introduction

In the context of the climate crisis, the continuously increasing demand for parcel delivery in city centers poses a problem. Last mile delivery is typically achieved with space-consuming vans powered by fossil fuels. Alternative concepts that reduce overall emissions and space consumption are therefore desired.

Recently, the idea of using cargo bikes for last-mile delivery in urban centers has gained traction [1]. Cargo bikes have a much lower capacity and movement speed compared to vans but are less space-consuming and emission free. Since the depots of courier express parcel (CEP) service providers are often located outside of city centers, trams are being considered for transport from depots to so-called city hubs in urban centers. These city hubs then act as micro-depots for cargo bikes. A city hub constitutes a transfer point for deliveries from tram to cargo bike and for pickups from cargo bike to tram.

Depending on the implementation, the trams may have a special compartment for parcels or they may transport the parcels alongside the passengers. In the latter case, the passengers have priority over parcels. As a result, the volume of parcels that can be delivered by tram varies throughout the day, for example due to commuter traffic. This means that some parcels cannot be delivered by tram and cargo bike. These remaining parcels are then delivered by conventional means.

This thesis proposes both a modeling and an algorithm designed for the new approach for parcel delivery using a combination of tram and cargo bike. The algorithm creates tour plans for the used vehicles and minimizes the lengths of the tours to save costs.

The problem of creating the optimal tour plans can be considered a variant of the Vehicle Routing Problem (VRP) which is a generalization of the well-known Traveling Salesman Problem (TSP). The VRP was introduced in 1959 by Dantzig and Ramser [4] and has some variants which integrate capacity, time and other constraints into the problem. The VRP and its variants are NP-hard, which requires the use of heuristics for large-sized instances.

There are manifold approaches to solve different variants of the VRP. However, there is a lack of approaches which combine all the requirements of an intermodal tour algorithm. Therefore, this thesis seeks to adapt the existing approach MACS-VRPTW.

MACS-VRPTW was proposed by Gambardella et al. [8] and employs Ant Colony Optimization (ACO). In its originally proposed form, the algorithm does not support certain requirements, such as multiple depots and pickups, but the adaptability of ACO allows us to use most of the approach anyways. The important changes made on MACS-VRPTW are presented in-depth in Section 5.1.1. A complete description of the algorithm in pseudocode is given in Section 5.2.

Furthermore, the algorithm is tested against various real-world data from the OpenStreetMap project[1] to demonstrate real-world usability.

The remainder of the thesis is structured as follows: Chapter 2 gives an overview of similar existing approaches. Chapter 3 introduces the underlying optimization problems and approaches. In chapter 4, mathematical descriptions of the problem and solution models are provided. The proposed algorithm is presented in chapter 5. Chapter 6 evaluates the algorithm. A conclusion is provided in chapter 7.

[1] `https://www.openstreetmap.org`

2 Related Work

Dantzig and Ramser [4] first described the Vehicle Routing Problem (VRP) applied to gasoline delivery between one terminal and service stations and called it “The Truck Dispatching Problem”. They proposed an algorithm to solve the problem using an integer linear programming (ILP) formulation.

Since then, there has been great research effort regarding the VRP and its diverse variants. Some of the proposed approaches focus on finding exact solutions. Examples for those are ILP-based formulations and dynamic programming algorithms. Other approaches focus on approximations and heuristics in order to achieve low computation time even on big-sized instances. Examples for those are the Clarke-Wright algorithm and the tabu search metaheuristic [13].

2.1 Exact Solution Algorithms

Wassan and Nagy [21] proposed an ILP model to solve the VRP with pickups, deliveries and capacity and time constraints. However, it has problems at realistic-sized instances. Their computational results show that even for relatively small-sized instances at around 30 (pickup and/or delivery) jobs, the computation time exceeds 3 hours. It is foreseeable that the instances of intermodal tour planning in urban centers will become much larger than that.

Laporte and Norbert [14] reported applicability of dynamic programming to several variants of the VRP, among them the VRP with capacity and time constraints. The authors propose a combination of state-space relaxation and branch and bound algorithm. For the capacity-constrained problem, the largest tested instance has 125 jobs which could be solved in 15 minutes.

2.2 Approximate Solution Algorithms

Exact solution algorithms fail if the problem instance gets too large. Approximate solution algorithms trade off some of the solution quality for a faster termination time.

There are approximate solution algorithms based on heuristics that are customized for specific problems. Apart from that, metaheuristics offer general-purpose mechanisms for a variety of problems [20].

2.2.1 Heuristic Algorithms

Clarke and Wright [3] developed the savings algorithm, which is a heuristic for simple tour planning problems. The algorithm starts assigning every job (delivery) to an own tour of a vehicle. It subsequently tries to merge tours while favoring merges yielding a high reduction of tour length (saving). While it is a simple solution approach, it neither supports time constraints nor pickups nor more than one depot.

Ratnagiri et al. [17] designed a procedure to solve VRPs with focus on fast solution generation instead of solution quality. The procedure is based on clustering and greedy search. First, the (pickup and/or delivery) jobs are N-clustered, where N is the number of vehicles available. The authors propose complete-linkage clustering for this purpose. Every cluster contains jobs for one vehicle. Subsequently, the path of the vehicle is generated using a greedy nearest-neighbor heuristic. Time and capacity constraints as well as pickups are not considered, but could be integrated. The integration of micro-depots pose a problem with this procedure because the clusters are generated without consideration of the location of the depots. A further problem is that the nearest neighbor heuristic may generate poor solutions on realistic instances.

2.2.2 Metaheuristic Algorithms

Tabu search is a flexible metaheuristic that is applicable to many optimization problems. It tries to prevent the search from getting stuck at local optimums. This is done with a tabu list that contains solutions which may not be visited (again). The algorithm starts with an initial solution generated by another solution method and repeatedly searches the neighborhood of the current solution in order to find better solutions. The tabu search algorithm usually stops after a certain number of iterations [10].

Gambardella et al. [8] proposed MACS-VRPTW, a procedure to solve single depot VRPs with capacity and time constraints. It employs the Ant Colony Optimization metaheuristic and focuses on acceptable solution quality in acceptable time even on large-sized instances. It does not support pickups, but the adaptability of Ant Colony Optimization makes it possible to integrate pickups and multiple depots as well.

Zheng et al. [22] developed a delivery planning algorithm for cities based on a metro line. The idea is to deliver parcels by truck from the depot to a nearby metro entry station. At the entry station, the parcels are loaded into a compartment of a metro train. The parcels are then transported to an exit station. This transport makes up the most of the distance the parcel is transported. From the exit station, the parcels are transported to the customers. The algorithm supports time windows and capacity of vehicles. It comprises two steps. First, the parcels are assigned to a metro exit station. After that, the actual tours from exit station to customers are calculated. Zheng et al. use a combination of the Clarke-Wright algorithm and tabu search. The tours respect the schedule of the metro line, but assume that there is always enough capacity on the metro trains to transport the parcels. Furthermore, they enforce that all parcels are transported with the metro. A direct transport from depot to customer is not considered.

The rich VRP solver Jsprit [18] by GraphHopper supports a number of variations of VRP such as CVRP, VRPTW, MDVRP and VRPPD. It is written in Java. The input parameters of Jsprit are encapsulated as Java classes. Jsprit makes use of the ruin-and-recreate principle, a large neighborhood search metaheuristic. Jsprit does not support multi-leveled VRPs, where packages are first transported to a number of hubs and from there transported to the customers.

3 Foundations

Intermodal tour planning can be traced back to several well-studied optimization problems. The Vehicle Routing Problem (VRP) is concerned with the planning the shortest possible tours for vehicles that need to visit a number of service points. The VRP is used to plan tours for cargo bikes and sprinters (a type of van). The Knapsack problem is concerned with the optimal selection of items in a container of limited capacity. The Knapsack problem is used for the optimal assignment of sets of parcels to trams. These optimization problems are introduced below.

3.1 Multiple Knapsack Problem with Assignment Restrictions

The Knapsack problem is a well-known combinatorial problem. The Multiple Knapsack Problem with Assignment Restrictions (MKAR) is a variant of the Knapsack problem with multiple knapsacks instead of one and restrictions on which item can be assigned to which knapsack.

In this thesis, the following formulation adapted from Dawande et al. [5] will be used:

- Let $N = \{1, ..., n\}$ be a set of n items.
- Let $M = \{1, ..., m\}$ be a set of m knapsacks.
- Let $w_j \in \mathbb{R}_+$ be the weight of item $j \in N$.
- Let $c_i \in \mathbb{R}_+$ be the capacity of knapsack $i \in M$.
- The subset $B_i \subseteq N$ defines which items can be assigned to knapsack $i \in M$.
- Find subsets $S_i \subseteq B_i$ that maximize $\sum_{i \in M} \sum_{j \in S_i} w_j$ and respect
  (1) $\forall i \in M : \sum_{j \in S_i} w_j \leq c_i$ and
  (2) $\forall i_1, i_2 \in M, i_1 \neq i_2 : S_{i_1} \cap S_{i_2} = \{\}$

Constraint (1) requires the compliance with the knapsack capacities c_i. (2) requires that items are not contained in multiple knapsacks. The MKAR problem is NP-hard [5].

The Knapsack problem without any variations can be solved in pseudo-polynomial time using dynamic programming [16].

Dawande et al. proved in [5] that the MKAR problem is NP-hard in the strong sense. That means that there cannot exist an algorithm that solves the MKAR problem in

Algorithm 1 Pseudocode of the Successive Knapsack Algorithm

```
1: procedure SuccessiveKnapsack($N, M, B$)
2:     for all $i \in M$ do
3:         $S_i \leftarrow$ Solve the Knapsack problem for $i$ with items $N \cap B_i$
4:         $N \leftarrow N \setminus S_i$
```

pseudo-polynomial time. However, Dawande et al. proposed a polynomial-time $\frac{1}{2}$-approximation algorithm called *Successive Knapsack Algorithm*. The pseudo code is stated in Algorithm 1.

The solutions of $|M|$ Knapsack problems required by the Successive Knapsack Algorithm to solve the MKAR problem must be solved by another algorithm. Dawande et al. propose dynamic programming.

3.2 Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is a combinatorial problem. It is a generalization of the well-known Travelling Salesman Problem (TSP). For the sake of completeness, we introduce the TSP here.

3.2.1 Travelling Salesman Problem

- Let $G = (V,E)$ be a directed graph.
- Let $S \subseteq V$ be the set of vertices to visit.
- The goal is to find a path P in G with minimal length, such that all $s \in S$ are visited. The first vertex and last vertex in P must be equal.

3.2.2 Vehicle Routing Problem (Without Variations)

In this thesis, we use the following formulation of the VRP:

- Let $G = (V,E)$ be a directed graph with $N = |V|$, $M = |E|$, a set of vertices $V = \{v_1, ..., v_N\}$ and a set of directed edges $E = \{e_1, ..., e_M\}$ where an edge $e = (v_i, v_j)$, $i \neq j$.
- Let $d^* \in V$ be the depot where the tour starts and ends.
- Let $D \subseteq V$ be a set of demands, meaning vertices to be visited.
- Let $F = \{f_1, ..., f_{|F|}\}$ be a fleet of vehicles that can traverse the edges of G.
- Find paths $P = (d^*, p_1, ..., p_K, d^*)$ in G that contain each $d \in D$ at least once and minimize the total distance driven. There must not be more paths than there are vehicles in the fleet.

As the VRP is a generalization of the NP-hard TSP, VRP is also NP-hard [9]. It has multiple variants, some of which we present next:

3.2.3 Capacitated Vehicle Routing Problem

The Capacitated Vehicle Routing Problem (CVRP) is a variant of the VRP [12].

- Each demand gets a capacity usage property: $D = \{(v_1, c_1), ..., (v_{|D|}, c_{|D|})\}$ where $\forall i \in \{1, ..., |D|\} : c_i \in \mathbb{R}^+$
- Each vehicle f gets an individual capacity constraint c.
- A vehicle starts with the summed capacity usages of the demands it serves. When a demand is served, the capacity usage is subtracted from the current load of the vehicle. There must not be capacity violations at any time.

3.2.4 Vehicle Routing Problem with Time Windows

The Vehicle Routing Problem with Time Windows (VRPTW) is a variant of the VRP [12].

- Each demand gets a time window property: $D = \{(v_1, tw_1), ..., (v_{|D|}, tw_{|D|})\}$ where a time window $tw = (start, end), start, end \in \mathbb{R}^+, start < end$
- A demand can only be served within its time window.

3.2.5 Vehicle Routing Problem with Pickups and Deliveries

The Vehicle Routing Problem with Pickups and Deliveries (VRPPD) is a variant of the VRP.

- Demands may not just be deliveries (from depot to customer), but also pickups (from customer to depot).
- For the Capacitated Vehicle Routing Problem, this implies
    a) pickups are excluded from the initial vehicle's load at the depot and
    b) when a pickup is served, its capacity usage is added to the load of the vehicle instead of subtracted.

3.2.6 Multi Depot Vehicle Routing Problem

The Multi Depot Vehicle Routing Problem (MDVRP) is a variant of the VRP.

- There is not just one depot $d^* \in V$, but a set of depots $D^* \subset V$, from where tours can start and end.
- The tour of every vehicle must end at the same depot where it started.

3.3 Metaheuristics for the Vehicle Routing Problem

The VRP and the variants listed above are all NP-hard. VRPs with simple constraints and small size can be solved exactly. But as constraints become more complex and size increases, obtaining exact solutions becomes infeasible. Metaheuristics can be used to tackle this [2].

Talbi [20] categorizes metaheuristics as a branch of optimization that can be used when exact algorithms fail to yield the solution in reasonable time. This is often the case for big instances of NP-hard problems. The mechanisms of metaheuristics usually are very abstract, which makes them applicable to a large variety of problems. Metaheuristics are often inspired by nature, e.g. evolutionary algorithms.

3.3.1 Ant Colony Optimization

Ant Colony Optimization (ACO) introduced by Dorigo et al. [7] is a metaheuristic inspired by the behavior of ants. Ants use pheromone trails to communicate with each other. They leave pheromones that can be detected by other ants, making it more likely that other ants take the same way. This kind of swarm intelligence is used in the ACO metaheuristic and can be applied to path finding problems in graphs.

The underlying mechanism works as follows [7]. Let $G = (V,E)$ be a directed graph. Let $\tau_{i,j}$ be the pheromone value between two vertices i and j. These pheromones are initialized with the same value τ_0. First, ant solutions are constructed based on the current pheromone trails. A solution is constructed by traversing the graph starting at a vertex and choosing the next vertex probabilistically based on the respective pheromone trails. The formula to calculate the probabilities of next steps is displayed in Equation (3.1). $p_{i,j}$ is the probability to take edge (i,j) next, $N(s^p)$ the set of feasible next steps where s^p is the solution constructed up to the current point, η is a heuristic value that encourages ants to take better paths more frequently, α and β are algorithm parameters that weigh the pheromones and heuristic value.

(3.1)
$$p_{i,j} = \begin{cases} \dfrac{\tau_{i,j}^{\alpha} * \eta_{i,j}^{\beta}}{\sum\limits_{(i,j)\in N(s^p)} \tau_{i,j}^{\alpha} * \eta_{i,j}^{\beta}}, & \text{if } (i,j) \in N(s^p) \\ 0, & \text{otherwise} \end{cases}$$

After the solutions have been constructed, the pheromone values are updated (offline pheromone update). Implementations differ in how they perform these updates. Some use the global best solution, others use the best solution of the current iteration. One proposed formula is $\tau_{i,j} := (1-\rho) * \tau_{i,j} + \rho * \Delta\tau_{i,j}$ for all edges (i,j) that are part of the best solution, where ρ is the evaporation rate and $\Delta\tau_{i,j}$ is the increase of pheromones. The actual value of $\Delta\tau_{i,j}$ varies depending on implementation.

Then, a new solution is constructed. These steps are repeated until some termination condition is met, e.g. number of iterations, a time threshold or a threshold of solution improvement. The precise implementation must be chosen depending on the specific problem.

Application to the Capacitated Vehicle Routing Problem One classical application of the ACO metaheuristic is the TSP and the related VRP. In case of the CVRP, the solution construction phase would yield solutions respecting the capacity constraints. At a vertex, the next vertex is chosen probabilistically, but excluding already-visited vertices and vertices violating a capacity constraint. The global best solution is updated if a better solution (shortest route) is found. The application to VRPTW is very similar.

3.3.2 Ant Colony Optimization Algorithms

Ant Colony System

One important ACO algorithm is Ant Colony System (ACS) [7]. It introduces a so-called local pheromone update. It is not to be confused with the pheromone update after the solution construction. A local pheromone update is performed after each edge traversing. In such an local pheromone update, the pheromone value of edge (i,j) is updated as follows: $\tau_{i,j} = (1-\rho) * \tau_{i,j} + \rho * \tau_0$, where ρ is the decay of pheromones. According to Dorigo et al. [7], this makes it less likely that following ants take the same path as previous ants, hence diversifying the solutions produced by ants. Diverse solutions are desirable for ACO because they make the search examine other parts of the solution space. It has been proved by Stutzle and Dorigo [19] that the probability that ACS yields an optimal solution (i.e. a solution of minimal length) can be made arbitrarily close to 1 if a sufficiently large number of iterations is given.

Multiple Ant Colony System for Vehicle Routing Problems with Time Windows

Multiple Ant Colony System for Vehicle Routing Problems with Time Windows (MACS-VRPTW) is an ACS-based algorithm to solve the VRPTW [8]. In this case, VRPTW implicitely means the combination of CVRP and VRPTW. It uses two separate instances of ACS with different objectives. The first objective is to reduce the number of vehicles used. The respective ACS instance (colony) is named ACS-VEI. The second objective is to reduce the overall time vehicles spend traveling. The respective ACS instance (colony) is named ACS-TIME. The first objective is always preferred over the second objective, meaning that a longer overall tour duration is accepted if a lower number of vehicles is achieved.

MACS-VRPTW takes as inputs a (directed) graph, the depot node and a set of demands and their respective time windows. At the beginning, MACS-VRPTW calculates an initial (feasible) solution by using the greedy local neighbor heuristic. This solution is not bound to a number of vehicles. Thus, the main MACS-VRPTW procedure tries to reduce the number of vehicles in the first place.

The ACS-VEI and ACS-TIME colonies are executed in parallel, using a common variable keeping the global best. ACS-VEI tries to find any feasible solution with one vehicle less than the current global best solution has. ACS-TIME tries to find a feasible solution with less time spent. When ACS-TIME has found a feasible solution with the same amount of vehicles, the global best solution is updated, but neither colony is stopped. When either ACS-TIME or ACS-VEI has found a feasible solution with less vehicles, both colonies are stopped. The global best solution is updated and both colonies are started again. The colonies are run until a termination condition is met.

The solution construction functions nearly the same in both colonies. The procedure first sets the depot as *current location*, where a location is either the depot or a demand. Then, a sequence of locations is generated one by one. For each demand, an attractiveness matrix $\eta_{i,j}$ is computed first. i is the *current location* and j is any other location. If it is infeasible to move from location i to location j, the algorithm sets $\eta_{i,j} = 0$. This may be the case if demand j's time window or capacity usage is incompatible or if demand j has already been served. A next demand is either chosen by maximum probability $p_{i,j}$ ("exploitation") or probabilistically with probabilities $p_{i,j}$ ("exploration"). Whether exploration or exploitation is performed, is decided at random. The parameter q_0 determines the probability of exploitation being performed. Exploration is thus being performed with probability $1 - q_0$. MACS-VRPTW uses Equation (3.1), taking the attractiveness matrix $\eta_{i,j}$ as heuristic value and $\alpha = 1$. Following the ACS approach, a local pheromone update is then performed. The solution construction is repeated until there are no more feasible demands. Solution construction is allowed to conclude without necessarily having visited all demands.

To improve the computed solution, an insertion procedure and a local search are optionally performed. The insertion procedure inserts yet unvisited demands into the path based on a heuristic. Even after that, the solution may not be feasible as there could still be missing demands. Infeasible solutions are filtered by ACS-VEI and ACS-TIME. ACS-VEI additionally keeps track of how often a demand has been not included in a solution and takes this into account in the attractiveness matrix.

The authors claim that MACS-VRPTW is competitive with the best existing solvers when taking into account solution quality and computation time. They performed tests on several benchmark instances and were able to improve the best known solution of several of those instances [8].

4 Problem Statement and Analysis

In this chapter the problem *"Intermodal Tour Planning for Delivery and Pickup Activities for CEP Service Providers"* described in the introduction is analyzed. The problem will be referenced as $\mathscr{P}$. First, $\mathscr{P}$ is described thoroughly. After that, a decomposition of $\mathscr{P}$ into smaller parts is proposed.

4.1 Description

In this section, the problem model, solution model and objective of $\mathscr{P}$ are given.

4.1.1 Problem Model

Table 4.1 provides an overview of the problem model. The problem model contains 14 items. $G_s = (V_s, E_s)$ and $G_b = (V_b, E_b)$ are strongly directed loop-free graphs. G_s and G_b may be equal. The vertex sets V_s and V_b contain 2-tuples $(latitude, longitude)$ holding the latitude and longitude of the vertex. $d^* = (e_d, latitude, longitude)$ is the depot, containing latitude and longitude of the depot and a corresponding edge $e_d \in E_s$. e_d must be contained in the sprinter graph vertex set, but not necessarily in the cargo bike vertex set. H is the set of city hubs. A city hub $h = (e, latitude, longitude) \in H$ contains the same attributes as the depot. The corresponding edge must be contained in both sprinter and cargo bike graph ($e \in E_s \cap E_b$). $G_t = (V_t, E_t)$ is the tram graph (or better called, tram schedule). The graph contains only edges from the depot d^* to any of the city hubs $h \in H$ or vice-versa. An edge additionally contains a capacity c that tells the maximum-transportable amount of demands and a time window t_s, t_e. The time window tells when demands can be delivered to or picked up from the city hub h. To be precise: If the edge $e \in E_t$ is an edge from the depot d^* to any of the city hubs $h \in H$ (delivery), then the time window means the time window of arrival at the city hub. If the edge $e \in E_t$ is an edge from any of the city hubs $h \in H$ to the depot d^* (pickup), then the time window means the time window of departure at the city hub. Time windows have a start time t_s and an end time t_e. E.g. an edge $(\cdot, \cdot, c = 5, t_s = 2, t_e = 6) \in E_t$ (capacity of 5 within the time window from 2 to 6) means that from 2 hours[1] to 6 hours the maximum

[1] hours since the start time

Index	Name	Constraints	Description
1	$G_s = (V_s, E_s)$	-	sprinter graph
2	$G_b = (V_b, E_b)$	-	cargo bike graph
3	$d^* = (e_d, lat, lon)$	$e_d \in E_s$	depot
4	H	-	city hubs
5	$G_t = (V_t, E_t)$	-	tram graph (schedules)
6	P	-	pickup demands
7	D	-	delivery demands
8	r_b	$\in \mathbb{R}^+$	cargo bike delivery radius in meters
9	N_s	$\in \mathbb{N}$	number of sprinter vehicles available
10	N_b	$\in \mathbb{N}$	number of cargo bike vehicles available
11	C_s	$\in \mathbb{R}^+$	capacity of sprinter vehicles in cu
12	C_b	$\in \mathbb{R}^+$	capacity of cargo bike vehicles in cu
13	v_s	$\in \mathbb{R}^+$	speed of sprinter vehicles in meters per hour
14	v_b	$\in \mathbb{R}^+$	speed of cargo bike vehicles in meters per hour

to 1 and 2: $V_x = \{v_{x1}, ..., v_{x|V_x|}\}$, where a vertex $v_x = (latitude, longitude) \in V_x$,
$E_x = \{e_{x1}, ..., e_{x|E_x|}\}$, where an edge $e_x = (v_{xi}, v_{xj}) \in E_x$, $i \neq j$, $x \in \{s, b\}$

to 4: $H = \{h_1, ..., h_{|H|}\}$, where a city hub $h = (e, lat, lon)$, $e \in E_s \cap E_b$

to 5: $V_t = \{e_d\} \cup H$, $E_t = \{e_{t1}, ..., e_{t|E_t|}\}$, $e_{ti} = (v_{ti}, v_{tj}, c, t_s, t_e)$ where $v_{ti} = e_d \oplus v_{tj} = e_d$, c is the transportable capacity in the time window from t_s to t_e

to 6 and 7: $P = \{p_1, ..., p_{|P|}\}$, $D = \{d_1, ..., d_{|D|}\}$, where a pickup or delivery $demand = (e, c, t_s, t_e)$, $e \in E_s \cap E_b$ is the corresponding edge, $c \in \mathbb{R}^+$ the capacity usage in cu, $t_s, t_e \in \mathbb{R}_0^+$ the start and end of the time window in hours

Table 4.1: Input of problem $\mathscr{P}$

capacity to be transported is 5. The time windows are non-overlapping. P and D are pickup and delivery demands that shall be served by the vehicles. A demand (e, c, t_s, t_e) has a corresponding edge e it belongs to, a capacity usage c and a time window t_s, t_e in which it must be served. $r_b \in \mathbb{R}^+$ is the cargo bike delivery radius in meters. $N_s \in \mathbb{N}$ is the number of sprinter vehicles available. $N_b \in \mathbb{N}$ is the number of cargo bike vehicles available. $C_s \in \mathbb{R}^+$ is the capacity of sprinter vehicles in cu[1]. $C_b \in \mathbb{R}^+$ is the capacity of cargo bike vehicles in cu. $v_s \in \mathbb{R}^+$ is the speed of sprinter vehicles in meters per hour. $v_b \in \mathbb{R}^+$ is the speed of cargo bike vehicles in meters per hour.

Figure 4.1 visualizes how street (i.e. sprinter and cargo bike) and tram graphs might look like together. Assume that the sprinter graph equals the cargo bike graph. The depot is located at the edge marked with the envelope. There are two city hub edges, marked by gray cylinders. For each of them, there is a tram connection through which parcels can be transported. Sprinters start at the depot edge. Cargo bikes start at one of the city hubs. Demands can be located at any of the edges.

[1]cu (capacity unit) is an arbitrary unit for capacities

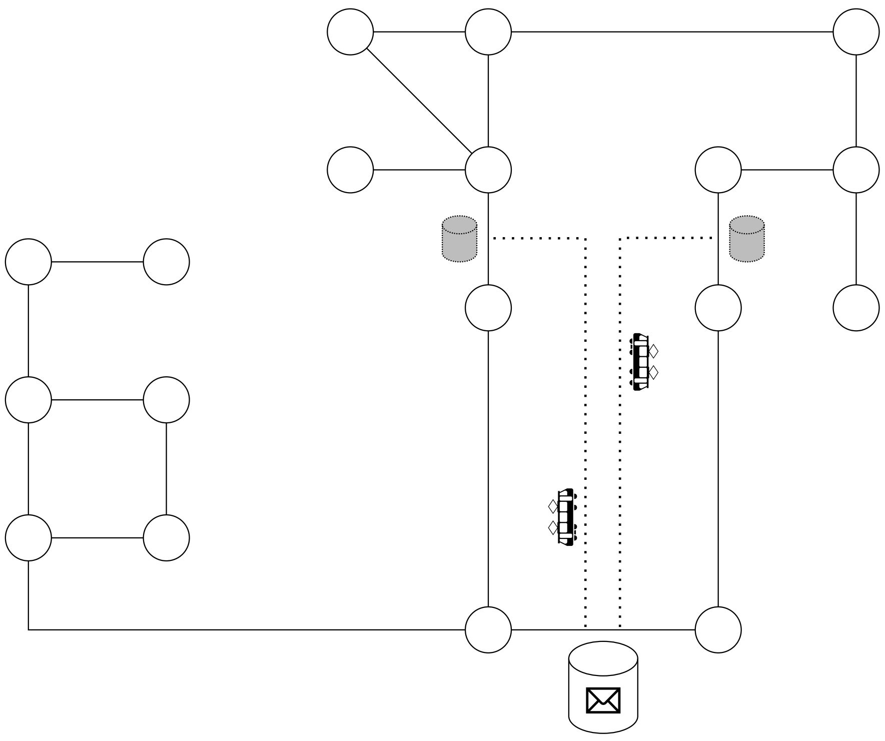


Figure 4.1: Visualization of street and tram graphs with a depot and city hubs

4.1.2 Solution Model

Index	Name	Description
1	$T = (t_1, ..., t_{\vert T\vert})$	tour plan of vehicles
2	$A = (a_1, ..., a_{\vert A\vert})$	assignment of demands to tour

to 1: $t_i = (type, route)$, where $type \in \{$*sprinter*, *cargo bike*, *tram*$\}$ is the vehicle type and $route$ is a list of edges that are sequentially used by the vehicle

to 2: $a_i = (demand, tours)$, where $demand \in P \cup D$ is the demand and $tours \subset T$ is a set of tours the demand is part of

Table 4.2: Output of problem $\mathscr{P}$

The solution of $\mathscr{P}$ contains two single parts T and A. T is the tuple of tour plans. Each tour plan contains the type of vehicle that performs the tour and a route that the vehicle takes. The route is a list of edges in the respective graph. A contains exactly $|P \cup D|$ assignments of a demand to the $tours$ the demand is part of. In case the demand is not delivered, $tours$ remains empty. In case the demand is delivered by sprinter, $tours$ contains exactly one tour, namely the sprinter tour that serves the demand. In case the demand is delivered by tram and cargo bike, $tours$ contains exactly two tours, namely the corresponding tram and cargo bike tour. Other cases do not exist.

4.1.3 Objective

There are two objectives of $\mathscr{P}$:

1. maximize $|\{a \mid a = (demand, tours) \in A, |tours| = 2\}|$
2. minimize $\sum_{t \in T} tourlength(t)$, where $tourlength(t = (type, route))$ is the time needed to take the $route$

The first and superior objective is to maximize the number of demands that are delivered by tram and cargo bike. The second objective is to minimize the time vehicles spend to serve the demands. Valid solutions (T, A) are subject to the following constraints:

- The routes of the tours T are valid paths in G_s, G_b or G_t respectively.
- The tours T serve all demands. A demand is served if the corresponding edge is traversed by the vehicle that is ought to serve the demand.
- The load of every vehicle does not exceed the vehicle's capacity at any time.
- The time window of every served demand is respected.
- The number of available vehicles is not exceeded.

4.2 Decomposition

$\mathscr{P}$ is a composition of different VRPs. It is a CVRP because of the capacity restrictions C_s and C_b. It is a VRPTW because there are time windows t_s and t_e for every demand (e, c, t_s, t_e). It also is a VRPPD as there are pickup and delivery demands.

This special combination of VRPs is not found in literature up to this point and thus requires an individual approach to solve it. For this purpose, a decomposition is proposed. $\mathscr{P}$ is broken down into two Capacitated Vehicle Routing Problems with Pickups and Deliveries and Time Windows (CVRPPDTWs) and one Multiple Knapsack Problem with Assignment Restrictions (MKAR problem). For each of these partial problems, approaches and solvers already exist. Solutions for the partial problems can be combined into a solution for the full problem using greedy algorithms and special heuristics.

In principle, there are two VRPs to solve: The first is the VRP for the cargo bikes, the second is the VRP for the sprinters. As cargo bikes emit no greenhouse gases, we aim to serve as many requests as possible using cargo bikes rather than polluting sprinters. Sprinters are second-tier vehicles that serve all demands the cargo bikes fail to serve. As each demand served by a cargo bike first needs to be transported to a city hub using a tram, we also need to optimize the usage of cargo tram capacity. However, the tram schedule may prevent that demands are transported to a city hub, for example due to commuter traffic. Any demand that could not be transported by tram will also be served by sprinters.

4.3 Transformation

In this section, the transformation of the problem $\mathscr{P}$ into the decomposed subproblems is stated. Note that in the following it is neglected that demands of $\mathscr{P}$ are assigned to edges whereas the VRP assumes demands are assigned to vertices. This issue is treated in Section 5.1.1.

4.3.1 Capacitated Vehicle Routing Problem with Pickups and Deliveries and Time Windows

The Capacitated Vehicle Routing Problem with Pickups and Deliveries and Time Windows (CVRPPDTW) is a composition of several VRP variants, which are discussed individually below.

Vehicle Routing Problem The VRP requires a directed graph, a depot, a set of demands and a fleet of vehicles. The graph is taken from either the sprinter graph G_s or cargo bike graph G_b of $\mathscr{P}$. The depot is taken from the depot of $\mathscr{P}$. The demands are taken from P and D as in $\mathscr{P}$. The fleet of vehicles is defined as $\{f_i \mid 1 \leq i \leq N\}$ where $N = N_s$ for sprinters and $N = N_n$ for cargo bikes.

Capacitated Vehicle Routing Problem The CVRP requires demands to have a capacity usage and vehicles to have a maximum capacity. The capacity usages of demands are taken from the capacity usage parameter c of pickup and delivery demands P and D of $\mathscr{P}$. The capacity of vehicles is defined as C_s or C_b as in $\mathscr{P}$ for every vehicle.

Vehicle Routing Problem Time Windows The VRPTW requires demands to have a time window. That is taken from the time window t_s, t_e of the demands P and D of $\mathscr{P}$.

Vehicle Routing Problem with Pickups and Deliveries The VRPPD allows and separates pickups and deliveries. The separation is specified by the sets P and D of $\mathscr{P}$.

The goal of the CVRPPDTW remains to minimize the total distance driven.

4.3.2 Multi-Depot Capacitated Vehicle Routing Problem with Pickups and Deliveries and Time Windows

The Multi-Depot Capacitated Vehicle Routing Problem with Pickups and Deliveries and Time Windows (MDCVRPPDTW) includes all transformations from CVRPPDTW, but replaces the depot d^* by D^* as follows.

Multi-Depot Vehicle Routing Problem The MDVRP requires not one depot, but (possibly) multiple depots. These depots are taken from the city hubs H of $\mathscr{P}$.

4.3.3 Multiple Knapsack Problem with Assignment Restrictions

Assume that a $\mathscr{P}$-solver has yielded a solution for the cargo bike delivery stage. Two Multiple Knapsack Problem with Assignment Restrictions (MKAR problem) instances are then constructed. One for deliveries from depot to city hubs (1), one for pickups from city hubs to depot (2).

The set of items N is taken from the cargo bike tours. Every cargo bike tour is transformed into one item $j \in N$. The weight w_j is the summed capacity usage of deliveries in the cargo bike tour (1) or the summed capacity usage of pickups in the cargo bike tour (2). The set of knapsacks M is taken from the edges from of the tram graph E_t. For (1), the edges from the depot d^* to any city hub $h \in H$ are taken. For (2) vice versa. The edges represent the summed maximum capacity usages that can be transported in a time slot. Every one is transformed into one knapsack $\in M$. The capacity usage c_i ($i \in M$) is the summed maximum capacity usage of the corresponding edge. Recall that the assignment restriction B_i for knapsack ($i \in M$) defines which $j \in N$ can be assigned to which knapsacks $B_i \subseteq N$. An item $j \in N$ is contained in B_i of a knapsack $i \in M$ if and only if the time windows of the corresponding cargo bike tour of j overlaps with the time window of the corresponding tram graph edge of i. In other words, if the rider of the cargo bike is able to receive the demands of the tour within the time window of the tram graph edge corresponding to i and can serve all demands on the tour without violating any time window of a demand, then and only then the item j corresponding to the tour is contained in B_i.

5 Algorithm

This chapter describes the intermodal tour algorithm $\mathscr{A}$ solving the problem $\mathscr{P}$ described in chapter 4.

5.1 Structure

The algorithm $\mathscr{A}$ is divided into five steps that are executed after another. Figure 5.1 gives an overview of the steps. Each of the steps has access to the output of the previous step. A description of the steps is given below.

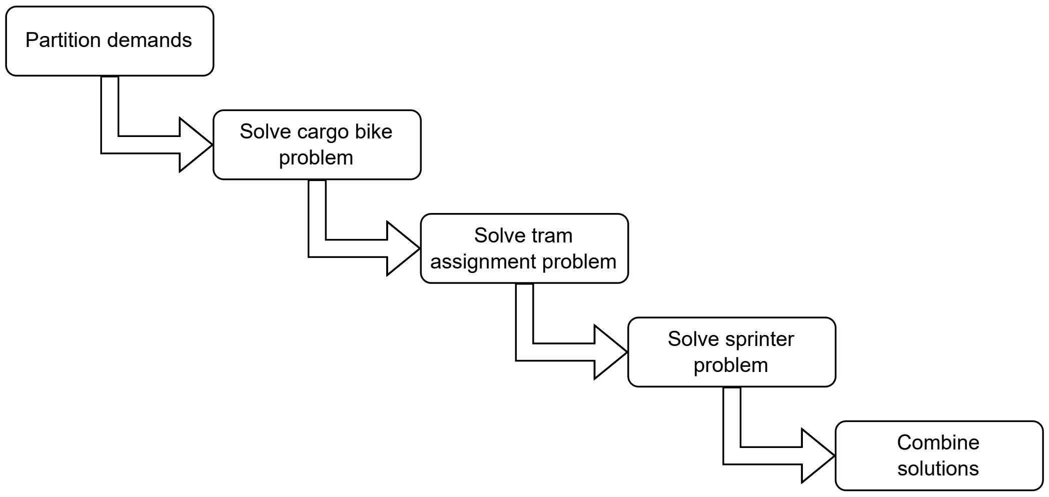


Figure 5.1: Overview of Algorithm $\mathscr{A}$

The first step partitions the demands based on their distance to the depot. The first part includes all demands within the range of a depot. The other part includes the remaining demands out of range of any depot. The range is defined by r_b.

Subsequently, the cargo bike problem is calculated. The cargo bike problem centers around serving demands within the range r_b of a city hub by using only cargo bikes. The city hubs hereby act as depots. The respective multi-depot problem[1] is solved.

[1] Multi-Depot Capacitated Vehicle Routing Problem with Pickups and Deliveries and Time Windows (cf. Section 4.3.2)

The solution of the cargo bike problem is passed to the tram assignment problem. This step calculates the deliveries from the depot to the city hubs and the pickups from the city hubs to the depot by tram. The respective tram assignment problem[1] is solved. There might be failed transports, depending on the capacities of the trains. The respective demands are then removed from the cargo bike deliveries and added to the demands pool of the sprinter problem.

These remaining demands are then passed to the next step. In this step, the sprinter problem is calculated. The sprinter problem centers around serving remaining demands. Unlike the cargo bike problem, there is only one depot d^*. The respective single-depot problem[2] is solved.

Finally, the solutions from the steps are combined and transformed to the correct output format.

The main sub problems are the sprinter problem problem, the cargo bike problem and the tram assignment problem. The algorithm frame allows the solvers to be interchanged. The single-depot ant algorithm is used as solver for the sprinter problem. The multi-depot ant algorithm is used as solver for the cargo bike problem. The greedy tram assignment algorithm, an adaption of the successive Knapsack algorithm, is used as solver for the tram assignment problem.

5.1.1 Cargo Bike and Sprinter Problem

The MACS-VRPTW[3] approach by Gambardella et al. [8] presented in Section 3.3.2 can be used to solve the CVRPTW, which is missing multi-depot and pickup capabilities that are required for the cargo bike problem and the sprinter problem. To solve them, the algorithm uses an adaption of MACS-VRPTW, which is presented here.

Edge Routing

The MACS-VRPTW approach expects the customers at vertices of the graph, whereas $\mathscr{P}$ specifies that customers are assigned to an edge. This is resolved by using edge routing instead of vertex routing. This means that paths are not a sequence of vertices (vertex routing), but a sequence of edges (edge routing). This raises the question of how the distance between two edges is determined:

Figure 5.2 illustrates an example of edge routing. Assume that we want to calculate the

[1]Multiple Knapsack Problem with Assignment Restrictions (cf. Section 4.3.3)

[2]Capacitated Vehicle Routing Problem with Pickups and Deliveries and Time Windows (cf. Section 4.3.1)

[3]Multiple Ant Colony System for Vehicle Routing Problems with Time Windows (cf. Section 3.3.2)

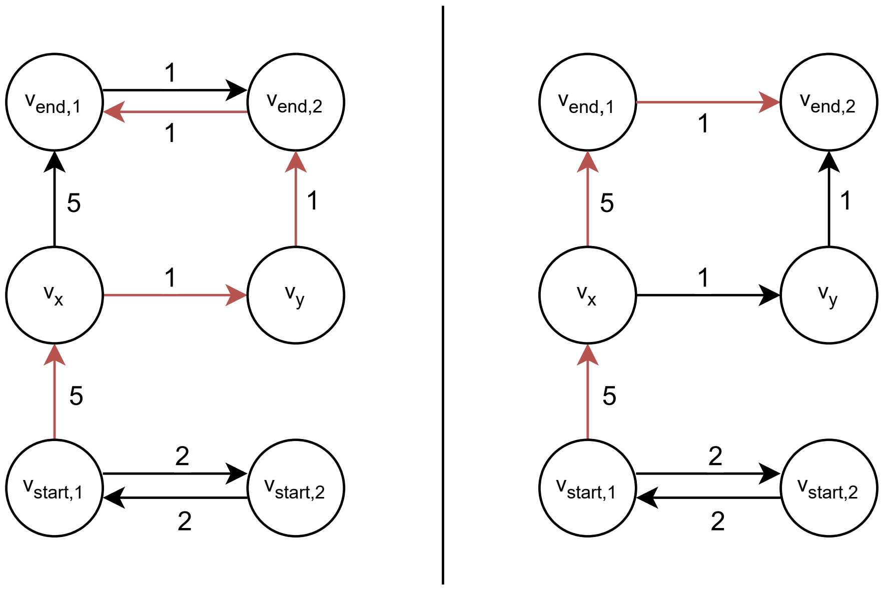


Figure 5.2: Illustration of edge routing.

distance between the two edges $e_{start} = (v_{start,1}, v_{start,2})$ and $e_{end} = (v_{end,1}, v_{end,2})$. Further assume that the reversed edge $(v_{end,2}, v_{end,1})$ exists in the graph (left case) and that $v_{start,1}$ is the current location. The distance between e_{start} and e_{end} is calculated as the minimum length from $v_{start,1}$ to either $v_{end,1}$ or $v_{end,2}$, plus the length of edge e_{end}. This ensures that the shortest route is taken and the the edge is traversed. If no reversed edge $(v_{end,2}, v_{end,1})$ exists in the graph (right case), then the distance between e_{start} and e_{end} is calculated as the minimum length from $v_{start,1}$ to $v_{end,1}$ plus the length of edge e_{end} to respect the one-way edge.

The current location is then updated accordingly. In Figure 5.2 the next edge is $v_{end,1}$ (left) or $v_{end,2}$ (right).

The exact formula to calculate the lengths between vertices (Equation (5.1)) or between a vertex and an edge (Equation (5.2)) is called $sp(\cdot,\cdot)$.

$$sp(v_1, v_2) := \text{length of the shortest path from } v_1 \text{ to } v_2 \tag{5.1}$$

$$sp(v, e = (v_1, v_2)) := \begin{cases} min\{sp(v, v_1), sp(v, v_2)\} + length(e) & \text{if } (v_2, v_1) \text{ exists} \\ sp(v, v_1) + length(e) & \text{otherwise} \end{cases} \tag{5.2}$$

Equation (5.1) requires the length of shortest paths. The shortest paths can be

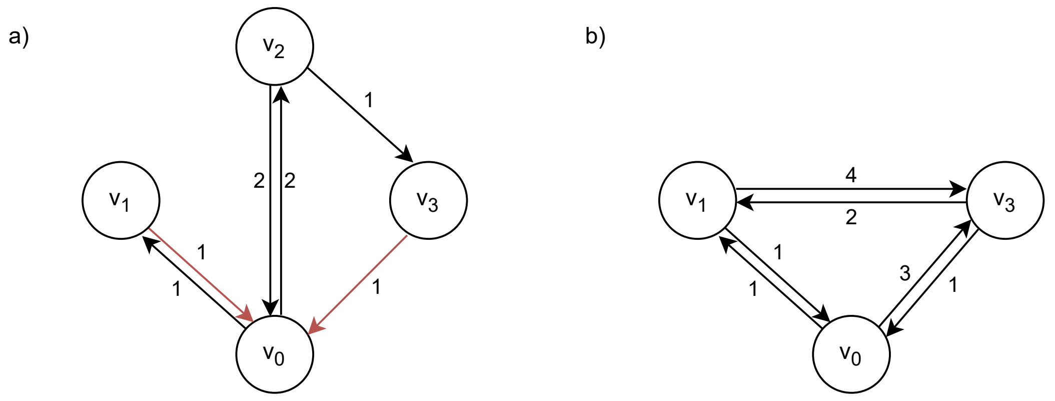


Figure 5.3: Illustration of unassigned edges.

precalculated in order to save computation time during the later calculations. However, not all lengths of paths between all pairs of vertices are relevant. Vertices that are not part of any edge that is assigned to a demand can be excluded. The result is a complete graph containing only vertices incident to demand edges. This procedure abstracts away the search for paths between demands, leaving only the problem of determining an optimal sequence in which to process the demands.

Figure 5.3 presents this procedure. Subfigure a) shows the source graph, Subfigure b) shows the result. Assume that there are two demands assigned to the edges (v_1, v_0) and (v_3, v_0) (highlighted in red). Consequently, the induced vertices are v_0, v_1 and v_3. Subfigure b) shows the complete graph containing only these vertices. Note that the rest of the graph was still used for calculating the shortest paths.

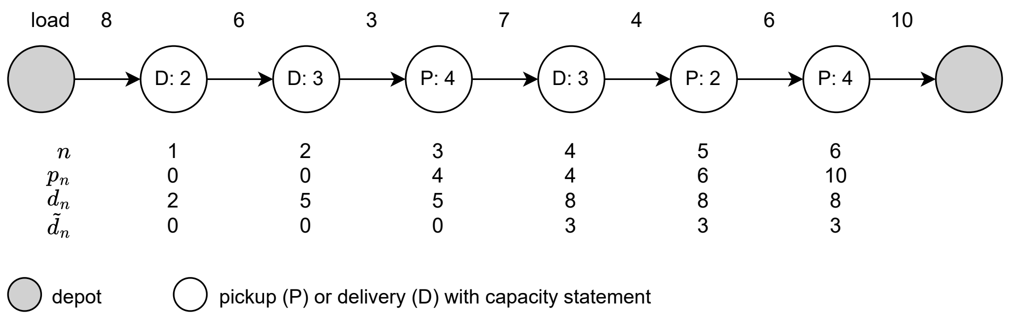
Pickups

The MACS-VRPTW approach does not support pickups at all, but they are required by $\mathscr{P}$. If a tour contains only deliveries but no pickups (or vice versa), the way to calculate the load of vehicles is easy: The capacity usages of deliveries are summed up. If the sum exceeds the capacity of the vehicle, the tour cannot include the demand. Mixed deliveries and pickups are not compatible with this approach. Therefore, the calculations of capacity limits during tour generation have to be adapted.

The tours are generated by subsequently appending demands to the tour. Pickup and delivery demands do not take place at the same time and have different consequences for the load of the transporting vehicle. Appending a pickup to the tour increases the load of the vehicle on the following path as the vehicle must transport the pickup from the time of loading the pickup until reaching the depot. Thus, appending a pickup does not affect previous loads. Appending a delivery to the tour increases the load of the vehicle on the previous path as the vehicle must transport the delivery from the beginning of the tour until unloading it. Thus, appending a delivery does not affect following loads.

There are three variables p_n, d_n and $\tilde{d_n}$ used to ensure compliance with capacity constraints during tour generation ($1 \leq n \leq$ # demands of tour). p_n is the sum of all capacity usages of all pickups included in the first n demands of a tour. d_n is the sum of all capacity usages of all deliveries included in the first n demands of a tour. $\tilde{d_n}$ is the sum of all capacity usages of all deliveries included between the first pickup demand and the n'th demand (inclusive) of a tour. If only deliveries are included in the tour, then $\tilde{d_n} = 0$ for all n. If the first demand in the tour is a pickup, then $\tilde{d_n} = d_n$ for all n. A tour can include a further pickup of capacity usage c at position $n+1$ if $p_n + c$ does not exceed the capacity of the vehicle (1). A tour can include a further delivery of capacity usage c at position $n+1$ if $d_n + c$ does not exceed the capacity of the vehicle (2) and $\tilde{d_n} + p_n + c$ does not exceed the capacity of the vehicle (3).

The constraints (1) and (2) ensure that the capacity constraints of individual pickups or deliveries are met. Constraint (3) ensures that a new delivery does not break the constraints of previous demands. If a new delivery of capacity usage c is added to a tour, it must be transported from the depot up to the current point in the tour, adding the capacity usage c to the current load of every previous point in the tour. The deliveries that are served before the first pickup are not taken into account because they are only decreasing the load of the vehicle, thus not affecting the ability to serve deliveries at a later point.



n	1	2	3	4	5	6
p_n	0	0	4	4	6	10
d_n	2	5	5	8	8	8
$\tilde{d}_n$	0	0	0	3	3	3

Figure 5.4: Illustration of mixed pickups and deliveries

Figure 5.4 shows an example of 6 mixed pickups and deliveries. Assume the capacity of the transporting vehicle to be 10. Additionally to the variables p_n, d_n and $\tilde{d_n}$, the load of the vehicle is displayed to show that the capacity is never exceeded.

Multiple Depots

The MACS-VRPTW approach does not support multiple depots, but assumes that there is only one depot where all vehicles start from. This is only needed for the cargo bike problem and not the sprinter problem as the tour origin for sprinters is always set as the central depot instead of city hubs.

To augment the MACS-VRPTW approach to handle multiple depots, the choice of the depot is encoded in the problem formulation using pheromone values modeling the selection of a specific depot: Every $depot$ gets a pheromone value δ_{depot} that determines the probability of the depot to be chosen. The pheromone value is handled the same as regular pheromones of edges. A depot gets local pheromone updates when it is chosen as initial depot and offline pheromone updates if it is included in a solution that is subject to an offline pheromone update.

Encoding the depot selection within the ant colony problem also allows the selection of a depot which already depleted all the demands in its range. As a remedy, a depot blacklist can be used. A depot is contained in the blacklist if and only if there are no demands in the range this depot that are not-yet served. If a depot is contained in the blacklist, its probability to be chosen as initial depot is 0. If all depots are blacklisted, the solution generation is aborted. The blacklist is only kept within one solution generation procedure. Every solution generation starts with an empty blacklist.

5.1.2 Tram Assignment Problem

The tram assignment problem is a MKAR problem[1]. MKAR problems can be solved using the successive Knapsack algorithm [5]. The successive Knapsack algorithm requires the solutions of regular Knapsack problems. The number of required solutions depends on the amount of time slots the tram schedule contains. For this, the dynamic programming algorithm for solving Knapsack problems can be used.

However, the successive Knapsack algorithm is not the best approach for this type of problem, because it does not take the length of time windows into account when it chooses a time window for a demand. Figure 5.5 illustrates the problem. There are two demands and two time slots of capacity 5. The first demand has a capacity usage of 5 and fits in both time slots. The second demand has a capacity usage of 4 an fits only in the first time slot. The successive Knapsack algorithm will then choose the first demand for the first time slot because the first demand has a higher capacity usage than the second demand. Subsequently, the second demand can not be chosen for the second time slot, because it could only be fit into the first time slot. But the successive

[1]Multiple Knapsack Problem with Assignment Restrictions (cf. Section 4.3.3)

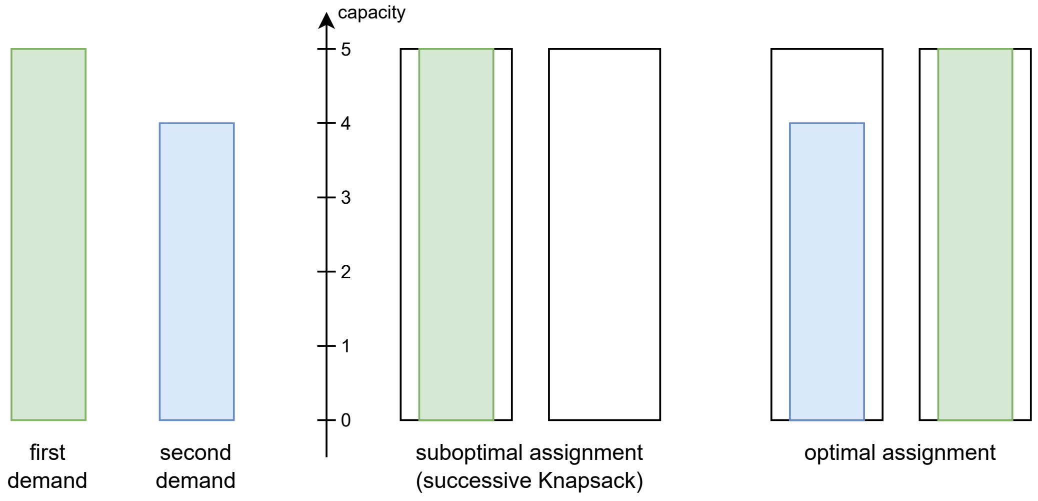


Figure 5.5: Illustration of unfavorable greedy choice by successive Knapsack

Knapsack algorithm does not respect this and makes overly greedy choices.

The highlighted issue can be remedied by the greedy tram assignment algorithm, which is used for the tram assignment problem. The procedure is described for deliveries. The procedure for pickups is done respectively.

First, the demands are sorted by the latest time in their time window (ascending) and their capacity usage (descending). The capacity usage is only taken into account if the time of two demands is equal. Similar to the successive Knapsack algorithm, the greedy tram assignment algorithm iterates over the time slots. It checks whether the demand with the lowest index that has not-yet been assigned to a time slot can be assigned to the current time slot and assigns it if possible.

The greedy tram assignment algorithm is not optimal at assigning optimally-many demands to a time slot, but takes the length of the time windows into account. A comparison of the performance of the successive Knapsack algorithm and the greedy tram assignment algorithm is given in Section 6.3.

5.2 Pseudo Code

In this section, the pseudo code of the algorithm $\mathscr{A}$ is given. The pseudo code of the algorithm frame (Section 5.2.1), the greedy tram assignment algorithm (Section 5.2.2) and the ant algorithm (Section 5.2.3) is given.

Algorithm 2 Intermodal Tour Algorithm (Frame)

```
 1: procedure ALGORITHM($\mathscr{P}$-Instance)
 2:     $sp(\cdot,\cdot) \leftarrow$ SHORTESTPATHS($G_s, G_b,$)
 3:     $(cargoBikeDemands, sprinterDemands) \leftarrow$ PARTITIONDEMANDS( )
 4:     $cargoBikeTours \leftarrow$ MDMACS($cargoBikeDemands$, $sp$, $H$, $N_b$, $C_b$, $v_b$)
 5:     $(deliveredByTram, tramDeliveryFailed) \leftarrow$ TRAMDELIVERY($cargoBikeTours$)
 6:     $deliveredBySprinter \leftarrow$ MACS($sprinterDemands \cup tramDeliveryFailed$, $sp$, $d^*$, $N_s$,
        $C_s$, $v_s$)

 7: function SHORTESTPATHS($G_s, G_b$)
 8:     $distances \leftarrow \{\}$
 9:     for all $vertex \in V_s \cup V_b$ do
10:         $distances \leftarrow distances \cup dijkstra(vertex, G_s \cup G_b)$
    return $\lambda from.\ \lambda to.\ distances[from][to]$

11: function PARTITIONDEMANDS( )
12:     $inRange \leftarrow \{\}$
13:     $outOfRange \leftarrow \{\}$
14:     for all $d \in D$ do
15:         $minimalDistance \leftarrow \{sp(h,d) \mid h \in H\}$
16:         if $minimalDistance < r_b$ then
17:             $inRange \leftarrow inRange \cup \{d\}$
18:         else
19:             $outOfRange \leftarrow outOfRange \cup \{d\}$
20:     return $(inRange, outOfRange)$
```

5.2.1 Algorithm Frame

The frame of the intermodal tour algorithm gets an instance of problem $\mathscr{P}$ as parameter and heuristically computes a solution of the given instance. It comprises five steps. The pseudo code is given in Algorithm 2.

In the first step, the shortest path from every vertex to every other vertex is calculated using Dijkstra's algorithm [6]. This is done for both the sprinter graph G_s and the cargo bike graph G_b The shortest path between vertices is $sp(\cdot,\cdot)$ as defined in Section 5.1.1. The second step PARTITIONDEMANDS partitions the demands into $cargoBikeDemands$ for demands within delivery range by bike and $sprinterDemands$. Next, the multi-depot ant algorithm (MDMACS) is used to compute a feasible solution of cargo bike deliveries for the $cargoBikeDemands$. Then, the TRAMDELIVERY function calculates which of the $cargoBikeTours$ can be supplied by tram. Lastly, the single-depot ant algorithm (MACS) is used to compute a feasible solution of sprinter deliveries for the $unassignedPackages = sprinterDemands \cup tramDeliveryFailed$.

5.2.2 Greedy Tram Assignment Algorithm

The pseudo code of the greedy tram assignment algorithm is given in Algorithm 3. The TRAMDELIVERY function calculates which of the *cargoBikeTours* can be supplied by tram. It first calculates the summed capacity usage of pickups and the summed capacity usage of deliveries. If the summed capacity usage of the pickups is greater than 0, it creates a joint (non-atomic) pickup demand with the summed capacity usage of the pickups. The time windows of the tour are taken from the tour. The targeted edge is the edge assigned to the city hub. A joint delivery is created the same way if applicable. Then, the TRAMDELIVERY function calls the FITDEMANDS function for each city hub and individually for pickups and deliveries. In each case, the FITDEMANDS function returns a set of successfully assigned joint demands and a set of failed joint demands. The TRAMDELIVERY function then transforms the joint demands back to atomic demands and adds atomic demands that are part of a successfully assigned joint demand to *delivered*. It respectively adds atomic demands that are part of a failed joint demand to *remaining*.

The FITDEMANDS function calculates which joint demands are delivered by tram. The function first sorts the demands by their time window (ascending) and their capacity usage (descending). The time window has precedence over capacity usage. In case of pickups, the earliest time the tour can possibly end is taken. In case of deliveries, the latest time the tour can possibly start is taken. In this order, the function iterates over the demands for each time slot, reducing the capacity usage of d from the time slot, removing the demand from the *remaining* list and adding the demand to the *delivered* set if an applicable time slot has been found. The delivered and undelivered demands are then returned. Hereafter, for every accumulated demand, it is checked whether the trams can serve the demand. The demands are handled in descending order by number of accumulated demands. If the request at G_t was successful, the capacity usage of the accumulated demand is subtracted from G_t. The served join demands and the not served joint demands are returned separately.

5.2.3 Single-Depot and Multi-Depot Ant Algorithms

The only difference between the single-depot ant algorithm (MACS) and the multi-depot ant algorithm (MDMACS) is the number of depots, which is why they are described together. The description only refers to the multi-depot ant algorithm. The differing parts are highlighted at the end of Section 5.2.3.

Multi-Depot Ant Algorithm

The multi-depot ant algorithm solves the multi-depot problem. The multi-depot ant algorithm is an adaption of MACS-VRPTW by Gambardella et al. [8] presented in Section 3.3.2. The pseudo code of the multi-depot ant algorithm is given in Algorithms 5 - 8. The parameters of the MDMACS function, which contains the multi-depot ant algorithm, is the following:

- A list of $demands \subseteq P \cup D$.
- The shortest paths function $sp(\cdot,\cdot)$.
- A list of $depots$. A single depot is a tuple (e, lat, lon), just as d^* in $\mathscr{P}$.
- The maximum number of usable vehicles $maxVehicles$, an integer.
- The $vehicleCapacity$, a floating point number.
- The $vehicleSpeed$, a floating point number.

The multi-depot algorithm uses a greedy solution generator to generate an initial solution. The pseudo code of the greedy solution generator is given in Algorithm 4. The greedy solution generator complies with all constraints to the multi-depot problem except the number of vehicles available. It builds tours and always begins a new tour with a newly assigned vehicle as soon as a constraint is broken. After the generation of a greedy solution, the algorithm tries to reduce the number of vehicles if it exceeds the number of vehicles available.

The GREEDY function calls the ASSIGNDEPOT function to assign the nearest depot to each demand. Then, the SINGLEDEPOTGREEDY function is used to generate paths for a single depot serving all assigned demands. First, the demand nearest to the depot is added to a new path in the SINGLEDEPOTGREEDY function. Subsequently, the closest feasible and not-yet served demand is added. If there is no feasible and not-yet served demand, the path is ended and a new path is begun. In the end, all paths are returned. With each path, the following attributes are returned: The $demands$ served in this path, the $depot$, $timeWindowStart$, the first possible time to begin the tour, and $timeWindowEnd$, the last possible time to begin the tour. The tours for all depots are collectively returned by the GREEDY function.

After the multi-depot algorithm has computed an initial solution using the greedy algorithm, it tries to optimize the initial solution using Ant-Colony-Optimization.

MDMACS function The MDMACS function starts the multi-depot algorithm algorithm. It first assigns the greedy solution to the $globalBestSolution$ and then asynchronously calls the ACS-VEI and ACS-TIME procedures which do the optimization work. The former tries to find a solution using less vehicles than the current $globalBestSolution$. The latter tries to find a solution taking less time than the current

$globalBestSolution$. After they have started, the only job of the MDMACS function is to accept solutions by the two procedures and update the $globalBestSolution$. If the new $globalBestSolution$ additionally uses less vehicles than the current $globalBestSolution$, then the executions of both ACS-VEI and ACS-TIME are stopped and restarted. Note that the ACS-VEI and ACS-TIME procedures always have access to the current $globalBestSolution$. The optimization process is stopped as soon as a predefined stopping criterion holds true.

ACS-VEI procedure The ACS-VEI procedure is trying to find solutions with less vehicles. First, it calls the GREEDY solution generator to find a solution using at most $numVehicles$ vehicles. This solution may not serve all demands. It subsequently tries to find other ant solutions and rewards solutions serving more demands. For all demands $d \in demands$, $in(d)$ counts for how many subsequent times d has not been included in a solution generated by NEWACTIVEANT. in is handed to NEWACTIVEANT as parameter where it changes the probability for demands to be chosen next. As soon as a solution serving all customers is found, it is sent to MDMACS.

ACS-TIME procedure The ACS-TIME procedure is trying to find shorter solutions. Like ACS-VEI, it generates ant solutions, but it rewards shorter solutions. As soon as a complete solution shorter than the current shortest solution is found, it is sent to MDMACS. A solution is regarded complete if it is not missing out any demand.

NEWACTIVEANT function The NEWACTIVEANT function generates new solutions based on a mixed-parameter heuristic and pheromones. The $paths$ variable stores all paths generated. The $path$ variable stores only the single currently generated path. First, a depot is chosen by the CHOOSEDEPOT function. It is added as the beginning of the first $path$. After that, the subsequent path items are added until the maximum number of vehicles is reached. In order to perform this, the attractiveness of each demand and depot is calculated. The attractiveness of a demand or depot is a heuristic value that prevents infeasible legs[1] and makes presumably bad legs less likely, thus decreasing the time to find good-quality solutions. The exact calculation is shown in Equation (5.7). An infeasible leg could arise from a demand that has already been visited or would lead to excess of capacity. An example for a presumably bad leg would be a demand that is very far away from the current position. As soon as all demands have been served, the solution generation is stopped. Otherwise, the next demand or depot is chosen based on pheromones τ and attractiveness η. The exact calculation is shown in Equation (5.8). The next leg is saved to the path and the local pheromone

[1] A leg $(from,to)$ means a „step“ from a demand or depot to another demand or depot.

update (see Section 3.3.2) is performed. Additionally, if the next leg leads to one of the depots, the current $path$ is ended and added to the $paths$. A new starting depot is chosen in the same way as at the beginning of the function. At the end, the $paths$ and the $demands$ that have not been served by any of the $paths$ are returned. With each path, the following attributes are returned: The $demands$ served in this path, the $depot$, $timeWindowStart$, the first possible time to begin the tour, and $timeWindowEnd$, the last possible time to begin the tour.

OFFLINEPHEROMONEUPDATE procedure The OFFLINEPHEROMONEUPDATE procedure performs the offline pheromone update. It is performed for the best solution(s) after an iteration of an ACS. For every $path$ that is part of the solution, the depot pheromones are updated. After that, for all legs $(from,to)$ in the $path$, the pheromones of that legs are also updated.

LOCALPHEROMONEUPDATE procedure The LOCALPHEROMONEUPDATE procedure performs the local pheromone update. It is performed when a leg $(from,to)$ is added during solution construction. The pheromone value of $(from,to)$ is updated.

CHOOSEDEPOT function The CHOOSEDEPOT function probabilistically chooses a depot to start a tour at and performs the respective local depot pheromone update immediately.

Offline pheromone update For every leg $(from,to)$ in a solution with length L, the pheromones $\tau_{from,to}$ are updated as follows:

$$\tau_{from,to} \leftarrow (1-\rho) * \tau_{from,to} + \rho * \frac{1}{L} \tag{5.3}$$

where ρ is the pheromone decay.

Local pheromone update For a leg $(from,to)$ with initial pheromones value τ_0, the pheromones $\tau_{from,to}$ are updated as follows:

$$\tau_{from,to} \leftarrow (1-\rho) * \tau_{from,to} + \rho * \tau_0 \tag{5.4}$$

where ρ is the pheromone decay.

Offline depot pheromone update For the $depot$ of every tour in a solution with length L, the depot pheromones τ_{depot} are updated as follows:

$$\delta_{depot} \leftarrow (1-\rho) * \delta_{depot} + \rho * \frac{1}{L} \tag{5.5}$$

where ρ is the pheromone decay.

Local depot pheromone update For a $depot$ with initial depot pheromones value δ_0, the pheromones δ_{depot} are updated as follows:

$$\delta_{depot} \leftarrow (1-\rho) * \delta_{depot} + \rho * \delta_0 \tag{5.6}$$

where ρ is the pheromone decay.

Leg attractiveness The attractiveness of a leg $(from,to)$ is defined as follows: $\eta_{from,to} = 0$ if the leg $(from,to)$ is infeasible. Reasons for this are:

- to has already been visited
- $from$ and to are both depots
- to is a depot $\neq$ the depot where the tour started
- to would be arrived after the time window
- to would exceed the $vehicleCapacity$

For all other legs $(from,to)$, where $to = (e,c,t_s,t_e)$ and t is the current time, η is defined as follows:

$$\eta_{from,to} = \frac{1}{max(1, distance_{from,to} - in(to))} \tag{5.7}$$

where $distance_{from,to} = (deliveryTime_{to} - t) * (e - t)$
and $deliveryTime_{to} = max(t + sp(from,to), s)$

Leg choice The next demand or depot given that $from$ is the current demand or depot is chosen as follows:
A random number $q \in [0,1)$ is generated. It decides whether exploitation or exploration is performed. Exploitation (case $q < q_0$) is performed with probability q_0. Exploration (other case) is performed with probability $1 - q_0$:

$$next = \begin{cases} to \in feasible_{from} \text{ with maximum } p_{from,to} & \text{if } q < q_0 \\ to \text{ chosen probabilistically with probabilities } p_{from,to} & \text{otherwise} \end{cases} \tag{5.8}$$

Where $\forall to \in feasible_{from}: \quad p_{from,to} = \frac{\tau_{from,to} * \eta^{\beta}_{from,to}}{\sum\limits_{d \in feasible_{from}} \tau_{from,d} * \eta^{\beta}_{from,d}}$

and $feasible_{from} = \{to \in demands$, where $\eta_{from,to} > 0\}$

Depot choice The depot of a tour is chosen at random with the following probabilities for each depot:

$$\text{(5.9)} \qquad \forall depot \in depots: \quad p_{depot} = \begin{cases} \frac{\delta_{depot}}{\sum\limits_{d \in depots \setminus blacklist} \delta_d} & \text{if } depot \notin blacklist \\ 0 & \text{if } depot \in blacklist \end{cases}$$

Single-Depot Ant Algorithm

The single-depot ant algorithm (MACS) is a slimmed version of the multi-depot ant algorithm. If only one depot exists, there is potential so save some computation time. In the single-depot version, the depot assignment (cf. ASSIGNDEPOT) and the depot selection (cf. CHOOSEDEPOT), including the depot blacklist and depot pheromones can be omitted. This is why two versions of the algorithm were considered instead of using the multi-depot ant algorithm with just one depot.

In the MDMACS function of the multi-depot ant algorithm, the initial *globalBestSolution* is calculated using the SINGLEDEPOTGREEDY function instead of the GREEDY function. Hereby, the unnecessary assignment is omitted. The ACS-VEI and ACS-TIME procedures are not changed. In the NEWACTIVEANT function, the initial and following depots are not chosen. In the OFFLINEPHEROMONEUPDATE procedure, the offline depot pheromone update is omitted. The CHOOSEDEPOT function is omitted.

Algorithm 3 Greedy Tram Assignment Algorithm

```
1: function TRAMDELIVERY($cargoBikeTours$)
2:     $jointPickups \leftarrow []$
3:     $jointDeliveries \leftarrow []$
4:     for all $(demands, cityHub, timeWindowStart, timeWindowEnd, path) \in$
           $cargoBikeTours$ do
5:         $pickupCapacity \leftarrow sum(\{c \mid p = (e,c,t_s,t_e) \in (demands \cap P)\}$
6:         $deliveryCapacity \leftarrow sum(\{c \mid d = (e,c,t_s,t_e) \in (demands \cap D)\}$
7:         if $pickupCapacity > 0$ then
8:             $p \leftarrow (cityHub, pickupCapacity, timeWindowStart, timeWindowEnd)$
9:             $jointPickups \leftarrow jointPickups + [p]$
10:        if $deliveryCapacity > 0$ then
11:            $d \leftarrow (cityHub, deliveryCapacity, timeWindowStart, timeWindowEnd)$
12:            $jointDeliveries \leftarrow jointDeliveries + [d]$
13:    $delivered \leftarrow \{\}$
14:    $remaining \leftarrow \{\}$
15:    for all $cityHub \in H$ do
16:        FITDEMANDS($cityHub, \{d \mid d \in jointPickups, d$ is assigned to $cityHub\}$)
17:        FITDEMANDS($cityHub, \{d \mid d \in jointDeliveries, d$ is assigned to $cityHub\}$)
18:        Transform the joint demands returned by FITDEMANDS back to atomic de-
           mands and add them to $delivered$ and $remaining$
19:    return $(delivered, remaining)$

20: function FITDEMANDS($cityHub$, $jointDemands$)
21:    $delivered \leftarrow \{\}$
22:    if the $jointDemands$ are pickups then
23:        $remaining \leftarrow$ sort $jointDemands$ by the earliest time of time window and
           capacity usage
24:    else
25:        $remaining \leftarrow$ sort $jointDemands$ by the latest time of time window and capacity
           usage
26:    for all time slots $(start, end) \in G_t$ do
27:        for all $d \in remaining$ do
28:            if $d$ can be assigned to time slot $(start, end)$ then
29:                subtract the capacity usage of $d$ from time slot $(start, end)$
30:                $remaining \leftarrow remaining \setminus \{d\}$
31:                $delivered \leftarrow delivered \cup \{d\}$
32:    return $(delivered, remaining)$
```

Algorithm 4 Greedy Solution Generator

```
1:  function GREEDY($demands$, $sp$, $depots$, $maxVehicles$, $vehicleCapacity$, $vehicleSpeed$)
2:      $part \leftarrow$ ASSIGNDEPOT($depots, demands$)
3:      $tours \leftarrow \{\}$
4:      for all $depot \in depots$ do
5:          $depotDemands \leftarrow \{d \in demands$ where $part(d) = depot\ \}$
6:          $tours \leftarrow tours \cup$ SINGLEDEPOTGREEDY($depot, depotDemands$,
                $vehicleCapacity$)
7:      return $tours$

8:  function ASSIGNDEPOT($depots, demands$)
9:      for all $demand \in demands$ do
10:         Find $depot \in depots$ with minimal $sp(depot, demand)$
11:         $part(demand) = depot$
12:     return $part$

13: function SINGLEDEPOTGREEDY($demands$, $sp$, $depot$, $maxVehicles$, $vehicleCapacity$,
        $vehicleSpeed$)
14:     $paths \leftarrow \{\}$
15:     while there is a not-yet served $demand \in demands$ left do
16:         $path \leftarrow$ empty path
17:         Add $depot$ to $path$
18:         $demand \leftarrow depot$
19:         repeat
20:             if there is no feasible and not-yet served $d \in demands$ then
21:                 break
22:             $next \leftarrow$ feasible and not-yet served $d \in demands$ with minimal
                $sp(demand, d)$
23:             Add $next$ to $path$, mark as served
24:             $demand \leftarrow next$
25:         until all $demands$ are served
26:         Add $depot$ to $path$
27:         $paths \leftarrow path \cup \{path\}$
28:     return $\{(demands, depot, timeWindowStart, timeWindowEnd, path) \mid path \in paths\}$
```

Algorithm 5 Multi-Depot Ant Algorithm (Part 1)

```
 1: function MDMACS($demands, sp, depots, maxVehicles, vehicleCapacity, vehicleSpeed$)
 2:     $globalBestSolution \leftarrow$ GREEDY($demands$, $sp$, $depots$, $maxVehicles$, $vehicleCapacity$,
            $vehicleSpeed$)
 3:     repeat
 4:         $numVehicles \leftarrow$ Number of vehicles used by $globalBestSolution$
 5:         if $numVehicles > maxVehicles$ then
 6:             Call ACS-VEI($numVehicles$ - 1) asynchronously
 7:         Call ACS-TIME($numVehicles$) asynchronously
 8:         while ACS-VEI and ACS-TIME are not stopped do
 9:             Wait until a better feasible solution $s$ has been found
10:             $globalBestSolution \leftarrow s$
11:             if Number of vehicles used by $s$ < $numVehicles$ then
12:                 Cancel the execution of ACS-VEI and ACS-TIME
13:                 continue
14:     until The stopping criterion has met
15:     return $globalBestSolution$
```

Algorithm 6 Multi-Depot Ant Algorithm (Part 2)

```
16: procedure ACS-VEI($numVehicles$)
17:     repeat
18:         $maxDemandsSolution \leftarrow$ GREEDY($depots, demands, vehicleCapacity$)
19:         $\forall d \in demands : in(d) \leftarrow 0$
20:         for $k = 1, ..., antCount$ do
21:             $(solution^k, missing^k) \leftarrow$ NEWACTIVEANT($numVehicles, in$)
22:             $\forall d \in missing^k : in(d) \leftarrow in(d) + 1$
23:         $iterationMaxDemandsSolution \leftarrow solution^k$ with maximum number of served
            demands
24:         if $iterationMaxDemandsSolution$ serves more demands than
            $maxDemandsSolution$ then
25:             $maxDemandsSolution \leftarrow iterationMaxDemandsSolution$
26:             $\forall d \in demands : in(d) \leftarrow 0$
27:             if $iterationMaxDemandsSolution$ serves all demands then
28:                 Send $iterationMaxDemandsSolution$ to MDMACS
29:         OFFLINEPHEROMONEUPDATE($maxDemandSolution$)
30:         OFFLINEPHEROMONEUPDATE($globalBestSolution$)
31:     until it is cancelled by MDMACS

32: procedure ACS-TIME($numVehicles$)
33:     repeat
34:         for $k = 1, ..., antCount$ do
35:             $(solution^k, missing^k) \leftarrow$ NEWACTIVEANT($numVehicles, 0$)
36:         $iterationShortestPathSolution \leftarrow solution^k$ with minimal path length and
            $missing^k$ is empty
37:         if $iterationShortestPathSolution$ is shorter than $globalBestSolution$ then
38:             Send $iterationShortestPathSolution$ to MDMACS
39:         OFFLINEPHEROMONEUPDATE($globalBestSolution$)
40:     until it is cancelled by MDMACS
```

Algorithm 7 Multi-Depot Ant Algorithm (Part 3)

```
41: function NEWACTIVEANT($numVehicles, in$)
42:     $paths \leftarrow \{\}$
43:     $path \leftarrow$ empty path
44:     $depot \leftarrow$ CHOOSEDEPOT($\{d \mid d \in depots,$ all demands in range of $d$ are served $\}$)
45:     $current \leftarrow depot$
46:     Add $current$ to $path$
47:     repeat
48:         calculate the attractiveness values $\eta_{current,\cdot}$ according to Equation (5.7)
49:         if No demand or depot is feasible then
50:             Add $depot$ to $path$
51:             $paths \leftarrow paths \cup path$
52:             break
53:         $next \leftarrow$ choose next demand or depot according to Equation (5.8)
54:         Add $next$ to $path$
55:         LOCALPHEROMONEUPDATE($current, next$)
56:         if $next$ is a depot then
57:             $paths \leftarrow paths \cup path$
58:             $path \leftarrow$ empty path
59:             $depot \leftarrow$ CHOOSEDEPOT($\{d \mid d \in depots,$ all demands in range of $d$ are served $\}$)
60:             $current \leftarrow depot$
61:             Add $current$ to $path$
62:         else
63:             $current \leftarrow next$
64:     until $|paths| = numVehicles \lor \forall d \in depots:$ all demands in range of $d$ are served
65:     $solution \leftarrow \{(demands, depot, timeWindowStart, timeWindowEnd, path) | path \in$
        $paths\}$
66:     $missing \leftarrow$ all $demands$ that are not contained in any $solution$
67:     return $(solution, missing)$
```

Algorithm 8 Multi-Depot Ant Algorithm (Part 4)

```
68: procedure OFFLINEPHEROMONEUPDATE($solution$)
69:     for all $(demands, depot, timeWindowStart, timeWindowEnd, path) \in solution$ do
70:         Update $\delta$ for $depot$ according to Equation (5.5)
71:         for all legs $(from, to)$ in the $path$ do
72:             Update $\tau$ for $(from, to)$ according to Equation (5.3)

73: procedure LOCALPHEROMONEUPDATE($from, to$)
74:     Update $\tau$ for $(from, to)$ according to Equation (5.4)

75: function CHOOSEDEPOT($blacklist$)
76:     $depot \leftarrow$ choose depot with probabilities according to Equation (5.9)
77:     Update $\delta$ for $depot$ according to Equation (5.6)
78:     return $depot$
```

6 Evaluation

In this section, the intermodal tour algorithm is evaluated. Section 6.1 introduces the instances used for the evaluation and explains how they have been generated. The multi-depot ant algorithm (Section 6.2) and the greedy tram assignment algorithm (Section 6.3) are evaluated separately. Section 6.4 evaluates the full intermodal tour algorithm.

The intermodal tour algorithm is implemented in Java SE 18 using Apache Commons Lang 3.12.0 and FasterXML Jackson Databind 2.16.1. The tests are executed on a machine with an Intel Xeon E-2288G (8x 3.70 GHz) CPU and 128 GB of memory.

6.1 Test Instances

The evaluation of the developed algorithm is carried out on a set of instances that is introduced here. The city of Karlsruhe in Baden-Württemberg, Germany, is the first instance. To verify that the algorithm can be applied to other cities with different layouts as well, two more cities have been chosen. The city of Santa Ana in California, USA has surprisingly similar properties compared to Karlsruhe, which is why it has been chosen as second instance. The third instance, the city of Amsterdam in the Netherlands, is interesting regarding the infrastructure for bicycles. In the following, the instances are introduced.

6.1.1 Karlsruhe

Karlsruhe in the German state of Baden-Württemberg is a city with about 304 000[1] residents. It is a planned city with streets arranged in the shape of a fan in the city center. A special feature of Karlsruhe is trams running on standard broad-gauge tracks. The tracks are connected to the regular German track network which makes it possible to transport people and goods from outside the city to the city center without transshipment. This makes package delivery particularly interesting applied to the city of Karlsruhe.

[1] https://www.karlsruhe.de/mobilitaet-stadtbild/stadtentwicklung/statistik-und-zensus

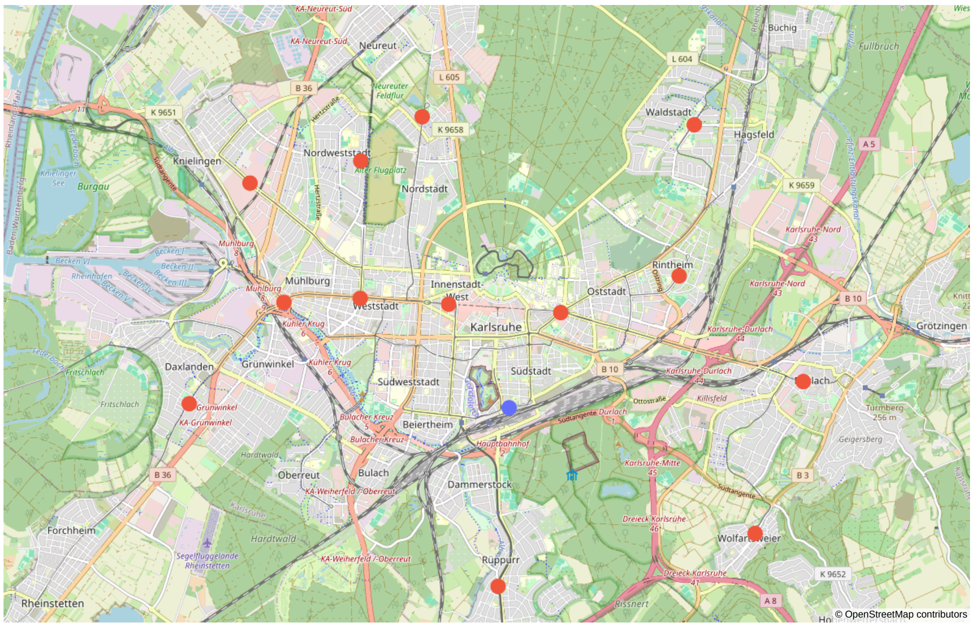
The Graph

The graph has been created based on data from OpenStreetMap, downloaded from Geofabrik[1]. The data has been converted to binary format using KaRRi's graph converting functionality[2]. The graph generated contains only the largest strongly connected component of OpenStreetMap's street network and does not contain any pedestrian walkways and areas. The latitude and longitude of the vertices and the length, speed limit and road category[3] of the edges are extracted.

The Depot and City Hubs

The depot has been placed at a depot of the German mail service (Deutsche Post AG), which is located near the railway station with connection to the tram network. The exact coordinates are: $48.99570, 8.40626$.

The city hubs have been placed at several stations of the Karlsruhe tram network. An overview is given in Table A.1 and Figure 6.1.



blue: depot, red: city hub

Figure 6.1: Depot and city hubs of the Karlsruhe instance

[1] http://download.geofabrik.de/europe/germany/baden-wuerttemberg/karlsruhe-regbez.html

[2] https://github.com/molaupi/karri/blob/main/RawData/ConvertGraph.cc

[3] https://wiki.openstreetmap.org/wiki/Template:Map_Features:highway

Generating Synthetic Parcel Demand

Demands are assigned to an edge (part of a street) they must be delivered at. Not all edges are reasonable for parcel delivery, e.g. parking lots or highways. The following edges meeting the following conditions are considered valid edges for parcel delivery:

- the length of the edge must be greater than 50 meters
- the speed limit of the street must not be higher than 50 kilometers per hour
- the road category must be one of { residential, living_street, secondary, tertiary }

Edges with a length lower than 50 meters are excluded to prevent to many demands in a certain areas because of increased fragmentation of streets. The speed of 50 km/h is the maximum speed of regular streets inside towns and cities. Streets with higher speed limits are not reasonable for city bike delivery. For the Karlsruhe instance, the number of valid edges is 8 934. An overview of them is given in Figure 6.2. 2 938 (33%) of the valid edges are within the 1 kilometer radius of any city hub.

It is not realistic for each of the edges to have a demand assigned. Thus, edges are uniformly chosen at random. In order to test different amounts of demands, there are two sets of demands. One contains demands associated to ~10% of the edges. The other contains demands associated to ~50% of the edges. The capacity usage of the demands is chosen as random integer between 1 and 4 (inclusive). The probability that a demand is a delivery is 95%. Thus, the probability that a demand is a pickup is 5%.

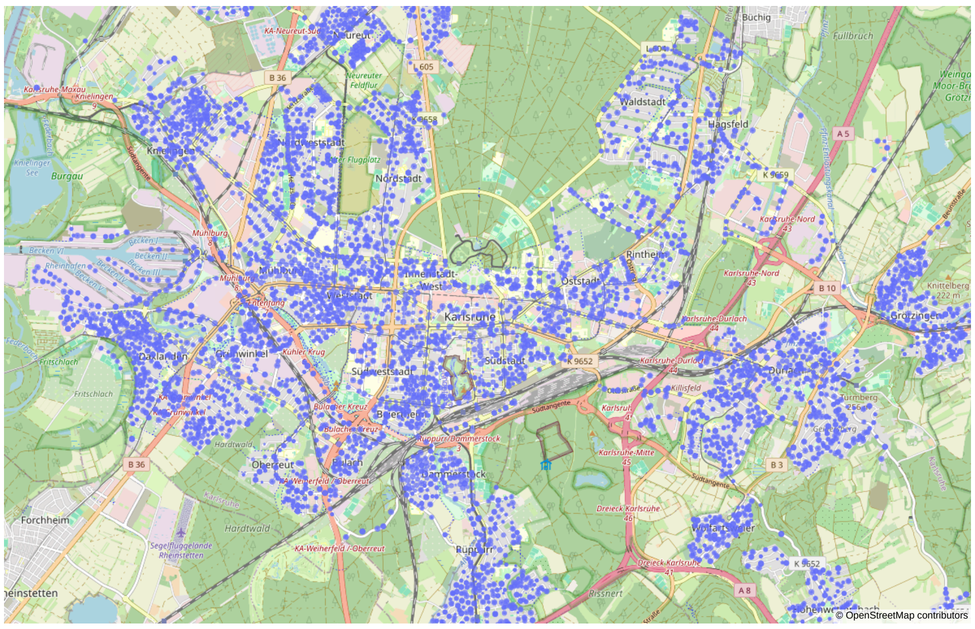


Figure 6.2: Valid edges of the Karlsruhe instance

Other Attributes

The cargo bike delivery radius r_b is set to 1000 meters. The maximum number of cargo bike tours N_b and the maximum number of sprinter tours N_s are set to a sufficiently high value of 99 so that the algorithm focuses on short paths instead of minimizing the number of vehicles. The capacity of cargo bikes C_b is set to 100 capacity units (cu). Thus, a cargo bike is able to transport between 25 and 100 parcels at a time, given that a single parcel has a capacity usage of at least 1 to at most 4. The capacity of sprinters C_s is set to 10 times C_b as proposed by [15]. The speed of cargo bikes v_b is set to 20.7 kilometers per hour. The speed of sprinters v_s is set to twice the speed of cargo bikes.

6.1.2 Santa Ana

Santa Ana in the U. S. state of California is a city with about 308 000[1] residents. It is located near Los Angeles in a very urban area. Most of its street network is arranged in a grid pattern. The different arrangement of streets in combination with similar number of residents compared to Karlsruhe makes the city important for testing purposes. Unlike Karlsruhe, Santa Ana does not have an exhaustive tram network, but a bus network with stops at many crossings.

The Graph

The graph has been created based on data from OpenStreetMap[2] the same way as the Karlsruhe instance.

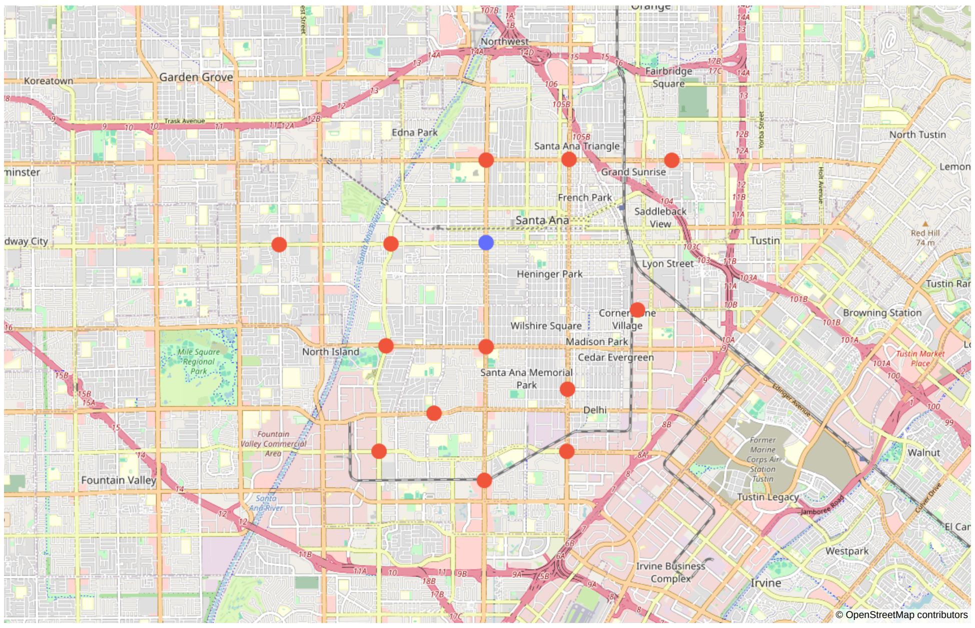
The Depot and City Hubs

The depot has been placed at the 1st-Bristol crossing in central Santa Ana. The exact coordinates are: $33.745464, -117.885096$.

The city hubs have been placed at several crossings in Santa Ana, which correspond to stations of the bus network. The small Orange County streetcar network is not exhaustive enough to be used. An overview is given in Table A.2 and Figure 6.3.

[1] https://www.census.gov/quickfacts/santaanacitycalifornia

[2] https://download.geofabrik.de/north-america/us/california/socal.html



blue: depot, red: city hub

Figure 6.3: Depot and city hubs of the Santa Ana instance

Generating Synthetic Parcel Demand

The demands are calculated in the same way as for the Karlsruhe instance. This results in 9 111 valid edges, which is roughly the same amount as in the Karlsruhe instance. An overview of them is given in Figure 6.4. 3 440 (38%) of the valid edges are within the 1 kilometer radius of any city hub, which also roughly corresponds to Karlsruhe's value.

Other Attributes

The other attributes of Santa Ana are the same as the other attributes of Karlsruhe, see Section 6.1.1.

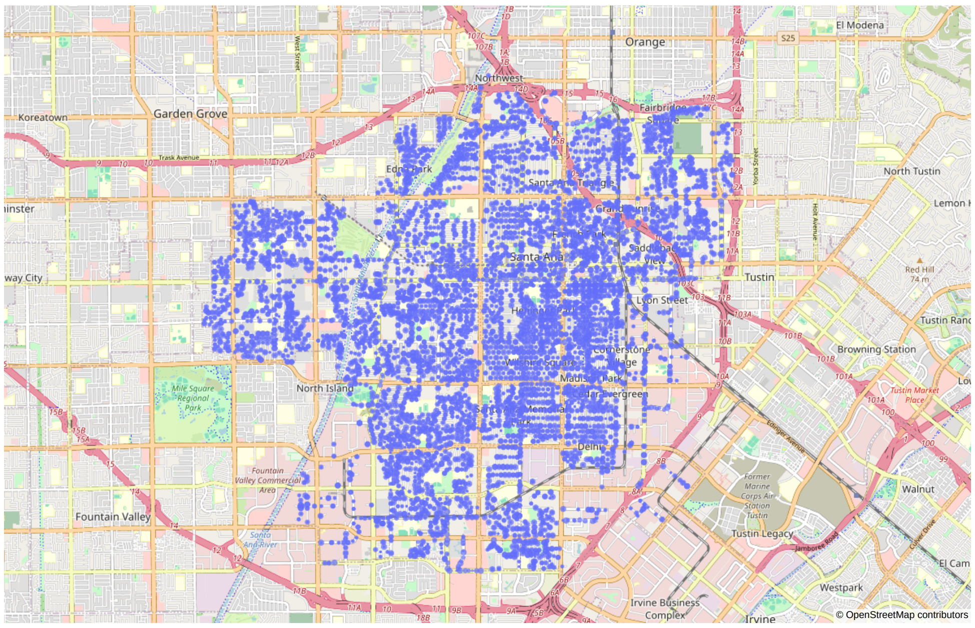


Figure 6.4: Valid edges of the Santa Ana instance

6.1.3 Amsterdam

Amsterdam in the Netherlands is a city with about 918 000[1] residents. The city is known for the good infrastructure for cyclists and their pioneering role in modern transport. The tram network is very dense in the southern part of the city.

The Graph

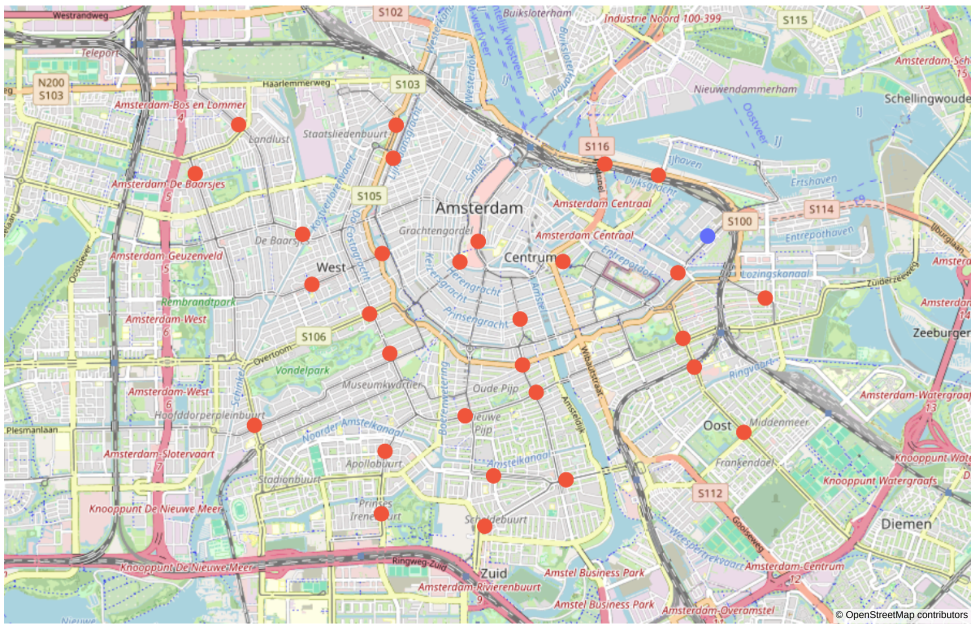
The graph has been created based on data from OpenStreetMap[2] the same way as the Karlsruhe instance. Because the city is divided in a bigger part where the city center is located and a smaller part in the south east that are not connected by any streets, only the central part of Amsterdam inside the bounds defined by the A10 highway[3] has been included.

[1]https://opendata.cbs.nl/#/CBS/en/dataset/37259eng/table?ts=1709544739543
[2]https://download.geofabrik.de/europe/netherlands/noord-holland.html
[3]https://www.openstreetmap.org/relation/13537038

The Depot and City Hubs

The depot has been placed in the center of Amsterdam with connection to the tram network and near the railway station. The exact coordinates are: $52.37024, 4.92900$.

The city hubs have been placed at stations of the Amsterdam tram network. Because the network is too dense, 30% of the stations have been randomly chosen as city hubs. Note that there are no stations in the northern part of Amsterdam. An overview is given in Table A.3 and Figure 6.5.



blue: depot, red: city hub

Figure 6.5: Depot and city hubs of the Amsterdam instance

Generating Synthetic Parcel Demand

The demands are calculated in the same way as for the Karlsruhe instance. This results in 7 515 valid edges. An overview of them is given in Figure 6.6. 4 470 (59%) of the valid edges are within the 1.5 kilometer radius of any city hub.

Other Attributes

The other attributes of Amsterdam are mostly the same as the other attributes of Karlsruhe, see Section 6.1.1. The only exception is the cargo bike radius r_b is set

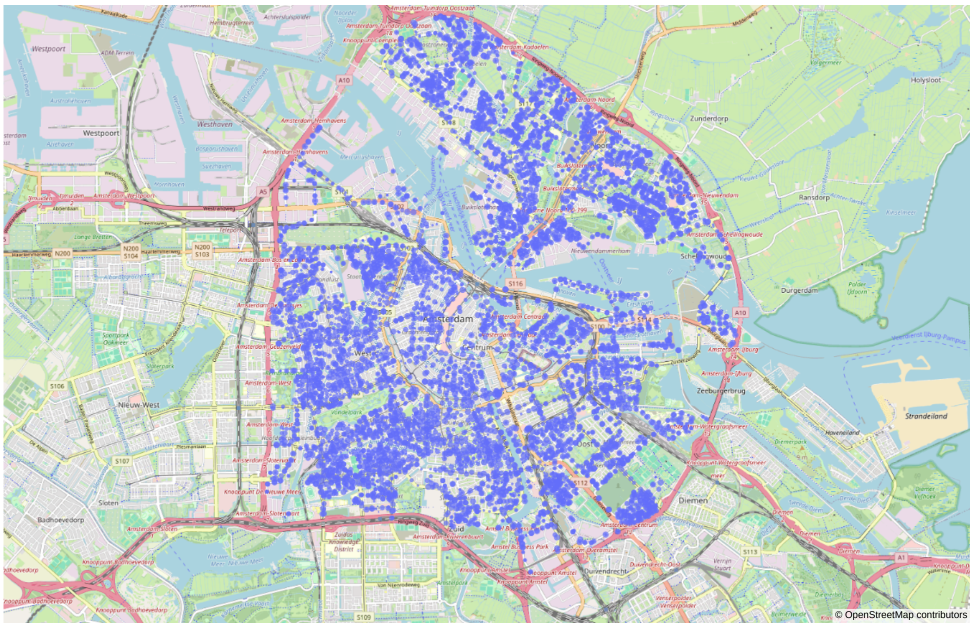


Figure 6.6: Valid edges of the Amsterdam instance

to 1500 meters instead of 1000 meters in order to cover more streets close to the boundary in the southern part of Amsterdam.

6.2 Multi-Depot Ant Algorithm

6.2.1 Algorithm Parameters

In this section, the multi-depot ant algorithm algorithm will be tested against a parameter space. The goal is to find which values produce good results for parameters for the algorithm. The algorithm has four parameters that can be specified independently from the input problem instance: $antCount$, β, ρ and q_0. $antCount$ specifies the number of ants that are used within one iteration. Each ant conforms to one generated solution, that can be feasible or not. β specifies the influence of the attractiveness value η on the probability of a next leg (see Equation (5.8)). β can only accept values $\beta > 0$ because $\eta_{from,to} = 0 \implies p_{from,to} = 0$. A value of 1 means that the pheromone value τ is weighted the same as the attractiveness η. A value < 1 increases the influence of τ, a value > 1 increases the influence of η. $\rho \in [0,1]$ specifies the share of influence of the previous pheromone value in a pheromone update (see Equations 5.3 - 5.6). A value of 0 means that the previous pheromone value has full influence on the next pheromone value (meaning that the initial pheromone value never changes). A value of 1 means that the pheromone update value has full influence on the next pheromone value. The scale between 0 and 1 is linear. Lastly, $q_0 \in [0,1]$ specifies the probability of exploitation being performed. Exploration is performed with probability $1 - q_0$.

In the first part of the test, only the parameters $antCount$, β and ρ are tested. The second part includes q_0, but not $antCount$. It is performed on a more narrow parameter space.

Test Setup (1)

The test is performed on the Karlsruhe instance using the smaller demand set including ~10% of the possible edges. Only demands within the range of a city hub have been taken into account. The timeout of each test has been set to 2 minutes which poses a good balance between solution quality and practicability. The test has been performed 10 times for each combination of parameter values. The tour lengths of each combination of parameter values is summed up. The following parameter space has been chosen:

- $antCount \in \{1,2,5,10,20,50,100\}$
- $\beta \in \{1.0,2.0,3.0,4.0,8.0,12.0,16.0\}$
- $\rho \in \{0.0,0.05,0.30,0.50,0.70,0.95,1.0\}$

Results (1)

Figure 6.7 shows a plot of the tour length depending on the three parameters $antCount$, β and ρ. The dots are arranged in a grid and every dot represents 10 measurements with the same parameter values. The color of the dots shows the solution quality, namely the length of the tour. A yellow dot represents a longer tour length and thus a worse solution. A blue dot represents a shorter tour length and thus a better solution.

Values for β of $1,2,3$ and 4 consistently yield bad solutions for every combination of $antCount$ and rho. The tour length of $1{,}392{,}790$ (meters) of these dots corresponds to the tour length found by the initial solution generator. This means, that there has been no improvement found at all.

The $antCount$ also has influence on solution quality. $antCounts$ between 1 and 20 perform equally well, with small benefits among smaller numbers. Other than that, higher numbers than 20 perform worse.

ρ values mostly perform equally well with overall better values in the mid of the interval.

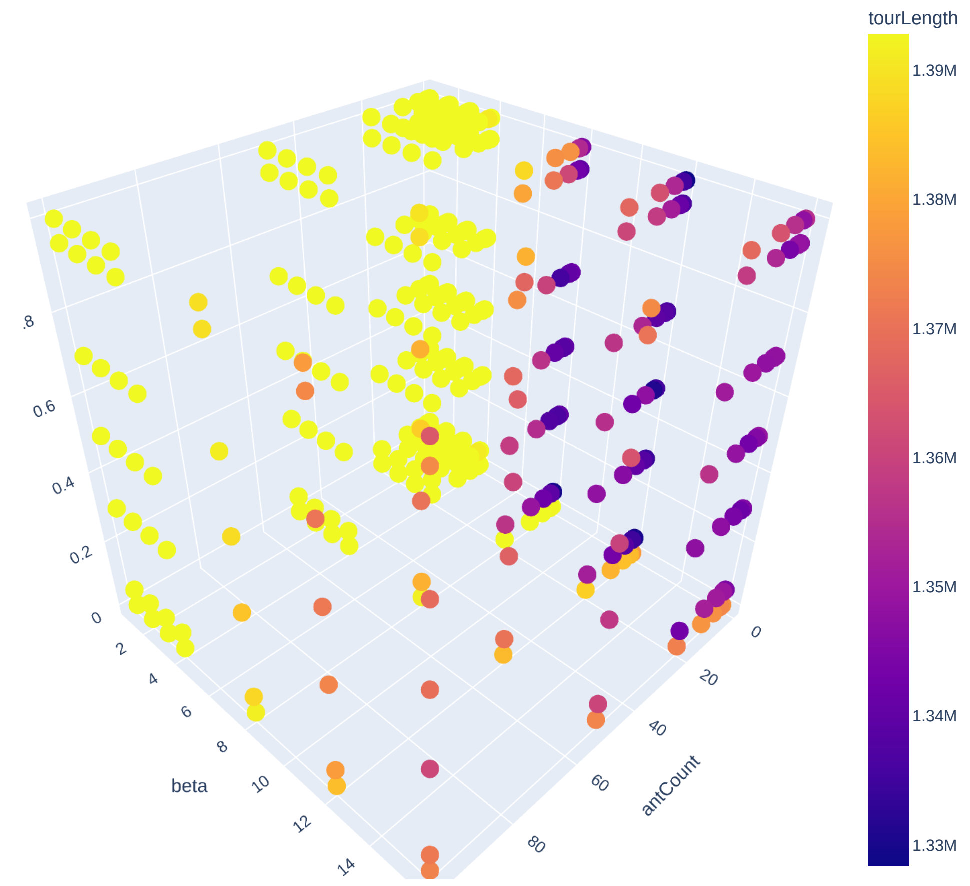


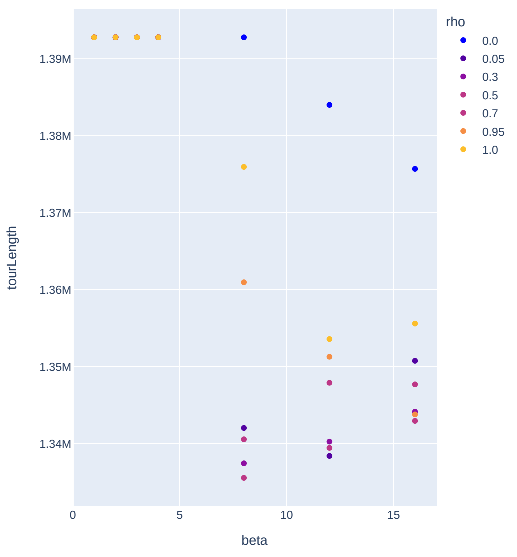
Figure 6.7: 3D plot of tour lengths depending on $antCount$, β and ρ

The best parameter space according to this test is $\{(antCount, \beta, \rho) \mid antCount \in \{1{,}2{,}5{,}20\}, \beta \in \{8{,}12{,}16\}, \rho \in \{0.3, 0.5, 0.7\}\}$.

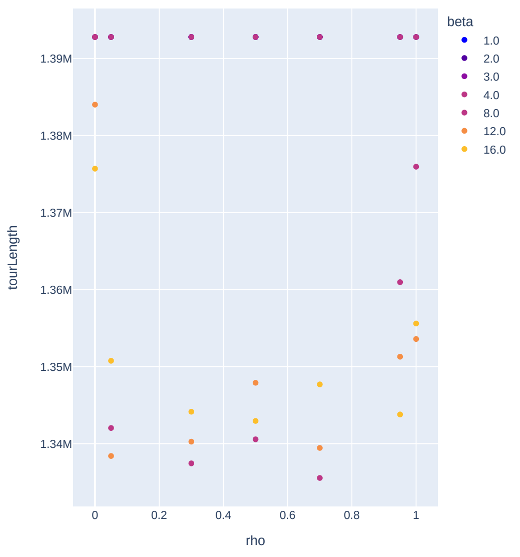
Figure 6.8 shows the same data for a fixed $antCount$ of 5. In plot a) it can be seen that the values of $\beta \in \{8, 12, 16\}$ yield similarly-well results with small benefits at $\beta = 8$. Plot b) shows that extreme ρ values of 0 and 1 yield worse results compared to others. ρ values of 0.05 and 0.95 perform a lot better than their nearby values 0 and 1.

Interpretation and Explanation (1)

The data shows that high $antCount$ values lead to bad solutions. A high number of ants for each iteration also reduces the number of updates to the pheromones τ when the computation time is equal. A high number of pheromone updates is important for a



a) $\beta \mapsto$ tour length plot



b) $\rho \mapsto$ tour length plot

Figure 6.8: Visualization for $antCount = 5$

good solution quality. The data also shows that too low β values contribute to bad solution quality. This can be explained by the fact that β values determine the influence of the attractiveness η. The attractiveness value is important to reduce the probability to make bad local choices. This leads to the question if the use of pheromone values τ is important at all. This is covered in Section 6.2.2. Lastly, the data shows that extreme ρ values of 0 and 1 lower solution quality. With such an extreme value, the pheromone values after each iteration are solely determined by the initial pheromone value (case $\rho = 0$) or the currently best solution (case $\rho = 1$). This must lead to worse solutions because the algorithm should neither consider only the initially-found nor only the lastly-found solution.

Test Setup (2)

The second part of the test is performed exactly the same as the first test, but with the following different parameter space. Some of the underperforming parameter values have been excluded:

- $\beta \in \{8.0, 12.0, 16.0\}$
- $\rho \in \{0.30, 0.50, 0.60\}$
- $q_0 \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$

Results (2)

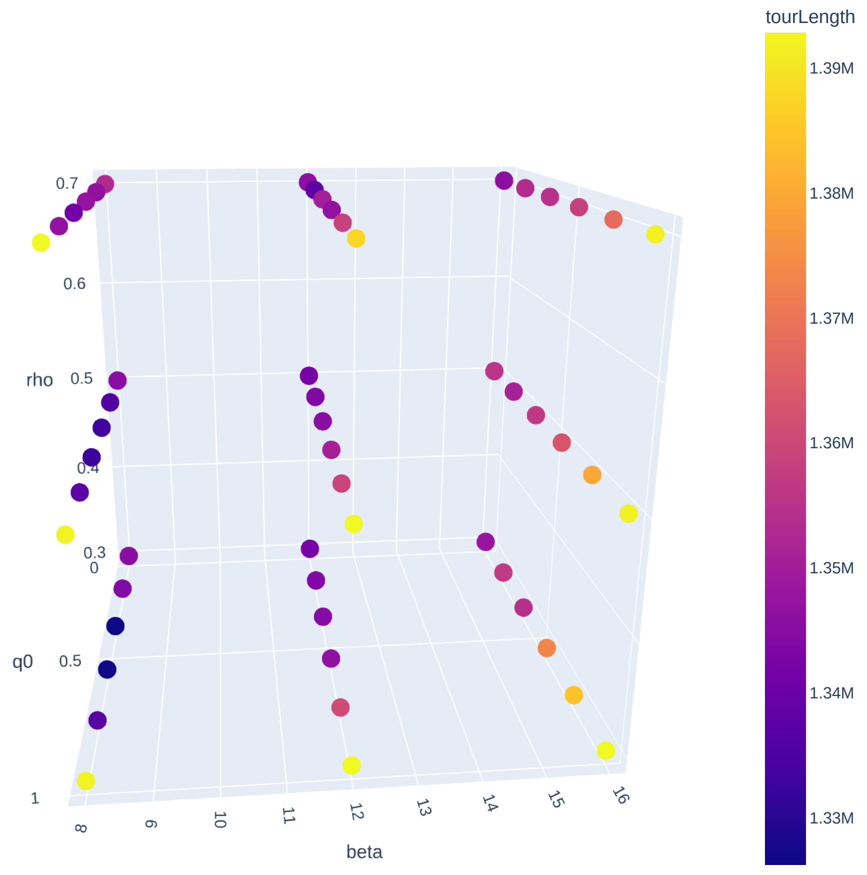


Figure 6.9: 3D plot of tour lengths depending on β, ρ and q_0

Figure 6.9 shows that both extremes $q_0 = 0$ and $q_0 = 1$ yield worse tour lengths than values in between with $q_0 = 1$ performing much worse. $q_0 = 0.4$ and $q_0 = 0.6$ perform best with parameter $\beta = 8.0$. The tour respective average tour lengths are 1 335 481 ($q_0 = 0.4$) and 1 333 309 ($q_0 = 0.6$). With other values for β, the best value is $q_0 = 0.2$, but the tour length is higher.

Interpretation and Explanation (2)

The data shows that $q_0 = 1$ yields bad solutions. This is reasonable. It means that exploitation is always performed, which leads to a high share of equal solutions among generated solutions. The only way to explore other solutions is to wait for the pheromone value to drop due to evaporation. A high value of ρ accelerates this process. This is why there are slightly better results for $\rho = 0.7$.

In general, values $q_0 \in \{0.2, 0.4, 0.6\}$ yield better results. There is an interplay with parameter β. A value of q_0 should be chosen accordingly.

6.2.2 Comparison with Disabled Pheromones and Attractiveness

In this section, the influence of disabled pheromones and attractiveness in the multi-depot ant algorithm will be tested. The pheromones τ store global information about good solutions, whereas the attractiveness η stores local information about good solutions. The goal is to test the algorithm with parts of information disabled.

Test Setup

The test is performed in the same way as described in Section 6.2.1, but the test is performed with a timeout of 2 hours. The following setups are tested:

a) pheromones τ disabled, attractiveness η enabled
b) pheromones τ enabled, attractiveness η disabled
c) both pheromones τ and attractiveness η disabled ($\iff$ purely random choice)
d) both pheromones τ and attractiveness η enabled ($\iff$ normal procedure)

Results

Table 6.1 shows the result of the test. Each of the four cells represents one of the tested setups. The first value is the tour length in meters. The second value shows the percentage of "useful" solutions covering every demand and the total number of solutions generated. Equal solutions may be counted more than once.

The results show that during the test, the number of solutions covering all demands is 0 if the attractiveness is disabled. If the attractiveness is enabled, but the pheromones disabled, the number is roughly the same as in the normal run with both enabled. Nevertheless, there has been no improvement found compared to the initial solution.

Interpretation and Explanation

The test shows that the interplay of pheromones τ and attractiveness η is extremely important. If one of both is disabled, the solutions generated are unusable.

Table 6.1: Comparison of multi-depot ant algorithm runs with disabled pheromones or attractiveness

		τ enabled		τ disabled	
		tour length	useful solutions	tour length	useful solutions
η	enabled	128 120 m	86% (of 725 984)	139 279 m	81% (of 853 919)
	disabled	139 279 m	0% (of 713 518)	139 279 m	0% (of 1 227 910)

6.2.3 Comparison with Jsprit

In this section, the performances of the multi-depot ant algorithm and the open source rich VRP solver Jsprit [18] are compared on several instances. Jsprit version 1.9.0 is used.

Test Setup

The test is performed on the Karlsruhe instance. One iteration is performed with the same set of edges as in Section 6.2.1, the other iteration is performed with another set of ~50% of the edges. Only edges in range of any city hub are taken into account. First, Jsprit is run with the standard iteration threshold. The time Jsprit takes to terminate is measured. This exact time is then assigned as timeout of the multi-depot ant algorithm.

Results

Table 6.2 shows the results of the first iteration. The Jsprit algorithm had a run time of 1507 seconds (25.1 minutes). It yielded a result of 119 707 meters using 14 vehicles. The runs of the multi-depot ant algorithm with $\beta = 12.0$ yielded worse results than the runs with $\beta = 8.0$. The latter varied between a best of 124 816 meters and a worst of 125 512 meters using an equal amount of 13 vehicles. Figures 6.10 and 6.11 show the routes of the tours taken.

Table 6.3 shows the results of the second iteration. The Jsprit algorithm had a run time of 7304 seconds (2 hours). It yielded a result of 301 898 meters using 41 vehicles. In this iteration, $\beta = 12.0$ yielded better results. The best result of the multi-depot ant algorithm is 380 211 meters using 43 vehicles.

Table 6.2: Comparison of Jsprit and the multi-depot ant algorithm results for instance with ~10% of edges

algorithm	*antCount*	β	ρ	q_0	# vehicles	tour length	% tour length
Jsprit	-	-	-	-	**14**	**119 707 m**	**100**
MDAA	5	8.0	0.3	0.2	13	125 409 m	105
MDAA	5	8.0	0.3	0.4	13	125 512 m	105
MDAA	5	8.0	0.3	0.6	**13**	**124 816 m**	**104**
MDAA	5	12.0	0.3	0.2	13	127 285 m	106
MDAA	5	12.0	0.3	0.4	13	131 672 m	110
MDAA	5	12.0	0.3	0.6	14	129 824 m	108

Table 6.3: Comparison of Jsprit and the multi-depot ant algorithm results for instance with ~50% of edges

algorithm	*antCount*	β	ρ	q_0	# vehicles	tour length	% tour length
Jsprit	-	-	-	-	**41**	**301898 m**	**100**
MDAA	5	8.0	0.3	0.4	43	386968 m	128
MDAA	5	8.0	0.3	0.6	43	386968 m	128
MDAA	5	12.0	0.3	0.4	**43**	**380211 m**	**126**
MDAA	5	12.0	0.3	0.6	42	381470 m	126

6.2.4 Solution Generation in the Course of Time

This section investigates the process of solution generation by the multi-depot ant algorithm over a long period of time. This helps to evaluate how good the algorithm is at calculating good solutions after a certain period of time. The parameter q_0 will be varied by two values to see typical differences in solution quality. The processes of solution generation by the multi-depot ant algorithm and Jsprit are compared.

Test Setup

The test is performed in the same way as described in Section 6.2.1, but the test is performed with a timeout of 1 hour. The parameters $antCount = 5$, $\beta = 8.0$, $\rho = 0.3$ have been chosen. q_0 is varied by 0.4 and 0.6. During the test, every solution considered "useful" (namely those covering all demands) is taken into account. Not "useful" solutions are not considered as their possibly lower tour lengths might disrupt the plot.

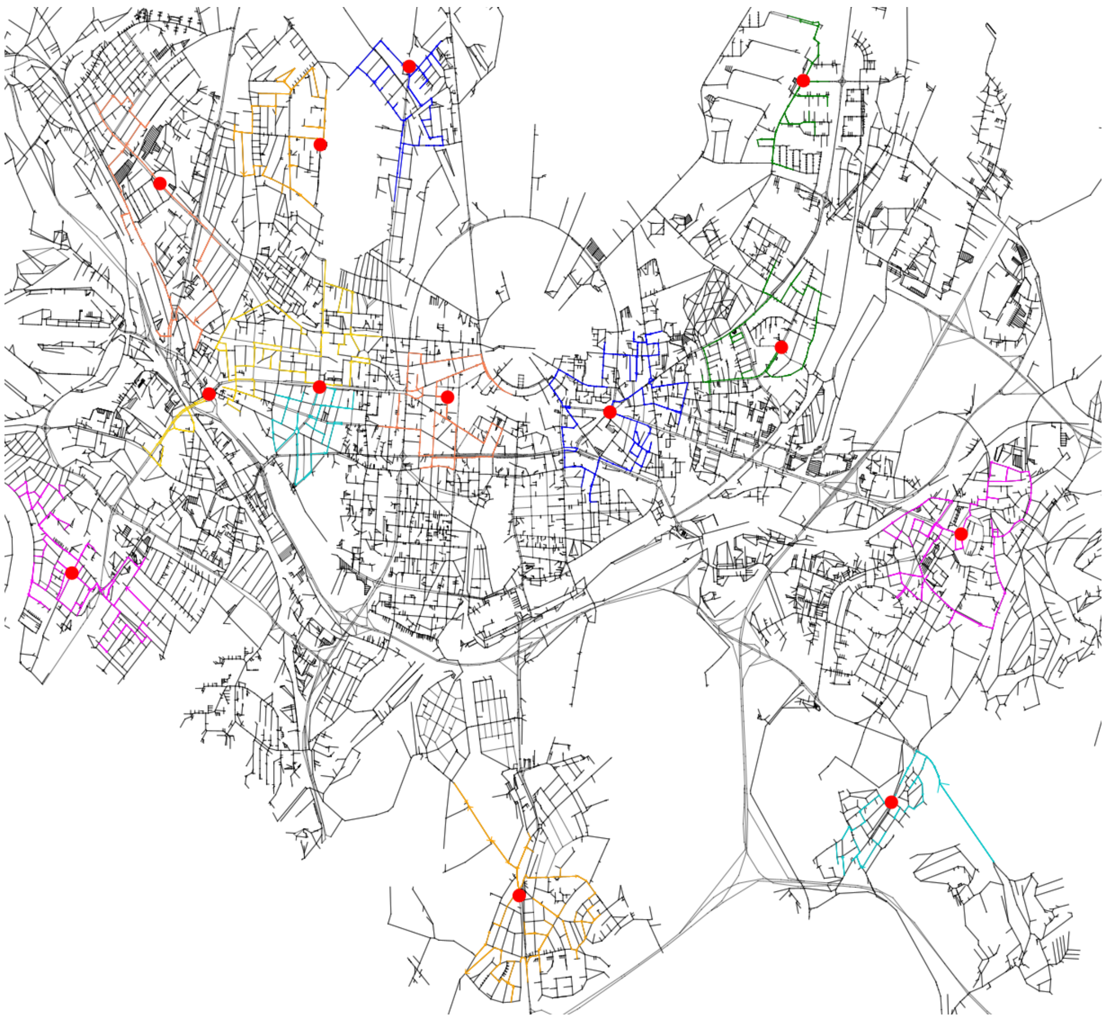


Figure 6.10: Jsprit results for instance with ~10% of edges

Results

Figure 6.12 shows the results of the runs in one plot. The x axis describes the time in seconds, the y axis the tour length in meters. The two upper lines represent the mean value over 10 values. The two lower lines represent the minimum value over 10 values.

The path length has been reduced from ~139 300 to ~124 700 ($q_0 = 0.6$) and ~123 000 ($q_0 = 0.4$). Both pairs of lines are near to each other. $q_0 = 0.6$ has a lower average than $q_0 = 0.4$ over the whole time span. $q_0 = 0.6$ also has a lower minimum value until ~400 seconds, after which $q_0 = 0.4$ almost always has a lower minimum value. Until ~1500 seconds, the average and minimum values drop permanently. After that, the mean values drop slowly but steadily and the minimum values keep approximately the same with more variation in case of $q_0 = 0.4$.

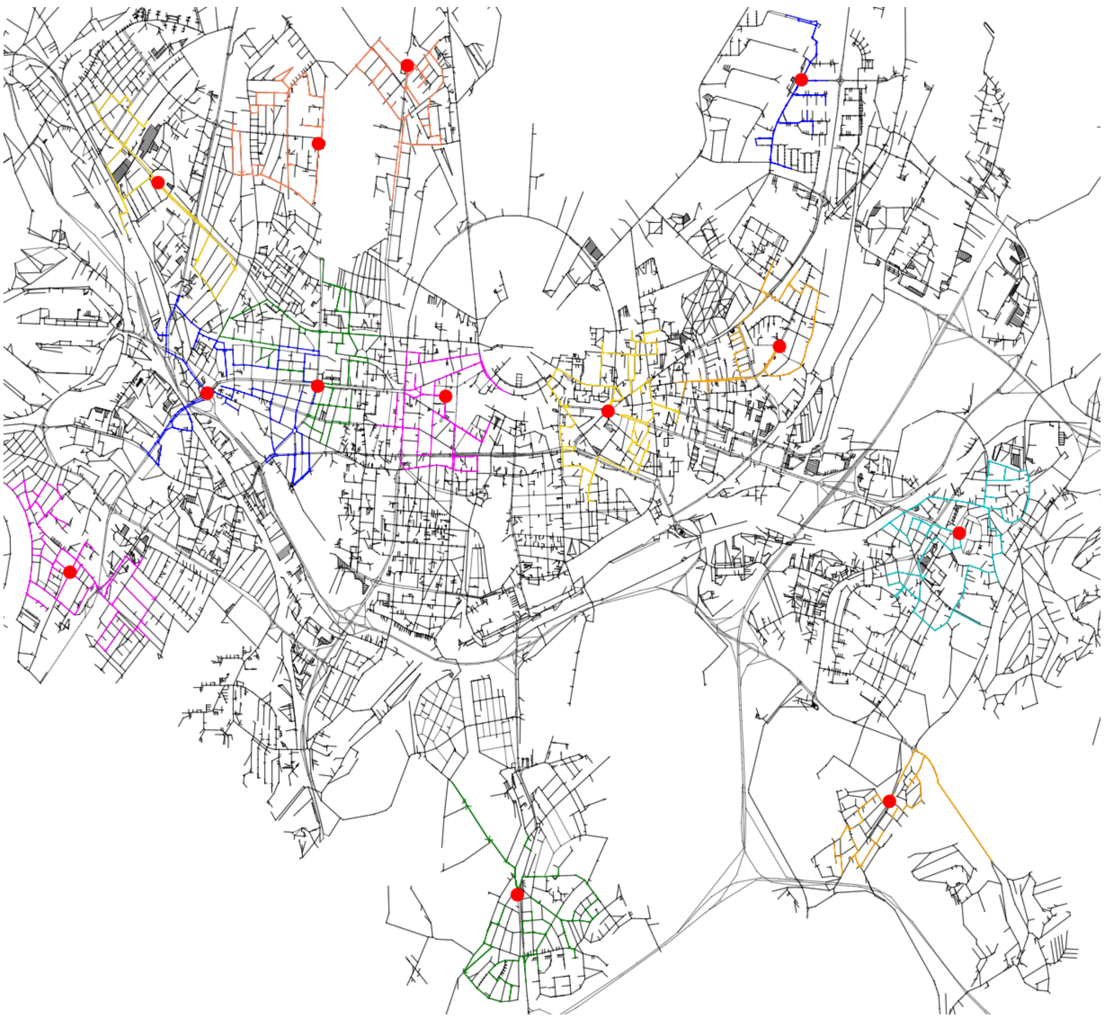


Figure 6.11: Multi-Depot Ant Algorithm results for instance with ~10% of edges

Figure 6.13 shows the same results as Figure 6.12, but separated by value of q_0 and with every measurement represented as dot.

Interpretation and Explanation

In general, $q_0 = 0.4$ performed better than $q_0 = 0.6$, but $q_0 = 0.4$ had a slower start. This is because more exploitation than exploration leads to more frequent selection of (currently) best choices. However in the long run, increased exploration succeeds. The more frequent exploration can also be observed in the more frequent peaks and lows in case of $q_0 = 0.4$, while $q_0 = 0.6$ has more level sections. $q_0 = 0.4$ has a lower tour length after ~1500 seconds although the mean of $q_0 = 0.6$ remains lower during the full period. This is also due to more frequent exploitation of best choices.

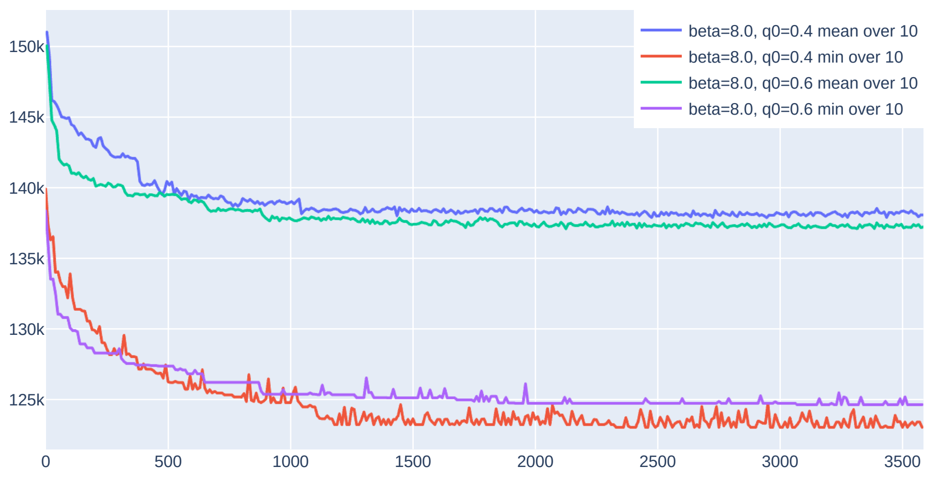
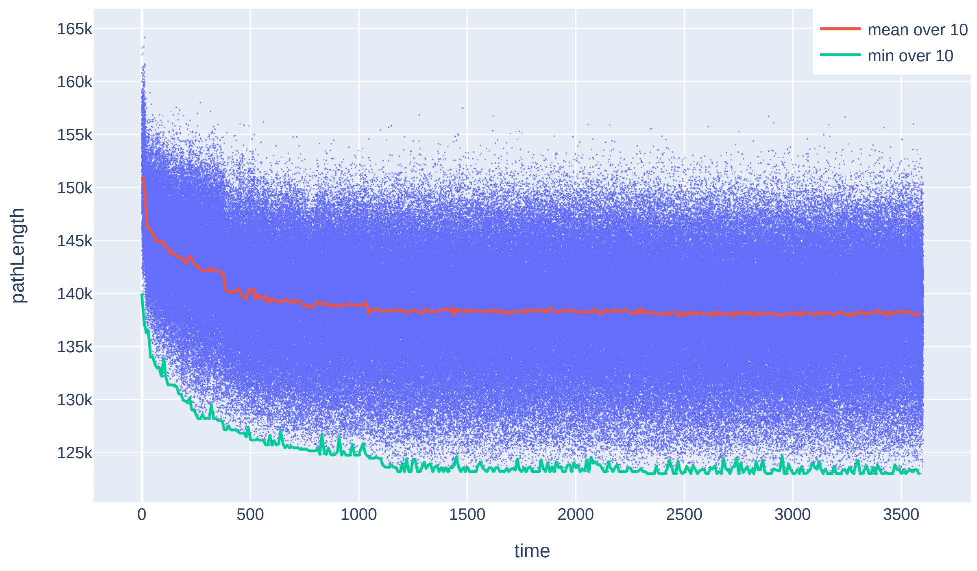


Figure 6.12: Mean and minimum of solution quality during 1h runs of the multi-depot ant algorithm with $q_0 \in \{0.4, 0.6\}$

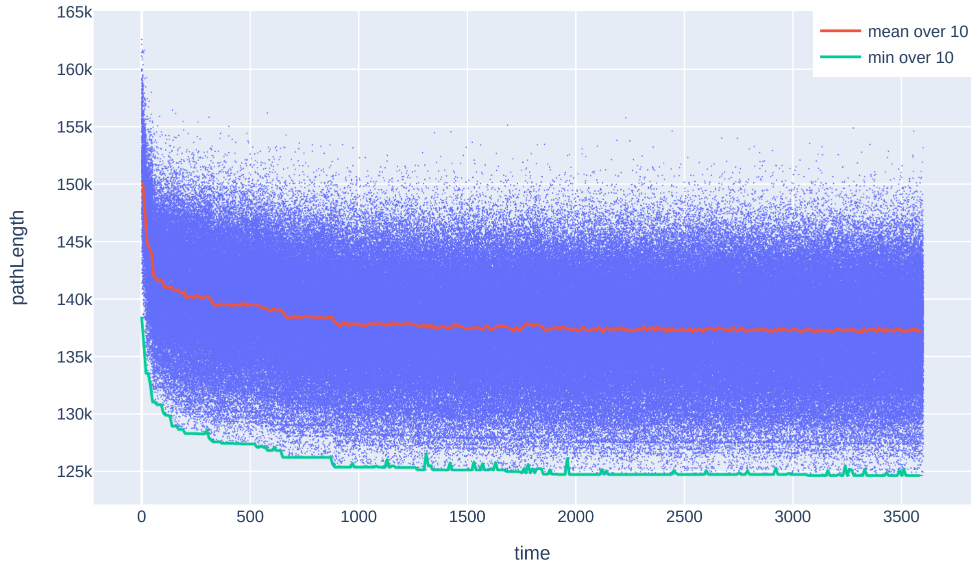
Comparison to Jsprit

Figure 6.14 shows the Jsprit's process of solution generation, as outputted by Jsprit itself. There are two things to note: The diagrams shows the mean solution quality depending on the iteration, not the time. Also, the best results as derived from the figure are 105 000 and 240 000, which does not fit to the tour lengths in Tables 6.2 and 6.3. This is because the tour lengths are re-calculated afterwards so that they are comparable to the tour lengths of the multi-depot ant algorithm. Jsprit does not support edge routing (cf. Section 5.1.1), which leads to a different distance between demands.

Jsprit's path length does not drop as much as the path length of the multi-depot ant algorithm at the very beginning. This is due to the fact that Jsprit spends more time on an initial solution than the multi-depot ant algorithm. The drop thereafter is correspondingly weaker.

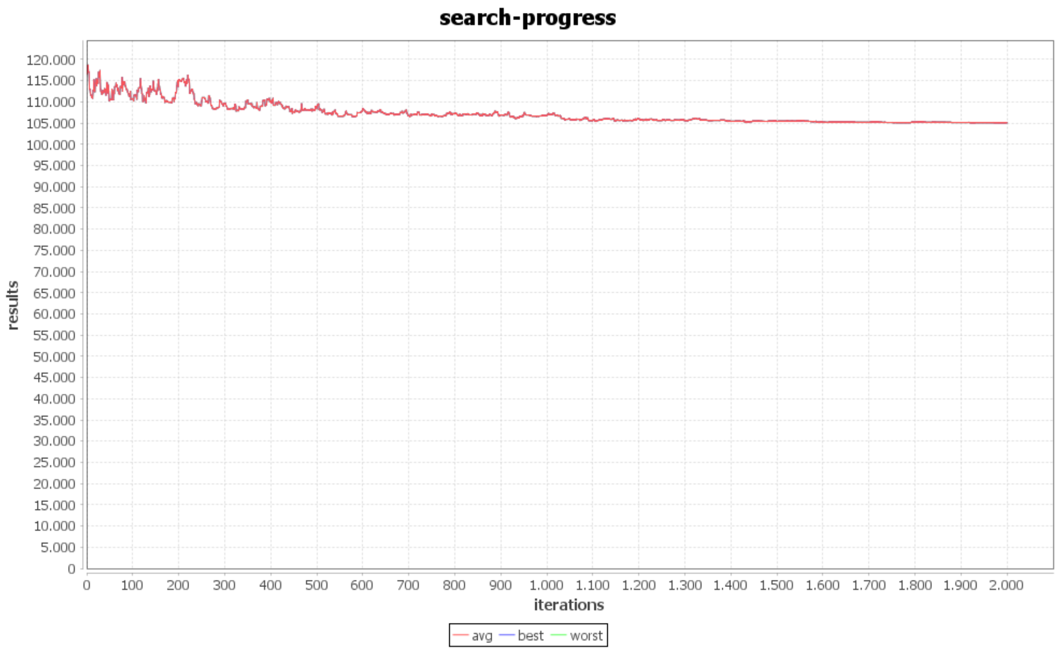


a) $q_0 = 0.4$

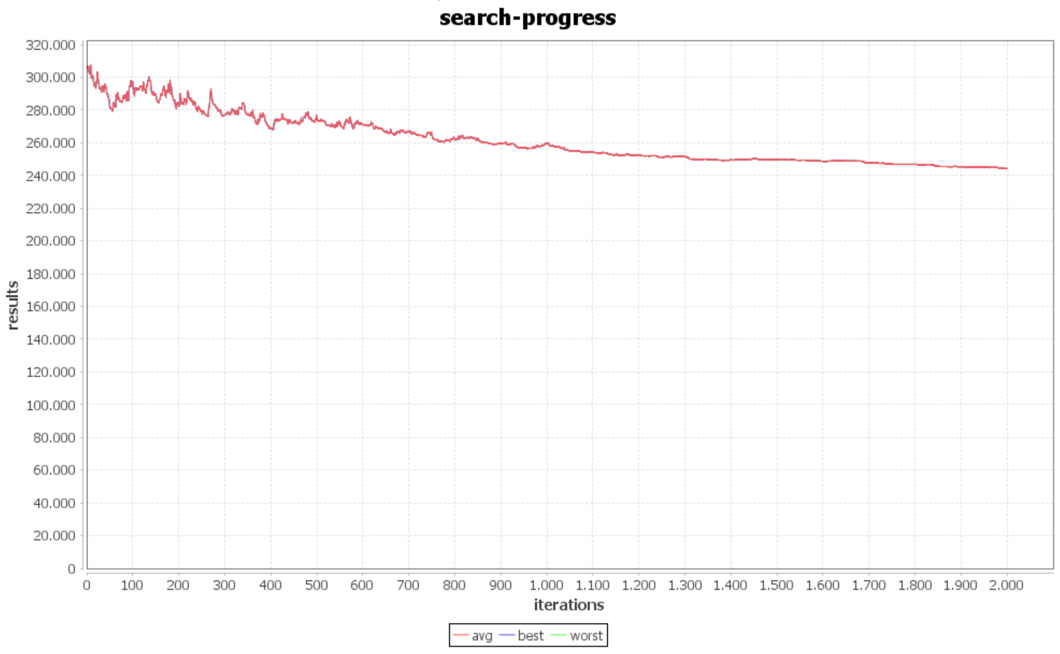


b) $q_0 = 0.6$

Figure 6.13: Solution qualities during 1h runs of the multi-depot ant algorithm with $q_0 \in \{0.4, 0.6\}$



a) Jsprit run in Table 6.2



b) Jsprit run in Table 6.3

Figure 6.14: Solution qualities during runs of Jsprit

6.3 Greedy Tram Assignment Algorithm

In this section, the greedy tram assignment algorithm is tested in two steps. The first step uses a randomly generated solution of a last mile delivery. The second step uses a previously-calculated solution by the multi-depot ant algorithm.

6.3.1 Randomized Last Mile Solution

The test using a random last mile solution compares the performance of the greedy tram assignment algorithm with the successive Knapsack algorithm.

Test Setup

The test is performed on an instance that has been randomly generated. The instance contains 40 tours each having a random summed delivery capacity usage $d \in [15,75)$. The values are taken from typical solutions of the multi-depot ant algorithm. Pickups are not considered in this test. The tour length is $(p+d)/90d$. The tour starts at a random time $t \in [0,0.6)$. The tram schedule is determined by a summed capacity. The summed capacity is varied by the values $\{1000, 1500, 2000, 2500\}$. The tram schedule contains five time slots $[0,0.25),[0.25,0.5),[0.5,0.75),[0.75,1),[1,1.25)$. Furthermore, two types of capacity distributions are evaluated. The first time, an even distribution of capacity is assigned to the tram schedule, where every time slot gets a capacity of $0.16*$ summed capacity. The second time, the distribution is the following: $[0.05,0.1,0.3,0.3,0.25]$, each multiplied by the summed capacity. This resembles a scenario where in the morning and evening, the tram capacities are lower than during the day due to commuting.

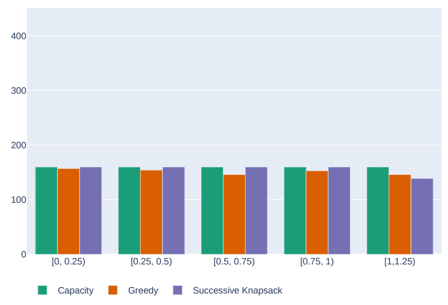
Results

Figure 6.15 and Table 6.4 show the results for even distribution, Figure 6.16 and Table 6.5 for uneven distribution. The green bars show the available capacity in a time slot. The red bars show the load calculated by the greedy tram assignment algorithm. The blue bars show the load calculated by the successive Knapsack algorithm.

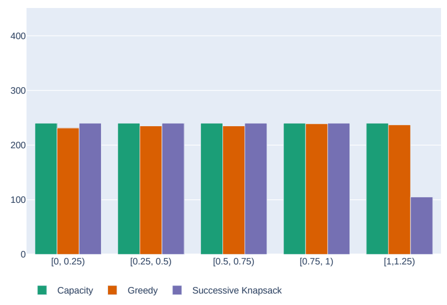
It should be noted that the successive Knapsack algorithm is an $\frac{1}{2}$-approximation algorithm [5]. This means that twice the result of the successive Knapsack algorithm is an upper bound of the optimal solution in Tables 6.4 and 6.5. The optimal solution might be below 100%.

The load of the successive Knapsack algorithm is higher than the load of the greedy tram assignment algorithm in the first three time slots. In the other two time slots, the reverse is the case. This can be observed in both tests and for all four capacities.

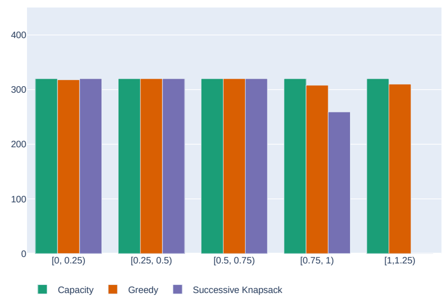
The greedy tram assignment algorithm has an overall better performance in both tests, with one exception in the uneven distribution test at the capacity of 1 000.



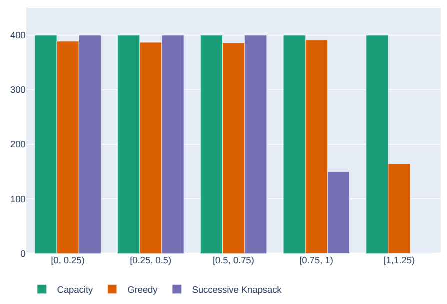
a) Capacity of 1 000



b) Capacity of 1 500



c) Capacity of 2 000



d) Capacity of 2 500

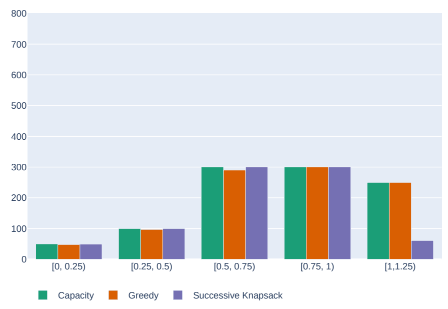
Figure 6.15: Tram assignment results at even distribution of capacity

Table 6.4: Percentage of load at even distribution of capacity

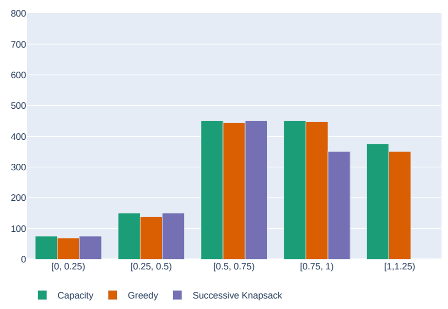
algorithm	capacity [%]			
	1 000	1 500	2 000	2 500
Greedy	60.8	89.5	92.4	96.4
Knapsack	50.0	63.3	65.5	69.4

Interpretation and Explanation

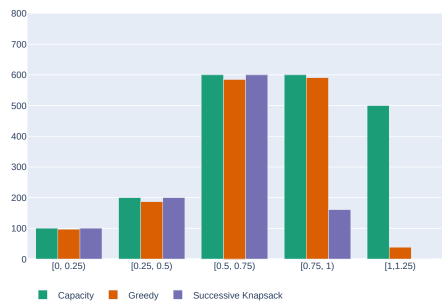
The effect of a higher load in the first three time slots by the successive Knapsack algorithm can be explained by the plausible fact that the dynamic programming Knapsack algorithm calculates the best solution to a Knapsack problem and at the beginning, more demands can be chosen. At later time slots, the successive Knapsack algorithm has made choices it cannot undo. It does not choose demands according to how far in



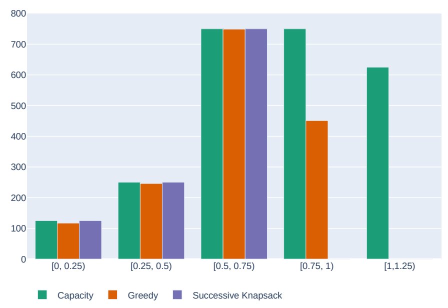
a) Capacity of 1 000



b) Capacity of 1 500



c) Capacity of 2 000



d) Capacity of 2 500

Figure 6.16: Tram assignment results at uneven distribution of capacity

Table 6.5: Percentage of load at even distribution of capacity

algorithm	capacity [%]			
	1 000	1 500	2 000	2 500
Greedy	44.0	68.5	91.8	100.0
Knapsack	45.4	62.0	71.0	78.6

time it can be postponed, which is what the greedy tram assignment algorithm does. This effect leads to an overall worse performance of the successive Knapsack algorithm in both even and uneven distribution. This shows that the greedy tram assignment algorithm should be chosen for this type of problem, although the successive Knapsack algorithm may seem like a better choice.

6.3.2 Last Mile Solution of the Multi-Depot Ant Algorithm

This section presents the performance of the greedy tram assignment algorithm on two solutions from Section 6.2.3. The best-performing solutions with parameters $\beta = 8.0$, $q_0 = 0.6$ and $\beta = 12.0$, $q_0 = 0.4$ are used as input.

Table 6.6: Percentage of load by the greedy tram assignment algorithm

instance	distribution	capacity [%]						
		100	200	300	400	500	600	700
$\beta = 8.0,\, q_0 = 0.6$	even	0.0	11.9	30.8	76.3	100	100.0	100.0
$\beta = 8.0,\, q_0 = 0.6$	uneven	2.8	30.8	87.9	100.0	100.0	100.0	100.0
$\beta = 12.0,\, q_0 = 0.4$	even	0.3	2.6	9.9	15.8	73.5	96.1	100.0
$\beta = 12.0,\, q_0 = 0.4$	uneven	1.5	9.9	46.7	81.8	87.1	97.4	100.0

6.4 Intermodal Tour Algorithm

In this section, the performance of the whole proposed intermodal tour algorithm (IMTA) is evaluated. Karlsruhe, Santa Ana and Amsterdam serve as instances. For each of the cities, three cases are tested. The first case is performed with ~10% of the edges and 10 hours of computation time. The second is performed with ~50% of the edges and 16 hours of computation time. The third is equally performed with ~50% of the edges and 16 hours of computation time, but only sprinter delivery is enabled and cargo bike and tram delivery is disabled. For more realistic results, for every demand, 2 minutes of time needed to pickup or deliver the demand are estimated. The time window of all demands is set to an interval of eight hours ($t_s = 0$ and $t_e = 8$). For the Karlsruhe and Santa Ana instances, the tram schedule is set in such a way, that all demands could be transported by tram to see the performance of the intermodal tour algorithm without the greedy tram assignment algorithm. For the Amsterdam instance, the tram schedule is set to a capacity of 300 per city hub during the whole time. Pickups and deliveries have a separate capacity of 300 each.

The solutions are broken down by the tours of a vehicle type. The number of stops, the maximum utilization of the vehicle at any point during the tours, the length of the tours, the time needed to drive the tours without stopping, the overall working time including the time to drive the tours and load and unload the demands and the emissions of CO_2 are presented. To calculate the CO_2 emissions of a tour, an emission factor of $0.087\frac{l}{km} * 2700\frac{g}{l} \approx 235\frac{g}{km}$ has been applied to sprinter vehicles[1] [11].

The Tables 6.7 - 6.15 show the results calculated by the intermodal tour algorithm. For results broken down by tour, the reader is referred to the Tables B.1 - B.10. The results give an idea of how many and which tours would be needed for intermodal parcel deliveries and pickups, given that there is approximately the same demand at each valid edge. In reality, there would be more demands and the demand volume would more depend on the particular area. Please note that the results show tours, not

[1]`https://www.mercedes-benz.de/vans/models/sprinter/panel-van/overview.html#technical-data`

work assignments of employees. It would be reasonable to assign multiple tours to one employee if applicable.

For Karlsruhe and Santa Ana, the number of stops served by cargo bikes is approximately half the number of stops served by sprinters. Nevertheless, the number of cargo bike tours is higher than the number of sprinter tours. This is due to the higher capacity of sprinters.

The working time is around 2 hours for cargo bike tours and 8 hours for sprinter tours. This is because of different constraints restricting the admission of demands. Cargo bike tours are mostly ended after a shorter time because the load of the cargo bike reaches the relatively low capacity. This can also be observed in the high utilization of cargo bike tours. Sprinter tours are mostly ended after a longer time because of the end of the demands' time windows at 8 hours and not because of the load reaching the capacity. This results in a lower utilization that can be observed.

The possible savings of CO_2 according to the intermodal tour algorithm can be observed in Tables 6.8 and 6.9 and Tables 6.11 and 6.12. It strikes out that the CO_2 savings of Karlsruhe's instance (~26%) are not comparable to Santa Ana's (~35%) although the number of demands are comparable. This can be ascribed to the different city layout. Karlsruhe does not have a grid layout as Santa Ana has which leads to longer tours in outer areas of the city. This effect is intensified by the fact that sprinter delivery is mostly performed outside of the city center because the city hubs are located there.

The results of Amsterdam are differing from the results of Karlsruhe and Santa Ana. The number of stops of cargo bikes is much higher because there are more demands within the range of a city hub. The demands in the southern and central part of Amsterdam are almost completely covered by the range of the city hubs. Thus, in the southern and central part of Amsterdam, there are almost only cargo bikes used. This leads to CO_2 savings of ~69% (cf. Tables 6.14 and 6.15).

Other than the Karlsruhe and Santa Ana instances, the Amsterdam instances have a lower tram capacity limit. For the first case (~10% edges), this has no effect because of the lower amount of demands. For the second case (~50% edges), this leads to 8 cargo bike tours that could not be performed because of the tram schedule. Accordingly, the demands of these tours have been served by sprinter vehicles. The 8 cargo bike tours are not included in the tour overview to avoid double accounting, but Table B.9 shows an overview of them.

Table 6.7: IMTA results for Karlsruhe and 10% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1 - 13	cargo bike	319	60%	126 km	6:04 h	16:42 h	0 kg
14 - 17	sprinter	525	32%	315 km	7:36 h	25:06 h	74 kg
1 - 17	all	844	39%	441 km	13:40 h	41:48 h	74 kg

Table 6.8: IMTA results for Karlsruhe and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1 - 43	cargo bike	1473	81%	354 km	17:05 h	66:11 h	0 kg
44 - 58	sprinter	3018	48%	891 km	21:32 h	122:08 h	210 kg
1 - 58	all	4491	55%	1 245 km	38:37 h	188:19	210 kg

Table 6.9: IMTA sprinter only results for Karlsruhe and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1 - 22	sprinter	4491	49%	1 199 km	28:57 h	178:39 h	282 kg

Table 6.10: IMTA results for Santa Ana and 10% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1 - 13	cargo bike	304	53%	139 km	6:44 h	16:52 h	0 kg
14 - 17	sprinter	568	33%	248 km	6:00 h	24:56 h	58 kg
1 - 17	all	872	38%	388 km	12:44 h	41:48 h	58 kg

Table 6.11: IMTA results for Santa Ana and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1 - 47	cargo bike	1664	84%	453 km	21:54 h	77:22 h	0 kg
48 - 61	sprinter	2833	48%	695 km	16:46 h	111:12 h	163 kg
1 - 61	all	4497	57%	1 148 km	38:40 h	188:34 h	163 kg

Table 6.12: IMTA sprinter only results for Santa Ana and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1 - 22	sprinter	4497	49%	1 058 km	25:33 h	175:27 h	249 kg

Table 6.13: IMTA results for Amsterdam and 10% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1 - 28	cargo bike	436	38%	237 km	11:27 h	25:59 h	0 kg
29 - 30	sprinter	288	35%	154 km	3:42 h	13:18 h	36 kg
1 - 30	all	724	37%	391 km	15:09 h	39:17 h	36 kg

Table 6.14: IMTA results for Amsterdam and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1 - 64	cargo bike	2029	77%	693 km	33:29 h	101:07 h	0 kg
65 - 73	sprinter	1657	44%	488 km	11:47 h	67:01 h	115 kg
1 - 73	all	3686	58%	1 181 km	45:16 h	168:08 h	115 kg

Table 6.15: IMTA sprinter only results for Amsterdam and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1 - 19	sprinter	3686	47%	1 078 km	26:02 h	148:54 h	253 kg

7 Conclusion

This thesis proposed an intermodal tour planning algorithm that integrates modern sustainable mechanisms for CEP service providers, such as transport by cargo bike and tram. The thesis introduced problem and solution models for intermodal tour planning. In order to solve the intermodal tour planning problem, it is decomposed and traced back to problem variants of the Vehicle Routing Problem and the Knapsack Problem. The algorithm uses Ant Colony Optimization with an adapted version of MACS-VRPTW[1] [8]. The changes therefore made to MACS-VRPTW have been explained in-depth. The variable tour generator makes the algorithm extensible. The thesis demonstrated using three examples that the algorithm can be run on real-world data and produces useful results.

[1] Multiple Ant Colony System for the Vehicle Routing Problem with Time Windows

8 Future Work

The proposed intermodal tour algorithm can be enhanced in several ways. Currently, the algorithm is split in three parts (cargo bike, tram, sprinter) which are merged with greedy choices. The Ant Colony approach would be suitable to merge the parts into one procedure that does all three parts together without greedy choices. Also, the initial solution generator could be enhanced by a more-sophisticated heuristic than the nearest neighbor heuristic. In the current approach, the cargo bike problem solver does not get information on the tram schedules. It is reasonable that it should know about the tram schedule in order to not generate tours that are not possible due to the tram schedule.

Despite the real-world data the algorithm has been tested on, most of the other input data has been arbitrarily chosen or synthetically/randomly generated. Therefore, real world impacts of e. g. demographic attributes have not been considered. Testing the algorithm on more sophisticated data would further reveal the practicality.

The proposed algorithm is an offline algorithm and was intended to be a such. However, the Ant Colony approach would allow to convert it to an online algorithm, where the algorithm is able to handle unexpected incidents after the tours have already started. This is particularly interesting for practical applications, e. g. of CEP service providers.

Bibliography

[1] L. Barthelmes, E. Görgülü, J. Kübler, M. Kagerbauer, and P. Vortisch. Modellbasierte Ermittlung von verkehrlichen Potentialen eines stadtbahnbasierten Gütertransports im Projekt LogIKTram in Karlsruhe. *Journal für Mobilität und Verkehr*, (16):50–58, Mar. 2023. ISSN 2628-4154. doi:10.34647/jmv.nr16.id103. Number: 16.

[2] J. Caceres-Cruz, P. Arias, D. Guimarans, D. Riera, and A. Juan. Rich Vehicle Routing Problem: Survey. *ACM Computing Surveys*, 47(2):32:1–32:28, 2014. ISSN 0360-0300. doi:10.1145/2666003.

[3] G. Clarke and J. Wright. Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, 12(4):568–581, Aug. 1964. ISSN 0030-364X, 1526-5463. doi:10.1287/opre.12.4.568.

[4] G. Dantzig and J. Ramser. The Truck Dispatching Problem: Management Science. *Management Science*, 6(1):80–91, Oct. 1959. ISSN 00251909. doi:10.1287/mnsc.6.1.80.

[5] M. Dawande, J. Kalagnanam, P. Keskinocak, F. Salman, and R. Ravi. Approximation Algorithms for the Multiple Knapsack Problem with Assignment Restrictions. *Journal of Combinatorial Optimization*, 4(2):171–186, June 2000. ISSN 1573-2886. doi:10.1023/A:1009894503716.

[6] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec. 1959. ISSN 0945-3245. doi:10.1007/BF01386390.

[7] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, Nov. 2006. ISSN 1556-6048. doi:10.1109/MCI.2006.329691. Conference Name: IEEE Computational Intelligence Magazine.

[8] L. Gambardella, E. Taillard, and G. Agazzi. MACS-VRPTW: a multiple ant colony system for vehicle routing problems with time windows. In *New ideas in optimiza-*

tion, pages 63–76. McGraw-Hill Ltd., UK, GBR, Jan. 1999. ISBN 978-0-07-709506-2. URL https://dl.acm.org/doi/10.5555/329055.329067.

[9] R. Geisberger. *Advanced route planning in transportation networks*. PhD thesis, Karlsruher Institut für Technologie, 2011. URL https://doi.org/10.5445/IR/1000021997.

[10] F. Glover. Tabu Search: A Tutorial | Interfaces. *INFORMS Journal on Applied Analytics*, 20(4):1–185, Aug. 1990. ISSN 2644-0873. doi:10.1287/inte.20.4.74.

[11] A. Jakhrani, A. Rigit, A. Othman, S. Samo, and S. Kamboh. Estimation of carbon footprints from diesel generator emissions. In *2012 International Conference on Green and Ubiquitous Technology*, pages 78–81, July 2012. doi:10.1109/GUT.2012.6344193.

[12] N. Jozefowiez, F. Semet, and E. Talbi. Multi-objective vehicle routing problems. *European Journal of Operational Research*, 189(2):293–309, Sept. 2008. ISSN 0377-2217. doi:10.1016/j.ejor.2007.05.055.

[13] G. Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345–358, June 1992. ISSN 0377-2217. doi:10.1016/0377-2217(92)90192-C.

[14] G. Laporte and Y. Nobert. Exact Algorithms for the Vehicle Routing Problem. In S. Martello, G. Laporte, M. Minoux, and C. Ribeiro, editors, *North-Holland Mathematics Studies*, volume 132 of *Surveys in Combinatorial Optimization*, pages 147–184. North-Holland, Jan. 1987. doi:10.1016/S0304-0208(08)73235-3.

[15] C. Llorca and R. Moeckel. Study of cargo bikes for parcel deliveries under different supply, demand and spatial conditions. In *2020 Forum on Integrated and Sustainable Transportation Systems (FISTS)*, pages 39–44, Nov. 2020. doi:10.1109/FISTS46898.2020.9264864.

[16] S. Martello and P. Toth. Algorithms for Knapsack Problems. In S. Martello, G. Laporte, M. Minoux, and C. Ribeiro, editors, *North-Holland Mathematics Studies*, volume 132 of *Surveys in Combinatorial Optimization*, pages 213–257. North-Holland, Jan. 1987. doi:10.1016/S0304-0208(08)73237-7.

[17] M. Ratnagiri, C. O'Dwyer, E. Beaver, H. Bang, B. Chalaki, and A. Malikopoulos. A Scalable Last-Mile Delivery Service: From Simulation to Scaled Experiment. In

2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC), pages 4163–4168, Oct. 2022. doi:10.1109/ITSC55140.2022.9921797.

[18] S. Schröder and GraphHopper. Jsprit, Apr. 2020. URL `https://github.com/graphhopper/jsprit`.

[19] T. Stutzle and M. Dorigo. A short convergence proof for a class of ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation*, 6(4): 358–365, Aug. 2002. ISSN 1941-0026. doi:10.1109/TEVC.2002.802444. Conference Name: IEEE Transactions on Evolutionary Computation.

[20] E. Talbi. *Metaheuristics: from design to implementation*. John Wiley & Sons, Hoboken, N.J, June 2009. ISBN 978-0-470-49691-6. doi:10.1002/9780470496916.

[21] N. Wassan and G. Nagy. Vehicle routing problem with deliveries and pick-ups: Modelling issues and meta-heuristics solution approaches. Apr. 2014. doi:10.14257/IJT.2014.2.1.06.

[22] C. Zheng, Y. Gu, J. Shen, and M. Du. Urban Logistics Delivery Route Planning Based on a Single Metro Line. *IEEE Access*, 9:50819–50830, 2021. ISSN 2169-3536. doi:10.1109/ACCESS.2021.3069415. Conference Name: IEEE Access.

A Detailed City Hub Data

Table A.1: City Hubs of the Karlsruhe instance

#	Name	Latitude	Longitude
1	Europaplatz	49.010072	8.393473
2	Yorckstraße	49.010872	8.374780
3	Durlacher Tor/KIT	49.008900	8.417167
4	Entenfang	49.010320	8.358662
5	Siemensallee	49.026844	8.351491
6	Kurt-Schumacher-Straße	49.029900	8.374894
7	Neureut-Heide	49.036032	8.387882
8	Daxlanden Karl-Delisle-Straße	48.996268	8.338646
9	Rüppurr Tulpenstraße	48.970952	8.403928
10	Wolfartsweier Nord	48.978280	8.458247
11	Durlach Friedrichsschule	48.999328	8.468429
12	Rintheim Forststraße	49.013992	8.442191
13	Waldstadt Zentrum	49.034920	8.445386

Table A.2: City Hubs of the Santa Ana instance

#	Name	Latitude	Longitude
1	Bristol-17th	33.759972	-117.885072
2	Main-17th	33.760088	-117.867568
3	17th-Concord	33.759928	-117.845856
4	1st-Newhope	33.745116	-117.928792
5	1st-Fairview	33.745264	-117.905192
6	McFadden-Hathaway	33.733636	-117.853104
7	Edinger-Fairview	33.727336	-117.90624
8	Edinger-Bristol	33.727176	-117.88508
9	Main-Saint Gertrude	33.71972	-117.86788
10	Warner-Raitt	33.7155	-117.896088
11	Fairview-Segerstrom	33.7088	-117.907696
12	Main-Dyer	33.708776	-117.868032
13	Bristol-Alton	33.703692	-117.885456

Table A.3: City Hubs of the Amsterdam instance

#	Name	Latitude	Longitude
1	Meester Visserplein	52.3677535	4.9058474
2	Nassaukade	52.3810685	4.8791525
3	Marnixplein	52.3778393	4.8786695
4	Prinses Irenestraat	52.3431074	4.8767749
5	Muziekgebouw	52.3772761	4.912547
6	Gerrit van der Veenstraat	52.3492157	4.877365
7	Stadhouderskade	52.3576504	4.8993709
8	Maasstraat	52.3468163	4.8947303
9	Dintelstraat	52.341887	4.893324
10	Van Woustraat	52.3549659	4.9015662
11	Hoogte Kadijk	52.3666491	4.9242477
12	Prinsengracht	52.3621338	4.8989773
13	Bos en Lommerweg	52.3811265	4.8539231
14	Willem de Zwijgerlaan	52.3704221	4.8641586
15	Hugo de Vrieslaan	52.351066	4.934801
16	Oostpoort	52.3574304	4.9268761
17	Wijttenbachstraat	52.3602514	4.9250583
18	Nicolaas Beetsstraat	52.365525	4.8656566
19	Elandsgracht	52.368541	4.8768971
20	Amstelveenseweg	52.3517531	4.8564472
21	Kattenburgerstraat	52.3761672	4.9211099
22	Victorieplein	52.3464278	4.9063478
23	Eerste Constantijn Huygensstraat	52.3626351	4.874895
24	Rokin	52.3697221	4.892274
25	Koningsplein	52.3677188	4.8893427
26	Javaplein	52.3641725	4.9382411
27	De Pijp	52.3526769	4.8901771
28	Van Baerlestraat	52.3587793	4.8781266
29	Erasmusgracht	52.3763312	4.8469734

B Detailed Intermodal Tour Algorithm Results

Table B.1: Detailed IMTA results for Karlsruhe and 10% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	cargo bike	9	22%	7 214 m	0:20 h	0:38 h	0 g
2	cargo bike	42	100%	13 376 m	0:38 h	2:02 h	0 g
3	cargo bike	15	36%	7 041 m	0:20 h	0:50 h	0 g
4	cargo bike	30	75%	10 166 m	0:29 h	1:29 h	0 g
5	cargo bike	25	65%	8 194 m	0:23 h	1:13 h	0 g
6	cargo bike	42	90%	12 777 m	0:37 h	2:01 h	0 g
7	cargo bike	20	48%	7 832 m	0:22 h	1:02 h	0 g
8	cargo bike	30	65%	12 163 m	0:35 h	1:35 h	0 g
9	cargo bike	20	52%	7 402 m	0:21 h	1:01 h	0 g
10	cargo bike	12	34%	7 053 m	0:20 h	0:44 h	0 g
11	cargo bike	17	47%	9 110 m	0:26 h	1:00 h	0 g
12	cargo bike	27	71%	12 376 m	0:35 h	1:29 h	0 g
13	cargo bike	30	71%	11 003 m	0:31 h	1:31 h	0 g
14	sprinter	174	41%	90 325 m	2:10 h	7:58 h	21 226 g
15	sprinter	170	41%	101 097 m	2:26 h	8:06 h	23 758 g
16	sprinter	164	39%	104 954 m	2:32 h	8:00 h	24 664 g
17	sprinter	17	5%	18 598 m	0:26 h	1:00 h	4 371 g
1 - 13	**cargo bike**	**319**	**60%**	**126 m**	**6:04 h**	**16:42 h**	**0 g**
14 - 17	**sprinter**	**525**	**32%**	**315 m**	**7:36 h**	**25:06 h**	**74 kg**
1 - 17	**all**	**844**	**39%**	**441 km**	**13:40 h**	**41:48 h**	**74 kg**

Table B.2: Detailed IMTA results for Karlsruhe and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	cargo bike	23	59%	4 762 m	0:13 h	0:59 h	0 g
2	cargo bike	46	100%	8 012 m	0:23 h	1:55 h	0 g
3	cargo bike	44	97%	8 501 m	0:24 h	1:52 h	0 g
4	cargo bike	47	100%	7 925 m	0:22 h	1:56 h	0 g
5	cargo bike	40	100%	6 091 m	0:17 h	1:37 h	0 g
6	cargo bike	40	100%	6 435 m	0:18 h	1:38 h	0 g
7	cargo bike	45	100%	6 988 m	0:20 h	1:50 h	0 g
8	cargo bike	43	96%	11 454 m	0:33 h	1:59 h	0 g
9	cargo bike	36	91%	7 550 m	0:21 h	1:33 h	0 g
10	cargo bike	47	93%	6 782 m	0:19 h	1:53 h	0 g
11	cargo bike	44	95%	6 876 m	0:19 h	1:47 h	0 g
12	cargo bike	46	100%	9 427 m	0:27 h	1:59 h	0 g
13	cargo bike	29	70%	5 312 m	0:15 h	1:13 h	0 g
14	cargo bike	42	100%	10 861 m	0:31 h	1:55 h	0 g
15	cargo bike	39	93%	5 794 m	0:16 h	1:34 h	0 g
16	cargo bike	40	100%	10 151 m	0:29 h	1:49 h	0 g
17	cargo bike	29	64%	7 484 m	0:21 h	1:19 h	0 g
18	cargo bike	18	49%	6 487 m	0:18 h	0:54 h	0 g
19	cargo bike	37	100%	9 161 m	0:26 h	1:40 h	0 g
20	cargo bike	36	73%	4 901 m	0:14 h	1:26 h	0 g
21	cargo bike	47	100%	6 514 m	0:18 h	1:52 h	0 g
22	cargo bike	38	100%	7 931 m	0:22 h	1:38 h	0 g
23	cargo bike	41	100%	9 550 m	0:27 h	1:49 h	0 g
24	cargo bike	2	5%	5 530 m	0:16 h	0:20 h	0 g
25	cargo bike	9	29%	5 067 m	0:14 h	0:32 h	0 g
26	cargo bike	35	88%	6 422 m	0:18 h	1:28 h	0 g
27	cargo bike	32	75%	8 230 m	0:23 h	1:27 h	0 g
28	cargo bike	38	94%	5 996 m	0:17 h	1:33 h	0 g
29	cargo bike	26	68%	10 541 m	0:30 h	1:22 h	0 g
30	cargo bike	22	51%	7 077 m	0:20 h	1:04 h	0 g
31	cargo bike	42	99%	7 694 m	0:22 h	1:46 h	0 g
32	cargo bike	41	100%	9 623 m	0:27 h	1:49 h	0 g
33	cargo bike	38	100%	7 442 m	0:21 h	1:37 h	0 g
34	cargo bike	38	94%	17 097 m	0:49 h	2:05 h	0 g
35	cargo bike	8	19%	5 820 m	0:16 h	0:32 h	0 g
36	cargo bike	34	90%	11 532 m	0:33 h	1:41 h	0 g
37	cargo bike	16	37%	5 350 m	0:15 h	0:47 h	0 g
38	cargo bike	31	70%	11 229 m	0:32 h	1:34 h	0 g
39	cargo bike	42	100%	11 186 m	0:32 h	1:56 h	0 g
40	cargo bike	32	73%	12 673 m	0:36 h	1:40 h	0 g
41	cargo bike	41	100%	13 988 m	0:40 h	2:02 h	0 g
42	cargo bike	22	54%	4 972 m	0:14 h	0:58 h	0 g
43	cargo bike	27	74%	11 304 m	0:32 h	1:26 h	0 g
44	sprinter	213	49%	38 856 m	0:56 h	8:02 h	9 131 g
45	sprinter	210	51%	41 913 m	1:00 h	8:00 h	9 850 g
46	sprinter	205	50%	54 679 m	1:19 h	8:09 h	12 850 g
47	sprinter	202	48%	59 710 m	1:26 h	8:10 h	14 032 g
48	sprinter	209	51%	50 623 m	1:13 h	8:11 h	11 896 g
49	sprinter	202	48%	56 204 m	1:21 h	8:05 h	13 208 g
50	sprinter	208	50%	49 877 m	1:12 h	8:08 h	11 721 g
51	sprinter	199	49%	64 067 m	1:32 h	8:10 h	15 056 g
52	sprinter	208	49%	52 436 m	1:15 h	8:11 h	12 322 g
53	sprinter	195	48%	70 225 m	1:41 h	8:11 h	16 503 g
54	sprinter	196	48%	67 404 m	1:37 h	8:09 h	15 840 g
55	sprinter	181	42%	87 709 m	2:07 h	8:09 h	20 612 g
56	sprinter	189	46%	70 911 m	1:42 h	8:00 h	16 664 g
57	sprinter	204	49%	57 719 m	1:23 h	8:11 h	13 564 g
58	sprinter	197	48%	69 182 m	1:40 h	8:14 h	16 258 g
1 - 43	**cargo bike**	**1473**	**81%**	**354 km**	**17:05 h**	**66:11 h**	**0 g**
44 - 58	**sprinter**	**3018**	**48%**	**891 km**	**21:32 h**	**122:08 h**	**210 kg**
1 - 58	**all**	**4491**	**55%**	**1 245 km**	**38:37 h**	**188:19**	**210 kg**

Table B.3: Detailed IMTA sprinter only results for Karlsruhe and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	sprinter	214	52%	40 548 m	0:58 h	8:06 h	9 529 g
2	sprinter	215	50%	37 116 m	0:53 h	8:03 h	8 722 g
3	sprinter	212	52%	40 715 m	0:59 h	8:03 h	9 568 g
4	sprinter	215	52%	38 843 m	0:56 h	8:06 h	9 128 g
5	sprinter	203	51%	56 316 m	1:21 h	8:07 h	13 234 g
6	sprinter	203	49%	56 825 m	1:22 h	8:08 h	13 354 g
7	sprinter	204	48%	51 522 m	1:14 h	8:02 h	12 108 g
8	sprinter	208	49%	50 105 m	1:12 h	8:08 h	11 775 g
9	sprinter	208	50%	50 892 m	1:13 h	8:09 h	11 960 g
10	sprinter	212	52%	45 243 m	1:05 h	8:09 h	10 632 g
11	sprinter	214	53%	41 090 m	0:59 h	8:07 h	9 656 g
12	sprinter	211	51%	47 213 m	1:08 h	8:10 h	11 095 g
13	sprinter	200	45%	57 350 m	1:23 h	8:03 h	13 477 g
14	sprinter	205	48%	54 834 m	1:19 h	8:09 h	12 886 g
15	sprinter	205	49%	52 846 m	1:16 h	8:06 h	12 419 g
16	sprinter	202	50%	59 974 m	1:26 h	8:10 h	14 094 g
17	sprinter	195	46%	67 586 m	1:37 h	8:07 h	15 883 g
18	sprinter	189	41%	76 747 m	1:51 h	8:09 h	18 036 g
19	sprinter	202	50%	59 093 m	1:25 h	8:09 h	13 887 g
20	sprinter	205	49%	56 199 m	1:21 h	8:11 h	13 207 g
21	sprinter	177	42%	84 814 m	2:02 h	7:56 h	19 931 g
22	sprinter	192	47%	73 162 m	1:46 h	8:10 h	17 193 g
1 - 22	**sprinter**	**4491**	**49%**	**1 199 km**	**28:57 h**	**178:39 h**	**282 kg**

Table B.4: Detailed IMTA results for Santa Ana and 10% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	cargo bike	37	93%	15 332 m	0:44 h	1:58 h	0 g
2	cargo bike	41	96%	15 454 m	0:44 h	2:06 h	0 g
3	cargo bike	29	59%	11 569 m	0:33 h	1:31 h	0 g
4	cargo bike	19	39%	10 697 m	0:31 h	1:09 h	0 g
5	cargo bike	37	85%	13 106 m	0:37 h	1:51 h	0 g
6	cargo bike	19	41%	11 265 m	0:32 h	1:10 h	0 g
7	cargo bike	9	21%	5 590 m	0:16 h	0:34 h	0 g
8	cargo bike	22	48%	9 220 m	0:26 h	1:10 h	0 g
9	cargo bike	9	25%	3 989 m	0:11 h	0:29 h	0 g
10	cargo bike	13	31%	9 455 m	0:27 h	0:53 h	0 g
11	cargo bike	29	62%	11 200 m	0:32 h	1:30 h	0 g
12	cargo bike	19	44%	12 051 m	0:34 h	1:12 h	0 g
13	cargo bike	21	48%	10 512 m	0:30 h	1:12 h	0 g
14	sprinter	181	43%	81 565 m	1:58 h	8:00 h	19 168 g
15	sprinter	185	45%	77 396 m	1:52 h	8:02 h	18 188 g
16	sprinter	182	40%	79 582 m	1:55 h	7:59 h	18 702 g
17	sprinter	20	4%	9 888 m	0:14 h	0:54 h	2 324 g
1 - 13	**cargo bike**	**304**	**53%**	**139 km**	**6:44 h**	**16:52 h**	**0 g**
14 - 17	**sprinter**	**568**	**33%**	**248 km**	**6:00 h**	**24:56 h**	**58 kg**
1 - 17	**all**	**872**	**38%**	**388 km**	**12:44 h**	**41:48 h**	**58 kg**

Table B.5: Detailed IMTA results for Santa Ana and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	cargo bike	38	100%	7 957 m	0:23 h	1:39 h	0 g
2	cargo bike	46	100%	10 645 m	0:30 h	2:02 h	0 g
3	cargo bike	41	99%	8 666 m	0:25 h	1:47 h	0 g
4	cargo bike	44	100%	5 851 m	0:16 h	1:44 h	0 g
5	cargo bike	43	100%	11 179 m	0:32 h	1:58 h	0 g
6	cargo bike	38	100%	8 317 m	0:24 h	1:40 h	0 g
7	cargo bike	40	97%	7 162 m	0:20 h	1:40 h	0 g
8	cargo bike	48	100%	15 097 m	0:43 h	2:19 h	0 g
9	cargo bike	48	98%	10 458 m	0:30 h	2:06 h	0 g
10	cargo bike	39	98%	8 835 m	0:25 h	1:43 h	0 g
11	cargo bike	42	100%	9 257 m	0:26 h	1:50 h	0 g
12	cargo bike	38	83%	8 380 m	0:24 h	1:40 h	0 g
13	cargo bike	40	100%	7 122 m	0:20 h	1:40 h	0 g
14	cargo bike	43	100%	11 777 m	0:34 h	2:00 h	0 g
15	cargo bike	43	100%	12 989 m	0:37 h	2:03 h	0 g
16	cargo bike	45	100%	6 805 m	0:19 h	1:49 h	0 g
17	cargo bike	43	100%	17 209 m	0:49 h	2:15 h	0 g
18	cargo bike	43	100%	11 492 m	0:33 h	1:59 h	0 g
19	cargo bike	40	90%	7 731 m	0:22 h	1:42 h	0 g
20	cargo bike	41	100%	9 523 m	0:27 h	1:49 h	0 g
21	cargo bike	12	34%	8 465 m	0:24 h	0:48 h	0 g
22	cargo bike	45	100%	15 842 m	0:45 h	2:15 h	0 g
23	cargo bike	46	100%	12 238 m	0:35 h	2:07 h	0 g
24	cargo bike	31	74%	8 936 m	0:25 h	1:27 h	0 g
25	cargo bike	35	81%	5 474 m	0:15 h	1:25 h	0 g
26	cargo bike	26	61%	5 838 m	0:16 h	1:08 h	0 g
27	cargo bike	41	100%	7 874 m	0:22 h	1:44 h	0 g
28	cargo bike	36	100%	9 362 m	0:27 h	1:39 h	0 g
29	cargo bike	9	23%	5 396 m	0:15 h	0:33 h	0 g
30	cargo bike	42	100%	7 765 m	0:22 h	1:46 h	0 g
31	cargo bike	38	98%	13 459 m	0:39 h	1:55 h	0 g
32	cargo bike	43	100%	11 787 m	0:34 h	2:00 h	0 g
33	cargo bike	7	21%	5 112 m	0:14 h	0:28 h	0 g
34	cargo bike	39	100%	14 974 m	0:43 h	2:01 h	0 g
35	cargo bike	50	100%	13 259 m	0:38 h	2:18 h	0 g
36	cargo bike	37	97%	12 452 m	0:36 h	1:50 h	0 g
37	cargo bike	1	4%	1 327 m	0:03 h	0:05 h	0 g
38	cargo bike	33	78%	10 900 m	0:31 h	1:37 h	0 g
39	cargo bike	19	47%	8 237 m	0:23 h	1:01 h	0 g
40	cargo bike	43	100%	7 800 m	0:22 h	1:48 h	0 g
41	cargo bike	40	100%	13 334 m	0:38 h	1:58 h	0 g
42	cargo bike	45	100%	15 321 m	0:44 h	2:14 h	0 g
43	cargo bike	27	63%	13 338 m	0:38 h	1:32 h	0 g
44	cargo bike	27	64%	9 854 m	0:28 h	1:22 h	0 g
45	cargo bike	3	11%	3 098 m	0:08 h	0:14 h	0 g
46	cargo bike	13	29%	6 363 m	0:18 h	0:44 h	0 g
47	cargo bike	33	81%	9 219 m	0:26 h	1:32 h	0 g
48	sprinter	219	52%	31 843 m	0:46 h	8:04 h	7 483 g
49	sprinter	214	52%	37 813 m	0:54 h	8:02 h	8 886 g
50	sprinter	212	47%	41 114 m	0:59 h	8:03 h	9 662 g
51	sprinter	216	54%	34 945 m	0:50 h	8:02 h	8 212 g
52	sprinter	203	50%	53 905 m	1:18 h	8:04 h	12 668 g
53	sprinter	210	49%	43 066 m	1:02 h	8:02 h	10 121 g
54	sprinter	204	48%	51 032 m	1:13 h	8:01 h	11 993 g
55	sprinter	210	50%	43 473 m	1:03 h	8:03 h	10 216 g
56	sprinter	202	46%	58 257 m	1:24 h	8:08 h	13 690 g
57	sprinter	194	44%	67 862 m	1:38 h	8:06 h	15 948 g
58	sprinter	204	49%	53 599 m	1:17 h	8:05 h	12 596 g
59	sprinter	204	49%	52 057 m	1:15 h	8:03 h	12 233 g
60	sprinter	201	47%	59 045 m	1:25 h	8:07 h	13 876 g
61	sprinter	140	38%	66 688 m	1:36 h	6:16 h	15 672 g
1 - 47	**cargo bike**	**1664**	**84%**	**453 m**	**21:54 h**	**77:22 h**	**0 g**
48 - 61	**sprinter**	**2833**	**48%**	**695 m**	**16:46 h**	**111:12 h**	**163 kg**
1 - 61	**all**	**4497**	**57%**	**1 148 km**	**38:40 h**	**188:34 h**	**163 kg**

Table B.6: Detailed IMTA sprinter only results for Santa Ana and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	sprinter	213	51%	41 898 m	1:00 h	8:06 h	9 846 g
2	sprinter	212	48%	42 447 m	1:01 h	8:05 h	9 975 g
3	sprinter	214	49%	38 559 m	0:55 h	8:03 h	9 061 g
4	sprinter	214	51%	36 385 m	0:52 h	8:00 h	8 550 g
5	sprinter	211	49%	43 425 m	1:02 h	8:04 h	10 205 g
6	sprinter	209	51%	46 784 m	1:07 h	8:05 h	10 994 g
7	sprinter	218	54%	32 259 m	0:46 h	8:02 h	7 581 g
8	sprinter	217	52%	33 440 m	0:48 h	8:02 h	7 858 g
9	sprinter	208	50%	48 390 m	1:10 h	8:06 h	11 372 g
10	sprinter	213	51%	39 743 m	0:57 h	8:03 h	9 340 g
11	sprinter	206	46%	51 873 m	1:15 h	8:07 h	12 190 g
12	sprinter	212	51%	41 741 m	1:00 h	8:04 h	9 809 g
13	sprinter	202	47%	56 768 m	1:22 h	8:06 h	13 340 g
14	sprinter	212	50%	44 344 m	1:04 h	8:08 h	10 421 g
15	sprinter	205	51%	50 350 m	1:12 h	8:02 h	11 832 g
16	sprinter	204	51%	53 115 m	1:16 h	8:04 h	12 482 g
17	sprinter	206	49%	51 748 m	1:14 h	8:06 h	12 161 g
18	sprinter	206	48%	51 997 m	1:15 h	8:07 h	12 219 g
19	sprinter	185	44%	82 815 m	2:00 h	8:10 h	19 462 g
20	sprinter	205	48%	51 938 m	1:15 h	8:05 h	12 205 g
21	sprinter	191	46%	72 986 m	1:45 h	8:07 h	17 152 g
22	sprinter	134	30%	45 273 m	1:05 h	5:33 h	10 639 g
1 - 22	**sprinter**	**4497**	**49%**	**1 058 km**	**25:33 h**	**175:27 h**	**249 kg**

Table B.7: Detailed IMTA results for Amsterdam and 10% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	cargo bike	8	13%	6 663 m	0:19 h	0:35 h	0 g
2	cargo bike	37	90%	14 770 m	0:42 h	1:56 h	0 g
3	cargo bike	20	50%	11 631 m	0:33 h	1:13 h	0 g
4	cargo bike	11	29%	5 126 m	0:14 h	0:36 h	0 g
5	cargo bike	21	54%	7 305 m	0:21 h	1:03 h	0 g
6	cargo bike	10	28%	8 429 m	0:24 h	0:44 h	0 g
7	cargo bike	4	13%	3 572 m	0:10 h	0:18 h	0 g
8	cargo bike	11	24%	5 460 m	0:15 h	0:37 h	0 g
9	cargo bike	8	20%	3 750 m	0:10 h	0:26 h	0 g
10	cargo bike	11	20%	7 960 m	0:23 h	0:45 h	0 g
11	cargo bike	5	15%	3 371 m	0:09 h	0:19 h	0 g
12	cargo bike	21	55%	13 954 m	0:40 h	1:22 h	0 g
13	cargo bike	20	47%	10 706 m	0:31 h	1:11 h	0 g
14	cargo bike	38	89%	16 387 m	0:47 h	2:03 h	0 g
15	cargo bike	22	53%	8 497 m	0:24 h	1:08 h	0 g
16	cargo bike	6	15%	5 656 m	0:16 h	0:28 h	0 g
17	cargo bike	2	6%	1 685 m	0:04 h	0:08 h	0 g
18	cargo bike	10	21%	6 763 m	0:19 h	0:39 h	0 g
19	cargo bike	38	98%	20 883 m	1:00 h	2:16 h	0 g
20	cargo bike	9	23%	6 262 m	0:18 h	0:36 h	0 g
21	cargo bike	20	45%	13 719 m	0:39 h	1:19 h	0 g
22	cargo bike	15	41%	8 187 m	0:23 h	0:53 h	0 g
23	cargo bike	3	3%	4 599 m	0:13 h	0:19 h	0 g
24	cargo bike	3	4%	2 186 m	0:06 h	0:12 h	0 g
25	cargo bike	37	87%	15 033 m	0:43 h	1:57 h	0 g
26	cargo bike	13	33%	6 119 m	0:17 h	0:43 h	0 g
27	cargo bike	17	48%	9 221 m	0:26 h	1:00 h	0 g
28	cargo bike	16	38%	9 229 m	0:26 h	0:58 h	0 g
29	sprinter	169	41%	102 577 m	2:28 h	8:06 h	24 106 g
30	sprinter	119	28%	51 084 m	1:14 h	5:12 h	12 005 g
1 - 28	**cargo bike**	**436**	**38%**	**237 m**	**11:27 h**	**25:59 h**	**0 g**
29 - 30	**sprinter**	**288**	**35%**	**154 m**	**3:42 h**	**13:18 h**	**36 kg**
1 - 30	**all**	**724**	**37%**	**391 km**	**15:09 h**	**39:17 h**	**36 kg**

Table B.8: Detailed IMTA results for Amsterdam and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	cargo bike	46	99%	20 152 m	0:58 h	2:30 h	0 g
2	cargo bike	6	21%	4 132 m	0:11 h	0:23 h	0 g
3	cargo bike	40	96%	8 161 m	0:23 h	1:43 h	0 g
4	cargo bike	40	100%	8 042 m	0:23 h	1:43 h	0 g
5	cargo bike	17	49%	8 372 m	0:24 h	0:58 h	0 g
6	cargo bike	45	100%	16 528 m	0:47 h	2:17 h	0 g
7	cargo bike	41	99%	17 491 m	0:50 h	2:12 h	0 g
8	cargo bike	2	4%	3 249 m	0:09 h	0:13 h	0 g
9	cargo bike	46	100%	10 592 m	0:30 h	2:02 h	0 g
10	cargo bike	19	51%	7 138 m	0:20 h	0:58 h	0 g
11	cargo bike	5	12%	3 224 m	0:09 h	0:19 h	0 g
12	cargo bike	43	100%	9 168 m	0:26 h	1:52 h	0 g
13	cargo bike	38	100%	8 894 m	0:25 h	1:41 h	0 g
14	cargo bike	9	27%	3 342 m	0:09 h	0:27 h	0 g
15	cargo bike	39	100%	15 153 m	0:43 h	2:01 h	0 g
16	cargo bike	7	20%	3 741 m	0:10 h	0:24 h	0 g
17	cargo bike	40	100%	12 823 m	0:37 h	1:57 h	0 g
18	cargo bike	4	12%	3 813 m	0:11 h	0:19 h	0 g
19	cargo bike	40	95%	15 479 m	0:44 h	2:04 h	0 g
20	cargo bike	11	32%	6 104 m	0:17 h	0:39 h	0 g
21	cargo bike	37	87%	11 157 m	0:32 h	1:46 h	0 g
22	cargo bike	46	100%	11 666 m	0:33 h	2:05 h	0 g
23	cargo bike	35	89%	11 303 m	0:32 h	1:42 h	0 g
24	cargo bike	38	100%	10 959 m	0:31 h	1:47 h	0 g
25	cargo bike	5	14%	4 140 m	0:11 h	0:22 h	0 g
26	cargo bike	43	100%	10 996 m	0:31 h	1:57 h	0 g
27	cargo bike	39	100%	10 787 m	0:31 h	1:49 h	0 g
28	cargo bike	31	77%	13 427 m	0:38 h	1:40 h	0 g
29	cargo bike	41	100%	8 659 m	0:25 h	1:47 h	0 g
30	cargo bike	39	100%	13 641 m	0:39 h	1:57 h	0 g
31	cargo bike	40	100%	13 587 m	0:39 h	1:59 h	0 g
32	cargo bike	44	100%	10 844 m	0:31 h	1:59 h	0 g
33	cargo bike	41	100%	7 920 m	0:22 h	1:44 h	0 g
34	cargo bike	43	100%	11 718 m	0:33 h	1:59 h	0 g
35	cargo bike	41	100%	10 085 m	0:29 h	1:51 h	0 g
36	cargo bike	38	100%	8 316 m	0:24 h	1:40 h	0 g
37	cargo bike	33	84%	14 680 m	0:42 h	1:48 h	0 g
38	cargo bike	39	100%	14 744 m	0:42 h	2:00 h	0 g
39	cargo bike	3	9%	2 578 m	0:07 h	0:13 h	0 g
40	cargo bike	13	27%	5 329 m	0:15 h	0:41 h	0 g
41	cargo bike	42	97%	15 763 m	0:45 h	2:09 h	0 g
42	cargo bike	19	45%	11 671 m	0:33 h	1:11 h	0 g
43	cargo bike	46	100%	19 496 m	0:56 h	2:28 h	0 g
44	cargo bike	40	100%	13 671 m	0:39 h	1:59 h	0 g
45	cargo bike	40	100%	13 480 m	0:39 h	1:59 h	0 g
46	cargo bike	42	100%	14 080 m	0:40 h	2:04 h	0 g
47	cargo bike	4	12%	4 102 m	0:11 h	0:19 h	0 g
48	cargo bike	41	92%	11 688 m	0:33 h	1:55 h	0 g
49	cargo bike	40	100%	15 256 m	0:44 h	2:04 h	0 g
50	cargo bike	41	100%	14 003 m	0:40 h	2:02 h	0 g
51	cargo bike	45	98%	12 975 m	0:37 h	2:07 h	0 g
52	cargo bike	35	100%	14 485 m	0:41 h	1:51 h	0 g
53	cargo bike	15	42%	5 322 m	0:15 h	0:45 h	0 g
54	cargo bike	16	39%	10 414 m	0:30 h	1:02 h	0 g
55	cargo bike	14	32%	9 686 m	0:28 h	0:56 h	0 g
56	cargo bike	52	100%	17 463 m	0:50 h	2:34 h	0 g
57	cargo bike	39	100%	10 616 m	0:30 h	1:48 h	0 g
58	cargo bike	39	100%	14 716 m	0:42 h	2:00 h	0 g
59	cargo bike	42	100%	15 712 m	0:45 h	2:09 h	0 g
60	cargo bike	18	46%	10 234 m	0:29 h	1:05 h	0 g
61	cargo bike	43	94%	11 926 m	0:34 h	2:00 h	0 g
62	cargo bike	29	63%	9 699 m	0:28 h	1:26 h	0 g
63	cargo bike	40	88%	15 667 m	0:45 h	2:05 h	0 g
64	cargo bike	20	54%	8 863 m	0:25 h	1:05 h	0 g
65	sprinter	200	49%	56 941 m	1:22 h	8:02 h	13 381 g
66	sprinter	212	49%	43 454 m	1:02 h	8:06 h	10 212 g
67	sprinter	180	46%	86 214 m	2:04 h	8:04 h	20 260 g
68	sprinter	199	48%	62 889 m	1:31 h	8:09 h	14 779 g
69	sprinter	207	50%	51 004 m	1:13 h	8:07 h	11 986 g
70	sprinter	208	49%	49 070 m	1:11 h	8:07 h	11 531 g
71	sprinter	207	49%	51 438 m	1:14 h	8:08 h	12 088 g
72	sprinter	196	49%	66 047 m	1:35 h	8:07 h	15 521 g
73	sprinter	48	10%	21 308 m	0:30 h	2:06 h	5 007 g
1 - 64	**cargo bike**	**2029**	**77%**	**693 km**	**33:29 h**	**101:07 h**	**0 g**
65 - 73	**sprinter**	**1657**	**44%**	**488 km**	**11:47 h**	**67:01 h**	**115 kg**
1 - 73	**all**	**3686**	**58%**	**1 181 km**	**45:16 h**	**168:08 h**	**115 kg**

Table B.9: Tours with failed tram delivery (Detailed IMTA results for Amsterdam and 50% edges)

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	cargo bike	3	10%	3 642 m	0:10 h	0:16 h	0 g
2	cargo bike	36	100%	9 203 m	0:26 h	1:38 h	0 g
3	cargo bike	22	54%	9 790 m	0:28 h	1:12 h	0 g
4	cargo bike	40	100%	7 025 m	0:20 h	1:40 h	0 g
5	cargo bike	37	100%	11 569 m	0:33 h	1:47 h	0 g
6	cargo bike	9	22%	8 702 m	0:25 h	0:43 h	0 g
7	cargo bike	17	44%	5 022 m	0:14 h	0:48 h	0 g
8	cargo bike	4	13%	6 386 m	0:18 h	0:26 h	0 g
1 - 8	**cargo bike**	**168**	**55%**	**61 339 m**	**2:57 h**	**8:33 h**	**0 g**

Table B.10: Detailed IMTA sprinter only results for Amsterdam and 50% edges

tour #	vehicle	stops	utilization	length	driving time	working time	CO_2
1	sprinter	209	52%	43 800 m	1:03 h	8:01 h	10 293 g
2	sprinter	214	51%	39 441 m	0:57 h	8:05 h	9 269 g
3	sprinter	194	47%	64 792 m	1:33 h	8:01 h	15 226 g
4	sprinter	201	47%	55 492 m	1:20 h	8:02 h	13 041 g
5	sprinter	195	49%	64 233 m	1:33 h	8:03 h	15 095 g
6	sprinter	210	47%	46 863 m	1:07 h	8:07 h	11 013 g
7	sprinter	205	53%	52 959 m	1:16 h	8:06 h	12 445 g
8	sprinter	211	51%	45 118 m	1:05 h	8:07 h	10 603 g
9	sprinter	208	49%	47 414 m	1:08 h	8:04 h	11 142 g
10	sprinter	203	46%	56 003 m	1:21 h	8:07 h	13 161 g
11	sprinter	199	49%	62 613 m	1:30 h	8:08 h	14 714 g
12	sprinter	202	48%	58 838 m	1:25 h	8:09 h	13 827 g
13	sprinter	200	48%	58 719 m	1:25 h	8:05 h	13 799 g
14	sprinter	196	47%	62 423 m	1:30 h	8:02 h	14 669 g
15	sprinter	190	48%	75 255 m	1:49 h	8:09 h	17 685 g
16	sprinter	187	44%	79 354 m	1:55 h	8:09 h	18 648 g
17	sprinter	198	47%	65 801 m	1:35 h	8:11 h	15 463 g
18	sprinter	204	50%	58 064 m	1:24 h	8:12 h	13 645 g
19	sprinter	60	14%	40 940 m	0:59 h	2:59 h	9 621 g
1 - 19	**sprinter**	**3686**	**47%**	**1 078 km**	**26:02 h**	**148:54 h**	**253 kg**