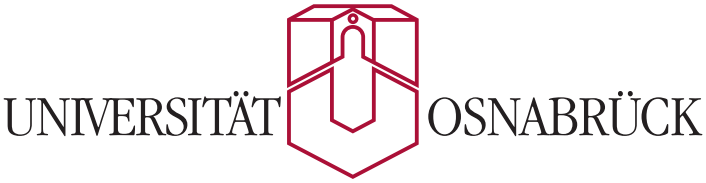


Efficient heuristics for large-scale vehicle routing problems

Dissertation
zur Erlangung des Doktorgrades (Dr. rer. nat.)
des Fachbereichs Mathematik / Informatik
der Universität Osnabrück

vorgelegt von

Benjamin Graf, M.Sc.

1. März 2021

Gutachter:
Prof. Dr. Sigrid Knust, Universität Osnabrück, Deutschland
Prof. Dr. Stefan Irnich, Johannes Gutenberg-Universität Mainz, Deutschland

Mitglieder der Promotionskommission:
Prof. Dr. Sigrid Knust
Prof. Dr. Stefan Irnich
Prof. Dr. Markus Chimani
Dr. Fritz Bökler

Tag der Disputation: 23. Juli 2021

Contents

Chapter 1

Introduction

Over the past decades the amount of goods shipped globally has increased drastically. For example, according to a recent study [15], the total number of parcels sent per year in Germany has increased from roughly 1.5 billion in the year 2000 to above 3.5 billion in 2019. Projections indicate that the annual volume could increase to 4.3 billion by the year 2024. Data collected by *Pitney Bowes*, a company specialized in services for postal systems, shows that in 2019 the global volume of parcels exceeded 100 billion for the first time [86]. The reasons for this increase are numerous, but the global relocation and centralization of production facilities for economic reasons as well as the shift towards online shopping are important factors.

As to these developments the logistics sector is confronted with a growing demand for transportation services, increasing customer expectations, ongoing legislative changes and a decreasing pool of possible employees due to the growth of the sector. The quality of service as perceived by the customers is especially important as multiple logistics service providers compete in the same market. From the customer's point of view, the logistics service provider should guarantee rather accurate time windows and stick to these reliably, such that the customers can integrate the stream of arriving parcels into their daily routine. The ongoing changes in the logistics sector and general trends of society are reflected in legislative changes w.r.t. environmental concerns or working hour regulations of drivers and other logistics personnel. The combination of increasing demand, increasing expectations, stricter regulations, the competition, costs for fuel, toll and vehicles provides a strong motivation for the logistics service providers to further optimize their operations. Facilitating the optimization and the efficient implementation of appropriate processes requires the logistics service providers to tackle a large set of additional challenges besides their day-to-day transportation and handling tasks. These challenges result in complex systems, spanning topics from data management, databases, hardware, software to adequate human-computer-interaction interfaces for customers and employees.

Vehicle routing problems (VRPs) arise in these complex systems in stages associated with planning, from long-term decision making to day-to-day operations. They address the routing of vehicles like bikes, trucks, ships and airplanes. In this context a routing consists of a sequence of operations interleaved with vehicle movements, for example the delivery of goods from a central depot to multiple customers with a delivery truck. The truck starts at the depot loaded with the required amount of goods and then moves from one customer to the next according to a specified route. At each customer location the requested amount of goods is unloaded and delivered. In the majority of realistic scenarios, the routing decisions are subject to constraints, e.g., the capacities of the considered vehicles or time windows negotiated with the customers need to be respected. Besides these hard constraints, additional criteria may be considered to assess and quantify the quality of a routing. The exact criteria depend on the specific scenarios and may include the travel cost induced by the route, the total duration of a set of routes and quality of service indicators like delays or maximum waiting times. Regarding these criteria the goal is to find a best or high-quality routing.

Research in the field of optimization problems and VRPs in particular is well-motivated

by the challenges encountered in logistics, from practical and theoretical perspectives alike. The research area covers a wide range of topics, from modeling of new problems and their variants to the study of efficient and effective algorithmic approaches for well-established problems. Modeling allows researchers to take a look at novel approaches and decisions on the strategic, tactical and operational levels and evaluate their economic benefits and implications in a controlled environment. By applying the models to real-world or synthetic data representative of real-world scenarios, managerial insights can be gained. For certain well-established problems that arise regularly, either in isolation or as components of larger combined problems, the research focus shifts towards efficient and effective techniques to solve these problems.

As outlined above, the scale of logistics processes increases in at least two dimensions: the number of items shipped and the size of the operational area. This leads to a certain discrepancy between practice and research as the latter tends to focus on smaller, more restricted problems with a focus on novel ideas. Generalizing performance results obtained on these comparably small instances is questionable. Large-scale instances in conjunction with small amounts of computational resources to derive best possible solutions provides a valid motivation for further research.

In this thesis, we consider challenging problems that represent specific aspects of complex real-world problems. We focus on efficient heuristic solution techniques that are effective even on large-scale instances and under tight restrictions of the computational budget. To achieve that, we focus on combinations of global and local search approaches embedded in higher order methods that take the available computational budget into account and allocate the available computing resources adaptively.

Organization of the thesis

In Chapter 2 we give descriptions of the background required for the remainder of this thesis. First, we provide a high-level description of two fundamental VRPs, associated special cases and the underlying motivation derived from real-world problems. Second, regarding efficient solution techniques, we give a quick overview over heuristic solution approaches including local search and metaheuristics. Finally, we take a look at empirical research in the context of heuristics and discuss implications and pitfalls to keep in mind when designing and performing experiments and computational studies.

In Chapter 3 we consider the *vehicle routing problem with unit demands* (VRPU), a unit-demand variant of the *capacitated vehicle routing problem* (CVRP) and develop a heuristic based on the *multi-insertion neighborhood* (MIN), an exponential neighborhood proposed in the literature. This highlights how constraints like unit and homogeneous demands can be exploited to derive global search procedures that can be used efficiently, especially on large problem instances and relatively constrained computing resources.

In Chapter 4 we tackle the *preemptive stacker crane problem* (PSCP) that has been studied for more than three decades. We study theoretical properties of the problem and extend the existing literature regarding bounds and solution representations. We propose constructive algorithms for the PSCP based on algorithms for well-known routing problems. In a large computational study, we evaluate the proposed algorithms and observe that they outperform the state-of-the-art heuristics in both solution quality and required computation time. As such, these algorithms are suitable for larger problem sizes.

In Chapter 5 we tackle a complex multi-period vehicle and technician routing problem with synchronization constraints. The problem considers the delivery of machines from a central depot to customers and subsequent installations of these machines by technicians. The delivery trucks and the technicians require their own routes and schedules, respec-

tively. Additionally, delivery trucks and technicians need to be synchronized. We extend the *adaptive large neighborhood search* (ALNS) method to realize a decomposition approach w.r.t. the overall problem that efficiently allocates computing resources to the subproblems in such a way as to derive solutions of high quality under tight restrictions of the computational budget. In the *restricted resources challenge* of the *VeRoLog Solver Challenge 2018–2019* (VSC2019), our method outperformed all other competing methods and achieved the first rank.

In Chapter 6 this thesis is concluded with a summary, its contribution and an outlook on future research opportunities regarding efficient heuristic solution methods for large-scale VRPs.

Chapter 2

Background

In this chapter we provide background information that is referred to throughout this thesis. This information encompasses the basics of vehicle routing problems and heuristic solution methods. All descriptions are provided on a rather abstract level to convey the basic ideas and problems at hand. We provide references to surveys and in-detail descriptions for each topic covered.

In Section 2.1 we give brief descriptions of *vehicle routing problems* (VRPs) and *pickup and delivery problems* (PDPs). We discuss a few special cases and their relationships, e.g., the *traveling salesman problem* (TSP) and the *vehicle routing problem with unit demands* (VRPU). In Section 2.2 we discuss the basics of heuristic solution approaches including local search and metaheuristics. Furthermore, we discuss aspects to keep in mind when implementing algorithms, performing experiments and interpreting the results.

2.1 Vehicle routing problems

Routing problems are optimization problems that deal with the generation of routes in a given network, possibly subject to additional constraints. The generality of this notion allows for a wide area of application, e.g., the routing of trains, trucks or data packets in rail, road or computer networks, respectively.

VRPs are routing problems historically stemming from logistics and transportation contexts. Widely encountered constraints include vehicle capacities, time windows, maximum vehicle distances or maximum travel times. Usual objective functions consider the cost or time required to perform a given set of transportation requests. Among others, the costs may be derived from the number of vehicles used, the distances traveled by the vehicles and delays or other penalties.

One of the most simple routing problems imaginable is the TSP. Colloquially speaking, the TSP asks for a shortest round trip that connects a given set of locations under the constraint that each location is visited exactly once. A detailed treatment of the TSP's history and results of the earlier research are provided by Lawler et al. [73].

Problem 2.1 (*Traveling salesman problem* (TSP))**.** *Given a complete directed graph* $G = (V, A)$ *with nodes* $V = \{1, \dots, n\}$ *and costs* c_{ij} *for all arcs* $(i, j) \in A$*, find a permutation* π *of the nodes* V *such that the cost of the closed tour implied by* π *is minimum.*

Despite the simple description of the TSP, the associated decision problem is $\mathcal{NP}$-complete by reduction from the *Hamiltonian cycle problem*, which in turn belongs to the 21 classic $\mathcal{NP}$-complete problems of Karp [63]. The TSP remains $\mathcal{NP}$-complete for Euclidean distances in two dimensions [82] and thus for symmetric and metric distances.

Although the TSP has a large area of application, a lot of real-world applications are more complex and need to consider additional requirements. In this regard, VRPs typically generalize the TSP in some way, e.g., by additional constraints or by combining the TSP with other planning and optimization problems.

In the remainder of this section we give a brief description of two fundamental VRPs and their relationships to the TSP, other optimization problems and their areas of application.

2.1.1 The capacitated vehicle routing problem (CVRP)

Consider the task of delivering a few paper envelopes to the mailboxes of a dozen facilities located in the limits of a small town. Usually a single vehicle or even a bicycle is sufficient to deliver all of these envelopes in a single day. Given these circumstances, it is enough to consider the TSP, find an optimal or good enough route and call it a day.

The circumstances may change in various ways, e.g., (i) instead of a small town, the facilities may be spread out over a large area, (ii) instead of a single day, all envelopes need to be delivered in a single hour or (iii) instead of envelopes, large boxes of different sizes need to be transported. Smaller changes may be alleviated by, e.g., choosing a larger or faster vehicle or increasing the working hours, but at some point these measures will reach physical or legal limits and, thus, the TSP alone will not be sufficient to model the problem.

Cases (i) and (ii) are comparable in that the relationship between the distances to be traveled and the provided time frame changes. Case (iii) does not incorporate any notion of time, instead it does only consider the sizes of the transported goods and the vehicle's capacity, i.e., the total amount to be delivered may exceed the capacity of the considered vehicles. Hence, the delivery vehicle needs to return to its depot and start a new tour. Alternatively, multiple vehicles may perform tours simultaneously. The *capacitated vehicle routing problem* (CVRP) corresponds to this scenario.

The CVRP was first introduced by Dantzig and Ramser [31] and is a straightforward generalization of the TSP that adds concepts of a central depot, vehicle capacity and customer demands. As such, the problem arises in two equivalent settings, (i) the supply of customers from a central depot, e.g., parcels, and (ii) the collection of goods from customers and their delivery to a central depot, e.g., the collection of waste and its delivery to a central dump.

Problem 2.2 (CVRP)**.** *Given a complete directed graph* $G = (V, A)$ *with nodes* $V = \{0, 1, \dots, n\}$*, costs* c_{ij} *for all arcs* $(i, j) \in A$*, a vehicle capacity* $C \geqslant 1$*, demands* $0 \leqslant d_i \leqslant C$*,* $i \in \{1, \dots, n\}$*, and a maximum number of vehicles* $K \geqslant 1$*.*

Find at most K *tours, each starting and ending at the depot* 0*, such that each customer* $i \in \{1, \dots, n\}$ *is visited exactly once in one of the tours, the total demand of each tour does not exceed the vehicle capacity* C *and the total cost of the tours is minimized.*

All decisions made to derive a TSP solution are concerned with the order of the locations, i.e., the *routing*. In contrast, the decisions made to derive a CVRP solution also contain an additional problem dimension, the *grouping*. The grouping decisions result in disjoint sets of customers such that the total demand of the customers in a single set does not exceed the vehicle capacity. Then, all customers in a single set including the depot are routed to derive an actual tour. Hence, the grouping of the CVRP corresponds to the decision problem of the *bin-packing problem* (BPP) and the routing to the optimization problem of the TSP.

The CVRP and various of its variants have been extensively researched. An overview of various models and exact, approximation and heuristic methods to solve them are given in Laporte [72], Toth and Vigo [98] and Cordeau et al. [29].

In Chapter 3 we consider a special case of the CVRP called the VRPU. The VRPU corresponds to the CVRP with the exception that all customer demands $d_i = 1$ are equal. Thus, the BPP associated with the CVRP becomes trivially solvable, which in turn can be used to develop efficient heuristic algorithms for the VRPU by exploiting this unit-demand structure.

2.1.2 Pickup and delivery problems (PDPs)

The role of the central depot in the CVRP is twofold. On the one hand, it is the origin and destination of the vehicles. On the other hand, it is either the origin or the destination of the goods being transported. In other words, the central depot ties the production or consumption of goods to their delivery or collection.

In contrast, consider a taxicab company. The taxi vehicles begin and end their shift at the company's central depot. However, clients call to request a taxi ride from one location to another location in the company's area of operations. Usually neither of these locations coincide with the depot. As such, the direct connection between the depot and the origin or destination of goods or clients is dissolved resulting in the more general class of *pickup and delivery problems* (PDPs).

Savelsbergh and Sol [93] introduce the *generalized pickup and delivery problem* (GPDP) that generalizes a wide range of VRPs and PDPs. Furthermore they survey the early literature on problems that are generalized by the GPDP, different objective functions, constraints and solution methods. As the GPDP is rather general we further discuss two extensively researched special cases, namely the *pickup and delivery problem with time windows* (PDPTW) and the *dial-a-ride problem* (DARP).

Problem 2.3 (PDPTW)**.** *Given a complete digraph* $G = (V, A)$ *with nodes* $V = \{0, 1, \dots, n\}$*, travel times* $t_{ij} \geqslant 0$ *for all arcs* $(i, j) \in A$*, a vehicle capacity* $C \geqslant 1$*, a maximum number of vehicles* $K \geqslant 1$ *and a set of requests* $R = \{1, \dots, m\}$*. Each request* $r \in R$ *is associated with a demand* $q_r \geqslant 1$ *that needs to be transported from a pickup location* $v_r^- \in V$ *to a delivery location* $v_r^+ \in V \backslash \{v_r^-\}$*. The pickup and delivery operations are associated with time windows* $[e_r^-, \ell_r^-]$ *and* $[e_r^+, \ell_r^+]$*, and service times* s_r^- *and* s_r^+*, respectively. Node* 0 *corresponds to the depot and is associated with a time window* $[t_0, t_{\max}]$*.*

Find a set of at most K *routes, each starting after time* t_0 *and ending before* $t_{\max}$ *at the depot* 0*, such that all requests are satisfied exactly once, and the vehicle capacity is never exceeded. A request is satisfied if its pickup and delivery operations are performed in the same tour (coupling) and the pickup is performed prior to the delivery (precedence). Additionally, all pickup and delivery operations need to start inside their respective time windows. If a vehicle arrives at a location too early it must wait until the corresponding time window opens. When performing a pickup or delivery operation, the corresponding service time is incurred.*

A set of tours is optimal iff it minimizes the number of vehicles used as a primary criterion and the total travel distance as a secondary criterion. The total travel distance is derived from the sum of travel times t_{ij} *over all arcs traveled by the vehicles.*

The PDPTW as stated considers a lexicographic objective function of two values. Nevertheless, the exact objective functions used in the literature vary between publications, e.g., Li and Lim [74] additionally consider the total schedule duration corresponding to the sum of durations of all routes and the total waiting time corresponding to the sum of all waiting times as third and forth minimization criteria, respectively.

The time windows associated with the depot, the pickup operations and the delivery operations increase the complexity of the problem considerably. Without the time windows, finding a feasible solution is trivial because all requests could be performed by a single vehicle in any order as long as the precedence and capacity constraints remain satisfied, e.g., a tour $\pi = \langle 0, v_1^-, v_1^+, \dots, v_m^-, v_m^+, 0 \rangle$. In contrast, finding a feasible solution for the PDPTW is already an $\mathcal{NP}$-hard problem (cf. [93]).

Considering the initially mentioned taxicab company, the description of the PDPTW may seem insufficient as there is no notion of client satisfaction. As such, the DARP is a specific extensions that adds so called *maximum ride-time constraints* limiting the maximum time that any client (request) can spend in transit between pickup and delivery. Note that Savelsbergh

and Sol [93] state that the DARP is a unit-demand problem, i.e., $q_r = 1$, such that each request corresponds to exactly one client. However, recent publications tend to allow for groups of $q_r \geqslant 1$ people that may not be split up. Thus, the essential difference between the DARP and the PDPTW is the existence of maximum ride-time constraints or possibly other restrictions related to the quality of service as perceived by human clients.

For a description of further PDP variants we refer the interested reader to the classification and survey articles by Berbeglia et al. [14] and Parragh et al. [83].

2.2 Heuristics

Developing a naive algorithm to solve a $\mathcal{NP}$-hard combinatorial optimization problem, e.g., the TSP, is usually simple. The number of possible solutions is finite, thus a *full enumeration* of all possible candidate solutions is sufficient. During the enumeration, the current best feasible solution is maintained and corresponds to an optimal solution after all solutions have been enumerated. Otherwise, if no candidate solution is feasible, then the problem instance itself is infeasible.

However, the number of solutions to consider in a full enumeration is exponential in the size of the problem instance. Therefore, the full enumeration is impractical. Structures of the problem may be exploited to derive algorithms that improve on full enumeration, e.g., *branch-and-bound*, *mixed integer programming* (MIP), constraint programming or dynamic programming. Nevertheless, for certain problems and input sizes encountered in real-world instances in combination with runtime requirements often render these improved exact methods also impractical. One way to combat this issue is the use of heuristics that trade the guarantee of proven optimality for reduced computation time.

On the conceptual level, a heuristic is a procedure to derive a decision or solution for a problem. For example, consider shopping for groceries in a supermarket. When approaching the check-out with the selected items, one may have the choice between multiple queues and cashiers. At this point, the problem may be as follows: select a queue such that the time to the completion of the check-out is minimized. A simple heuristic would be to select a queue that is shortest in terms of the number of people in the queue or the amount of items on the conveyor belt. The experienced reader will know that such a heuristic is rarely optimal. As such, selecting a queue at random is also a reasonable heuristic.

With respect to combinatorial optimization problems and VRPs in particular, the situation is similar at first. For example, for the TSP any random permutation of the nodes implies a feasible solution and therefore, selecting such a random permutation is sufficient to derive a solution. However, heuristics are often designed using knowledge about the problem structure. For example, instead of choosing a random permutation, an arbitrary initial node is chosen and the nearest, not yet visited neighbor is chosen. Starting from this chosen neighbor, the next not yet visited neighbor is chosen and so on, until a complete tour is formed. This so-called *nearest neighbor heuristic* for the TSP (cf. [12]) tends to generate solutions of higher quality while consuming more computation time. Thus, different heuristics for the same problem may realize different trade-offs between solution quality and computation time, just as it is the case for exact methods and heuristics in general.

Heuristics can be classified w.r.t. different properties. One of these is the distinction between *construction* and *improvement heuristics*. Construction heuristics take an input instance and produce a solution from scratch. In contrast, improvement heuristics usually take an already existing solution as input with the goal to improve it w.r.t. the objective function value. The distinction between both classes is not strict, i.e., construction heuristics may use components from improvement heuristics and vice versa.

The literature on heuristics is vast, from reference works over surveys on the individual heuristic methods to works on specific applications. For a comprehensive overview of heuristic methods in general, we refer the reader to Martí et al. [77] and Gendreau and Potvin [46]. In the following, we will focus on the description of some fundamental building blocks of heuristics that will occur regularly throughout this thesis.

2.2.1 Local search

The idea underlying *local search* is that an existing solution may be improved by performing local and small changes. For a detailed treatment of local search and its theory, we refer the reader to Michiels et al. [78].

One of the essential concepts of local search is that of a *neighborhood*. Assume a minimization problem with cost function $c(\cdot)$ and the set of all solutions $\mathcal{S}$. For a single solution $S \in \mathcal{S}$, its neighborhood $\mathcal{N}(S) \subseteq \mathcal{S}$ corresponds to a subset of all possible solutions. A solution $S' \in \mathcal{N}(S)$ is called *neighbor* of S in $\mathcal{N}$. To improve a solution S its neighborhood $\mathcal{N}(S)$ is searched for a better solution $S' \in \mathcal{N}(S)$ with $c(S') < c(S)$ and S is replaced by S'.

A solution S is *locally optimal* w.r.t. a neighborhood $\mathcal{N}$ iff $c(S) \leqslant c(S'), \forall S' \in \mathcal{N}(S)$, i.e., the solution is at least as good as its best neighbors. A locally optimal solution S is *globally optimal* iff $c(S) \leqslant c(S'), \forall S' \in \mathcal{S}$. Consider the directed graph $G_{\mathcal{N}} = (\mathcal{S}, A)$ with $A = \{(S, S') \colon S \in \mathcal{S}, S' \in \mathcal{N}(S)\}$ induced by the neighborhood $\mathcal{N}$. A neighborhood $\mathcal{N}$ is *connected* iff for each pair of distinct solutions $S, S' \in \mathcal{S}$ there exists a directed path from S to S' in $G_{\mathcal{N}}$. Let $p_{S,S'}$ be such a path in $G_{\mathcal{N}}$ with the minimal number of arcs among all paths from S to S'. The *diameter* of a neighborhood $\mathcal{N}$ is the maximum number of arcs over all these paths, i.e., $\max_{S,S' \in \mathcal{S}} \{|p_{S,S'}|\}$.

The size of a neighborhood corresponds to the number of neighbors $|\mathcal{N}(S)|$. However, often the exact size may vary depending on the specific input problem and solution. Therefore the size is regularly discussed in terms of asymptotic growth in $\mathcal{O}$-notation. The size of the neighborhood provides an upper bound for the computational complexity of finding a best neighbor. Besides the sizes, different neighborhoods can be analyzed w.r.t. the sets of neighbors they contain. A neighborhood $\mathcal{N}_1$ is *contained* in another neighborhood $\mathcal{N}_2$ iff $\mathcal{N}_1(S) \subseteq \mathcal{N}_2(S)$, $\forall S \in \mathcal{S}$. As such, the containing neighborhood may be used in place of the contained neighborhood although its possibly larger size may incur a larger time to find a best or improving neighbor.

The neighborhood concept is rather broad and in general, neighborhoods may be arbitrarily defined. However, neighborhoods are often derived from operations performing well defined local changes to the solution or its representation. For example, in the context of permutations, possible operations may interchange two adjacent elements or swap two arbitrary distinct elements of the permutation. The neighborhoods resulting from these operations are called $\mathcal{N}_{\text{api}}$ (*adjacent pairwise interchange*) and $\mathcal{N}_{\text{swap}}$, respectively.

Example 2.1. *Consider the permutation $\pi_1 = (1, 2, 3)$. Its neighbors in $\mathcal{N}_{api}$ are $\mathcal{N}_{api}(\pi_1) = \{(2, 1, 3), (1, 3, 2)\}$. The neighborhood $\mathcal{N}_{swap}(\pi_1) = \{(2, 1, 3), (1, 3, 2), (3, 2, 1)\}$ contains one additional neighbor $(3, 2, 1)$ while all neighbors in $\mathcal{N}_{api}$ are also contained in $\mathcal{N}_{swap}$. For $|\pi| = 2$, both neighborhoods are equivalent, but for $|\pi| \geqslant 3$ the property $\mathcal{N}_{api}(\pi) \subset \mathcal{N}_{swap}(\pi)$ generally holds.*

While the neighborhoods $\mathcal{N}_{\text{api}}$ and $\mathcal{N}_{\text{swap}}$ can be applied to any problem whose solutions are representable by permutations of objects, a lot of problem-specific neighborhoods are proposed in the literature. For instance, the k-opt neighborhood of Lin [76] is a rather well-known neighborhood for the TSP and thus for other routing problems containing TSP subproblems as well. For an overview and description of neighborhoods and other local search methods for VRPs we refer the reader to Funke et al. [43] and Cordeau et al. [29].

Regardless of the specific problem, the local search operators for VRPs can be classified into *intra-tour* and *inter-tour* operators. The former operate on a single tour while the latter operate on multiple tours at once. Note that operators may belong to both classes simultaneously. While some operators target the specific structure of the considered VRP variant, others like k-opt are more general and in widespread use.

Searching neighborhoods The way a neighborhood operator is used depends on the broader algorithmic framework it is embedded in. However, the most common uses include *sampling* and *enumeration*. In case of sampling, a random neighbor is chosen from the neighborhood. In case of enumeration the neighborhood is searched in a systematic way, either until an improving neighbor (*first fit*) or a best neighbor (*best fit*) has been found. A simple procedure called *iterative improvement* searches for an improving or best neighbor and repeats the search starting from this neighbor until no improving neighbors are found indicating that the resulting solution is locally optimal.

Combining neighborhoods Multiple neighborhoods may be combined into a larger neighborhood. A common way to combine a set of neighborhoods is by ordering them in a fixed sequence. The first neighborhood in the sequence is iteratively searched until no improving neighbors can be found. Then the process continues with the second neighborhood and so on. The process terminates when the current solution is locally optimal w.r.t. all considered neighborhoods. This process is called *variable neighborhood descent* (VND) and a recent review of this method along with successful applications in the literature is provided by Duarte et al. [35].

2.2.2 Large neighborhoods

The local search operators described above induce neighborhoods of polynomial size and can thus be searched for best or improving neighbors in polynomial time as well. However, for a variety of problems, larger neighborhoods usually lead to better local optima on the one hand and larger computation times on the other. Consider neighborhoods of increasing size. At some point, the size of the neighborhood will actually be exponential w.r.t. the input problem size, leading to the concept of so-called *exponential neighborhoods* or *large neighborhoods* in general. For a more detailed overview of different large neighborhood methods, we refer the reader to Pisinger and Ropke [85].

Given a neighborhood of exponential size, searching for a best or improving neighbor by full enumeration becomes impractical. However, for some exponential neighborhoods the problem of finding a best neighbor is still polynomial-time tractable. Heuristics for combinatorial optimization problems applying neighborhoods from this class of exponential neighborhoods are called *very large-scale neighborhood search* (VLSN) heuristics. A survey of these methods is provided by Ahuja et al. [2]. Most of the research articles on exponential neighborhoods study theoretical properties like the neighborhood diameter or the complexity of finding a best neighbor (cf. [3, 22, 32, 55, 56, 87]). Other articles report results of computational studies (cf. [20, 59]).

If finding a best or improving neighbor is not possible in polynomial time, it makes sense to resort to a heuristic approach that samples neighbors. One popular approach introduced by Shaw [94] is the so-called *large neighborhood search* (LNS). Thereby a possibly exponential neighborhood is generated through a combination of *destroy* and *repair* operators. The sampling of a neighbor is performed in two stages. First, the destroy operator reduces the solution by removing objects from it and returns the reduced partial solution and the set of removed objects. Subsequently, the repair operator inserts the removed objects into

the partial solution thereby recreating a new solution with possibly lower cost. To utilize the repair and destroy operators in an iterative approach, the neighbors are sampled by randomizing either the destroy operator, the repair operator or both. Note that the randomization does not need to be uniform, i.e., the operators may derive probabilities for certain destroy or repair operations from their respective input solutions and considerations regarding the problem structure. One rather straightforward option is to couple a randomized destroy operator with a deterministic greedy insertion heuristic.

2.2.3 Metaheuristics

Neighborhoods provide a way to move from one solution $S \in \mathcal{S}$ to a neighbor solution $S' \in \mathcal{N}(S)$. This property on its own is often not sufficient to derive good solutions for given problem instances. To mitigate that issue, neighborhoods are embedded in higher order methods that guide the search process by orchestrating the use of problem-specific heuristics and solution management. These higher order methods are called *metaheuristics*, because they are often similar across different problems and thus problem independent. Metaheuristics may operate on a single solution or a so-called *population* of solutions.

The central issues addressed by metaheuristics are the diversification and intensification of the search space. The former corresponds to a broad sampling of the search space to identify promising regions. The latter corresponds to an intensified search of promising regions to identify solutions of high quality. Given that most optimization processes are subject to limited computing resources, metaheuristics also need to balance diversification and intensification to realize a trade-off resulting in solutions of high quality.

The field of metaheuristics is rather huge due to its versatility and success in tackling a broad range of optimization problems. In the following, we will describe certain metaheuristic concepts that are relevant in the scope of this thesis. For a further in-depth treatment of different metaheuristics, we refer the reader to Gendreau and Potvin [46] and additionally to Sörensen [96] for an analysis of the field w.r.t. the variety of methods and the resulting problems to keep track of new developments and to tell apart new from known approaches.

Simulated annealing

The iterative improvement method described above only accepts improving neighbors until a local optimum is reached. As such, iterative improvement realizes a rather strong intensification but no diversification of the search space. The other extreme is realized by performing a random walk through the graph induced by the neighborhood, i.e., any neighbor solution is accepted, improving or not. *Simulated annealing* (SA) realizes a trade-off between these two extremes by accepting non-improving candidates stochastically while taking the difference between the current and the candidate solution as well as the progress of the search process into account.

The SA metaheuristic was proposed by Kirkpatrick et al. [68] and is inspired by the physical process of annealing metals. Abstractly speaking, as long as the temperature of the system under consideration is relatively high, drastic state changes are more likely. When the temperature decreases, the likelihood of state changes decreases as well. In the context of the SA metaheuristic this concept is replicated by a so-called *temperature* with higher temperatures corresponding to an increased probability that a non-improving neighbor is accepted. Note that despite the name, in usual applications of SA the connection to the annealing of metals is only metaphorical and not an actual simulation of a physical system.

A detailed description of the SA heuristic is provided in Algorithm 2.1. In each iteration a candidate solution is derived, for example by choosing a neighbor solution from the neighborhood of the current solution. Let S be the current solution and S' the candidate solution.

In the original method as proposed by Kirkpatrick et al. [68] an improving candidate is always accepted. The probability that a worse candidate is accepted

$$p(S, S', \tau) = e^{-\left(\frac{c(S')-c(S)}{\tau}\right)} \tag{2.1}$$

depends on the difference between the current and the candidate solutions, and the current temperature τ. Higher temperatures and smaller differences increase the probability of acceptance. During the search, the temperature is regularly modified according to a so-called cooling schedule. A common schedule is the *geometric cooling* $\tau_{t+1} = \alpha\tau_t$ for a parameter $0 < \alpha < 1$ resulting in a monotonically decreasing temperature trajectory.

Algorithm 2.1 Simulated annealing (SA)

Input: Initial solution S
Output: Solution

$S^\star \leftarrow S$
$\tau \leftarrow \tau_0$ ▷ initial temperature
while time or iteration limit is not reached **do**
  $S' \leftarrow \text{SELECT}(\mathcal{N}(S))$ ▷ derive candidate
  if $c(S') < c(S)$ **or** $\text{RAND}(0,1) \leqslant e^{-\left(\frac{c(S')-c(S)}{\tau}\right)}$ **then** ▷ accept
    $S \leftarrow S'$
  end if
  if $c(S') < c(S^\star)$ **then**
    $S^\star \leftarrow S'$
  end if
  $\tau \leftarrow \text{COOLING}(\tau)$ ▷ update temperature
end while
return $S^\star$

To use a SA based heuristic in practice it is necessary to decide (i) how to derive an initial temperature τ_0, (ii) which cooling schedule to use and (iii) how candidate solutions are derived. Note that the acceptance probability given in Equation (2.1) uses the temperature τ to scale the difference between the objective function values of two solutions. This implies that the temperature τ is not robust against scaling of objective function coefficients, e.g., multiplying all costs in a TSP instance by a constant yields an equivalent problem instance but would require different temperatures to derive the same acceptance probabilities for structurally equivalent solutions. Thus, choosing initial temperatures and cooling schedule parameters may have to be done dynamically w.r.t. a specific problem instance.

Franzin and Stützle [39] provide a recent survey and comparative study of the various variants of SA based heuristics. A dynamic adjustment of the cooling schedule w.r.t. a given time limit seems to be rather rare throughout the literature. In contrast, we will use such an approach in Chapters 3 and 5 to derive SA based heuristics that ensure that a certain temperature trajectory is traversed in a fixed, user specified time limit.

Iterated local search

Another way to overcome local optima is through the use of a *shaking procedure* that takes a locally optimal solution and generates another solution that is hopefully not locally optimal and may be improved by local search beyond the quality of the input solution. In other words, the current solution is moved away from a local optimum and the search is restarted.

An overview of the *iterated local search* (ILS) method is shown in Algorithm 2.2. Each

iteration starts with a shaking phase and the resulting solution is improved by a local search phase. The locally optimal candidate solution S'' is then accepted according to a specified acceptance criterion, e.g., only improving solutions are accepted. Another option is to combine ILS and SA by utilizing the SA acceptance criterion.

The essential component is the shaking procedure. On the one hand, it must perturb the current solution enough to overcome the current local optimum, i.e., the subsequent local search phase should not result in the same solution. On the other hand, it should not completely destroy all structures and attributes generated by the search process so far, otherwise the ILS corresponds to a process that iteratively applies local search to randomly generated initial solutions. As such, the shaking procedures are usually randomized while taking structural properties of the specific problem into account.

Algorithm 2.2 Iterated local search (ILS)

Input: Initial solution S
Output: Solution

```
 1: S* ← S                                          ▷ best solution
 2: while time or iteration limit is not reached do
 3:     S' ← SHAKE(S)                               ▷ diversify
 4:     S'' ← LOCAL SEARCH(S')                      ▷ improve by local search
 5:     if ACCEPT(S'', S) then
 6:         S ← S''
 7:     end if
 8:     if c(S'') < c(S*) then
 9:         S* ← S''
10:     end if
11: end while
12: return S*
```

Adaptive large neighborhood search

Adaptive large neighborhood search (ALNS) is an extension of the LNS method of Shaw [94] that allows for multiple repair and multiple destroy operators. The operators themselves are chosen randomly in each iteration, weighted by their past performance in the search process. Such an approach is beneficial if it is unclear how to develop a heuristic or operator that works equally well on all possible instances. Instead, multiple operators are provided and the appropriate choices are determined during the search process w.r.t. the specific problem instance.

Before we go into more detail regarding the ALNS method, we take a brief look at the general idea of *hyper-heuristics* (HHs). Informally, HHs are *heuristics to choose heuristics*. A more precise definition is given by Burke et al. [23] who survey HHs, give a classification of different methods and review applications. They state that "[the] defining feature of hyper-heuristics is that they operate on a search space of heuristics rather than directly on a search space of problem solutions". They distinguish between *generating* HHs and *selection hyper-heuristics* (SHHs). The former deal with the generation of heuristics and operators for a given problem. The latter consider the selection of heuristics from a fixed pool of candidate heuristics. A recent survey on SHHs is provided by Drake et al. [34].

Furthermore, SHH can be grouped into *online* and *offline* methods. In online methods, the selection of heuristics and adaptation is performed at runtime, i.e., while a specific instance of an optimization problem is solved. Offline methods select heuristics by using a prepared

training set of instances and evaluate various selection and parameter options. An underlying assumption is that the training set is representative of future, yet unknown instances, such that all generated selections and parameters remain applicable.

An example of an offline method is *automatic parameter tuning* (cf. [1, 7, 30, 60]) whereby the search for an optimal or rather good set of parameters for a solution method is regarded as a higher-level optimization problem. The parameter space corresponds to the search space. The objective function value associated with a specific parameter configuration is derived from results obtained by the solution method over a training set of instances. Note that both, online and offline methods can be combined appropriately by offline tuning a subset of the parameters and online tuning the remaining parameters. According to this classification, ALNS should be regarded as an online SHH.

Ropke and Pisinger [92] develop the ALNS method that extends LNS by allowing for multiple repair and multiple destroy operators, and an adaptive layer that is used to control which specific operators are applied. The method is illustrated in Algorithm 2.3. The adaptive layer is realized by a collection of weights W, one weight for each operator. In each iteration, a repair and a destroy operator are chosen w.r.t. probabilities derived from the weights, e.g., by dividing all weights corresponding to a single decision by the sum of these weights. The weights are adjusted regularly during the search process to adapt to the specific problem instance. The underlying idea is, that operators that lead to improving solutions more often should have higher weights and should be selected more often.

Detailed descriptions of LNS, ALNS, the adaptive layer, the weight adjustments and the operator selection alongside examples of successful applications of the methods are given in Pisinger and Ropke [85].

Algorithm 2.3 Adaptive large neighborhood search (ALNS)

Input: Initial solution S, set of destroy operators H_d, set of repair operators H_r
Output: Solution

```
1:  S* ← S                                                  ▷ best solution
2:  Initialize weights W for operators h ∈ H_d ∪ H_r
3:  while time or iteration limit is not reached do
4:      Choose destroy and repair operators h_d ∈ H_d and h_r ∈ H_r w.r.t. W
5:      S' ← h_r(h_d(S))                                    ▷ generate neighbor
6:      if ACCEPT(S', S) then
7:          S ← S'
8:      end if
9:      if c(S') < c(S*) then
10:         S* ← S'
11:     end if
12:     Update weights W                                    ▷ learning
13: end while
14: return S*
```

Similarities and Combinations

To conclude the overview of metaheuristics used in this thesis, we want to stress that despite the separate description of the individual methods and the large body of publications dealing with them in isolation, they are far from mutually exclusive. Both, ILS and ALNS may use SA acceptance criteria. The shaking phase of an ILS procedure may be realized through problem specific destroy and repair operators. Likewise, an ALNS approach may be augmented

with a local search phase. Hence, a problem specific heuristic may correspond to multiple metaheuristic approaches at the same time.

2.2.4 Empirical heuristics research

As noted before, heuristics trade the guarantee of proven optimality in favor of reduced computation time or other resources. For a given problem, multiple heuristics may be available, possibly with multiple actual implementations. Given the choice, this raises the general question of what heuristic to use. Clearly, there is no single answer, instead the answers depend on the specific requirements and use cases. For example, assume two heuristics for the same problem. If either one of the heuristics outperforms the other in terms of quality and time, then the choice may be relatively simple because there is no trade-off to be made. On the contrary, if one heuristic yields solutions of lower quality in a short time while the other provides solutions of better quality while consuming more time, then an actual decision has to be made, favoring either quality over time or vice versa.

Prior to such a decision, it needs to be established whether one heuristic does in fact outperform another w.r.t. quality, time or both. From a theoretical perspective, available tools include upper and lower bounds, approximation guarantees and the analysis of the runtime complexity. However, these results do not always map to results obtained on practical problem instances as well, e.g., a method with a better average case quality guarantee may still perform worse on average in practical settings because the set of practical instances is not representative of the complete set of instances. Additionally, a large set of heuristics currently elude theoretical analysis due to randomization and the complex nature of the operations they perform, i.e., a purely theoretical analysis is currently impractical although it may be possible. Thus, the performances of the heuristics have to be established empirically in computational studies through experimentation using computer hardware, software implementations of the heuristics under study and problem instances.

Computational studies and empirical research of algorithms has been considered repeatedly in the literature (cf. [11, 50, 53, 57, 58, 75, 90]). Issues discussed include (i) the selection of benchmark instances, (ii) the experimental setup, i.e., computers, software and reproducibility, (iii) the algorithms' sensitivities to external and fixed parameters like SA cooling schedules and (iv) appropriate statistical approaches, e.g., w.r.t. missing responses due to timeouts. The validity and generalization of the results obtained and inferred from computational studies strongly depend on sensible approaches to these issues. Rardin and Uzsoy [90] note that a fair way to compare possibly drastically different algorithms for the same problem is by providing them with the same budget of computational resources and compare the obtained solutions. Computational resources usually refer to a time limit or other quantities like the number of operations. In the following paragraphs we take a closer look at the selection of benchmark instances, the experimental setup and the implementation effort w.r.t. experiments that include a notion of time, either as restriction of computational resources or as a dependent variable that is reported along with the solution quality.

Instances

There exist several ways to obtain benchmark instances. In widespread use are *real-world instances*, *randomly generated instances* and combinations thereof. A published and generally available set of instances with known properties is a so-called *library* (cf. [53]). Performing computational studies over well-known instance libraries is the most prevalent practice. Using the same set of instances when comparing multiple algorithms is called *blocking on instances* by Rardin and Uzsoy [90]. They show that such an approach works better to uncover

differences between multiple algorithms as opposed to running the algorithms on different instances generated randomly from the same distribution.

Real-world instances are harder to obtain and therefore less numerous. Additionally, complete sets of real-world instances may originate from the same source which may negatively impact the set's diversity. This situation leads to frequent criticism questioning the validity of the results and the derived conclusions. Hooker [57] notes that criticizing the non-representativeness of library instances is always possible in principle because the reference, i.e., the set of instances that should be represented, is not made explicit nor is it obvious what the reference should be. One option is the set of all possible instances. However, this set may contain a lot of instances that are rather dissimilar to real-world or other more interesting instances. Therefore, the drawn conclusions may be inadequate regarding the practical settings that the algorithms are used in.

Random generation of instances is one option to remedy this issue (cf. [75]). Instead of a uniform distribution of instances, instances may be generated according to predefined ranges of parameters resulting in specific instance properties, e.g., the properties of distances and times like symmetry, satisfaction of the triangle inequality and clustering. In this way, algorithm performances can be compared and analyzed w.r.t. these instance properties. In conclusion, the selection of benchmark instances should depend on the research questions being tackled.

Environment

The environment in which experiments are conducted encompasses all hardware and software components as well as external or fixed parameters as discussed by Golden et al. [50], i.e., it corresponds to the laboratory and its equipment. In the majority of publications, a description of the environment is provided to varying degrees of detail. Rardin and Uzsoy [90] discuss the extent to which the setup of the environment should be described in order to ensure reproducibility of the results by other researchers. They note that the degree of detail required is a matter of ongoing discussion in the research community. Additionally, a description of the environment allows the reader to gain an intuition while interpreting measured quantities like durations, as the number of operations executed per time frame by a contemporary CPU exceeds the number of operations performed by a 40 year old CPU by multiple orders of magnitude.

Publications usually report the exact CPU model, the size of the available RAM, the operating system including its major version, the implementation programming language, the compiler, the version of its runtime and the versions of additional third-party libraries and tools, e.g., MIP solvers or graph algorithm libraries. While this information is sufficient for the reader to derive an intuition of the presented results, it is questionable how well these results generalize to other hardware and software architectures.

The following examples highlight the impact of modern hardware developments on the perceived performance of programs and thus, the influence on observed variables like durations required to perform a specific computation. Analogously, this performance impacts any computation that is limited to a certain time limit, like iterative heuristics or MIP solvers. Mytkowicz et al. [80] consider the so-called *measurement bias* in experiments conducted on computer systems, i.e., biases introduced by the experimental setup that skew the results such that wrong conclusions may be drawn. They show that minor changes, like running software under a different username on a UNIX system may impact the performance of a program by $\pm 30\%$. This impact is due to the fact that the user's environment variables are loaded into RAM together with the program. Thus, the size of the user's environment variables affects the relative position of the program in RAM which in turn affects the speed of

the memory access to the individual procedures of the program. Kaligosi and Sanders [62] consider runtime optimal choices of the pivot-element for the quicksort algorithm. They show that modern advanced CPU techniques like *branch-prediction* lead to runtime improvements, if instead of the theoretically optimal median pivot-element a different element is chosen, such that the branch-predictor of the CPU is more likely to correctly predict the branching outcomes. This shows how certain algorithmic choices may be beneficial on some hardware components while they may be not on others. Similarly, the so-called *speculative execution* is able to increase perceived processor speed by executing certain program paths before it is determined whether they should be executed at all. This led to security vulnerabilities (e.g., *Meltdown*, *Spectre*) which in turn can be mitigated via software updates. Whether these software-based mitigations are enabled or not may impact the perceived performance of programs, both positive and negative, as shown by Bowen and Lupo [19].

Performing computational studies that correct for all these influences and reporting all exact details of the implementations, settings, user's environment variables and so on is highly impractical. Nevertheless, it should be kept in mind that these influences exist and that timing results obtained in a specific environment may not simply generalize to another setup. An essential takeaway however is that computational studies comparing multiple algorithms and their implementations should be performed in the same environment, especially if they report CPU or wall-clock times.

Implementation effort

Comparing algorithm implementations in the same environment incurs additional challenges if one or more of the algorithms have been reported in the existing literature. Then, either implementations of the algorithms by other authors are available or they have to be implemented again from the descriptions provided by the original authors. Even if an existing implementation is available, the quality of that implementation must be assessed regarding its suitability for a fair comparison. This generally raises the question what kind of effort should be invested into implementations.

Rardin and Uzsoy [90] distinguish between a *research phase* and a *development phase*. The former corresponds to the study of new methods and algorithms to solve a problem in a mostly scientific environment to generate insights about the problem, its structure and suitable solution approaches. In contrast, the development phase is concerned with finding the most efficient implementation for a given problem and scenario, usually performed in a commercial setting with real-world applications. Rardin and Uzsoy [90] agree with Hooker [57] that "excessive emphasis" on the efficient implementation of algorithms in the research phase diverts time from the actual research. However, this tautological statement does not help to determine the actual effort, although it may imply that the efficiency of implementations may not be that important.

We think any limit on the implementation effort is arbitrary. However, deriving a generally most efficient implementation is impractical if not impossible, especially in the context of the influences of hardware and software components of the environment as explained above. As such, any limit of the effort does also realize a trade-off that should be made w.r.t. the goals of the computational study. As time related quantities frequently occur in computational studies, either as time limits or dependent variables, we assert that efficient implementations are in fact important. This can be done by selecting proper algorithms and data structures for subproblems and representations of problem instances and solutions, respectively. The choices should be evaluated and justified in preliminary computational experiments. In conclusion, efficient implementations are important and a certain amount of time should be invested into the implementation of algorithms.

Chapter 3

The vehicle routing problem with unit demands

In this chapter, we continue research on the exponential *multi-insertion neighborhood* (MIN) for the *vehicle routing problem with unit demands* (VRPU) that was proposed by Angel et al. [3] in a purely theoretical work. The VRPU is a special case of the *capacitated vehicle routing problem* (CVRP) that imposes unit demands for all customers. Thus, the vehicle capacity effectively limits the number of customers in each tour (cf. Section 2.1.1). The MIN is applied to a solution of the VRPU by selecting a set of so-called *mobile nodes* and removing it from the solution. Subsequently the mobile nodes are optimally reinserted into the remaining partial solution under the constraint that at most one mobile node is inserted between any pair of two consecutive remaining nodes in the tours of the partial solution. We extend the study of this exponential neighborhood in theory and practice. Parts of this chapter have already been published in the peer-reviewed research article Buckow, Graf, and Knust [21].

Contribution We contribute a theoretical analysis of the MIN by studying its connectivity, diameter and the quality of local optima in comparison to other well-known local search neighborhoods. We show that finding an optimal set of mobile nodes is $\mathcal{NP}$-hard. From a practical perspective we contribute an extensive computational study comparing different node selection approaches and we evaluate the proposed embedding of the MIN in a *simulated annealing* (SA) metaheuristic with other solvers from the literature. On the abstract level we show that using a global approach as provided by the MIN is beneficial especially at the beginning of a search process on large-scale instances or when the time limit is rather small. However, when sufficient computing resources are available, a combination of global and local approaches provides the best results.

Organization In the following Section 3.1 we provide an introduction to the MIN, the VRPU and related literature. In Section 3.2 we describe the considered VRPU and the MIN more precisely. In Section 3.3 we study the MIN from a theoretical point of view. We prove that the problem of finding a best set of mobile nodes for a solution is strongly $\mathcal{NP}$-hard, consider the question of how many nodes may be chosen as mobile, and study the connectivity of the neighborhood as well as the quality of local optima. Section 3.4 is devoted to a practical algorithm incorporating the MIN by combining it with a SA acceptance criterion in a two-stage approach akin to the *large neighborhood search* (LNS) approach: in the first stage a set of mobile nodes is selected and in the second stage the selected mobile nodes are optimally reinserted into the partial solution. For the first stage we propose different node removal heuristics. Computational results for all the considered approaches are reported in Section 3.5. The chapter closes with concluding remarks regarding the MIN and implications for this thesis in Section 3.6.

3.1 Introduction

The MIN generalizes an exponential neighborhood originally proposed for the *traveling salesman problem* (TSP) in Punnen [87] and Gutin [55]. It is based on first removing a set of mobile nodes from the tours and subsequently reinserting them in possibly different positions in the remaining tour. Angel et al. [3] show that in case of the VRPU and a chosen set of mobile nodes, a best reinsertion of these mobile nodes can be determined in polynomial time by solving a particular case of a generalized matching problem. However, no computational study was performed, i.e., it remains unclear how well a *very large-scale neighborhood search* (VLSN) procedure utilizing this neighborhood performs. Additionally, several details of such an approach were not specified by the original authors and there exist many possibilities to implement them. For example, how to choose the nodes that are to be removed from the tours before they are reinserted in a best possible way.

While a huge amount of literature considers the CVRP, its generalizations and several variants as described in Section 2.1.1, the VRPU has not been studied that often. Regarding the VRPU as special case of the CVRP, Campos et al. [24] studied the polytope and inequalities for a specific formulation, while Kudva et al. [71] developed a branch-and-cut method. The VRPU is also a special case of the *black and white traveling salesman problem* (BWTSP). Bourgeois et al. [18] solved the BWTSP heuristically and Ghiani et al. [47] proposed an exact branch-and-cut approach. Talluri [97] considered a special case without arc costs arising in the maintenance process of aircraft and proposed exact and heuristic methods.

To the best of our knowledge, there are no specialized heuristics for the VRPU itself. However, the problem has practical relevance since it appears in routing scenarios where all items have the same size and the vehicle capacity sets only a limit on the number of items transported simultaneously. These scenarios include the distribution of goods having equal size, e.g., containers of food or parcels and the transportation of workers or students between their homes and factories or schools, respectively. Another area of application is the routing of aircraft with maintenance stops as considered by Talluri [97].

3.2 The multi-insertion neighborhood for the VRPU

The VRPU can be stated as follows. Given a directed graph $G = (V, A)$ with node set $V = \{0, 1 \dots, n\}$ consisting of n customers and a single depot node 0. The arcs $(i, j) \in A$ are weighted with costs or distances c_{ij}. We do not assume that the costs satisfy the triangle inequality, i.e., $c_{ih} + c_{hj} \geqslant c_{ij}$ for all nodes i, j, h is not required. Additionally, there are K vehicles with capacity C available at the depot node 0. The number K may either be bounded ($K < n$) or unbounded ($K = n$). The goal is to find a set of at most K tours with minimum total cost such that every tour starts and ends at the depot 0, every customer is visited exactly once, and in each tour at most C customers are visited.

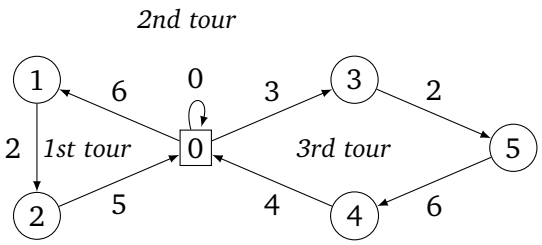
For every tour $k \in \{1, \dots, K\}$ we denote by ℓ_k the number of customer nodes in that tour. A tour k is called *active* if $\ell_k > 0$, otherwise it is *inactive*. For a solution S consisting of K tours, we denote by $\ell(S)$ the tuple $(\ell_1, \dots, \ell_K)$ and by $c(S)$ its costs.

As the VRPU is a special case of the CVRP, all mathematical formulations for the CVRP (e.g., Toth and Vigo [98]) can also be used to model the VRPU. However, depending on the formulation, the model may be simplified with respect to the capacity constraints.

Example 3.1. *Consider the VRPU instance shown in Figure 3.1 with* $n = 5$ *customers,* $K = 3$ *vehicles with capacity* $C = 3$*, and costs* c_{ij} *as displayed in Figure 3.1a. A feasible solution* S *for this instance with* $\ell(S) = (2, 0, 3)$ *is shown in Figure 3.1b. The first and third tours are active while the second tour is inactive. The total cost of* S *is* $c(S) = 13 + 0 + 15 = 28$.

	0	1	2	3	4	5
0	0	6	5	3	4	2
1	6	0	2	9	5	8
2	5	2	0	8	4	7
3	3	8	8	0	5	2
4	4	6	4	5	0	6
5	2	8	6	2	6	0

(a) Costs c_{ij}



(b) A feasible solution S

Figure 3.1: Example instance for the VRPU.

For the CVRP and other *vehicle routing problems* (VRPs) several neighborhoods have been suggested in the literature (cf. Funke et al. [43]). The neighborhoods $\mathcal{N}_{\text{relocate}}$ and $\mathcal{N}_{\text{swap}}$ are two simple and well-known neighborhoods. In $\mathcal{N}_{\text{relocate}}$, a neighbor solution is obtained by removing a single node and reinserting it into a possibly different tour at an arbitrary position while respecting the vehicle capacity C. In $\mathcal{N}_{\text{swap}}$, instead of a single node, two distinct nodes are selected and their positions are swapped, either in a single tour or across a pair of tours. Due to the unit demands, swapping is always feasible w.r.t. the vehicle capacity.

(a) Neighbor in $\mathcal{N}_{\text{relocate}}$ (b) Neighbor in $\mathcal{N}_{\text{swap}}$

Figure 3.2: Examples for the neighborhoods $\mathcal{N}_{\text{relocate}}$ and $\mathcal{N}_{\text{swap}}$ based on the solution S from Figure 3.1b.

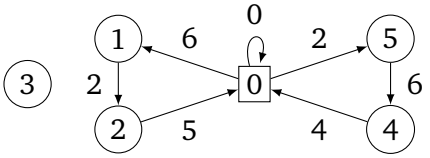
Example 3.2. *To illustrate these two neighborhoods, two neighbor solutions based on the solution S from Figure 3.1b are illustrated in Figure 3.2. The solution in Figure 3.2a is achieved by relocating the node* 4 *from the third to the first tour when using* $\mathcal{N}_{relocate}$*. On the other hand, the solution in Figure 3.2b is obtained by swapping the nodes* 2 *and* 4*, using* $\mathcal{N}_{swap}$*.*

In Angel et al. [3], the exponential MIN was introduced as follows. For a given solution of the VRPU consisting of a set of tours, first a subset of so-called *mobile nodes* is chosen. The remaining nodes and the depot are called *fixed nodes*. A neighbor solution is a set of tours where each mobile node has been inserted between two fixed nodes and where at most one mobile node is inserted between two fixed nodes. If m mobile nodes and m insertion positions have been chosen, there are $m!$ permutations of the nodes to be placed in the insertion positions. Thus, the number of neighbors in this neighborhood may be exponential in the number of nodes. However, in [3] it was shown that a best neighbor in this neighborhood can be calculated in polynomial time given that a set of mobile nodes has been chosen.

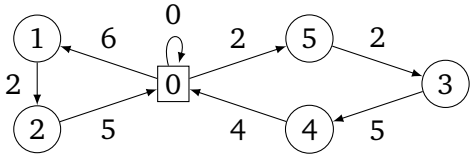
We extend this neighborhood by also varying the subsets of mobile nodes in the first step. For $m \in \mathbb{N}, m < n$ and a given solution S, let $\text{MIN}^m(S)$ be the neighborhood consisting of all neighbors where an arbitrary subset of m nodes in S is chosen as mobile and reinserted afterwards according to the requirements of the MIN. Note that there are $\binom{n}{m}$ possibilities to choose m mobile nodes among the n customers, however, not all choices lead to feasible solutions as we show in Section 3.3.2. Additionally, let $\text{MIN}^m_+(S)$ be the neighborhood

consisting of all neighbors that can be derived by choosing up to m nodes as mobile in S, i.e., $\mathrm{MIN}^m_+(S) = \bigcup_{\mu=1}^{m} \mathrm{MIN}^\mu(S)$.

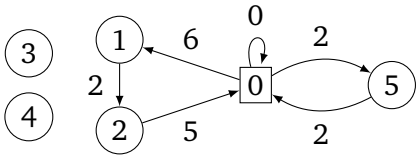
Note that in the original neighborhood defined in [3], the number of tours in a solution cannot increase. Either it remains the same or it decreases if in a tour the depot is the only fixed node and no mobile node is inserted into this tour. Since sometimes it may be advantageous to increase the number of tours, we additionally consider insertion positions in currently inactive tours.



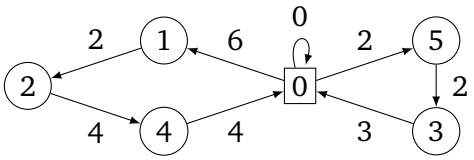
(a) S after deleting node 3



(b) Neighbor solution S_1



(c) S after deleting nodes 3 and 4



(d) Neighbor solution S_2

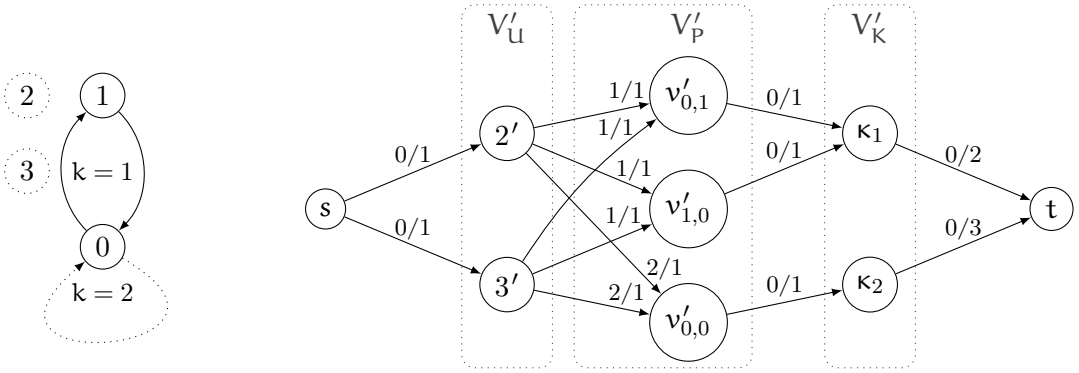
Figure 3.3: Examples for neighbors in $\mathrm{MIN}^1(S)$ and $\mathrm{MIN}^2(S)$ based on the solution S from Figure 3.1b.

Example 3.3. *Two example neighbors based on the solution S from Figure 3.1b are shown in Figure 3.3. Removing the mobile node 3 results in the partial solution show in Figure 3.3a. This corresponds to a move in the neighborhood MIN^1, because only a single mobile node is chosen. Reinserting node 3 between the two fixed nodes 5 and 4 results in the neighbor solution $S_1 \in \mathrm{MIN}^1(S)$ shown in Figure 3.3b. A neighbor in MIN^2 is obtained by removing and reinserting two mobile nodes. The remaining partial solution obtained by removing mobile nodes 3 and 4 as illustrated in Figure 3.3c. After reinserting node 3 between the fixed nodes 5 and 0 and reinserting node 4 between the fixed nodes 2 and 0, the neighbor $S_2 \in \mathrm{MIN}^2(S)$ shown in Figure 3.3d is obtained. Both neighbor solutions S_1, S_2 are contained in $\mathrm{MIN}^2_+(S) = \mathrm{MIN}^1(S) \cup \mathrm{MIN}^2(S)$.*

The neighborhood MIN^m is a generalization of the simple neighborhoods introduced above. It is easy to see that MIN^1 corresponds exactly to $\mathcal{N}_{\text{relocate}}$. A swap of two non-adjacent nodes in a tour can be simulated with MIN^2. In order to swap two adjacent nodes, we may instead relocate one of them. Thus, $\mathcal{N}_{\text{swap}}$ is contained in MIN^2_+, however, the reverse is not true as MIN^2_+ may contain several additional neighbors. On the other hand, the well-known 2-opt neighborhood for the TSP (remove and insert 2 arcs) is not contained in MIN^m_+ for any m, since the subsequence between the two exchanged arcs is effectively reversed in a 2-opt move.

As shown by Angel et al. [3], after a subset of mobile nodes has been selected, a best insertion can be calculated in polynomial time. Let $\hat{S}$ be a partial solution derived from a solution S by removing a set U of mobile nodes. Then, a minimum-cost insertion of the mobile nodes U into $\hat{S}$ can be calculated by reducing the problem to the so-called *minimum-weight particular restricted complete matching* problem (MIN-WPRCM). The MIN-WPRCM itself can be reduced to a specific *minimum-cost network flow problem* (MCFP), which is then solved in polynomial time. The corresponding MCFP s-t-network $G' = (V', A')$ is constructed as follows. The node set $V' = \{s, t\} \cup V'_U \cup V'_P \cup V'_K$ is composed of four disjoint

sets: the source and sink nodes $\{s, t\}$, a set of nodes $V'_U = \{u' : u \in U\}$ representing the mobile nodes U, a set of nodes $V'_K = \{\kappa_k : k \in \{1, \dots, K\}\}$ representing the tours of $\hat{S}$ and a set of nodes V'_P representing the insertion positions. The set V'_P contains a node v'_{ij} for every pair $(i, j) \in V \times V$ if i is visited directly before j in a tour of $\hat{S}$. Additionally, V'_P contains a copy of the node v'_{00} for each empty tour in $\hat{S}$. Each arc $a \in A'$ is associated with a cost c'_a and a capacity $Cap_a \geqslant 0$. The arc set $A' = A'_U \cup A'_P \cup A'_K \cup A'_C$ is composed of four disjoint sets. The arcs $A'_U = \{s\} \times V'_U$ connect the source node to all mobile nodes. The *insertion arcs* $A'_P = V'_U \times V'_P$ connect all mobile nodes to all insertion positions. The arcs $A'_K = \left\{(v'_{ij}, \kappa_k) : v'_{ij} \in V'_P, \text{tour } k \text{ contains arc } (i, j)\right\}$ connect all insertion positions v'_{ij} to the node representing the tour k that contains the nodes i and j. The *capacity arcs* $A'_C = V'_K \times \{t\}$ connect the tour nodes to the sink node. All arcs $a \in A'_U \cup A'_K$ have zero cost and capacity $Cap_a = 1$. The insertion arcs $(u', v'_{ij}) \in A'_P$ have cost $c'_{u'v'_{ij}} = c_{iu} + c_{uj} - c_{ij}$ and capacity $Cap_{u'v'_{ij}} = 1$. The capacity arcs $(\kappa_k, t) \in A'_C$ have zero cost and capacity $Cap_{\kappa_k t} = C - \ell_k$, i.e., the capacities of the arcs equal the remaining capacities of the respective tours in $\hat{S}$.



(a) Partial solution $\hat{S}$ and mobile node set U

(b) MCFP s-t-network

Figure 3.4: Example for the construction of the MCFP network. The arcs in Figure (b) are labeled c'_a / Cap_a.

Example 3.4. *We consider a VRPU instance with $K = 2$ vehicles with capacity $C = 3$, and unit costs, i.e., for arcs $(u, v) \in A$ with $u \neq v$ the costs are $c_{uv} = 1$ and $c_{vv} = 0$ otherwise. In Figure 3.4a, a partial solution $\hat{S}$ and the mobile node set $U = \{2, 3\}$ are shown. The corresponding MCFP network is given in Figure 3.4b.*

Angel et al. [3] consider the basic vehicle routing model as described in the beginning of this section. However, heterogeneous vehicle capacities and vehicle fixed costs may be incorporated into the model such that the ability to calculate minimum-cost insertions in polynomial time is maintained.

Heterogeneous capacities Instead of homogeneous vehicles with capacity C, the vehicles $k \in \{1, \dots, K\}$ may have individual capacities C_k. To incorporate these capacities, we associate each tour with exactly one specific vehicle and modify the capacity arcs $(\kappa_k, t) \in A'_C$ such that $Cap_{\kappa_k, t} = C_k - \ell_k$ for each tour k. All other arcs, capacities and costs remain unchanged.

Vehicle fixed costs In the basic model, the maximum number of vehicles K is fixed and only the travel costs determine the total solution cost. To consider a trade-off between the number of vehicles used and the travel costs, so-called fixed costs F_k can be associated with any active tour k. Recall that every node $v'_{00} \in V'_P$ represents a single insertion position in a currently inactive tour and using such an insertion position turns an inactive tour into a tour with a single customer. Therefore, we associate such an insertion with the additional fixed costs F_k and the costs become $c'_{u'v'_{00}} = c_{0u} + c_{u0} + F_k$ for every $u \in U$.

3.3 Theoretical properties of the multi-insertion neighborhood

In this section, we study the MIN from a theoretical point of view. In Section 3.3.1 we show that the problem of finding a best set of mobile nodes for a given solution is strongly $\mathcal{NP}$-hard. In Section 3.3.2 we consider the question of how many nodes may be chosen as mobile such that a feasible reinsertion can be performed. The connectivity of the neighborhood and the quality of its local optima are discussed in Sections 3.3.3 and 3.3.4, respectively.

3.3.1 Calculating a best set of mobile nodes is $\mathcal{NP}$-hard

As described above, the techniques in Angel et al. [3] can be used to efficiently determine best insertion positions for a fixed subset of mobile nodes. In this subsection, we consider the more general problem of finding a best subset of mobile nodes which leads to a best neighbor in the complete neighborhood MIN^m_+. We prove that this problem is strongly $\mathcal{NP}$-hard.

Proposition 3.1. *Calculating a best neighbor solution in the neighborhood MIN^m_+ is strongly $\mathcal{NP}$-hard.*

Proof. We use a reduction of *vertex cover* (VC), which is one of the classical 21 problems that are proven to be strongly $\mathcal{NP}$-hard by Karp [63]. A VC instance consists of an undirected graph $G' = (V', E')$ and a threshold $y \in \mathbb{N}$. It has to be decided whether a node subset $W \subseteq V'$ with $|W| \leqslant y$ exists such that $v \in W$ or $w \in W$ for every edge $\{v, w\} \in E'$.

Given an arbitrary instance of VC, we construct a VRPU instance with a complete graph $G = (V, A)$, and a feasible initial solution S, represented by K tours starting and ending at the depot. We want to decide whether there is a neighbor solution $S' \in \mathrm{MIN}^m_+(S)$ with costs $c(S') \leqslant y$.

W.l.o.g. we may assume that G' does not contain isolated nodes (they can never cover an edge). Let $M > |V'|$ be a large number. We define a VRPU instance with $n = 2|V'| + 3|E'|$ customers and vehicle capacity $C = \max_{u \in V'} \{|\mathcal{N}(u)|\} + 2$, where $\mathcal{N}(u) = \{u' : \{u, u'\} \in E'\}$ denotes the nodes adjacent to u in G'. All costs between nodes $v, w \in V$ with $v \neq w$ which are not explicitly specified below, are set to $c(v, w) = M$.

The graph G of the VRPU instance contains the following nodes V:

- One node 0 corresponding to the depot.
- For each original node $u \in V'$, we introduce two nodes $v_{u_a}, v_{u_b} \in V$. The arc costs are set to $c(0, v_{u_a}) = c(v_{u_b}, 0) = 0$ and $c(v_{u_a}, v_{u_b}) = 1$.
- For each original edge $z = \{u, w\} \in E'$, there are one node $e_z \in V$ as well as two node copies $v^{e_z}_u, v^{e_z}_w \in V$ of the nodes u and w which can cover edge z. The arc costs are $c(0, v^{e_z}_u) = c(v^{e_z}_u, e_z) = c(e_z, v^{e_z}_u) = c(v^{e_z}_u, 0) = 0$ and $c(0, v^{e_z}_w) = c(v^{e_z}_w, e_z) = c(e_z, v^{e_z}_w) = c(v^{e_z}_w, 0) = 0$.

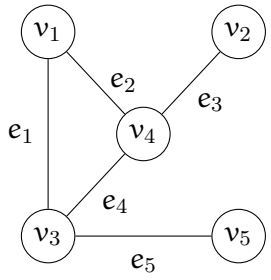
The initial solution S contains $K = |V'| + |E'|$ tours and is created as follows:

- Each node $e_z \in V$ representing an edge $z \in E'$ is arranged in a separate, so-called *edge tour*. The costs are $c(0, e_z) = M$ and $c(e_z, 0) = 0$, i.e., each of these tours has total costs of M.

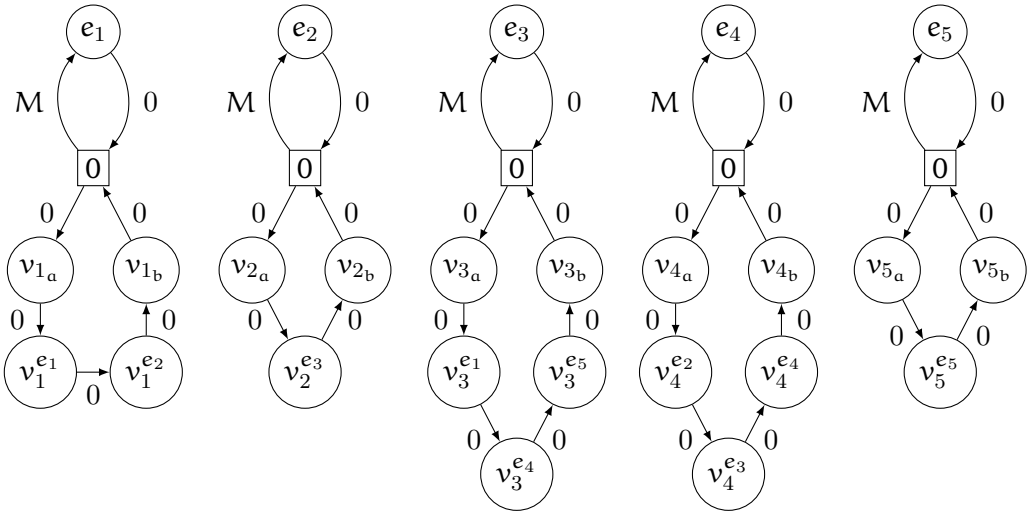
- For every node $u \in V'$ there is a so-called *node tour* with $|\mathcal{N}(u)| + 2$ nodes. The node v_{u_a} is visited at the beginning of this tour. After it, the $|\mathcal{N}(u)|$ nodes $v_u^{e_1}$, $v_u^{e_2}$, ..., $v_u^{e_{|\mathcal{N}(u)|}}$ belonging to the edges $e_1, e_2, \ldots, e_{|\mathcal{N}(u)|}$ incident to node u are visited. At the end of the tour the node v_{u_b} is visited. We set the arc costs $c(v_{u_a}, v_u^{e_1}) = c(v_u^{e_1}, v_u^{e_2}) = \ldots = c(v_u^{e_{|\mathcal{N}(u)|}}, v_{u_b}) = 0$, i.e., each of these tours has total costs of 0.

Obviously, the transformation is polynomial in the input length of VC.

In Figure 3.5, we illustrate the reduction by a small example using the graph G' with 5 nodes and 5 edges from Figure 3.5a. The resulting initial solution S is shown in Figure 3.5b where the depot node 0 is reproduced multiple times for better visibility. At the top, the *edge tours* with costs of M are shown, below, the *node tours* with costs 0 can be found.



(a) Graph G'



(b) Initial solution S corresponding to G'

Figure 3.5: Example for the reduction

In the following, we show that there is a VC of size at most y in G' if and only if there is a neighbor solution $S' \in \mathrm{MIN}_+^m(S)$ with $m = 2|E'|$ having costs $c(S') \leqslant y$.

"⇒": Starting with a vertex cover $W = \{v_1, \ldots, v_\gamma\} \subseteq V'$ with $\gamma \leqslant y$ nodes, a feasible neighbor solution S' can be created by selecting the nodes $v_1^{e_1}$, ..., $v_1^{e_{|\mathcal{N}(v_1)|}}$, ..., $v_\gamma^{e_1}$, ..., $v_\gamma^{e_{|\mathcal{N}(v_\gamma)|}} \in V$ as mobile. Since we select at most $\sum_{\mu=1}^{\gamma} |\mathcal{N}(v_\mu)| \leqslant 2|E'| = m$ mobile nodes, the resulting neighbor S' is contained in $\mathrm{MIN}_+^m(S)$.

The selected mobile nodes are inserted in the *edge tours* belonging to the edges which are incident to the corresponding nodes in the original graph. Because each node $e_z \in V$ corresponding to edge $z \in E'$ is in its own tour, there are exactly two insertion positions. If in the first position no mobile node has been inserted yet, we insert the node there, otherwise, in the other. Thus, both node copies can be inserted if an edge is covered by two nodes. Because W is a VC, each edge is covered by at least one node, which avoids the large costs M per edge. Consequently, all *edge tours* have costs 0. The *node tours* corresponding to the nodes not contained in W remain unchanged and hence have costs 0. In contrast, each *node tour* corresponding to a node $u \in W$ contains only the nodes v_{u_a} and v_{u_b} with total costs of 1. Because these are the only tours with positive costs and at most y nodes are in the vertex cover, $c(S') \leqslant y$ holds.

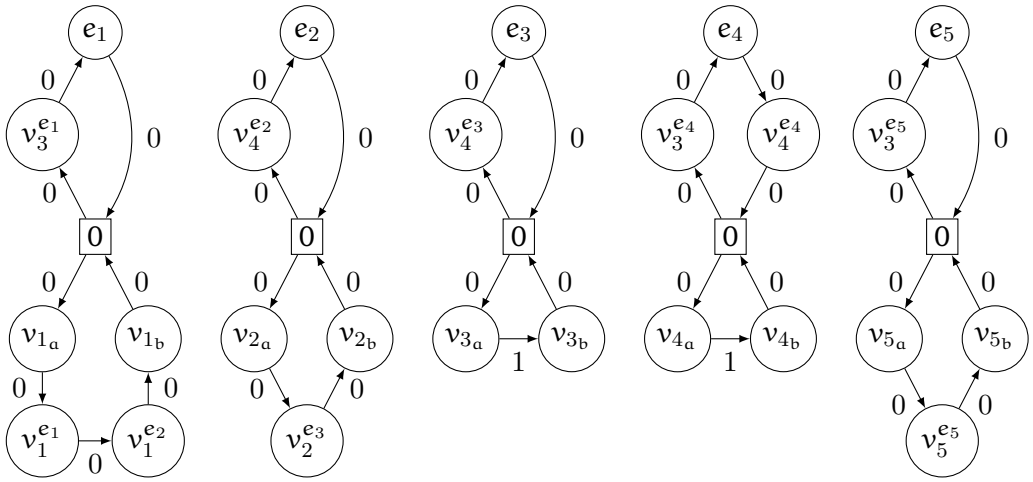


Figure 3.6: Neighbor solution $S' \in \mathrm{MIN}_+^m(S)$

In Figure 3.6, we show a neighbor solution S' with $c(S') = 2$ corresponding to the vertex cover $W = \{v_3, v_4\}$ of size 2 for the graph G' from Figure 3.5a. Note that here edge e_4 is covered by two nodes.

"$\Leftarrow$": Conversely, we show that from a neighbor solution S' with $c(S') \leqslant y$, a VC W with at most y nodes can be derived.

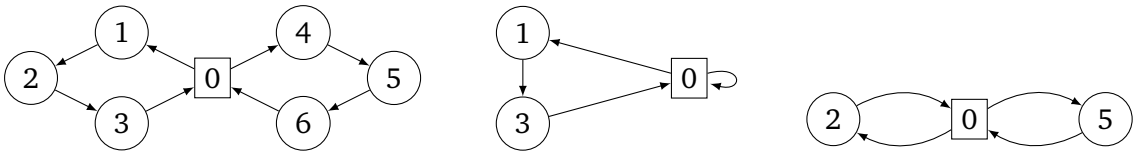
If the *node tour* belonging to an original node $u \in V'$ in the neighbor solution S' is changed, we include node u into the set W. On the other hand, u is not included in W if the tour is unchanged. First of all, it can be stated that in such a neighbor solution S' the edge nodes e_z for all $z \in E'$ are never chosen as mobile. Otherwise, the large costs $M > y$ would arise when inserting e_z into a *node tour* because e_z would be adjacent to at least one node that is not intended to cover the corresponding edge. Likewise, the cost M may not arise in the *edge tour* belonging to e_z. Therefore, a node covering e_z must have been chosen as mobile and inserted directly next to e_z in S'. Because two mobile nodes cannot be directly inserted next to each other, e_z is inevitably a fixed node. The resulting set W is a VC because each edge is covered to avoid the costs M in the corresponding *edge tour*. If nodes from a *node tour* are selected as mobile, they must be reinserted either in their original positions or in an *edge tour* whose edge is covered by the node. Otherwise, the costs M would arise. If at least one node has been moved from a *node tour* to an *edge tour*, this causes costs of at least 1. For every *node tour* corresponding to a node $u \in V'$, the nodes v_{u_a} and v_{u_b} are never adjacent in the initial solution S, because G' does not contain isolated nodes. Thus, in S' the costs of *node tours* corresponding to nodes not contained in the set W are 0. Because $c(S') \leqslant y$ holds, at most y *node tours* have been changed. Thus, the set W also contains at most y nodes. □

3.3.2 Mobile node selection

In this subsection, we study in more detail how many nodes may be chosen as mobile such that a feasible reinsertion exists. In the exponential removal-insertion neighborhood for the TSP suggested by Punnen [87] and Gutin [55], the nodes in a tour are first partitioned into two sets: a set of mobile nodes, and the remaining fixed nodes. Then, the neighborhood contains all solutions where each mobile node is inserted between two fixed nodes. Obviously, in order to have a sufficient number of insertion positions, in a TSP with n nodes at most $\lfloor n/2 \rfloor$ nodes may be chosen as mobile. However, in the VRPU case the situation is more involved. In the following we show that the maximum number of nodes which may be chosen as mobile, depends on the instance as well as the current solution's number of tours. First, we consider a solution with a single tour containing all n customers. This corresponds to a TSP tour with $n + 1$ nodes including the customers and the depot, such that at most $\lfloor (n + 1)/2 \rfloor$ customer nodes can be removed. Adding another tour to the solution introduces another copy of the depot and therefore another arc where a single customer node can be inserted. We conclude that in the case of K tours the number of mobile nodes can be at most $\lfloor (n + K)/2 \rfloor$.

Punnen's exponential neighborhood for the TSP permits a feasible reinsertion for any mobile node set of size $m \leqslant n/2$. Contrary to that, for the more general VRPU with $C < n$, there may be no feasible reinsertions for a given solution and some mobile node sets of size $m \leqslant \lfloor (n + K)/2 \rfloor$. This is due to the fact that the number of insertion positions does not only depend on the number of available arcs in the remaining tours, but also on the capacities of the vehicles.

Example 3.5. *Consider the initial solution S shown in Figure 3.7a for an instance with $n = 6$ nodes and $K = 2$ vehicles of capacity $C = 3$.*



(a) Solution S (b) Partial solution $\hat{S}_1$ (c) Partial solution $\hat{S}_2$

Figure 3.7: Existence of neighbors depends on the selected mobile nodes.

Not every subset of $m = \lfloor (n + K)/2 \rfloor = \lfloor (6 + 2)/2 \rfloor = 4$ mobile nodes can be chosen, as the example in Figure 3.7b shows, where the partial solution $\hat{S}_1$ results from removing the mobile node set $U = \{2, 4, 5, 6\}$. Only one mobile node can be inserted in the left tour due to the limited tour capacity. As the right tour is inactive, it provides only a single insertion position. Overall, only two mobile nodes can be inserted in the partial solution $\hat{S}_1$, even though there are four mobile nodes. Thus, there is no valid neighbor solution for the chosen set of mobile nodes. On the other hand, choosing the nodes $U = \{1, 3, 4, 6\}$ as mobile results in the partial solution $\hat{S}_2$ as shown in Figure 3.7c. There are two arcs in each tour. In addition, the vehicle capacity in each tour allows the insertion of two mobile nodes. Thus, there are valid neighbor solutions because all four mobile nodes can be reinserted. This example shows that the existence of neighbor solutions does not only depend on the number of mobile nodes, but also on the actually selected mobile nodes.

For the VRPU a feasible reinsertion for all mobile node sets $U \subset V$ with $|U| = m$ is only guaranteed for $m = 1$. To see this, consider a solution S with $\ell(S) = (C, \ldots, C, 2)$, i.e., $K - 1$ full tours and a single tour K with $\ell_K = 2$. Choosing both nodes from tour K as mobile,

leaves a single arc for insertions, i.e., this mobile node set U with $|U| = 2$ does not permit a feasible reinsertion.

In the following, we investigate the maximum number of mobile nodes such that at least one mobile node set of this size allows for a feasible reinsertion. Since all results can be obtained for the more general setting of heterogeneous vehicle capacities C_k, we study this setting.

Proposition 3.2. *Given a solution S with n customer nodes, $K \leqslant n$ tours and odd capacities C_k, then for all $m \leqslant \lfloor (n+K)/2 \rfloor$ there exists a set of mobile nodes $U \subset V$ with $|U| = m$ that permits a feasible reinsertion.*

Proof. Let S be a solution. For each tour k with $\ell_k > \lfloor C_k/2 \rfloor$, we can remove customer nodes until $\ell_k = \lfloor C_k/2 \rfloor$ since the number of remaining arcs $\ell_k + 1 = \lceil C_k/2 \rceil$ is exactly as large as the maximal number of nodes removed from the tour. Therefore, each removed node can be reinserted into the tour it was removed from. Furthermore, the number of fixed and depot nodes in each tour is at most as large as the remaining capacity. Hence, all arcs can be used for insertions without violating any capacity constraint.

Assuming that the total number of selected mobile nodes m' is still smaller than $\lfloor (n+K)/2 \rfloor$, we observe that the total number of remaining fixed and depot nodes $f > \lceil (n+K)/2 \rceil$ satisfies $f - m' \geqslant 2$. Since the number of remaining fixed and depot nodes f equals the number of insertion positions p and $p - m' \geqslant 2$ holds, we can select another mobile node decreasing p and increasing m' by one, respectively. Afterwards, we still have $p - m' \geqslant 0$ and a feasible insertion of all mobile nodes is ensured. The argument can be iterated until $m' = \lfloor (n+K)/2 \rfloor$. □

For even capacities C_k the situation is more complex as shown in the following example.

Example 3.6. *Consider an instance with $n = 16$ nodes, $K = 5$ tours, and capacity $C = 4$. Let S_1, S_2 be two solutions with $\ell(S_1) = (4,3,3,3,3)$ and $\ell(S_2) = (4,4,4,4,0)$. While 10 mobile nodes can be selected in S_1 (e.g., two from each tour), at most 9 mobile nodes can be selected in S_2. Removing 10 mobile nodes from the first four tours of S_2 implies that at least two nodes need to be moved to the fifth tour. This is impossible as the fifth tour has only a single insertion position. Hence, for even capacities the maximal number of mobile nodes also depends on the distribution of nodes among the tours.*

However, we can establish the following general bound, independent of the actual node distribution among the tours.

Proposition 3.3. *Given a solution S with n customer nodes, $K \leqslant n$ tours and even capacities C_k, then for all $m \leqslant \lceil n/2 \rceil$ there exists a mobile node set $U \subset V$ with $|U| = m$ that permits a feasible reinsertion.*

Proof. We consider each tour k with $\ell_k \leqslant C_k$ individually. Selecting $\lceil \ell_k/2 \rceil$ mobile nodes leaves $\lfloor \ell_k/2 \rfloor + 1 \geqslant \lceil \ell_k/2 \rceil$ arcs and at least $\lceil \ell_k/2 \rceil$ of available capacity. Thus, if we select at most $\lceil \ell_k/2 \rceil$ nodes in each tour a feasible reinsertion exists. Note that selecting exactly $\lceil \ell_k/2 \rceil$ nodes in every tour ensures that at least $\lceil n/2 \rceil$ mobile nodes have been selected in total. □

3.3.3 Connectivity of the multi-insertion neighborhood

In this subsection, we study the question for which values of m the neighborhood MIN^m is connected, i.e., each solution can be iteratively transformed into any other solution by a finite number of moves in the neighborhood. Note that exactly m mobile nodes must be selected for a move in MIN^m. However, it is often useful if also only $\mu < m$ mobile nodes

can be chosen. For this purpose, we introduce the concept of so-called *dummy nodes*. These are mobile nodes, which are chosen in addition to the μ actual selected mobile nodes and which can be inserted in their original positions in the target solution. For example, if only $\mu = 3$ mobile nodes are to be selected in MIN^{10}, we must choose $d = m - \mu = 10 - 3 = 7$ additional dummy nodes, which can be reinserted in their original positions.

The following lemma guarantees the existence of a minimum number d of possible dummy nodes. For that matter, two cases are distinguished. In the first case, the μ mobile nodes can be inserted in arbitrary positions. In the second case, the μ nodes are restricted to be inserted at the beginning or at the end of a tour, which implies that some more dummy nodes can be selected.

Lemma 3.1. *When using the neighborhood* MIN^{μ}*, at least* $d = \lceil (n - 3\mu)/2 \rceil$ *dummy nodes can be selected. If all* μ *nodes are restricted to be inserted at the beginning or at the end of a tour, at least* $d = \lceil n/2 \rceil - \mu$ *dummy nodes can be selected.*

Proof. Let S, S' be two arbitrary solutions. We show how additional dummy nodes can be selected to reach S' in the neighborhood $\mathrm{MIN}^{\mu}(S)$. The selection of these dummy nodes depends in particular on the target solution S'.

A node is called *blocked* if it is adjacent to one of the μ mobile nodes in the target solution S'. When using the MIN, two mobile nodes cannot be inserted next to each other. Thus, a valid dummy node must not be blocked. If one of the μ mobile nodes is located at the beginning or at the end of a tour in S', it is adjacent to at most one node besides the depot. Otherwise, at least one of the μ mobile nodes is neither the first nor the last node in a tour and has therefore exactly two adjacent nodes in S'. Hence, there are at most 2μ blocked nodes in the general case and at most μ blocked nodes if all μ mobile nodes are inserted at the beginning or at the end of a tour.

All μ mobile nodes and the corresponding blocked nodes cannot be chosen as dummy nodes. Consequently there remain $n - \mu - 2\mu = n - 3\mu$ (respectively $n - \mu - \mu = n - 2\mu$) possible dummy node candidates. Because dummy nodes are chosen as mobile, they cannot be inserted next to each other as well. This means that only half of the candidates can be chosen as dummy nodes. Thus, there are at least $d = \lceil (n - 3\mu)/2 \rceil$ (respectively $d = \lceil (n - 2\mu)/2 \rceil = \lceil n/2 \rceil - \mu$) valid dummy nodes that can be selected. □

Example 3.7. *Consider the example shown in Figure 3.8 with* $n = 8$ *nodes and* $\mu = 2$*. Assume that the two mobile nodes* 4 *and* 5 *are removed from the solution* S *and reinserted resulting in the solution* S'*. The nodes* 1*,* 6 *and* 7 *adjacent to* 4 *and* 5 *in* S' *are blocked. Hence, the dummy node candidates* 2*,* 3 *and* 8 *remain and only one of the adjacent candidates* 2 *and* 3 *can be chosen as dummy node. Overall,* $d = 2 \geqslant \lceil (8 - 3 \cdot 2)/2 \rceil = 1$ *dummy nodes can be selected (e.g., nodes* 2 *and* 8*).*

(a) Solution S (b) Solution S'

Figure 3.8: Example for choosing valid dummy nodes when using $\mathrm{MIN}^{\mu} = \mathrm{MIN}^{2}$.

Now we can show that the smaller neighborhood $\mathrm{MIN}^{m'}$ is contained in the neighborhood MIN^{m} for $2 \leqslant m' < m$ if there are a sufficient number of nodes.

Lemma 3.2. *For all* $m \geqslant 2$*, the neighborhood inclusion relationship* $\mathrm{MIN}^{m-1} \subseteq \mathrm{MIN}^{m}$ *holds if* $n \geqslant 3m - 1$.

Proof. To simulate the neighborhood MIN^{m-1} with MIN^{m}, we must be able to select one dummy node in addition to the $\mu = m - 1$ actual chosen mobile nodes. According to Lemma 3.1, at least $d = \lceil (n - 3(m-1))/2 \rceil$ dummy nodes can be selected. Thus, there exists at least one dummy node if $\lceil (n - 3(m-1))/2 \rceil \geqslant 1$ holds which is satisfied for $n \geqslant 3m - 1$. □

This lemma implies that for up to $m \leqslant (n+1)/3$ mobile nodes, MIN^{m} also contains all solutions which can be generated with fewer mobile nodes.

In Gutin and Yeo [56] it was shown that for the TSP with n nodes (n even) the exponential removal-insertion neighborhood in general is not connected if the maximal possible number $n/2$ of nodes is chosen as mobile. However, if less than $n/2$ nodes are chosen, the neighborhood is connected even with the additional requirement that all pairs of chosen mobile nodes are not adjacent in the tour. The following proposition shows that MIN^{m} is connected if up to $m = \lceil n/2 \rceil$ mobile nodes are chosen.

Proposition 3.4. *The neighborhood* MIN^{m} *is connected if* $2 \leqslant m \leqslant \lceil n/2 \rceil$.

Proof. At first, we define a smaller neighborhood which is connected and contained in MIN^{m} for $2 \leqslant m \leqslant \lceil n/2 \rceil$. Let $\mathcal{N}^{\text{first}}_{\text{relocate}} \subset \mathcal{N}_{\text{relocate}}$ denote the neighborhood which allows to remove one arbitrary node and reinsert it at the beginning of an arbitrary tour. Furthermore, let $\mathcal{N}^{\text{first}}_{\text{2relocate}}$ be the neighborhood which allows to remove two arbitrary nodes and reinsert them at the beginning of two different tours. With the neighborhood $\mathcal{N}^{\text{first}}_{\text{2relocate}}$, each distribution of nodes among the tours can be achieved by iteratively relocating nodes. Thus, for each given solution we can transform it into a solution having the desired numbers of nodes in the tours of the target solution. Moreover, the neighborhood $\mathcal{N}^{\text{first}}_{\text{relocate}}$ allows us to establish any sequence of nodes inside of each tour. Hence, $\mathcal{N}^{\text{first}}_{\text{relocate}} \cup \mathcal{N}^{\text{first}}_{\text{2relocate}}$ is connected.

According to Lemma 3.1, at least $d = \lceil n/2 \rceil - \mu$ dummy nodes can be selected when choosing μ mobile nodes and reinserting them at the beginning or at the end of the tours. Consequently, MIN^{m} includes the neighborhoods $\mathcal{N}^{\text{first}}_{\text{relocate}}$ (where $\mu = 1$ mobile node is chosen) and $\mathcal{N}^{\text{first}}_{\text{2relocate}}$ (where $\mu = 2$ mobile nodes are chosen) if at least $m - \mu$ additional dummy nodes can be selected. We have

$$d \geqslant m - \mu \Leftrightarrow \left\lceil \frac{n}{2} \right\rceil - \mu \geqslant m - \mu \Leftrightarrow \left\lceil \frac{n}{2} \right\rceil \geqslant m.$$

Since we must choose at least $\mu = 2$ mobile nodes to simulate $\mathcal{N}^{\text{first}}_{\text{2relocate}}$, we also must have $m \geqslant 2$. Thus, for $2 \leqslant m \leqslant \lceil n/2 \rceil$ the neighborhood MIN^{m} contains $\mathcal{N}^{\text{first}}_{\text{relocate}} \cup \mathcal{N}^{\text{first}}_{\text{2relocate}}$ and hence is also connected.

As mentioned above, for $m = 1$, the neighborhood MIN^{1} corresponds to $\mathcal{N}_{\text{relocate}}$, which in general is not connected. For $m > \lceil n/2 \rceil$, the TSP example from [56] shows that in this situation MIN^{m} may also not be connected. □

In Gutin and Yeo [56] it was additionally shown that for the TSP the diameter of the removal-insertion neighborhood graph is quite small: if among the n nodes, $\lfloor (n-1)/2 \rfloor$ nodes are chosen as mobile, every solution can be reached from any other solution in at most four steps in the neighborhood. Moreover, for any $\mu \leqslant n/2 - 1$ and $\lfloor (n - \mu)/2 \rfloor$ mobile nodes, the diameter of the neighborhood graph is bounded by 8. However, in the situation of more than one vehicle, we do not have a constant diameter since the following example shows that there are instances where at least $\Omega(\log n)$ steps are required.

Example 3.8. *For $\lambda \geqslant 1$ consider a VRPU instance with $n = 2^\lambda$ customers, capacity $C = n$ and a solution S with n non-empty tours, i.e., each tour serves exactly one customer. The target solution S' contains $n - 1$ empty tours and a single tour with n customers. We choose one designated tour in S, into which all customers will be inserted. As the designated tour initially contains 2 arcs, at most 2 customers can be inserted into the designated tour in the first neighborhood move. The resulting tour contains 3 customers and 4 arcs. Generally, in the μ-th neighborhood move, at most 2^μ customers can be inserted into the designated tour and after the μ-th neighborhood move the designated tour contains at most $2^{\mu+1} - 1$ customers. Hence, reaching solution S' from solution S requires at least $\log n = \lambda$ neighborhood moves.*

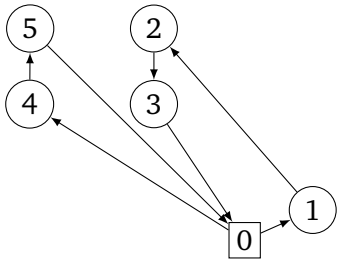
3.3.4 Quality of local optima

In this subsection, we study the quality of local optima w.r.t. MIN^m. When studying larger neighborhoods in comparison to smaller neighborhoods, it is often desirable that the former allow for better local optima. In this way, they compensate for the larger runtime cost associated with searching these larger neighborhoods for improving neighbor solutions. However, as shown in Ergun et al. [37], for some exponential neighborhoods this is not always true. For example, for the TSP the set of local optima w.r.t. the exponential compounded independent 2-opt neighborhood is the same as the set of local optima w.r.t. the much smaller 2-opt neighborhood of size $\mathcal{O}(n^2)$. However, the following example shows that for the VRPU, even when the triangle inequality is satisfied, there are solutions that are locally optimal w.r.t. both neighborhoods $\mathcal{N}_{\text{relocate}}$ and $\mathcal{N}_{\text{swap}}$ of size $\mathcal{O}(n^2)$, but that are not locally optimal w.r.t. the larger neighborhood $\mathrm{MIN}^2_+ \supset \mathcal{N}_{\text{relocate}} \cup \mathcal{N}_{\text{swap}}$.

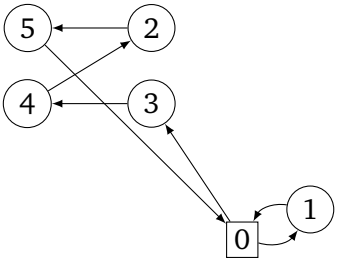
Example 3.9. *Consider an instance with $n = 5$ nodes, $K = 2$ vehicles with capacity $C = 4$, and the cost matrix with $c \geqslant 4$ in Figure 3.9a that satisfies the triangle inequality. Node 1 is very close to the depot, whereas the remaining nodes $\{2, \dots, 5\}$ have the larger distance c to the depot. Therefore, nodes $\{2, \dots, 5\}$ should be together in one tour, such that the large cost c to the depot is incurred only once on the way to these far away nodes and once on the way back.*

	0	1	2	3	4	5
0	0	1	c	c	c	c
1	1	0	c	c	c	c
2	c	c	0	1	3	2
3	c	c	1	0	2	3
4	c	c	3	2	0	1
5	c	c	2	3	1	0

(a) Costs c_{ij} with $c \geqslant 4$



(b) Solution S



(c) Solution $S' \in \mathrm{MIN}^2_+(S)$

Figure 3.9: Local optimum w.r.t. $\mathcal{N}_{\text{relocate}} \cup \mathcal{N}_{\text{swap}}$ which is not locally optimal w.r.t. MIN^2_+

A solution S with $c(S) = 4c + 3$ having nodes $\{4, 5\}$ in the first tour and nodes $\{1, 2, 3\}$ in the second tour, is shown in Figure 3.9b. The nodes within the two tours are arranged in such a way that an improvement of the solution is only possible if the nodes $\{2, \dots, 5\}$ are in the same tour. After any swap move, the nodes $\{2, \dots, 5\}$ are still in different tours, as only two nodes can be swapped. Thus, the solution S is locally optimal w.r.t. the neighborhood $\mathcal{N}_{swap}$. Since only one node can be shifted by a relocate move, but two nodes need to be shifted in order to insert the nodes $\{2, \dots, 5\}$ into a single tour, the solution is also locally optimal w.r.t. $\mathcal{N}_{relocate}$. On the other hand, when using the neighborhood MIN^2_+, an improving neighbor solution can be generated if nodes 2 and 3 are selected as mobile and then reinserted into the left tour between

nodes 4 *and* 5*. The resulting neighbor solution* S' *with* $c(S') = 2c+9$ *is visualized in Figure 3.9c. This is possible, since* MIN^2_+ *allows to perform two independent relocate moves simultaneously.*

3.4 A two-stage approach

The generation of neighbor solutions in MIN^m can be seen as a two-stage process. In the first stage, a set of m mobile nodes is selected and removed from the solution. Afterwards, as described in Angel et al. [3], a best reinsertion of the mobile nodes can be calculated in polynomial time. Since the subproblem of finding a best subset of mobile nodes in the first stage is $\mathcal{NP}$-hard (cf. Proposition 3.1), in this section, we propose different heuristics for this problem.

At first, we use so-called *node evaluators* which assign an individual non-negative priority value to each node. A larger priority value increases the probability that a node is chosen as mobile. Then, so-called *node selectors* choose the actual mobile nodes based on the priority values assigned by a node evaluator.

Note that such a two-stage approach can also be perceived as a LNS approach with a *destroy operator* that removes nodes from a solution and a *repair operator* that subsequently reinserts the removed nodes into the remaining partial solution (cf. Section 2.2.2). As such, the combination of a node evaluator and a selector corresponds to a destroy operator, and the optimal reinsertion corresponds to a repair operator, respectively.

The following three node evaluators have been implemented.

- **Random evaluator (RD)**. Assign to each node a random number, uniformly drawn from the interval $[0, 1] \subset \mathbb{R}$.
- **Node count evaluator (NC)**. Assign to each node the number ℓ_k of nodes in the corresponding tour k that contains the node.
- **Cost savings evaluator (CS)**. Assign to each node i the cost savings of removing the node from its current position. If node i is currently located between the nodes u and v, the cost savings are given by $\max\{c_{ui} + c_{iv} - c_{uv}, 0\}$.

The random evaluator has the advantage that it generates diversified sets of mobile nodes which may help to leave local optima. On the other hand, the random evaluator has the disadvantage that it does not use any information about the current solution. Obviously, in tours with many nodes there exist more insertion positions than in other tours with fewer nodes. Hence, when using the node count evaluator, nodes are chosen as mobile for which more insertion positions exist. The idea of the cost savings evaluator is that large deletion costs indicate that a node is placed in a suboptimal position. Deleting such a node and reinserting it in another position may have a good chance to improve the solution. Note that the 2nd and 3rd evaluator may be used in a static or dynamic way. While in a static procedure, all priority values are calculated only once, in a dynamic procedure, the priority values may change after selecting a mobile node, e.g., if a node from tour k is chosen as mobile, then the number ℓ_k of nodes in tour k is reduced by one.

After an evaluator has assigned a priority value to each node, we use one of the following selection heuristics to choose a feasible set of mobile nodes. For a given number $m \leqslant \lceil n/2 \rceil$, the selectors provide a set of m mobile nodes so that at least one feasible neighbor solution exists. To ensure this, each node selector chooses at most $\lceil \ell_k/2 \rceil$ nodes in each tour k (cf. the proof of Proposition 3.3).

- **General selector (GS).** Choose m nodes with the largest priority values ensuring that at most $\lceil \ell_k/2 \rceil$ nodes are chosen in each tour k.

– **Iterative selector (IS).**

1. Among the tours k with less than $\lceil \ell_k/2 \rceil$ chosen mobile nodes, select a node with largest priority as next mobile node.
2. If in total less than m nodes have been chosen, update the priority values and go to Step 1.

– **Roulette wheel selector (RW).**

1. Among the tours k with less than $\lceil \ell_k/2 \rceil$ chosen mobile nodes, select a node with a probability corresponding to its priority.
2. If in total less than m nodes have been chosen, update the priority values and go to Step 1.

In the general selector, the node evaluator is used in a static way, i.e., it is called only once at the beginning of the selection procedure. Contrary to that, the iterative and roulette wheel selectors use dynamic priority values which are updated each time a node has been chosen as mobile.

Note that with these three selectors, in principle every subset of m nodes can be selected. In order to search for good solutions in MIN^m, we implemented a SA procedure denoted by MIN-SA that is shown in Algorithm 3.1. The SA portion of the method uses a geometric cooling schedule with a cooling factor $f \in [0,1]$. The initial temperature $t = c(S)/p$ depends on the objective function value $c(S)$ of the initial solution and a fixed parameter $p > 0$.

Since good solutions tend to have a smaller number of tours, only initial solutions realizing the minimum possible number of tours $\lceil n/C \rceil$ are constructed. Two different start heuristics are considered. The so-called *first-fit* heuristic successively inserts nodes at the end of the first tour which has sufficient capacity. The so-called *best-fit* heuristic inserts each node in a position with minimum cost increase, i.e., minimum value $\Delta\mathrm{cost} = c_{uv} + c_{vw} - c_{uw}$ for all nodes v when inserted between the nodes u and w.

Algorithm 3.1 MIN simulated annealing (MIN-SA)

Input: Initial solution S, e.g., by *first-fit* or *best-fit*
Output: Solution $S^\star$ with $c(S^\star) \leqslant c(S)$

$S^\star \leftarrow S$ ▷ best solution
$t \leftarrow c(S)/p$
while time or iteration limit is not reached **do**
  Select heuristically a set U of m mobile nodes in S
  Remove nodes U from S to obtain the partial solution $\hat{S}$
  Solve the MCFP modeling the corresponding MIN-WPRCM
  Perform best insertions of nodes U into $\hat{S}$ and obtain neighbor $S' \in \mathrm{MIN}^m(S)$
  if $c(S') < c(S)$ **or** $\mathrm{RAND}(0,1) \leqslant e^{-\left(\frac{c(S')-c(S)}{t}\right)}$ **then**
    $S \leftarrow S'$
  end if
  if $c(S') < c(S^\star)$ **then**
    $S^\star \leftarrow S'$
  end if
  $t \leftarrow t \cdot f$
end while
return $S^\star$

3.5 Computational study

In this section, we present a detailed computational study of the MIN and solution approaches that incorporate the MIN. First, in Section 3.5.1, the quality of local optima is examined in comparison to those in smaller neighborhoods. In Section 3.5.2, different values for the number of chosen mobile nodes and the proposed node selection strategies are evaluated. Finally, in Section 3.5.3, different MIN-based algorithms are compared to other VRP heuristics.

For the following experiments, the four instance sets summarized in Table 3.1 have been used. The set VRPU from [89] (also collected in the TSPLIB [91]) contains VRPU instances with 16–48 nodes. Since these instances are rather small, we derived other sets of unit-demand instances CMTU, GoldenU and XU from the CVRP instances of Christofides et al. [27], Golden et al. [49] and Uchoa et al. [99], respectively. To derive these instances, the cost matrix was reused and all customer nodes were assigned a demand of 1. The original vehicle capacity C was adapted for unit demands by calculating the minimum number of tours K required by any feasible solution to the original CVRP instance. The new capacity $C' := \lceil n/K \rceil$ was set in such a way that the resulting number of tours in a VRPU solution is similar to the number of tours in a corresponding solution to the original CVRP instance.

Table 3.1: Instance sets

	$\lvert V \rvert$			
name	min.	max.	#inst.	derived from
VRPU	16	48	13	Ralphs [89]
CMTU	51	200	14	Christofides et al. [27]
GoldenU	201	484	20	Golden et al. [49]
XU	101	1001	100	Uchoa et al. [99]
all	16	1001	147	all aforementioned instance sets
all_S	16	200	27	VRPU $\cup$ CMTU

In the following experiments, the obtained results are reported relative to the values of *best known solutions* (BKSs). For the set VRPU, optimal solutions can be obtained at the web site [89]. For all derived instance sets CMTU, GoldenU and XU the BKS values originate from our experiments.

All algorithms were implemented in C++ and compiled with *GCC* version 5.5.0. To solve the matching problem MIN-WPRCM by network flow techniques, the *network simplex* algorithm implemented in the *Lemon* library [33] (version 1.3.1) was used. All experiments involving the neighborhood MIN^m and the HGS-CARP solver [100] were run on an Intel Core i7-3770 3.4GHz machine with 64Bit Ubuntu Linux 16.04 and 16GB RAM.

For comparison, we also used the *spreadsheet solver* [36] which is embedded in Microsoft Excel. Since this code is not executable on Linux based systems, all experiments related to the spreadsheet solver were run on a comparable Intel Core i7-6700 3.4GHz machine with 64Bit Windows 10 and 16GB RAM.

3.5.1 Quality of local optima

In the first experiment, we study the quality of local optima from a practical point of view. In Example 3.9 we have seen that theoretically there may be solutions which are locally optimal w.r.t. the smaller neighborhoods $\mathcal{N}_{\text{relocate}}$ and $\mathcal{N}_{\text{swap}}$, but that are not locally optimal w.r.t.

the larger neighborhood MIN^2_+. In the following, we want to see whether such situations also occur in practice on the aforementioned instance sets. For this purpose, we use an iterative improvement procedure to compare MIN^2_+ with the neighborhoods $\mathcal{N}_{\text{relocate}}$, $\mathcal{N}_{\text{swap}}$, and $\mathcal{N}_{\text{relocate}} \cup \mathcal{N}_{\text{swap}}$.

To use MIN^2_+, for each solution considered during the local search process, all subsets with one or two mobile nodes are enumerated. For each subset, a corresponding optimal solution is determined by solving the MIN-WPRCM as a MCFP. Moreover, the best-fit (BF) and first-fit (FF) heuristics are used to calculate initial solutions. The iterative improvement procedure is stopped when a local optimum is reached. For each combination of start heuristic, neighborhood and instance, a single run is performed as the search is deterministic.

Table 3.2 shows the quality of local optima w.r.t. different neighborhoods. The results are reported grouped by the instance sets and the start heuristics. Columns *avg. gap* report the average gaps between the local optimum of a specific start solver and neighborhood combination and the best local optimum found in any combination. Furthermore, column *avg. time* reports the average time to reach a local optimum.

Table 3.2: Quality of local optima w.r.t. different neighborhoods and start heuristics.

		avg. gap [%]			
start	neighborhood	VRPU	CMTU	$\texttt{all}_\texttt{S}$	avg. time [ms]
BF	MIN^2_+	0.70	1.18	0.95	30482.67
	$\mathcal{N}_{\text{relocate}} \cup \mathcal{N}_{\text{swap}}$	6.01	14.42	10.37	34.48
	$\mathcal{N}_{\text{relocate}}$	13.22	19.20	16.32	15.41
	$\mathcal{N}_{\text{swap}}$	9.25	19.81	14.73	14.67
FF	MIN^2_+	1.60	2.57	2.11	32414.74
	$\mathcal{N}_{\text{relocate}} \cup \mathcal{N}_{\text{swap}}$	6.86	13.09	10.09	36.30
	$\mathcal{N}_{\text{relocate}}$	8.31	19.67	14.20	19.74
	$\mathcal{N}_{\text{swap}}$	12.06	17.74	15.01	19.22

It can be seen that MIN^2_+ leads to the best local optima with average gaps smaller than 2.11% over all runs and smaller than 0.95% for the runs starting from an initial solution obtained with the BF heuristic. The second best is the neighborhood $\mathcal{N}_{\text{relocate}} \cup \mathcal{N}_{\text{swap}}$. Note that for a few instances, the neighborhood $\mathcal{N}_{\text{relocate}} \cup \mathcal{N}_{\text{swap}}$ found better local optima than MIN^2_+. On the other hand, the (in general not connected) neighborhoods $\mathcal{N}_{\text{relocate}}$ and $\mathcal{N}_{\text{swap}}$ behave very differently for different instances. This may be due to the fact that with $\mathcal{N}_{\text{relocate}}$ not every distribution of nodes in a solution can be achieved, while with $\mathcal{N}_{\text{swap}}$ only solutions which have the same numbers of nodes in the tours as in the start solution can be reached.

While MIN^2_+ provided the best results, the computation times to reach these local optima were up to three orders of magnitude larger than those of the other neighborhoods. Thus, there is an obvious trade-off between the quality of local optima and the required computation time.

When comparing the results of the two start heuristics, it can be seen that neither BF nor FF outperforms the respective other for all neighborhoods. For example, on average, BF performs better than FF for MIN^2_+, while FF performs better than BF for $\mathcal{N}_{\text{relocate}}$. However, the differences between the start heuristics are comparatively small compared to the differences between the neighborhoods. As BF outperforms FF for MIN^2_+, the BF heuristic is used for all further experiments considering approaches based on $\mathrm{MIN}^{\mathrm{m}}$ or $\mathrm{MIN}^{\mathrm{m}}_+$.

3.5.2 Mobile node selection

The next computational experiment compares the mobile node selection strategies as described in Section 3.4. Each selector was tested along with each node evaluator and different values for the number m of mobile nodes. Instead of using constant values for m, we vary them in relation to the total number of nodes n. Ratios between 5% and 50% with increments of 5% were tested. Each configuration was run in the MIN-SA algorithm with 10 replications and two stopping criteria: (i) a time limit of 60 s and (ii) an iteration limit of 100 000.

Preliminary experiments were conducted to derive the initial temperature t_0 and the final temperature $t_\star$ as functions of the initial solution S_0 and the current best solution $S^\star$, respectively:

$$t_0 = c(S_0) \cdot \frac{1}{-\ln(0.4)}$$
$$t_\star = c(S^\star) \cdot \frac{10^{-5}}{-\ln(10^{-5})}$$

Given the number of remaining iterations and the current best solution, the cooling factor f can be calculated such that the final temperature will be reached in the last iteration. In the case of the time limit criterion, the number of remaining iterations needs to be estimated regularly and the cooling factor is calculated w.r.t. this estimate.

In total, there are too many tested configurations to show all results in detail. At first, we give an overview of the best combinations of selectors and node evaluators, before studying different percentages of mobile nodes. We compare our results with BKSs for each instance.

Table 3.3 shows the average gaps to the BKSs over all instances and percentages of mobiles nodes. Table 3.3a shows the results for the time-based stopping criterion and Table 3.3b for the iteration-based stopping criterion. The average gaps range from around 2% to around 37% for both criteria. Generally, all completely deterministic combinations, i.e., the selectors GS and IS combined with the evaluators NC and CS perform worse than combinations including either randomized selectors, randomized evaluators or both. The reason is that the same nodes are chosen over and over again thereby impeding the exploration of the search space. The evaluator RD should yield equally good results regardless of the chosen selector, yet small differences remain for both stopping criteria.

Table 3.3: Average gaps to the BKSs [%] for different combinations of selectors and node evaluators, averaged over all instances and percentages of mobile nodes.

(a) Stopping criterion with time limit 60 s

	evaluator		
selector	RD	NC	CS
GS	1.98	25.62	23.11
IS	2.07	24.57	36.66
RW	2.11	2.07	4.15

(b) Stopping criterion $100\,000$ iterations

	evaluator		
selector	RD	NC	CS
GS	2.11	25.96	23.06
IS	2.20	24.93	36.66
RW	2.15	2.16	4.36

In the following, the most promising combinations for each evaluator are compared. These include GS-RD representing all combinations of the random evaluator RD and the partially randomized combinations RW-NC and RW-CS.

The results for the most promising combinations of the three evaluators for the time-based and iteration-based criterion are given in Tables 3.4 and 3.5, respectively. The tables report

Table 3.4: Average gaps to the BKSs [%] for the best three node selection heuristics with different percentages of mobile nodes. Time-based stopping criterion with 60 s.

			mobile nodes [%]									
instances	SEL	NE	5	10	15	20	25	30	35	40	45	50
	GS	RD	5.99	1.37	0.09	**0.00**	0.01	0.03	0.04	0.24	0.52	2.01
VRPU	RW	CS	7.03	1.47	0.11	0.05	0.09	0.22	0.46	1.11	2.14	4.51
	RW	NC	7.03	0.94	0.10	0.01	0.05	0.06	0.07	0.24	0.58	2.08
	GS	RD	0.77	0.47	0.37	0.48	0.92	1.57	2.79	4.44	6.55	9.39
CMTU	RW	CS	2.32	1.32	1.64	2.33	3.62	5.33	7.34	10.00	12.64	15.71
	RW	NC	0.62	0.43	**0.35**	0.45	0.74	1.50	2.58	4.35	6.40	9.67
	GS	RD	3.06	0.86	**0.24**	0.27	0.52	0.89	1.58	2.59	3.90	6.14
$\texttt{all}_\texttt{S}$	RW	CS	4.39	1.39	0.97	1.33	2.07	3.08	4.31	6.09	8.02	10.78
	RW	NC	3.44	0.65	**0.24**	0.26	0.44	0.86	1.48	2.54	3.84	6.33

Table 3.5: Average gaps to the BKSs [%] for the best three node selection heuristics with different percentages of mobile nodes. Iteration-based stopping criterion with $100\,000$ iterations.

			mobile nodes [%]									
instances	SEL	NE	5	10	15	20	25	30	35	40	45	50
	GS	RD	6.82	1.00	0.14	**0.01**	0.02	0.05	0.10	0.35	0.94	2.56
VRPU	RW	CS	7.38	1.66	0.41	0.21	0.26	0.40	0.82	1.50	2.96	5.77
	RW	NC	7.38	1.58	0.07	0.06	0.06	0.09	0.15	0.35	1.00	2.80
	GS	RD	1.14	0.53	**0.40**	0.53	0.84	1.42	2.64	4.21	6.21	9.15
CMTU	RW	CS	2.93	1.47	1.70	2.44	3.57	5.27	7.42	9.96	12.60	15.54
	RW	NC	1.35	0.55	0.48	0.51	0.77	1.36	2.40	3.96	6.22	9.45
	GS	RD	3.64	0.74	**0.28**	0.31	0.48	0.82	1.52	2.51	3.89	6.25
$\texttt{all}_\texttt{S}$	RW	CS	4.89	1.55	1.13	1.46	2.11	3.13	4.51	6.24	8.36	11.24
	RW	NC	4.00	1.00	0.30	0.31	0.46	0.80	1.41	2.37	3.92	6.53

the average gaps to the BKSs grouped by the percentage of mobile nodes. For each instance set, the best configuration is highlighted. The results show that the smallest average gaps to the BKSs are obtained with percentages of mobile nodes ranging from 10% to 30%. For the smaller VRPU instances, larger percentages of mobile nodes of approximately 20% perform best, whereas for the larger CMTU instances smaller percentages of 15% are better. This may be partially related to the fact that with increasing instance size, the time required to calculate an optimal insertion of the mobile nodes increases drastically.

On the other hand, the observation is consistent for both stopping criteria and may also originate from differences in the structural properties of the instance sets, e.g., the relationship between the number of nodes, the capacity and the resulting number of tours. Generally, the combinations GS-RD and RW-NC perform well, with only minor differences. In contrast, the combination RW-CS is consistently outperformed by the other two combinations, regardless of the stopping criterion. This is particularly apparent for larger percentages of mobile nodes and the larger CMTU instances.

For the iteration-based stopping criterion (Table 3.5), the completely random combination GS-RD consistently performs best, although the differences to the combination RW-NC are small. With respect to the time-based stopping criterion, when only the VRPU instances are

considered, the combination GS-RD performs best with average gaps of 0%. For the CMTU instances, the combination RW-NC with 15% mobile nodes performs best with an average deviation of 0.35%. Overall, on the set of all instances, the combinations GS-RD and RW-NC with 15% mobile nodes perform equally well with an average gap of 0.24%, nevertheless, the RW-NC combination performs better for 10% and 20% mobile nodes and therefore seems to depend less on the exact percentage. As the combination RW-NC performs better than combination GS-RD on the larger CMTU instances and is less dependent on the mobile node percentages, this combination with 15% mobile nodes is used in further experiments with time-based stopping criteria.

To establish how the time for calculating a best reinsertion scales with the percentage of mobile nodes, we recorded the computation times. Figure 3.10 shows the minimum, average and maximum time over all instances $\texttt{all}_\texttt{S}$ and different mobile node percentages ranging from 5% to 50%. The time required strictly increases with increasing mobile node percentages. Especially, for larger instances the average time required for mobile node percentages of 50% is six times larger than the time required for mobile node percentages of 5%.

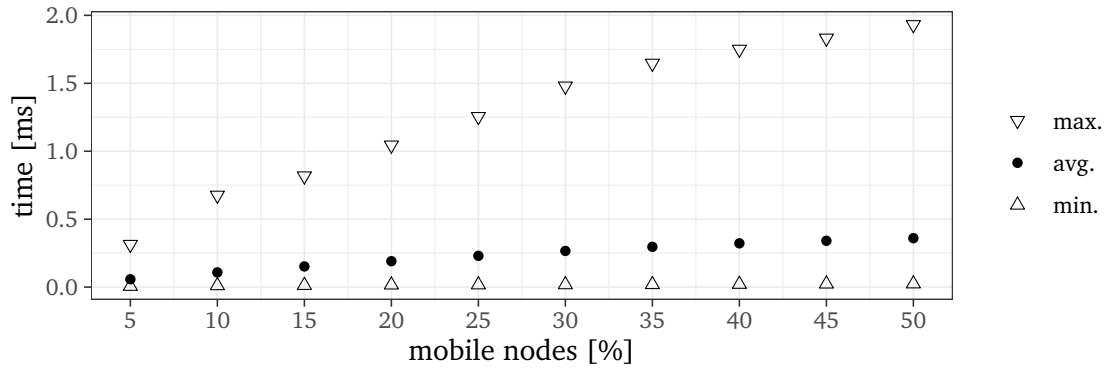


Figure 3.10: Maximum, average and minimum times required to find a best insertion.

To gain some deeper understanding on how the mobile node percentages relate to the performance of the neighborhood, we conducted a further experiment. The solver is modified in such a way that while a regular run is performed, all possible mobile node subsets $U \subseteq V$ are enumerated in each iteration. For each subset, we try to calculate a best insertion. If no feasible insertion exists, we record infeasibility of the subset, otherwise, the resulting neighbor solution's cost is recorded. The experiment was run on the VRPU instances gr-n17-k3 ($n = 17$), gr-n21-k3 ($n = 21$) and gr-n24-k4 ($n = 24$). Figure 3.11 shows the percentages of improving neighbors in the set of feasible neighbors over the first 100 iterations of the process. The resulting patterns are uniform w.r.t. the best ratio of improving to feasible neighbors and are similar among the considered instances. The best average ratios are obtained for $m = 4$ on instance gr-n17-k3, $m = 4$ on instance gr-n21-k3 and $m = 5$ on instance gr-n24-k4. These correspond to mobile node percentages of 23.5%, 19% and 20.8%, i.e., $m \approx n/5$. These findings reflect the results from Table 3.4 where the best results were obtained for percentages ranging from 10% to 30%.

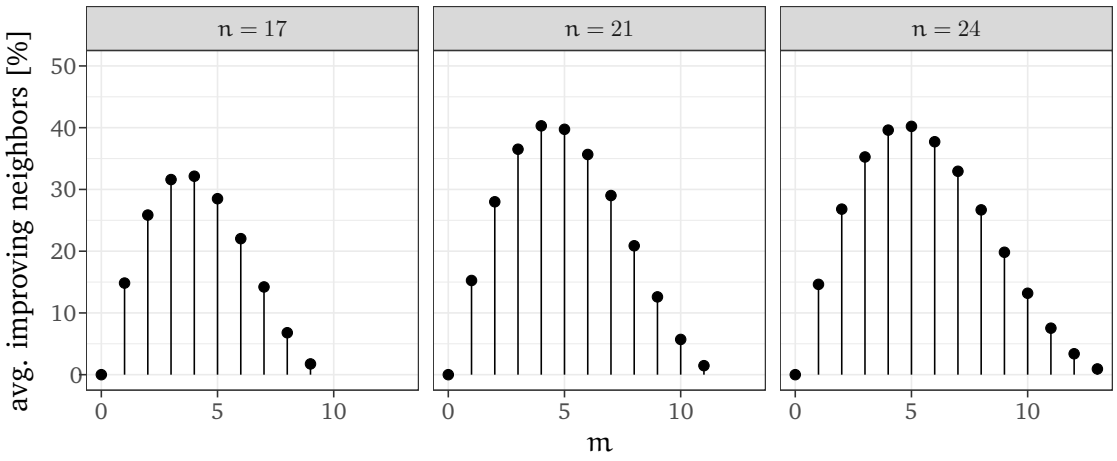


Figure 3.11: Average percentage of improving neighbors for different sizes $\mathfrak{m}$ of feasible subsets $\mathcal{U}$.

3.5.3 Comparison to other VRP heuristics

In this subsection, we compare different algorithms based on the exponential MIN to other heuristics from the literature. As MIN-based algorithms we use

- MIN-SA, corresponding to the proposed SA algorithm, using the best configuration, i.e., the selection procedure RW-NC and 15% mobile nodes,
- MIN-VND, corresponding to MIN-SA, augmented by a *variable neighborhood descent* (VND) with the smaller intra-tour 2-opt, inter-tour relocate and inter-tour string-swap (cf. Funke et al. [43]) neighborhoods.

The second variant was chosen since applications of VND heuristics to various VRPs have been rather successful in the past. They combine multiple neighborhoods to escape from local optima (cf. Section 2.2.1). Since smaller neighborhoods are faster to evaluate than performing one step in the MIN, this may be more efficient when given a time limit for the whole algorithm.

To see whether finding an optimal reinsertion of the mobile nodes in the second stage is really beneficial, we additionally compare the insertion procedure based on exactly solving the MIN-WPRCM with heuristic node reinsertion procedures from the literature. For this, we use two LNS variants:

- LNS-MC uses a parallel minimum-cost insertion heuristic, and
- LNS-R2 uses the regret-2 insertion heuristic of Ropke and Pisinger [92].

Since there are no specialized heuristics for the VRPU, for the comparison we use heuristics for the more general problem with arbitrary demands. Despite the large amount of literature on VRP solvers, only a few competitive implementations are freely available. We decided to use

(i) the HGS-CARP solver of Vidal [100], based on a *genetic algorithm* (GA), implemented in C++, and

(ii) the spreadsheet solver of Erdoğan [36], based on LNS, implemented in Visual Basic and running in Microsoft Excel.

Since the spreadsheet solver only accepts instances with at most 200 nodes, it was only tested on the instance sets VRPU and CMTU. All other solvers were tested on all instance sets. To ensure the comparability of the results, a time limit was set. Because the execution of programs written in Visual Basic is considerably slower than programs produced by optimizing C++ compilers, the developer of the spreadsheet solver recommends to choose a time limit of at least 120 s. To compensate for this speed difference, the three time limits 60 s, 300 s, and 600 s were chosen. For each combination we performed 10 replications because all solvers are randomized.

The computational results are summarized in Table 3.6. The results are grouped by time limits, solvers, and instance sets. For all tested combinations, the average gaps to the BKSs are reported. The smallest average gap over all instances is highlighted for each time limit.

Table 3.6: Results for all instance sets and solvers.

		avg. gap [%]					
time [s]	solver	VRPU	CMTU	all_S	GoldenU	XU	all
60	HGS-CARP	0.17	0.16	0.16	1.74	1.13	1.03
	spreadsheet	0.32	3.63	2.04	–	–	–
	MIN-SA	0.08	0.32	0.20	3.19	0.99	1.14
	MIN-VND	0.11	0.13	**0.12**	0.69	0.45	**0.42**
	LNS-MC	0.28	1.77	1.06	4.95	12.41	9.31
	LNS-R2	0.18	0.40	0.29	3.15	2.05	1.87
300	HGS-CARP	0.17	0.08	0.12	1.02	0.70	0.64
	spreadsheet	0.28	2.39	1.37	–	–	–
	MIN-SA	0.07	0.23	0.15	2.42	0.69	0.83
	MIN-VND	0.11	0.06	**0.09**	0.33	0.20	**0.20**
	LNS-MC	0.25	1.53	0.91	2.77	10.84	7.92
	LNS-R2	0.15	0.27	0.21	2.05	1.57	1.39
600	HGS-CARP	0.17	0.06	0.11	0.79	0.55	0.50
	spreadsheet	0.29	1.88	1.11	–	–	–
	MIN-SA	0.07	0.16	0.12	2.25	0.60	0.73
	MIN-VND	0.11	0.04	**0.08**	0.23	0.15	**0.15**
	LNS-MC	0.22	1.56	0.91	2.17	10.36	7.51
	LNS-R2	0.15	0.25	0.20	1.74	1.41	1.24

Overall, the gaps to the BKSs are rather small and below 4% for all solvers except LNS-MC over all time limits and instance sets. The spreadsheet solver is outperformed by all other solvers but is still competitive, especially on the smaller VRPU instances. However, the results of the spreadsheet solver for 600 s are still worse than those for the other solvers after 60 s. Except for the VRPU and the XU instances, the simple MIN-SA solver is outperformed by the HGS-CARP solver. Especially on the GoldenU instances the margin is comparatively large. Except for the VRPU instances, MIN-VND consistently outperforms all other solvers, i.e., augmenting the exponential MIN by a local search procedure clearly improves the results. The margin between MIN-SA and MIN-VND is notably larger on the GoldenU instances.

MIN-SA outperforms both LNS variants on all except the GoldenU instances. Among the LNS variants, LNS-R2 consistently outperforms LNS-MC. These results indicate that it is beneficial to spend some effort in finding a better or best reinsertion in the second step. Thus, for the VRPU the MIN-based insertion procedure is competitive with conventional LNS repair operators.

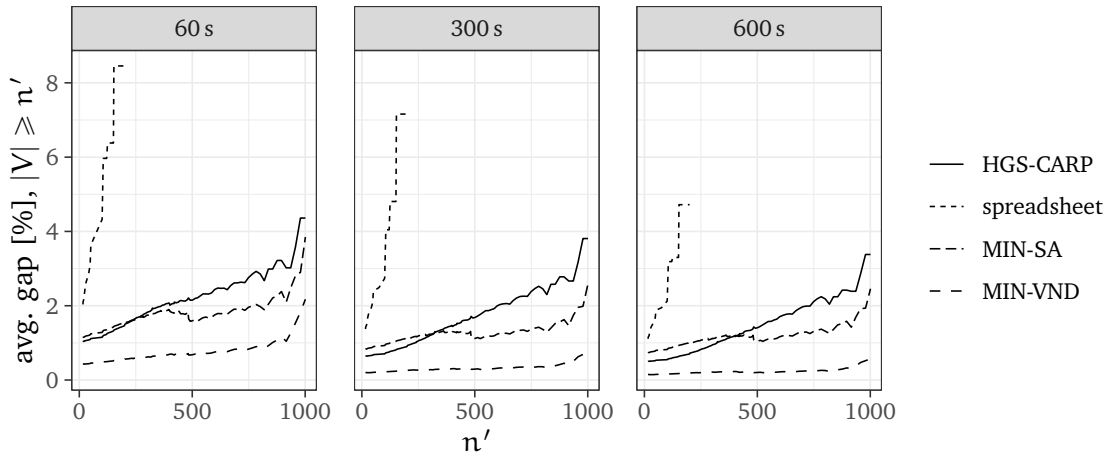


Figure 3.12: Average gaps for all solvers and instances with $|V| \geqslant n'$ nodes.

To get a clear picture w.r.t. the influence of the instance size, Figure 3.12 shows the average gaps to the BKSs obtained by the MIN-based solvers, the HGS-CARP solver and the spreadsheet solver over all instances with $|V| \geqslant n'$ nodes. Hence, the values at $n' = 0$ correspond to the values of column `all` of Table 3.6. For $n' \leqslant 450$ the general results remain consistent with those over all instances. For larger n', the average gap of MIN-SA remains below the average gap of the HGS-CARP solver, while MIN-VND is consistently outperforming the other solvers. This implies that, for the given time limits, the MIN-based solvers work rather well on larger instances and are able to reduce the total costs faster than the HGS-CARP solver, although the latter may outperform the MIN-based solvers for larger time limits. A possible reason is that the global optimization approach of the insertion procedure solving the MIN-WPRCM is especially beneficial at the beginning of the search process as it may perform a lot of relocations simultaneously. Thus, MIN-based solvers may be used to pre-optimize larger instances before continuing with better but slower solvers.

3.5.4 Large-scale instances

The instance sets summarized in Table 3.1 contain a few instances with more than 1000 nodes. To gain an intuition regarding the performance of the proposed heuristics w.r.t. large-scale instances, we perform an experiment over generated instances ranging from 1000 to 5000 nodes.

The generation of the instances was performed along the two dimensions *number of nodes* and *cost structure*. The considered numbers of nodes are $|V| \in \{1000, 2000, 5000\}$. The costs are either *Euclidean* or *asymmetric*. Initially the nodes are sampled randomly from the Euclidean plane. In case of Euclidean costs, the costs c_{ij} are derived from the distances between the sampled points i and j. For asymmetric costs, we start from the Euclidean costs and set each pair of nodes' direct connection cost $c_{ij} \coloneqq \infty$ with probability 0.5. Subsequently, we calculate *all-pairs shortest path* (APSP) over the complete directed graph with arc weights c_{ij} and replace all costs $c_{ij} = \infty$ with the cost of the corresponding shortest path. Thus, the costs c_{ij} satisfy the triangle inequality. For each combination of *number of nodes* and *cost structure*, nine instances were generated for a total of 54 instances.

The solvers HGS-CARP, MIN-SA and MIN-VND were run on all instances for time limits 1800 s and 3600 s with 10 replications. The results are summarized in Table 3.7. Column *structure* reports the cost structure, column *time* the time limit and column $|V|$ the number of nodes, respectively. The average gaps to the best obtained solutions are reported in columns

avg. gap and the number of best solutions reached in the respective group of cost structure, time limit and number of nodes is reported in columns *#best*. If a solver was not able to obtain solutions over all runs in a group, we removed the solver from the analysis.

Table 3.7: Results for the generated large instances.

			avg. gap [%]			#best		
structure	time [s]	$\|V\|$	HGS-CARP	MIN-SA	MIN-VND	HGS-CARP	MIN-SA	MIN-VND
euc	1800	1000	1.74	1.96	0.42	0	0	9
		2000	–	2.24	0.73	–	0	9
		5000	–	0.96	2.40	–	6	3
	3600	1000	1.33	1.79	0.22	0	0	9
		2000	2.04	2.00	0.26	0	0	9
		5000	–	0.39	1.92	–	6	3
asym	1800	1000	2.37	4.21	0.90	1	0	8
		2000	3.59	3.48	1.24	0	0	9
		5000	–	1.22	5.04	–	9	0
	3600	1000	1.70	3.87	0.48	2	0	7
		2000	2.79	3.22	0.33	0	0	9
		5000	–	0.25	4.53	–	8	1

With the given time limits, the HGS-CARP solver was unable to provide feasible solutions for the instances with 5000 nodes and for some instances with Euclidean cost structure and 2000 nodes. However, for all other groups HGS-CARP outperformed MIN-SA more often than not, especially for the larger time limit. Nevertheless, on average, HGS-CARP is always outperformed by MIN-VND.

Comparing MIN-SA and MIN-VND we observe that, on average, MIN-VND outperforms MIN-SA over all instances with 1000 and 2000 nodes while for instances with 5000 nodes the relationship is reversed. Our interpretation is as follows: For 5000 nodes the VND requires a rather large share of the total time limit to improve solutions while the overall number of iterations and thus, applications of the MIN, becomes rather small. At this point, performing more iterations of the MIN alone becomes beneficial as its global search approach can perform multiple improving small moves at the same time without searching for improving moves over and over again. We conjecture that the pattern observed for instances with 1000 and 2000 nodes does surface as well for instances with 5000 nodes if the time limit is increased appropriately.

In summary, this experiment highlights the advantages of the MIN's global approach in comparison to local search approaches like the VND: In the beginning of a search process or when the time or computational budget is strictly limited, the global approach can be used to efficiently improve solutions and move into promising regions of the search space. Then, more fine-grained local search approaches can be used to improve solutions even further.

3.5.5 Impact of MCFP algorithm implementations

The MIN-based solver implementations considered so far are using the *network simplex* implementation of the *Lemon* [33] graph algorithms library to solve the underlying MCFP. However, a variety of algorithms to solve the MCFP exists and for many of these algorithms multiple implementations across different graph algorithm libraries are available. For a general in-depth analysis of different MCFP algorithms and their implementations, we refer the reader to Kovács [70].

The scope of the following experiments is to establish an intuition of the impact that different choices of MCFP algorithms and their implementations have w.r.t. the structure of the MCFP arising in the MIN.

Table 3.8: MCFP algorithm implementations and graph libraries.

library	version	MCFP algorithm	identifier
BGL [95]	1.58.0	cycle canceling	BGL-CC
		successive shortest path	BGL-SSP
Lemon [33]	1.3.1	capacity scaling	Lemon-CaS
		cost scaling	Lemon-CoS
		cycle canceling	Lemon-CC
		network simplex	Lemon-NS
OGDF [25]	2020.02	network simplex	OGDF-NS

To facilitate the experiment, we reimplemented the MIN-SA and MIN-VND solvers using five different MCFP algorithms from the *Lemon* graph library, the *Boost Graph Library* (BGL) [95] and the *Open Graph Drawing Framework* (OGDF) [25]. The libraries and implementations are summarized in Table 3.8. The solvers were run on all instance sets `VRPU`, `CMTU`, `GoldenU` and `XU` for three time limits of 60 s, 300 s and 600 s with 10 replications for each combination, respectively. Each combination was required to perform at least a single iteration of the MIN or the MIN in conjunction with the VND. Combinations that failed to obtain a solution on any of the 10 replications are excluded from the corresponding analysis.

The average gaps to the BKSs for MIN-SA and MIN-VND are reported in Table 3.9. All MIN-SA combinations successfully obtained solutions. Over all instances, the average gaps for MIN-SA range from 1.35% to 51.17% and 0.80% to 34.25% for runs over 60 s and 600 s, respectively. Hence, the choice of the MCFP algorithm and its implementation may have a rather large impact. The BGL-CC algorithm performed worst by a large margin. However, the Lemon-CC algorithm performed clearly better and outperforms the BGL-SSP implementation, i.e., choosing one implementation over another of the same algorithm may have a larger impact than choosing another algorithm altogether. The Lemon-NS consistently performs best followed by the Lemon-CaS and OGDF-NS implementations.

For MIN-VND certain combinations with BGL-CC, BGL-SSP and Lemon-CaS were unable to perform a full iteration over all instances of the two larger instance sets `GoldenU` and `XU`. We observe that differences between the different algorithms and implementations are a lot smaller for MIN-VND as the time spent on solving the MCFP relative to the total time limit decreases due to the time spent on the VND. The Lemon-NS still consistently performs best but the results are marginal when compared to the other promising combinations with OGDF-NS, Lemon-CaS and Lemon-CoS.

In conclusion, the choice of algorithm and implementation combination can have a significant impact on the results reported. The results show that the choice for the Lemon-NS implementation is well-founded.

Implications In Section 2.2.4 we discussed empirical heuristics research and the question of the required implementation effort. In this context the above experiments comparing different MCFP algorithms to solve the MIN-WPRCM subproblem lead to the following conclusions.

When developing solution methods that utilize external codes in a blackbox fashion to solve subproblems, time needs to be invested to select appropriate implementations to solve

Table 3.9: MIN-SA and MIN-VND with different MCFP algorithm implementations.

		avg. gap [%]									
		MIN-SA					MIN-VND				
time [s]	MCFP alg.	VRPU	CMTU	GoldenU	XU	all	VRPU	CMTU	GoldenU	XU	all
60	BGL-CC	3.30	27.51	0.05	69.24	51.17	0.73	0.10	–	–	–
	BGL-SSP	0.64	8.84	0.02	5.38	4.93	0.26	0.08	1.39	–	–
	Lemon-CaS	0.37	4.48	0.01	1.47	1.65	0.21	0.08	1.00	–	–
	Lemon-CoS	0.51	5.20	0.02	1.70	1.91	0.26	0.08	1.05	0.71	0.66
	Lemon-CC	0.66	7.11	0.11	2.97	3.06	0.30	0.08	1.20	0.82	0.76
	Lemon-NS	0.36	3.71	0.01	1.19	1.35	0.23	0.08	1.05	0.66	0.62
	OGDF-NS	0.39	4.48	0.08	1.51	1.68	0.21	0.09	1.06	0.68	0.63
300	BGL-CC	1.52	21.64	0.04	53.52	39.50	0.47	0.09	5.63	–	–
	BGL-SSP	0.33	5.28	0.00	1.98	2.10	0.13	0.08	0.57	0.49	0.43
	Lemon-CaS	0.28	2.97	0.00	0.93	1.06	0.06	0.08	0.42	0.33	0.29
	Lemon-CoS	0.27	3.33	0.00	0.99	1.15	0.07	0.08	0.45	0.34	0.31
	Lemon-CC	0.38	4.12	0.09	1.38	1.55	0.13	0.08	0.53	0.40	0.37
	Lemon-NS	0.21	2.67	0.00	0.78	0.92	0.06	0.08	0.45	0.30	0.28
	OGDF-NS	0.23	2.98	0.07	0.93	1.06	0.08	0.09	0.48	0.32	0.30
600	BGL-CC	1.12	18.68	0.02	46.45	34.25	0.39	0.09	4.52	–	–
	BGL-SSP	0.29	4.31	0.00	1.48	1.62	0.06	0.08	0.44	0.35	0.31
	Lemon-CaS	0.22	2.65	0.00	0.78	0.91	0.08	0.08	0.34	0.23	0.22
	Lemon-CoS	0.27	2.86	0.00	0.84	0.99	0.05	0.08	0.30	0.24	0.22
	Lemon-CC	0.28	3.37	0.08	1.11	1.25	0.06	0.08	0.41	0.29	0.27
	Lemon-NS	0.18	2.36	0.00	0.67	0.80	0.04	0.08	0.32	0.23	0.21
	OGDF-NS	0.20	2.67	0.07	0.78	0.92	0.05	0.09	0.35	0.23	0.22

these subproblems. The time to invest should be proportional to the expected time the subproblem solution process consumes relative to the total solution time, i.e., more time needs to be invested for a subproblem that is solved multiple times than a subproblem that is solved once, e.g., to derive an initial solution. This holds for most research on heuristic and exact methods alike as either a fixed time limit is used or the time is reported for some other fixed parameter, e.g., iteration limit.

Failing to select suitable implementations may have the following consequences, among others, for the results obtained from empirical research: (i) proposed methods could be drastically improved by simply exchanging the implementation of subproblem solution processes and (ii) proposed methods seemingly outperforming methods relying on unsuitable subproblem solution implementations may actually be outperformed by the latter if a suitable choice had been made.

We suggest (i) to utilize battle tested implementations that have been rigorously evaluated and tested by the scientific community and (ii) to perform sensitivity analyses of the impact of different implementations if there is reason indicating that the performance of the overall solution method is tightly coupled to the performance of the subproblem solution method.

3.6 Conclusions

In this chapter we extended, analyzed and evaluated the MIN for the VRPU as proposed by Angel et al. [3].

To make the neighborhood $\mathrm{MIN}^{\mathrm{m}}$ usable in an actual solver implementation, first a subset

of m mobile nodes has to be chosen before these nodes can be inserted in a best possible way to generate a neighbor solution. While in [3] it was shown that a best insertion can be calculated in polynomial time, we proved that choosing a best mobile node set is strongly $\mathcal{NP}$-hard.

In contrast to the simpler TSP, we have shown that the maximum number of nodes which may be chosen as mobile, depends on the instance as well as the number of tours in the current solution. Furthermore, the existence of feasible neighbors additionally depends on the actual set of mobile nodes. Additionally, we have shown that for $2 \leqslant m \leqslant \lceil n/2 \rceil$ the neighborhood MIN^m is connected.

Since the subproblem of selecting a best subset of mobile nodes is $\mathcal{NP}$-hard, we proposed and implemented several node selection heuristics. The experiments have shown that randomized selection strategies perform well. Especially, the roulette wheel selector with probabilities based on the number of nodes in the tours performed best.

Finally, the neighborhood MIN^m was used in two variants of a SA algorithm and compared to two simple LNS heuristics as well as to two VRP heuristics from the literature. Variant (i) uses only MIN^m, while variant (ii) augments the algorithm with a VND procedure using four small neighborhoods. The results show that variant (i) performs quite well on all considered instances, outperforming the spreadsheet solver and the LNS-based solvers. On the other hand, it did not perform as well as the HGS-CARP solver. In contrast, variant (ii) consistently outperformed all other solvers.

Further experiments on synthetic large-scale instances have shown that for large instances and comparatively small time limits the MIN-SA approach obtained the best results. However, when the time limit is increased or the instance size is decreased, then the MIN-VND approach obtained the best results. This indicates that the MIN is especially useful in the early phase of the search process to improve the initial solution efficiently. Additionally, we tested different MCFP algorithms and implementations. The results show that the choice of the algorithm and implementation is important w.r.t. the final results.

In conclusion, the MIN utilizes structural properties of the VRPU and is able to exceed the solution quality of smaller neighborhoods. From a practical perspective, MIN^m on its own is easy to implement but is outperformed by a VND combined with smaller neighborhoods. The time required to setup the MCFP graph and to calculate a best reinsertion by exactly solving a matching problem requires a comparatively large amount of time and it is therefore better to use additionally smaller neighborhoods to find improving neighbor solutions. However, if the computational budget is strictly limited relative to the sizes of the considered instances, then it is beneficial to use the MIN in isolation to efficiently reach more promising regions of the search space and continue with additional, more granular neighborhoods from there. These results are in line with many results from the literature on exponential neighborhoods, where a better trade-off between solution quality and calculation time tends to be realized using smaller (non-exponential) neighborhoods.

Chapter 4

The preemptive stacker crane problem

In this chapter, we study the *preemptive stacker crane problem* (PSCP) from theoretical and practical perspectives. The PSCP is a single-vehicle unit-capacity one-to-one *pickup and delivery problem* (PDP) with preemption, the single-vehicle analogue to transshipments. Parts of this chapter have already been published in the peer-reviewed research article Graf [52].

Contribution On the theoretical side, we contribute new bounds for the maximum improvement enabled by allowing preemption and the maximum improvement enabled by allowing additional nodes that are only used for preemption but neither for pickups nor deliveries. We study these bounds for asymmetric and symmetric costs and provide example instances showing tightness for all provided bounds. Additionally, we identify two polynomial-time solvable subproblems that allow to constrain the search space to permutations of the input requests in theory. On the practical side, we contribute efficient and effective construction heuristics based on heuristics for the *asymmetric traveling salesman problem* (ATSP). Our heuristics outperform the state-of-the-art algorithms in the quality of the obtained solutions as well as the required computation time. Finally, a computational study provides further insights regarding the improvement made possible by allowing preemption on realistic instances.

Organization We first introduce the problem and its surrounding context and literature in Section 4.1. The PSCP and accompanying notation are introduced formally in Section 4.2. In Section 4.3 theoretical properties of the PSCP are considered. The main focus is on the benefits of preemption and explicit drop nodes, i.e., nodes not associated with any request, as well as reduced solution representations. Subsequently, in Section 4.5 we discuss three construction heuristics for the PSCP and their variants. An extensive computational study regarding the proposed algorithms is described in Section 4.6 followed by concluding remarks in Section 4.7.

4.1 Introduction

The PSCP has been studied in various variants over the past three decades. The goal is to derive a minimum-cost routing for a single unit-capacity vehicle satisfying a given set of one-to-one pickup and delivery requests with the possibility to preempt requests, i.e., to drop their payloads temporarily at arbitrary nodes.

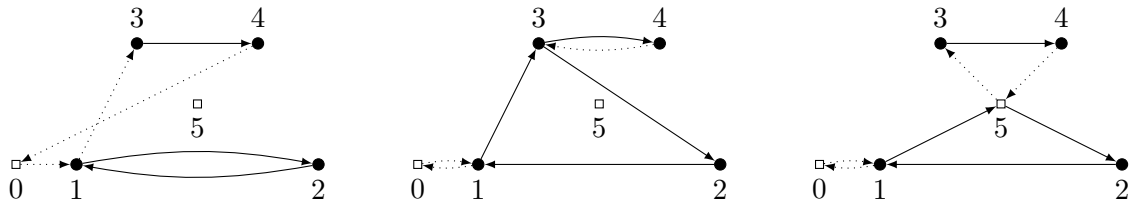
The preemption of requests allows the vehicle to intertwine the transportation of two or more requests despite its unit-capacity. Recall the precedence constraints resulting from a pickup and delivery setting, i.e., pickup locations need to be visited prior to their corresponding delivery locations. With preemption allowed, additional precedence constraints are implied as a vehicle can only pickup a request from an intermediate location that has been visited earlier and where the corresponding delivery has been performed. Colloquially

speaking, preemption may be interpreted to be the single-vehicle analogue of transshipments in multi-vehicle pickup and delivery settings.

The motivation to allow preemption is economic in that it enables cost savings in certain scenarios. In more detail, it allows to decrease the routing cost by reducing the costs associated with deadheadings, i.e., parts of the tour that the vehicle is performing empty without transporting any request. Similar to the multi-vehicle pickup and delivery variants with transshipments, additional locations may be used to facilitate preemption and reduce the costs even further. The differences between preemptive and non-preemptive tours with and without additional locations are illustrated in the following Example 4.1.

Table 4.1: Cost matrix with entries c_{ij} providing costs from i to j.

	0	1	2	3	4	5
0	0	2	9	5	8	6
1	2	0	7	4	7	4
2	9	7	0	7	4	4
3	5	4	7	0	4	3
4	8	7	4	4	0	3
5	6	4	4	3	3	0



(a) Non-preemptive tour Π_1 (b) Preemptive tour Π_2 (c) Preemptive tour Π_3

Figure 4.1: Preemptive and non-preemptive vehicle tours.

Example 4.1. *Consider a PSCP instance with nodes $v \in \{0, \dots, 5\}$ and three requests. The requests are defined by pairs of corresponding pickup and delivery nodes: $r_1 = (1, 2)$, $r_2 = (2, 1)$, $r_3 = (3, 4)$. Node $v = 0$ corresponds to the depot, where the vehicle is initially located and where it must be located again at the end of the tour. Node $v = 5$ is an additional node not associated with any pickup or delivery node and therefore does not have to be visited by the vehicle. Table 4.1 provides the symmetric costs satisfying the triangle inequality for all pairs of nodes. Figure 4.1 shows three feasible vehicle tours:*

$$\Pi_1 = \langle 0, 1, 2, 1, 3, 4, 0 \rangle$$
$$\Pi_2 = \langle 0, 1, 3, 4, 3, 2, 1, 0 \rangle$$
$$\Pi_3 = \langle 0, 1, 5, 3, 4, 5, 2, 1, 0 \rangle$$

Tour Π_1 performs each request directly from pickup to delivery and is thus non-preemptive with total cost $c(\Pi_1) = 32$. In contrast, tour Π_2 preempts request $r_1 = (1, 2)$ at node 3 to perform request $r_3 = (3, 4)$ resulting in total cost $c(\Pi_2) = 30$. The savings of Π_2 compared to Π_1 result from the combination of the deadheadings associated with request r_3 and the transportation of r_1. Thus, the total cost decreased despite the fact that the cost induced by the transportation of r_1 increased. Tour Π_3 is structurally similar to tour Π_2, i.e., only request r_1 is preempted in favor of request r_3. However, the preemption is performed at the additional node 5 instead of the pickup node 3 reducing the total cost $c(\Pi_3) = 29$ even further.

Literature

The *stacker crane problem* (SCP) is introduced by Frederickson et al. [42] as a mixed arc routing problem, i.e., all transportation requests are modeled as directed arcs while all other connections are undirected. They assume the triangle inequality and describe a $\frac{9}{5}$-approximation algorithm by first calculating a minimum-weight cycle cover (matching) followed by use of the Christofides heuristic (cf. [26]) for the metric *traveling salesman problem* (TSP). Frederickson and Guan [40] show that the SCP remains hard when the underlying network is restricted to tree structures and they develop a $\frac{5}{4}$-approximation algorithm. In contrast, for line and circular structures the SCP becomes polynomial-time solvable as shown by Atallah and Kosaraju [10].

Quilliot et al. [88] consider the (asymmetric) PSCP and derive a randomized *Monte-Carlo insertion heuristic* (MCI) and a *variable neighborhood descent* (VND) iterative-improvement heuristic. They also provide a Miller-Tucker-Zemlin (MTZ) (cf. Miller et al. [79]) style mixed-integer programming formulation and calculate optimal values for smaller benchmark instances. An exact branch-and-cut approach for the *single-vehicle preemptive pickup and delivery problem* (SPPDP) and for its special case the PSCP is developed in Kerivin et al. [65].

Like the SCP, the PSCP becomes solvable in polynomial time when it is restricted to certain graph structures. Atallah and Kosaraju [10] provide exact algorithms for line and circular structures while Frederickson and Guan [41] provide an exact algorithm for the PSCP on a tree.

Anily and Hassin [6] introduce the *swapping problem* (SP), a generalization of the PSCP that relaxes the one-to-one correspondence between pickup and delivery nodes. Each node may demand one unit of a commodity and may provide one unit of a possibly different commodity. Preemption is allowed for a specified commodity subset. As such, the original SP is also called the *mixed swapping problem* (MSP), while the SP and *preemptive swapping problem* (PSP) correspond to the non-preemptive and preemptive variants, respectively. Anily and Hassin [6] describe a $\frac{5}{2}$-approximation algorithm for the MSP. Anily et al. [5] consider the MSP on a line and propose an exact polynomial-time algorithm. Similarly, for the PSP on a tree, Anily et al. [4] develop a $\frac{3}{2}$-approximation algorithm. Furthermore Bordenave et al. [17] develop a heuristic for the MSP that is based on calculating a minimum-weight cycle cover for each commodity and a post-processing phase aimed to improve the solutions by use of preemption. They show that the cost of the initial minimum-weight cycle cover is a lower bound of the total solution cost. Additionally, Bordenave et al. [16] describe an exact branch-and-cut algorithm for the PSP.

4.2 Problem description and notation

The PSCP is formally defined as follows. Let V be a finite set of nodes and $v_0 \in V$ a specific depot node. Furthermore let $R = \{r_1, \ldots, r_m\}$ be a set of m requests, each request $r \in R$ is associated with a pickup node $v_r^- \in V$ and a delivery node $v_r^+ \in V$. Let $V_R^- \subset V$ be the set of pickup nodes and $V_R^+ \subset V$ be the set of delivery nodes, respectively. The set of requests implies a disjoint partition of the node set $V = V_R \cup V_U \cup \{v_0\}$ with the set of *implicit drop nodes* $V_R = V_R^- \cup V_R^+$ and the set of *explicit drop nodes* $V_U = V \backslash (V_R \cup \{v_0\})$ which are neither the pickup or the delivery of any request nor the depot. A cost function $c\colon V \times V \to \mathbb{R}_{\geqslant 0}$ satisfying the triangle inequality provides the costs for all connections $(v, w) \in V \times V$. To ease the notation, distances of node sequences are denoted by $c(v_i, \ldots, v_j) = c(v_i, v_{i+1}) + c(v_{i+1}, v_{i+2}) + \cdots + c(v_{j-1}, v_j)$.

A single vehicle with unit-capacity, initially located at the depot v_0, needs to transport each request exactly once and return to the depot. Due to the unit-capacity at most one request

may be transported at any time. When preemption is allowed, a currently loaded request $r \in R$ may be dropped at an intermediate node $v \in V \backslash \{v_r^-, v_r^+\}$ instead of being directly transported to its delivery node v_r^+. If a request has been dropped at a node, it needs to be reloaded, i.e., picked up again at a later time and is then either delivered to v_r^+ or another intermediate node and so on.

The goal is to find a feasible minimum-cost routing of the vehicle, starting and ending at the depot and satisfying all requests while respecting the unit-capacity constraint.

A straightforward representation of a solution S is by a finite sequence of nodes $\Pi = \langle v_{\Pi_1}, \ldots, v_{\Pi_{|\Pi|}} \rangle$ with $v_{\Pi_1} = v_{\Pi_{|\Pi|}} = v_0$ corresponding to the directed closed walk of the vehicle. Each arc $(v_{\Pi_{i-1}}, v_{\Pi_i})$, $2 \leqslant i \leqslant |\Pi|$ is either associated with a request $r \in R$ being transported on that arc or a deadheading. The cost of a solution is the sum of costs between consecutive nodes $c(\Pi) = \sum_{i=2}^{|\Pi|} c(v_{\Pi_{i-1}}, v_{\Pi_i})$. As the triangle inequality holds, (i) consecutive deadheadings and (ii) consecutive arcs transporting the same request may be joined, e.g., two deadheading arcs (v_i, v_j) and (v_j, v_k) are joined into the deadheading arc (v_i, v_k). As such, it is sufficient to consider sequences with consecutive arcs corresponding to either different requests or a request and a deadheading.

Additionally we use so-called *simplified alternating sequences* (SASs). In contrast to arbitrary sequences, a SAS consists of arcs alternating between transportation of requests and deadheadings, e.g., a sequence $\langle v_{r_1}^-, v_{r_2}^-, v_{r_2}^+ \rangle$ transporting r_1 from its pickup node to the pickup of r_2 to drop it there is expanded into the sequence $\langle v_{r_1}^-, v_{r_2}^-, v_{r_2}^-, v_{r_2}^+ \rangle$ with an additional zero-cost deadheading from $v_{r_2}^-$ to $v_{r_2}^-$. To simplify even further, each drop is associated with its own drop node, i.e., a drop node is copied for each drop in the sequence such that the sequence becomes $\langle v_{r_1}^-, u, v_{r_2}^-, v_{r_2}^+ \rangle$ where u is a copy of $v_{r_2}^-$ and associated with the drop of request r_1. Thus, in the Eulerian digraph induced by a SAS, all pickup and delivery nodes have indegree and outdegree of one while all drop nodes have indegree and outdegree of two, respectively.

As consecutive arcs transporting the same request are merged, each request is transported over a set of non-consecutive arcs which induce a directed path from v_r^- to v_r^+ via zero or more drop nodes. This path $p(r) = \langle v_r^-, \ldots, v_r^+ \rangle$ is called the *request path* of r (cf. [65]). Each drop node $v \in p(r) \backslash \{v_r^-, v_r^+\}$ implies a cycle in the tour Π corresponding to the arcs between the drop of r at v and its subsequent reload at v. This subsequence $\Pi(r, v) \subset \Pi$ is called *drop cycle*. Two drop cycles *overlap* if they share at least one arc. A drop cycle $\Pi(r, v)$ is *contained* in another drop cycle $\Pi(r', v')$ if all arcs of the former are contained in the latter, i.e., $\Pi(r, v) \subseteq \Pi(r', v')$.

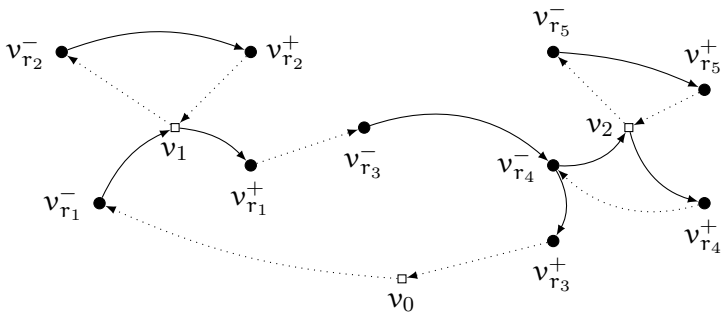


Figure 4.2: A solution satisfying five requests $R = \{r_1, \ldots, r_5\}$ starting at the depot v_0. Arcs that carry a request are drawn *solid* while deadheading arcs are drawn *dotted*.

Example 4.2. *Consider the solution illustrated in Figure 4.2. The drop cycle* $\Pi(r_1, v_1)$ *is isolated and does not contain any further drop cycles. In contrast, drop cycle* $\Pi(r_3, v_{r_4}^-)$ *contains the drop*

cycle $\Pi(r_4, v_2)$, i.e., the drop and the reload at $v^-_{r_4}$ occur before and after the transport of r_5, respectively. Nodes v_1, $v_2 \in V_U$ are explicit drop nodes and node $v^-_{r_4} \in V_R$ is an implicit drop node.

To identify different problem variants, SCP and PSCP denote the fully non-preemptive and fully preemptive problem variants, respectively. Additionally, $\text{PSCP}_\varnothing$ denotes a PSCP problem with $V_U = \varnothing$, i.e., without explicit drop nodes. Mixed problems are not considered explicitly.

4.3 Theoretical properties

The main results of this section consider the benefits of preemption, the benefits of explicit drop nodes and reduced representations of the problem. By combining two results from the literature, we show that preemption may improve the total cost by at most 50%. Additionally, it is shown that explicit drop nodes may improve the total cost by up to 50% in case of asymmetric distances and up to $33.\overline{3}\%$ in case of symmetric distances. In case of line- or circular structures, explicit drop nodes do not provide any benefit. Tightness is shown for all provided bounds.

4.3.1 Tree-structured solutions

First, a fundamental result of Quilliot et al. [88] is reproduced, as it is the basis for all newly presented results in this chapter. The proofs are provided constructively, i.e., all restructuring methods described in the proofs can be turned into algorithms.

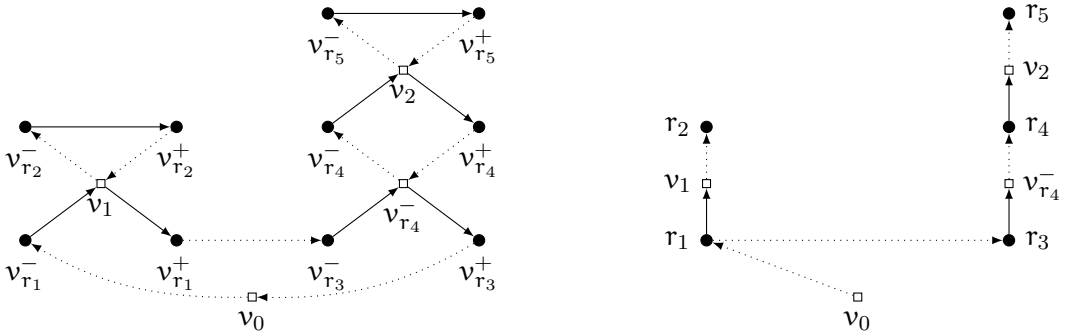
Definition 4.1 (Bipartite ordered tree (BOT))**.** *A bipartite ordered tree is a tree whose nodes are partitioned into two sets A, B such that (i) nodes in A only have children in B and vice versa and (ii) each node's children are arranged in a totally ordered sequence.*

For a given node $a \in A$ in tree T, $\sigma_T(a)$ denotes the child sequence of a consisting of nodes in B and vice versa for nodes $b \in B$, $\sigma_T(b)$ denotes the child sequence consisting of nodes in A. For a given sequence $\sigma_T(a) = \langle b_1, \dots, b_k \rangle$, we denote by $\tilde{\sigma}_T(a) = \sigma_T(b_1) \oplus \cdots \oplus \sigma_T(b_k)$ the sequence resulting from the concatenation of the children's child sequences. If the corresponding tree T is obvious from context, it is omitted and we write $\sigma(a)$ and $\tilde{\sigma}(a)$, respectively.

Quilliot et al. [88] introduce the concept of *bipartite ordered tree* (BOT) in the context of the asymmetric PSCP. They state that a BOT is *consistent* with a given instance if (i) the BOT is partitioned between the sets V and R, (ii) the root node is $v_0 \in V$, (iii) every request $r \in R$ occurs exactly once in the BOT and (iv) every leaf node in the tree is a request node. Note that not every explicit drop node needs to be present in the tree. They show that every consistent BOT corresponds to a PSCP solution.

Given a consistent BOT T and a drop node $v \in T$ with its child sequence $\sigma(v) = \langle r_1, \dots, r_k \rangle$, the arc $(v, r_1) \in T$ implies that the vehicle performs deadheadings from v to $v^-_{r_1}$ and later on from $v^+_{r_k}$ back to v. In between, the vehicle performs deadheadings from $v^+_{r_i}$ to $v^-_{r_{i+1}}$ for each arc $(r_i, r_{i+1}) \in T$, $i \in \{1, \dots, k-1\}$. For a request node r and its child sequence $\sigma(r) = \langle v_1, \dots, v_k \rangle$ the arc $(r, v_1) \in T$ implies that the vehicle transports request r from its pickup node v^-_r to v_1 and later on from v_k to its delivery node v^+_r. In between, the request is transported from v_i to v_{i+1} via arcs $(v_i, v_{i+1}) \in T$, $i \in \{1, \dots, k-1\}$. A leaf node r always implies a direct transportation from v^-_r to v^+_r.

In other words, a request node r with its child sequence of drop nodes $\sigma(r) = \langle v_1, \dots, v_k \rangle$ corresponds to the request path $p(r) = \langle v^-_r, v_1, \dots, v_k, v^+_r \rangle$. Similarly, the subtree rooted at a drop node $v \in \sigma(r)$ corresponds to the induced drop cycle $\Pi(r, v)$.



(a) Abstract view of the solution shown in Figure 4.2.

(b) BOT of the solution

Figure 4.3: (a) shows an abstract view of the solution shown in Figure 4.2. Note that in contrast to Figure 4.2, the drop of r_3 at $v^-_{r_4}$ and the pickup at $v^-_{r_4}$ are represented by two distinct nodes. (b) shows the corresponding BOT.

Example 4.3. *Figure 4.3 shows an abstract view of the solution depicted in Figure 4.2 and the corresponding BOT. The child sequence* $\sigma(r_1) = \langle v_1 \rangle$ *of request* r_1 *in the BOT corresponds to the request path* $p(r_1) = \langle v^-_{r_1}, v_1, v^+_{r_1} \rangle$ *containing only a single drop node. The subtree rooted at drop node* $v^-_{r_4}$ *corresponds to the drop cycle* $\Pi(r_3, v^-_{r_4})$ *and represents all vehicle movements after the drop of request* r_3 *at* $v^-_{r_4}$ *and prior to its subsequent reload at the same node* $v^-_{r_4}$*. The subtree at* $v^-_{r_4}$ *contains the subtree at* v_2 *and thus, the drop cycle* $\Pi(r_4, v_2)$ *is contained in the drop cycle* $\Pi(r_3, v^-_{r_4})$*.*

The BOT structure implies that if two drop cycles overlap, then one is fully contained in the other. The sequence of vehicle movements and the total costs can be derived from a BOT by a depth-first traversal. Quilliot et al. [88] further show that such a representation is actually sufficient to represent at least one optimal solution for any PSCP instance:

Theorem 4.1 (Quilliot et al. [88])**.** *For each instance of the asymmetric PSCP there exists a consistent BOT that corresponds to an optimal solution.*

Proof. We show that an arbitrary feasible solution Π can be transformed into a BOT-structured solution S without any cost increase, i.e., $c(S) \leqslant c(\Pi)$. For that, it is sufficient to transform the sequence Π into a so-called *BOT-structured sequence* Π'. We say a sequence Π' is BOT-structured if for all pairs of distinct drop-cycles one is fully contained in the other or vice versa:

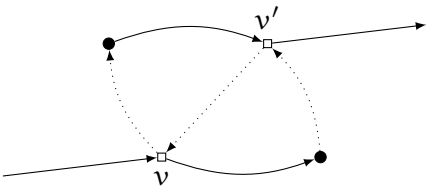
$$\forall \Pi'(r, v), \Pi'(r', v') \subseteq \Pi' \colon \Pi'(r, v) \subseteq \Pi'(r', v') \vee \Pi'(r', v') \subseteq \Pi'(r, v) \tag{4.1}$$

We call a pair of drop-cycles *non-conforming* iff they overlap but neither one is contained in the other. A BOT-structured sequence Π' can be transformed into a BOT-structured solution S trivially.

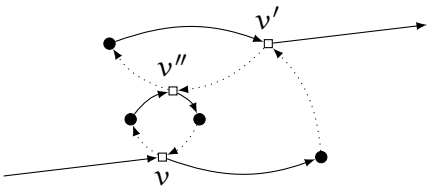
Initially, we assume the sequence Π to be a SAS, i.e., all pickup and delivery nodes have exactly one incoming and one outgoing arc while drop nodes have exactly two incoming and two outgoing arcs.

Assuming that Π does not conform to Property (4.1), then we can identify two drop-cycles $\Pi(r, v)$ and $\Pi(r', v')$ with underlying subsequence $p = \langle v, \ldots, v', \ldots, v, \ldots, v' \rangle \subseteq \Pi$. Let $p = \langle v \rangle \oplus p_1 \oplus \langle v' \rangle \oplus p_2 \oplus \langle v \rangle \oplus p_3 \oplus \langle v' \rangle$ be a decomposition of the subsequence into three disjoint subsequences such that subsequence p_1 corresponds to the sequence between the

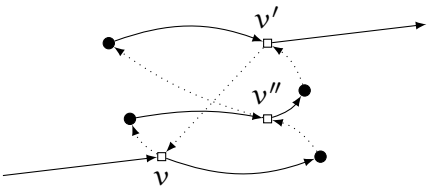
first visit of v and the first visit of v', p_2 corresponds to the sequence between the first visit of v' and the second visit of v and p_3 corresponds to the remaining subsequence between the second visit of v and the second visit of v'.



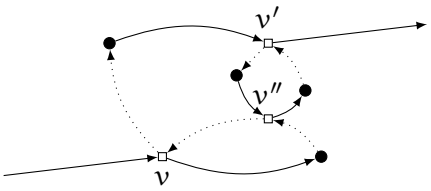
(a) No dependence



(b) Dependence $p_1 \rightarrow p_2$



(c) Dependence $p_1 \rightarrow p_3$



(d) Dependence $p_2 \rightarrow p_3$

Figure 4.4: Dependencies between the subsequences of non-conforming pairs of drop-cycles. The subsequence depicted in (a) is $p = \langle v, v_p, v', v, v_d, v' \rangle$ with $p_1 = \langle v_p \rangle$, $p_2 = \langle\rangle$ and $p_3 = \langle v_d \rangle$.

In fact, we can always identify two drop-cycles such that the subsequences p_1, p_2 and p_3 are pairwise *independent* and can be performed in any order without violating the precedences implied by the request paths as illustrated in Figure 4.4a.

Suppose a pair of non-conforming drop-cycles rooted at v and v', respectively. Assuming that the independence between subsequences p_1, p_2 and p_3 is violated, three cases need to be considered.

(i) There is a dependence $p_1 \rightarrow p_2$, i.e., p_1 has to be performed prior to p_2 as illustrated in Figure 4.4b. Then there must be a drop node v'' such that the drop-cycles $\Pi(r'', v'')$ and $\Pi(r', v')$ are non-conforming and their associated sequence

$$p' = \langle v'', \ldots, v', \ldots, v'', \ldots v' \rangle \subset p \Rightarrow |p'| < |p|$$

is a proper subsequence of p.

(ii) There is a dependence $p_1 \rightarrow p_3$ as illustrated in Figure 4.4c. Then there must be a drop node v'' such that the drop-cycles $\Pi(r'', v'')$, $\Pi(r', v')$ and the drop-cycles $\Pi(r, v)$, $\Pi(r'', v'')$ are non-conforming and their associated sequences

$$p' = \langle v'', \ldots, v', \ldots, v'', \ldots v' \rangle \subset p \Rightarrow |p'| < |p|$$
$$p'' = \langle v, \ldots, v'', \ldots, v', \ldots, v'' \rangle \subset p \Rightarrow |p''| < |p|$$

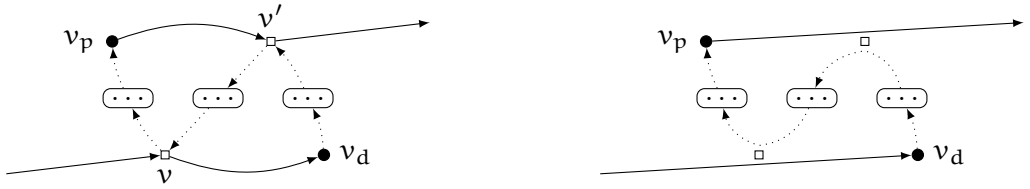
are proper subsequences of p.

(iii) There is a dependence $p_2 \rightarrow p_3$ as illustrated in Figure 4.4d. Then there must be a drop node v'' such that the drop-cycles $\Pi(r, v)$ and $\Pi(r'', v'')$ are non-conforming and their associated sequence

$$p' = \langle v, \ldots, v'', \ldots, v', \ldots, v'' \rangle \subset p \Rightarrow |p'| < |p|$$

is a proper subsequence of p.

Thus, as long as non-conforming pairs exist in Π, we can find a pair whose sequences p_1, p_2 and p_3 are independent by iterating cases (i)–(iii) and continuing the search over subsequence p' instead of p. The property $|p'| < |p|$ guarantees that the search terminates.



(a) Non-conforming drop-cycles (b) Transformed and cycles removed

Figure 4.5: Removal of non-conforming drop-cycles with independent subsequences.

Suppose v and v' are the drop nodes corresponding to non-conforming drop-cycles satisfying the independence condition for subsequences p_1, p_2 and p_3. Then we modify the sequence and perform the path $\langle v\rangle \oplus p_3 \oplus \langle v'\rangle \oplus p_2 \oplus \langle v\rangle \oplus p_1 \oplus \langle v'\rangle$ instead of p, thereby eliminating both drops and their corresponding cycles. This transformation is illustrated in Figure 4.5. The cost of the resulting sequence does not increase as the set of arcs remains the same, although consecutive deadheadings and arcs transporting the same request may be merged without cost increase due to the triangle inequality.

Thus, unless Π is BOT-structured, we can find two non-conforming drop-cycles and remove them by restructuring the sequence. This transformation does not introduce any new drop nodes and thus no further drop-cycles. Iterating the given approach transforms Π into a BOT-structured sequence Π' without increased cost. The BOT-structured solution S derived from Π' has equal cost, i.e., $c(S) = c(\Pi') \leqslant c(\Pi)$. Given that Π is optimal, then S is optimal as well. □

From Theorem 4.1 Quilliot et al. [88] conclude that solving the asymmetric PSCP is essentially a tree construction problem.

There are various direct implications of Theorem 4.1, such that there always exists a solution satisfying the following properties. First, if two drop cycles overlap, then one is contained in the other or vice versa. As empty drop cycles are unnecessary, each drop cycle consumes at least one request and therefore at most $|R| - 1$ drops are required and at least one request is never dropped. Also, as the child sequence of a BOT's drop node corresponds to a sequence of requests, the first node in a drop cycle corresponds to a pickup and the last node corresponds to a delivery, respectively. Besides these direct implications of the BOT structure, the following properties constrain the search space even further.

Proposition 4.1. *There exists an optimal solution for the PSCP such that*

(i) each node is used for at most one drop,

(ii) no request is dropped at the depot node v_0,

(iii) active implicit drop nodes contain their corresponding request at the beginning or end of their drop cycle.

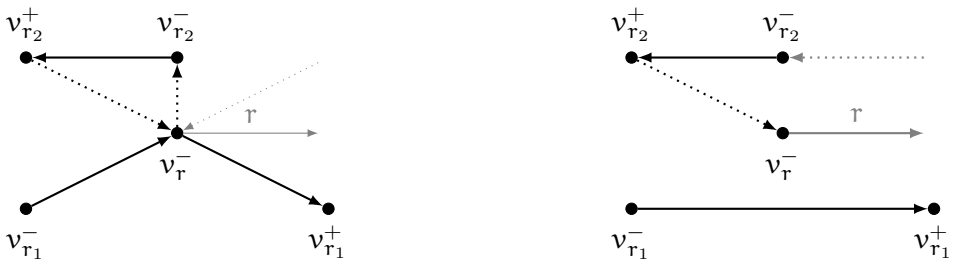
Proof. Properties (i) and (ii) are already proven in Anily and Hassin [6] for the PSP. Property (i) is proven again in Quilliot et al. [88] for the PSCP on solutions represented by sequences Π. With respect to BOTs, proofs for both properties are rather intuitive, hence, we include them for the sake of completeness. In contrast, to the best of our knowledge, property (iii) has not been stated before.

Suppose an optimal consistent BOT solution. All three properties will be proven by restructuring the BOT. Assuming property (i) is not satisfied, there must be at least two drop nodes $v_i = v_j$ in the BOT. Without loss of generality the child sequence of v_i is appended to the child sequence of v_j and v_i is removed from the BOT, i.e., $\sigma(v_j) \leftarrow \sigma(v_j) \oplus \sigma(v_i)$. Due to the triangle inequality, the cost of the resulting BOT does not increase. This approach is iterated until each node is used for at most one drop. Property (ii) is a special case of property (i) and proven in the same way, i.e., the child sequence of any drop node $v' = v_0$ associated with the depot node is appended to the depot's child sequence $\sigma(v_0) \leftarrow \sigma(v_0) \oplus \sigma(v')$ and removed from the BOT.

Regarding property (iii), suppose that a drop is performed at an implicit drop node $v \in \{v_r^-, v_r^+\}$ and that the request path of the corresponding request r is not contained in the induced drop cycle as illustrated in Figure 4.6a. As request r is not contained in $\sigma(v)$, there must be another drop node $v' \in V$ with $r \in \sigma(v')$. As such, the sequence $\sigma(v)$ is inserted either prior to r in $\sigma(v')$ if $v = v_r^-$ or after r if $v = v_r^+$ and the drop node v is removed from the tree. This scenario is illustrated in Figure 4.6b for $v = v_r^-$. The cost of the resulting BOT does not increase due to the triangle inequality.

This restructuring is impossible when r is in fact contained in $\sigma(v)$. Hence, a drop at an implicit drop node is only necessary if the request path of the corresponding request is contained in the induced drop cycle.

If r is contained in $\sigma(v)$ but neither at beginning nor the end, then $\sigma(v)$ can be reordered appropriately without cost increase due to the triangle inequality. □



(a) Drop at implicit drop node

(b) Rerouted and drop removed

Figure 4.6: (a) shows request r_1 being dropped at the implicit drop node v_r^-, but request r itself is not contained in the induced drop cycle. Hence, there is another visit of v_r^- to pickup r, shown in gray. Due to Proposition 4.1 the drop can be removed as shown in (b).

Therefore the search space can be restricted to consistent BOTs that satisfy the properties of Proposition 4.1. We call a consistent BOT satisfying these properties *canonical*.

4.3.2 Benefits of preemption

The main reason for preemption and transfers is the possible decrease in deadheadings and therefore in total cost. This gives rise to the question regarding the relationship between the costs of an optimal non-preemptive and an optimal preemptive solution. First, a result for the PSP on a tree is reproduced and then extended to the PSCP.

Theorem 4.2 (Anily et al. [4])**.** *The costs of optimal solutions to the preemptive swapping problem on a tree* $c(S^\star_{\text{PSP-tree}})$ *and the non-preemptive swapping problem on a tree* $c(S^\star_{\text{SP-tree}})$ *satisfy*

$$c(S^\star_{\text{SP-tree}}) \leqslant 2c(S^\star_{\text{PSP-tree}})$$

In other words, preemption may reduce the cost by up to 50%*. The given bound is tight.*

Anily et al. [4] provide a proof by transforming a preemptive solution into a non-preemptive solution while using each arc of the original solution at most twice. They also construct an example that actually realizes the given bound. The connections between the PSP-tree and the PSCP are twofold: (i) both are special cases of the PSP and (ii) both exhibit tree structured solutions due to the underlying tree structure in case of the PSP-tree and Theorem 4.1 in case of the PSCP, respectively. Hence, some results for the PSP-tree can be transferred to the PSCP, e.g., Algorithm 4.1 to transform a preemptive PSCP solution into a non-preemptive SCP solution.

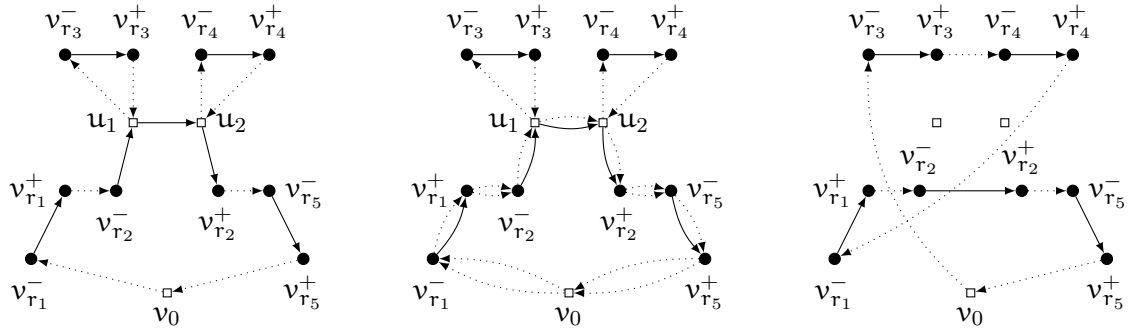
Algorithm 4.1 Remove preemption

Input: BOT rooted at drop node $v \in V$
Output: Non-preemptive solution sequence
Let $\sigma(v) = \langle r_1, \ldots, r_k \rangle$. We perform the following two-step recursive procedure:

1. If at least one of requests $r_i \in \sigma(v)$ has a non-empty subtree $|\sigma(r_i)| \geqslant 1$, then traverse the path
$$p_1 = \langle v \rangle \oplus \tilde{\sigma}(v) \oplus \langle v \rangle,$$
implied by the corresponding request paths. For each drop node $v' \in \tilde{\sigma}(v)$ on path p_1 this procedure is called recursively. After the traversal, arrive back at v.

2. Traverse the path
$$p_2 = \langle v, v^-_{r_1}, v^+_{r_1}, \ldots, v^-_{r_k}, v^+_{r_k}, v \rangle$$
implied by the children again, this time without visiting the drop nodes and instead performing the requests directly. After the traversal, arrive back at v.

To obtain a non-preemptive sequence of requests the procedure is started on the BOT's root node $v_0 \in V$.



(a) Preemptive solution (b) Additional deadheadings (c) Non-preemptive solution

Figure 4.7: Example transformation of a preemptive to a non-preemptive solution.

Example 4.4. *Consider the preemptive solution depicted in Figure 4.7a. When starting Algorithm 4.1 from the depot node* v_0*, we observe that the request path of request* $r_2 \in \sigma(v_0)$

contains two further drop nodes u_1 and u_2, i.e., the subtree of r_2 is non-empty. Thus, the algorithm introduces deadheadings along the path to u_1 as illustrated in Figure 4.7b. The single request r_3 in the subtree rooted at u_1 has an empty subtree itself, thus r_3 is performed directly. The same holds for r_4 and u_2. Then, additional deadheadings from u_2 back to v_0 are introduced concluding the first step of the algorithm. In the second step, all requests in $\sigma(v_0)$ are performed along their original request paths, but without performing the actual drops. Finally, consecutive deadheadings and consecutive transportations of the same request are merged resulting in the non-preemptive solution shown in Figure 4.7c.

Lemma 4.1. *Algorithm 4.1 constructs a feasible non-preemptive solution S' from a consistent BOT S with cost $c(S') \leqslant 2c(S)$.*

Proof. The proof is similar to the proof given by Anily et al. [4] for Theorem 4.2. Assume a drop node v in the BOT supplied to Algorithm 4.1. The path from v through all the pickups, request paths and deliveries of $\sigma(v)$ and back to v without contained drop cycles is

$$p = \langle v, v^-_{r_1} \rangle \oplus \sigma(r_1) \oplus \langle v^+_{r_1}, \ldots, v^-_{r_k} \rangle \oplus \sigma(r_k) \oplus \langle v^+_{r_k}, v \rangle.$$

The paths p_1 and p_2 are subsequences of p and therefore $c(p_1) + c(p_2) \leqslant 2c(p)$ holds. For each drop node v the algorithm traverses p_1 and p_2 thereby at most doubling the cost. Pickups and deliveries are only performed during the traversal of p_2 and are therefore non-preemptive and feasible. □

Using Algorithm 4.1 the result of Anily et al. [4] can be extended to the PSCP.

Proposition 4.2. *The costs of optimal solutions to the preemptive stacker crane problem $c(S^\star_{PSCP})$ and the non-preemptive stacker crane problem $c(S^\star_{SCP})$ satisfy*

$$c(S^\star_{SCP}) \leqslant 2c(S^\star_{PSCP})$$

The bound is tight, even for points on the line.

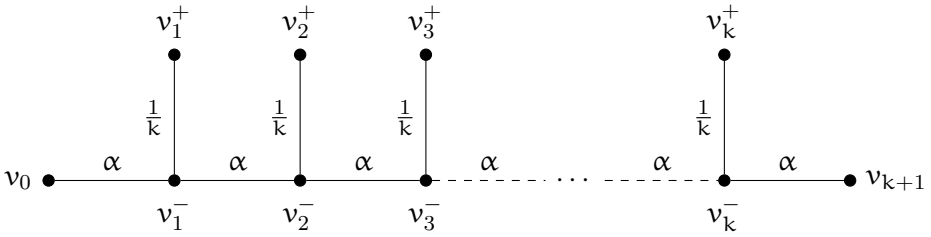


Figure 4.8: Instance construction adapted from Anily et al. [4] with $k + 2$ requests $R = \{(v_0, v_{k+1}), (v_{k+1}, v_0), (v_1^-, v_1^+), \ldots, (v_k^-, v_k^+)\}$. All costs are symmetric, $\alpha > 0$ and all unspecified costs derive from the shortest paths via specified edges. The depot coincides with v_0.

Proof. The relationship $c(S^\star_{SCP}) \leqslant 2c(S^\star_{PSCP})$ follows directly from Lemma 4.1. To prove the tightness, the proof given by Anily et al. [4] for the PSP on a tree is reproduced. Consider the instance provided in Figure 4.8. The optimal preemptive solution has cost $c(S^\star_{PSCP}) = 2(k + 1)\alpha + 2$. A non-preemptive solution may start satisfying the requests (v_r^-, v_r^+), $r \in \{1, \ldots, k\}$ from v_0, v_{k+1} or both. This will incur at least $2k\alpha$ additional cost. As k approaches infinity, the bound is realized:

$$\lim_{k\to\infty} \frac{c(S^\star_{SCP})}{c(S^\star_{PSCP})} = \lim_{k\to\infty} \frac{2(k+1)\alpha + 2k\alpha + 2}{2(k+1)\alpha + 2} = 2$$

Additionally, as the cost $\frac{1}{k}$ approaches zero, the constructed graph is collapsed to a line. □

Note that the result implies that each α-approximation algorithm for the SCP is 2α-approximation algorithm for the PSCP, e.g., the algorithm of Frederickson et al. [42] provides a $\frac{18}{5}$-approximation or better for the PSCP.

4.3.3 Benefits of explicit drop nodes

As noted in Section 4.1, additional nodes or *explicit drop nodes* not associated with pickup or delivery operations may enable further cost savings when preemption is allowed. In a realistic setting this leads to the question whether it makes sense to look out for these explicit drop nodes and increase the set of explicit drop nodes to facilitate an even larger reduction in total cost. Thus, in the following paragraphs we study the maximum benefits of explicit drop nodes.

Due to Proposition 4.2, the maximum improvement possible by considering an optimal set of explicit drop nodes is bounded by 50% because the benefit of explicit drop nodes cannot exceed the benefit of preemption itself. First, we show that it is not possible to improve the bound for asymmetric costs.

Proposition 4.3. *The costs for optimal solutions to the asymmetric PSCP* $c(S^\star_{PSCP})$ *and the asymmetric PSCP without explicit drop nodes* $c(S^\star_{PSCP_\emptyset})$ *satisfy*

$$c(S^\star_{PSCP_\emptyset}) \leqslant 2c(S^\star_{PSCP})$$

The bound is tight.

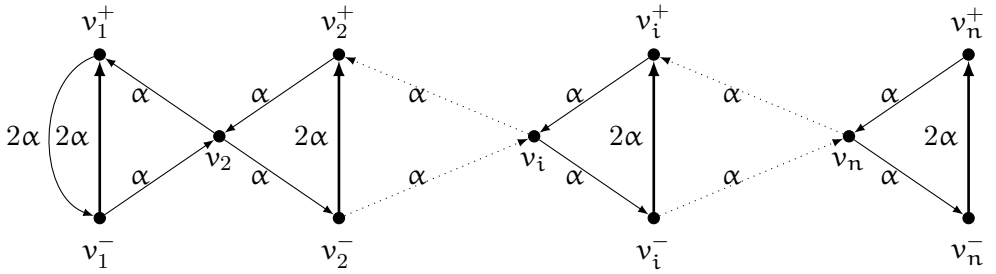


Figure 4.9: Instance construction for n requests $R = \{r_1, \ldots, r_n\}$. Specified costs are $\alpha > 0$ and unspecified costs equal the lengths of shortest directed paths via specified arcs, e.g. $c(v_1^-, v_n^-) = 2(n-1)\alpha$. The depot coincides with node v_1^-.

Proof. Due to $c(S^\star_{PSCP_\emptyset}) \leqslant c(S^\star_{SCP})$ the bound follows transitively from Proposition 4.2. The tightness is shown by the example instance given in Figure 4.9. The minimum cost between any pair of distinct pickup or delivery nodes $v_i, v_j \in \{v_1^-, \ldots, v_n^-, v_1^+, \ldots, v_n^+\}$ is $c(v_i, v_j) \geqslant 2\alpha$. Hence, the cost of a closed tour visiting all these nodes provides a lower bound $LB_{PSCP}(n) = 4n\alpha$ for the total cost of any preemptive solution.

The optimal preemptive solution starts by picking up request r_1 and drops it at v_2 for request r_2. Request r_2 is dropped at v_3 for r_3 and so on until request r_n is performed directly. Afterwards the dropped requests are reloaded and completed in reverse order for a total cost $c(S^\star_{PSCP}(n)) = 4n\alpha = LB_{PSCP}(n)$.

Assuming that drops at the explicit drop nodes $v_2, \ldots, v_n$ are prohibited, there are three options: (i) drop request $r_i \in R\backslash\{r_n\}$ at node v_{i+1}^-, (ii) at node v_{i+1}^+ or (iii) perform requests $r_i \in R$ directly, without a drop. All options incur additional cost of 4α per request dropped

in the preemptive solution $S^\star_{\text{PSCP}}$. This observation yields the lower bounds $\text{LB}_{\text{SCP}}(n) = \text{LB}_{\text{PSCP}_\varnothing}(n) = \text{LB}_{\text{PSCP}}(n) + 4(n-1)\alpha = 8(n-1)\alpha + 4\alpha$. A solution realizing $c(S^\star_{\text{PSCP}_\varnothing}(n)) = \text{LB}_{\text{PSCP}_\varnothing}(n)$ can be constructed by dropping requests $r_i \in R \backslash \{r_n\}$ at ν^-_{i+1} instead of ν_{i+1}.

Then, the ratio of costs $c(S^\star_{\text{PSCP}_\varnothing}(n))$ and $c(S^\star_{\text{PSCP}}(n))$ realizes the bound as n approaches infinity:

$$\lim_{n\to\infty} \frac{c(S^\star_{\text{PSCP}_\varnothing}(n))}{c(S^\star_{\text{PSCP}}(n))} = \lim_{n\to\infty} \frac{8(n-1)\alpha + 4\alpha}{4n\alpha} = \lim_{n\to\infty} \frac{8(n-1)\alpha + 4\alpha}{4(n-1)\alpha + 4\alpha} = 2$$

□

The established bound on the benefits of explicit drop nodes is as large as the bound on the benefits of preemption itself. While the bound on the benefits of preemption is tight even for points on the line, the bound on the benefits of explicit drop nodes can be further tightened for special cases. First, we consider symmetric costs.

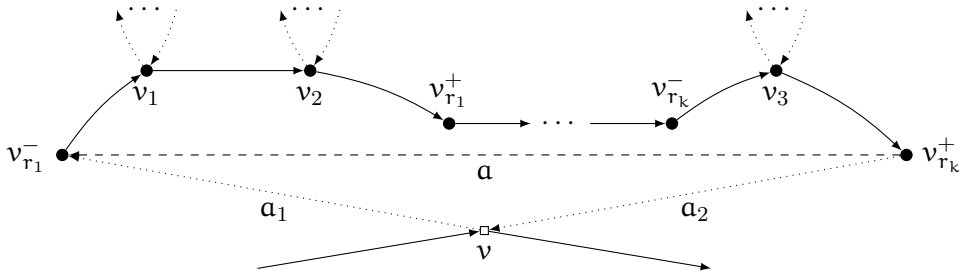


Figure 4.10: Example drop node ν with two deadheadings a_1, a_2 connecting it to its subsequence $\sigma(\nu) = \langle r_1, \ldots, r_k \rangle$ with $\sigma(r_1) = \langle \nu_1, \nu_2 \rangle$ and $\sigma(r_k) = \langle \nu_3 \rangle$. Assuming symmetric costs, the cost of arc a is bounded by both, the combined costs of arcs a_1, a_2 and by the cost of the solid path from $\nu^-_{r_1}$ to $\nu^+_{r_k}$.

Proposition 4.4. *The costs for optimal solutions to the symmetric PSCP* $c(S^\star_{PSCP})$ *and the symmetric PSCP without explicit drop nodes* $c(S^\star_{PSCP_\varnothing})$ *satisfy:*

$$c(S^\star_{PSCP_\varnothing}) \leqslant \frac{3}{2} c(S^\star_{PSCP})$$

That is, introducing explicit drop nodes may improve the costs by at most $33.\overline{3}\%$*. The bound is tight.*

Proof. Consider an optimal solution $S^\star_{\text{PSCP}}$ represented by a consistent BOT. This solution is transformed into a solution $S^\star_{\text{PSCP}_\varnothing}$ by replacing all drops at explicit drop nodes with drops at implicit drop nodes. Let ν be an explicit drop node in $S^\star_{\text{PSCP}}$ and let $\sigma(\nu) = \langle r_1, \ldots, r_k \rangle$ be its sequence of requests as depicted in Figure 4.10. The drop node ν is connected to its sequence via two deadheading arcs $a_1 = (\nu, \nu^-_{r_1})$ and $a_2 = (\nu^+_{r_k}, \nu)$. If $c(a_1) \leqslant c(a_2)$ then the drop at ν is replaced by a drop at $\nu^-_{r_1}$, otherwise it is replaced by a drop at $\nu^+_{r_k}$. In both cases the additional deadheading $a = (\nu^+_{r_k}, \nu^-_{r_1})$ is performed, either to get from the drop node $\nu^+_{r_k}$ to the first pickup of the drop cycle $\nu^-_{r_1}$ or to get from the last delivery of the drop cycle $\nu^+_{r_k}$ to the drop node $\nu^-_{r_1}$. The deadheadings a_1, a_2 of the prior drop node ν are replaced by either two traversals of a_1 or two traversals of a_2, respectively.

The path from the first pickup of the drop cycle through all the other pickups, request paths and deliveries of $\sigma(\nu)$ to the last delivery of the drop cycle without contained drop

cycles is

$$p = \langle \nu^-_{r_1} \rangle \oplus \sigma(r_1) \oplus \langle \nu^+_{r_1}, \ldots, \nu^-_{r_k} \rangle \oplus \sigma(r_k) \oplus \langle \nu^+_{r_k} \rangle$$

and due to symmetry, the cost of the deadheading a is bounded by the cost of this path: $c(a) \leqslant c(p)$. Note that path p may contain further explicit drop nodes ν', e.g. ν_1, ν_2, ν_3 in Figure 4.10, but does not contain any deadheadings connecting these drop nodes to their sequences $\sigma(\nu')$. Likewise, due to symmetry and the triangle inequality, the cost of the deadheading a is bounded by the costs of the deadheadings connecting the drop node ν to its sequence: $c(a) \leqslant c(a_1) + c(a_2)$. Without loss of generality, assume $c(a_1) \leqslant c(a_2)$. Then, the way from the prior drop node ν to the new drop node $\nu^-_{r_1}$ and back incurs cost $2c(a_1) \leqslant c(a_1) + c(a_2)$. Therefore, the cost associated with the drop ν in $S^\star_{\text{PSCP}}$ is $c_{\text{PSCP}} = c(a_1) + c(a_2) + c(p)$ and the cost associated with the drop $\nu^-_{r_1}$ in $S^\star_{\text{PSCP}_\varnothing}$ is $c_{\text{PSCP}_\varnothing} = 2c(a_1) + c(p) + c(a)$.

$$\begin{aligned}
\frac{c_{\text{PSCP}_\varnothing}}{c_{\text{PSCP}}} &= \frac{2c(a_1) + c(p) + c(a)}{c(a_1) + c(a_2) + c(p)} \\
&\leqslant \frac{c(a_1) + c(a_2) + c(p) + c(a)}{c(a_1) + c(a_2) + c(p)} \\
&= 1 + \frac{c(a)}{c(a_1) + c(a_2) + c(p)} \\
&\leqslant 1 + \frac{c(a)}{2c(a)} = \frac{3}{2}
\end{aligned}$$

That means, for each explicit drop node, the cost of its associated deadheading $a_1 + a_2$ is at most doubled. These deadheading arcs a_1, a_2 are themselves never contained in any path p, hence the cost being doubled is at most 50% of the total cost.

Tightness is shown by example. Again, the instance depicted in Figure 4.9 is used, but the arc orientations are ignored and only the undirected graph is considered, thus ensuring symmetric costs. As before, the optimal preemptive solution has total cost $c(S^\star_{\text{PSCP}}(n)) = 4n\alpha$. In contrast to the asymmetric case, prohibiting drops at the explicit drop nodes $\nu_2, \ldots, \nu_n$ does only incur additional costs of 2α per request dropped in the preemptive solution. This holds for all possible options, i.e., drops at ν^-_i, drops at ν^+_i and no drops at all. As such, the lower bounds on the total cost are $\text{LB}_{\text{SCP}}(n) = \text{LB}_{\text{PSCP}_\varnothing}(n) = \text{LB}_{\text{PSCP}}(n) + 2(n-1)\alpha = 6(n-1)\alpha + 4\alpha$. A solution realizing $\text{LB}_{\text{PSCP}_\varnothing}(n)$ can be constructed by dropping requests $r_i \in R \backslash \{r_n\}$ at nodes ν^-_{i+1}. The ratio of the costs realizes the bound as n approaches infinity:

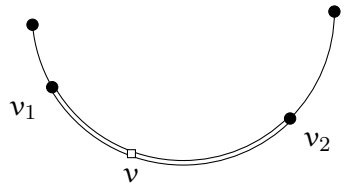
$$\lim_{n\to\infty} \frac{c(S^\star_{\text{PSCP}_\varnothing}(n))}{c(S^\star_{\text{PSCP}}(n))} = \lim_{n\to\infty} \frac{6(n-1)\alpha + 4\alpha}{4n\alpha} = \lim_{n\to\infty} \frac{6(n-1)\alpha + 4\alpha}{4(n-1)\alpha + 4\alpha} = \frac{3}{2}$$

□

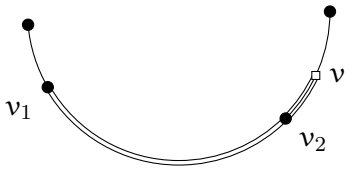
When the structure is constrained further to line- and circular geometric structures, explicit drop nodes do not provide any benefit, i.e., it is sufficient for drops to occur at implicit drop nodes.

Proposition 4.5. *Explicit drop nodes cannot improve the costs of optimal solutions to the PSCP$_\varnothing$ on line- and circular geometric structures.*

Proof. Suppose an optimal PSCP solution on a circle that contains a drop at an explicit drop node ν. Due to the symmetric distances, ν minimizes the sum of distances to exactly four nodes, the nodes before and after ν on the request path and the first and the last node of the induced drop cycle. As ν is part of an optimal solution, two of these nodes must



(a) Optimal drop node v



(b) Non-optimal drop node v

Figure 4.11: A drop node v connected to four points on a circle or line. The drop node v is optimally placed and minimizes the distance to all four nodes as long as it is located on the arc between v_1 and v_2.

lie on either side of v. Moving v to the right or left would increase the distance to the two nodes on the side being moved from, but would also decrease the distance to the two nodes on the side being moved to by the same margin. Therefore all points between the drop node's nearest right and left neighbors incur equal cost. This property is illustrated in Figure 4.11. Hence, v may be moved to the left or the right until it coincides with one of the four nodes it is connected to. Let v' be that node. Two cases need to be considered. If v' is an implicit drop node, then an explicit node has been transformed into an implicit drop node without increasing the cost. If otherwise v' is also an explicit drop node, then the drop cycles associated with v and v' can be merged into a single drop cycle due to Proposition 4.1. Both cases do not increase the cost of the solution. Hence, the argument is iterated until all explicit drop nodes are removed. The result also holds for points on the line, as a line can be transformed into a circle by connecting its endpoints by a sufficiently large arc. □

4.4 Reduced representations

In the context of the SPPDP and the PSCP, Kerivin et al. [65] consider the concept of *minimal solution representations*. They define a minimal solutions representation in the following way:

Definition 4.2 (Minimal representation [65])**.** *A* tractable representation *must contain enough information to assert in polynomial time whether or not a feasible solution can be obtained from it. A* (inclusionwise) minimal representation *is a tractable one from which no information can be removed without losing the polynomial tractability.*

A pedantic interpretation of this definition might assert that for any problem exhibiting a polynomial-time tractable feasibility problem the zero-bit empty word is a legitimate minimal representation because a feasible solution can always be obtained from the problem instance without any extra knowledge. However, we are confident that the objective value of the encoded solution is relevant as well, although it is not explicitly mentioned in the definition.

Therefore, we refrain from the use of the term *minimal representation* and instead use the term *reduced representation* adopting the notion of redundant information that can be recovered in polynomial time. In more detail, we assume a solution S in a specific representation, e.g., a BOT with cost $c(S)$. If information can be omitted from said representation and the remaining information is sufficient to obtain a solution S' with $c(S') \leqslant c(S)$ in polynomial time, then the original representation without this polynomial-time recoverable information corresponds to a *reduced representation*.

For the PSCP Kerivin et al. [65] show that the set of request paths in conjunction with the set of arcs traversed by the vehicle provide a reduced solution representation. A correspond-

ing optimal ordering of the vehicle's arcs that satisfies all request paths can be calculated in polynomial time.

In the following we take a similar route and show that canonical BOTs actually encode redundant, polynomial-time recoverable information. Nevertheless, canonical BOTs remain a sensible solution representation for heuristic construction and improvement methods for the PSCP. We develop two algorithms to (i) obtain a minimum-cost request path for a given request and a sequence of child requests and (ii) obtain a minimum-cost BOT for a sequence of requests. Both algorithms are based on the computation of shortest s-t-paths in *directed acyclic graphs* (DAGs) $G = (W, A)$ with $\mathrm{indeg}(s) = \mathrm{outdeg}(t) = 0$. Such a shortest path can be calculated in $\mathcal{O}(|A|)$ time by topological traversal.

4.4.1 The request path problem

Problem 4.1 (*Request path problem* (RPP))**.** *Given a request* $r \in R$ *and a sequence of unique requests* $\pi = \langle r_1, \dots, r_k \rangle$ *with* $r_i \neq r$*,* $1 \leqslant i \leqslant k$*. Find a minimum-cost request path* $\sigma(r)$ *such that* $\tilde{\sigma}(r) = \pi$*.*

In other words, the goal of the RPP is to determine a *composition* of the input request sequence π into $1 \leqslant k' \leqslant k$ consecutive non-overlapping segments and a sequence of drop nodes $\sigma(r) = \langle \nu_1, \dots, \nu_{k'} \rangle$ of length k'. The i-th segment corresponds to the child sequence $\sigma(\nu_i)$ of the i-th drop node $\nu_i \in \sigma(r)$. The cost of the resulting request path and its connections to the segments must be minimum.

Let $D = (W, A)$ be the auxiliary DAG with nodes $W = \{x_{iu} \colon 1 \leqslant i \leqslant k, u \in V\} \cup \{s, t\}$. The nodes s, t correspond to artificial source and sink nodes, respectively. A node $x_{iu} \in W$ represents the request r_i as part of a child request sequence rooted at drop node u, i.e., $r_i \in \sigma(u)$. The arc set $A = A_s \cup A_t \cup A_x$ is the union of the following disjoint arc sets.

$$
\begin{aligned}
A_s &= \{(s, x_{1u}) \colon x_{1u} \in W\} \\
A_t &= \{(x_{ku}, t) \colon x_{ku} \in W\} \\
A_x &= \{(x_{iu}, x_{i'u'}) \colon x_{iu}, x_{i'u'} \in W, i + 1 = i', 1 \leqslant i < k\}
\end{aligned}
$$

Arcs A_s connect the source node s to all nodes representing the first request in the sequence π. Analogously, arcs A_t connect all nodes representing the last request in the sequence to the sink node t. Arcs A_x connect nodes representing a request at position i with nodes representing the following request at position $i + 1$. The costs $c_a \in \mathbb{R}$ of the arcs $a \in A$ are defined as follows. All cases are illustrated in Figure 4.12.

$$
c_a = \begin{cases}
c(\nu_r^-, u, \nu_{r_1}^-) & \text{if } a = (s, x_{1u}) \in A_s \\
c(\nu_{r_k}^+, u, \nu_r^+) & \text{if } a = (x_{ku}, t) \in A_t \\
c(\nu_{r_i}^+, \nu_{r_{i'}}^-) & \text{if } a = (x_{iu}, x_{i'u'}) \in A_x, u = u' \\
c(\nu_{r_i}^+, u, u', \nu_{r_{i'}}^-) & \text{if } a = (x_{iu}, x_{i'u'}) \in A_x, u \neq u'
\end{cases}
$$

Algorithm 4.2 shows the complete procedure using the DAG D to obtain an optimal solution for a given instance of the RPP. An illustration is provided in Figure 4.13. Note that a drop node may occur multiple times in the resulting request path, although in non-consecutive positions. Such a sequence implies that the input sequence $\pi = \tilde{\sigma}(r)$ can be reordered to derive a canonical BOT without increased cost, according to Proposition 4.1

(a) $a \in A_s$ (b) $a \in A_x$, $u = u'$ (c) $a \in A_x$, $u \neq u'$ (d) $a \in A_t$

Figure 4.12: Illustration of the solution components contributing to the arc costs c_a. The contributing components are drawn in black, while surrounding components are drawn in gray. Deadheadings are drawn dotted, while request carrying arcs are drawn solid.

Algorithm 4.2 Derive request path

Input: Root request r with sequence of immediate child requests $\pi = \langle r_1, \ldots, r_k \rangle$
Output: Cost optimal sequence $\sigma(r)$ with $\tilde{\sigma}(r) = \pi$
Construct the auxiliary DAG $D = (W, A)$ and calculate a shortest s-t-path on D. The resulting path contains $k + 2$ nodes $\langle s, x_{1u_1}, x_{2u_2}, \ldots, x_{ku_k}, t \rangle$. The optimal request path $\sigma(r)$ is obtained in two steps.

1. Derive the path $\sigma(r)' = \langle u_1, \ldots, u_k \rangle$ with $\sigma(u_i) = \langle r_i \rangle$
2. Find all maximally long and consecutive subsequences $p = \langle u_i, \ldots, u_j \rangle \subseteq \sigma(r)'$ corresponding to the same drop node, i.e. $u_{i'} = u_{i''}$ for all $u_{i'}, u_{i''} \in p$ and merge them into a single occurrence of drop node u_i with $\sigma(u_i) \leftarrow \sigma(u_i) \oplus \cdots \oplus \sigma(u_j)$.

Proposition 4.6. *For a given request* $r \in R$ *and its sequence of immediate child requests* $\pi = \langle r_1, \ldots, r_k \rangle$*, Algorithm 4.2 calculates an optimal request path* $\sigma(r)$ *in polynomial time* $\mathcal{O}(|V|^3)$.

Proof. The arc set A_x contains $(k-1)|V|^2 < |V|^3$ arcs. The sets A_s and A_t contain $|V|$ arcs each, respectively. The total number of arcs is $|A| = (k-1)|V|^2 + 2|V| \in \mathcal{O}(|V|^3)$. Calculating a shortest s-t-path in a DAG requires visiting each arc exactly once, hence, the runtime is in $\mathcal{O}(|V|^3)$.

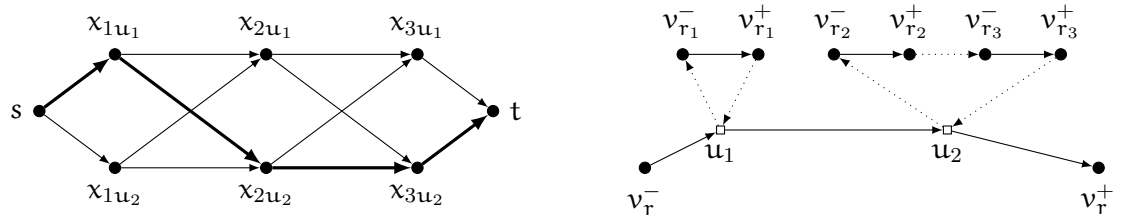
By definition of the arc weights c_a, the cost of an s-t-path corresponds exactly to the cost of the resulting request path including the costs connecting the drop nodes to the first pickup and the last delivery of their drop cycles, respectively. Hence, a shortest s-t-path induces a minimum-cost request path $\sigma(r)$ satisfying the required sequence π. □

The result shows that the search space can be reduced to ordered trees of request nodes, as the optimal request paths for all requests can be calculated in polynomial time. However, the idea of the RPP can be further extended to multiple request paths and thereby to entire BOTs.

4.4.2 The BOT derivation problem

Let S be a consistent BOT structured solution and $\prec_S$ a total order on the request set R with respect to the solution S:

$$r \prec_S r' \Leftrightarrow \begin{cases} \text{rank}_S(r) < \text{rank}_S(r') & \text{if } \text{level}_S(r) = \text{level}_S(r') \\ \text{level}_S(r) < \text{level}_S(r') & \text{otherwise} \end{cases}$$



(a) DAG D with s-t-path

(b) Resulting request path

Figure 4.13: Example with root request r, sequence $\pi = \langle r_1, r_2, r_3 \rangle$ and two drop nodes $u_1, u_2 \in V$. All other possible drop nodes are not depicted for clarity. (a) shows the DAG D and the shortest s-t-path. (b) shows the resulting solution component with sequences $\sigma(r) = \langle u_1, u_2 \rangle$, $\sigma(u_1) = \langle r_1 \rangle$, $\sigma(u_2) = \langle r_2, r_3 \rangle$ and $\tilde{\sigma}(r) = \pi$.

The level of a node a corresponds to the depth of the sequence that a is contained in w.r.t. the root of the BOT, i.e., $\mathrm{level}_S(r_i) = 0$ for $r_i \in \sigma(v_0)$, $\mathrm{level}_S(r_j) = 1$ for $r_j \in \tilde{\sigma}(r_i)$ and so on. The rank is defined for two distinct requests $r \neq r'$, $\mathrm{level}_S(r) = \mathrm{level}_S(r')$ on the same level. If request r is performed prior to r' in S, then $\mathrm{rank}_S(r) < \mathrm{rank}_S(r')$ holds.

As the ordering $\prec_S$ is total, each BOT structured solution S exhibits exactly one permutation $\pi_\prec(S)$ of the requests R. The number of BOTs mapping to the same permutation π is exponential in $|\pi|$ as each *composition* of the permutation into k' consecutive, non-overlapping subsequences corresponds to multiple distinct BOTs with levels $\{0, \ldots, k'-1\}$.

Problem 4.2 (*BOT derivation problem* (BOTDP))**.** *Given a permutation π of all requests R, find a minimum-cost BOT-structured solution*

$$S = \gamma(\pi) := \operatorname*{arg\,min}_{S' \in \mathcal{S}, \pi_\prec(S') = \pi} \{c(S')\}$$

in the set of all solutions $\mathcal{S}$ whose ordering $\pi_\prec(S)$ is consistent with π.

Like the RPP, the BOTDP is solved in polynomial time by calculating a shortest s-t-path in an auxiliary graph.

Without loss of generality, assume a permutation $\pi = \langle r_1, \ldots, r_m \rangle$ of the requests R. For ease of notation, an artificial request r_0 with $v^-_{r_0} = v^+_{r_0} = v_0$ corresponding to a zero cost request at the depot is prepended to the permutation π. Let $D = (W, A)$ be a DAG with nodes $W = \bigcup_{i=1}^{m} W_i$. All nodes in set W_i are associated with request r_i.

$$W_i = \{x_{i0v_o}\} \cup \{x_{iju} \colon 1 \leqslant j < i, u \in V\}$$

A node x_{iju} that is part of a shortest s-t-path implies that among all solutions $S \in \mathcal{S}$ whose ordering $\pi_\prec(S) = \pi$ is consistent with the input permutation π, there exists a minimum-cost solution with $r_i \in \sigma(u)$ and $u \in \sigma(r_j)$, i.e., the *subrequest* r_i is in the subtree of the *root request* r_j via drop node u. The arc set $A = A_s \cup A_t \cup A_x$ is the union of the following disjoint arc sets.

$$\begin{aligned}
A_s &= \{(s, x_{10v_0})\} \\
A_t &= \{(x_{mju}, t) \colon x_{mju} \in W_m\} \\
A_x &= \{(x_{iju}, x_{i'j'u'}) \colon x_{iju}, x_{i'j'u'} \in W, i+1 = i', j \leqslant j'\}
\end{aligned}$$

Arc set A_s contains a single arc from the source node s to the node $x_{1r_0v_0}$. Arc set A_t connects all nodes corresponding to the last request r_m to the sink node t. Finally, arc set

A_x connects all nodes corresponding to request r_i to all nodes corresponding to request r_{i+1} if the *root requests* j, j' satisfy $j \leqslant j'$. That means, given any s-t-path p in D, the relationship $i \leqslant i' \Leftrightarrow j \leqslant j'$ holds for all pairs of nodes x_{iju}, $x_{i'j'u'} \in p$, i.e., each root request will have at most one associated request path. The arc costs $c_a \in \mathbb{R}, a \in A$ are defined as follows.

$$c_a = \begin{cases} c(v_0, v^-_{r_1}) & \text{if } a \in A_s \\ c(v^+_{r_m}, u, v^+_{r_j}) & \text{if } a = (x_{mju}, t) \in A_t \\ c(v^+_{r_i}, v^-_{r_{i'}}) & \text{if } a = (x_{iju}, x_{i'j'u'}) \in A_x, j = j', u = u' \\ c(v^+_{r_i}, u, u', v^-_{r_{i'}}) & \text{if } a = (x_{iju}, x_{i'j'u'}) \in A_x, j = j', u \neq u' \\ c(v^+_{r_i}, u, v^+_{r_j}) + c(v^-_{r_{j'}}, u', v^-_{r_{i'}}) - c(v^-_{r_{j'}}, v^+_{r_{j'}}) & \text{if } a = (x_{iju}, x_{i'j'u'}) \in A_x, j \neq j' \end{cases}$$

The arc weights c_a ensure that the distance of a shortest s-t-path is exactly the variable cost of the solution represented by the path, i.e., each arc's distance corresponds to the cost incurred by the decisions associated with that arc.

Algorithm 4.3 Derive BOT

Input: Permutation of the requests $\pi = \langle r_1, \dots, r_m \rangle$
Output: BOT S with $\pi_\prec(S) = \pi$
Construct DAG $D = (W, A)$ from the sequence $\langle r_0, r_1, \dots, r_m \rangle$ including the dummy request r_0. Let $p = \langle s, y_1, \dots, y_m, t \rangle$ be a shortest s-t-path in D with nodes $y_i \in W_i$. Furthermore, let $u(y_i) \in V$ be the associated drop node and $j(y_i) \in \{0, \dots, i-1\}$ be the associated root request. The solution associated with p is obtained in two steps:

1. Find each maximal subsequence $p' = \langle y_i, \dots, y_k \rangle \subseteq p$ with $u(y_\ell) = u(y_{\ell'})$ and $j(y_\ell) = j(y_{\ell'})$ for all $y_\ell, y_{\ell'} \in p'$. Merge the subsequence into a drop cycle $\sigma(u(y_i)) \leftarrow \langle r_i, \dots, r_k \rangle$ and remove all nodes $p' \backslash \{y_i\}$ from path p.

2. Find each maximal subsequence $p'' = \langle y_i, \dots, y_k \rangle$ of p with $j(y_\ell) = j(y_{\ell'})$ for all $y_\ell, y_{\ell'} \in p''$. Merge the subsequence into a request path $\sigma(r_{j(y_i)}) \leftarrow \langle u(y_i), \dots, u(y_k) \rangle$. Note that due to the structure of the arc set A_x, there is at most one subsequence for each root request r_j.

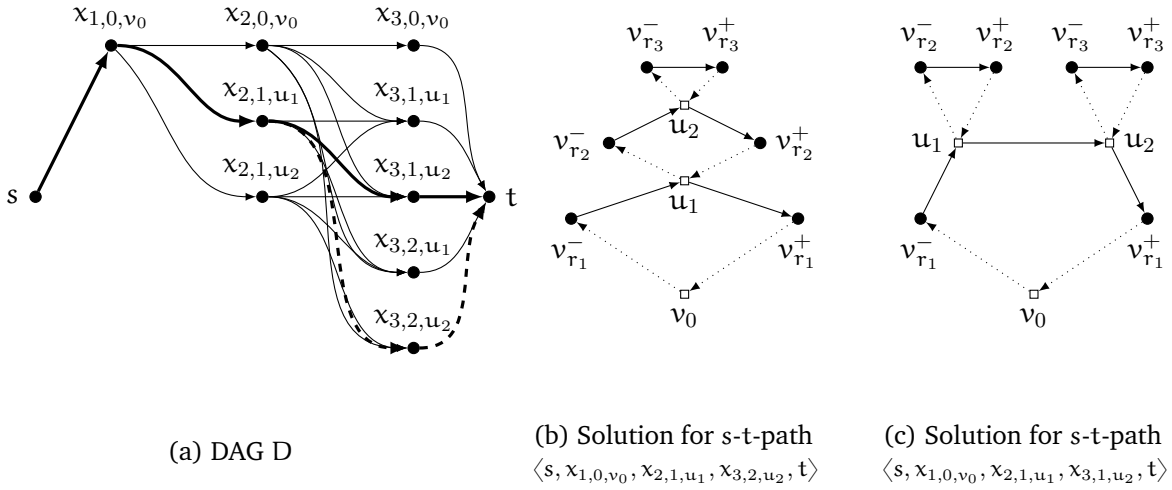
The complete algorithm to calculate $\gamma(\pi)$ is given in Algorithm 4.3 and an illustrative example is provided in Figure 4.14. The resulting BOT may not be canonical as single drop nodes may occur multiple times. The restructuring approaches described in the proof of Proposition 4.1 should be applied to obtain a canonical BOT. However, these restructurings may change the permutation $\pi_\prec(S)$ accordingly.

Proposition 4.7. *For a permutation π of the requests R, Algorithm 4.3 calculates $\gamma(\pi)$ in time $\mathcal{O}(|V|^5)$.*

Proof. Every solution S induced by an s-t-path in D satisfies $\pi_\prec(S) = \pi$. Conversely, for each canonical BOT structured solution S satisfying $\pi_\prec(S) = \pi$ there is an s-t-path in D. This follows directly from the definition of the arc set A.

By definition of the arc weights c_a, the cost of an s-t-path equals the cost of the solution without the fixed cost $\sum_{r \in R} c(v^-_r, v^+_r)$. As such, a shortest s-t-path induces a minimum-cost solution satisfying $\pi_\prec(S) = \pi$.

Each node set W_i contains $\mathcal{O}(|R| \cdot |V|) \subseteq \mathcal{O}(|V|^2)$ nodes and generates $\mathcal{O}(|V|^4)$ arcs to its subsequent node set W_{i+1}. Hence, the total number of arcs $|A| \in \mathcal{O}(|R| \cdot |V|^4) \subseteq \mathcal{O}(|V|^5)$ is polynomial in $|V|$. Calculating a shortest s-t-path on a DAG requires visiting each arc



(a) DAG D

(b) Solution for s-t-path $\langle s, x_{1,0,v_0}, x_{2,1,u_1}, x_{3,2,u_2}, t\rangle$

(c) Solution for s-t-path $\langle s, x_{1,0,v_0}, x_{2,1,u_1}, x_{3,1,u_2}, t\rangle$

Figure 4.14: Example BOTDP instance with $\pi = \langle r_1, r_2, r_3\rangle$, the depot v_0 and two drop nodes u_1, u_2. All other possible drop nodes are excluded for clarity. (a) shows the DAG, (b) shows the solution corresponding to the dashed path and (c) shows the solution corresponding to the solid path. The path $\langle s, x_{1,0,v_0}, x_{2,0,v_0}, x_{3,0,v_0}, t\rangle$ corresponds to the single non-preemptive solution S satisfying $\pi_\prec(S) = \pi$.

exactly once. Transforming the s-t-path into a BOT requires only $\mathcal{O}(|R|)$ time. Hence, the total runtime is bounded by the size of the arc set. □

The result implies that the search space for the PSCP is as large as the number of request permutations $|R|!$. If the number of explicit drop nodes is bounded by a polynomial in the number of requests $|V| \in \mathrm{poly}(|R|)$, then the search space is as large as for the ATSP on $|R|+1$ nodes.

4.4.3 Induced neighborhoods

Algorithms 4.2 and 4.3 give rise to exponential neighborhoods $\mathcal{N}_{\text{RPP}}$ and $\mathcal{N}_{\text{BOTPD}}$, respectively. For a solution S, the neighborhood induced by the RPP

$$\mathcal{N}_{\text{RPP}}(S) = \{S' \in \mathcal{S}\colon \sigma_{S'}(v_0) = \sigma_S(v_0), \forall r \in R\colon \tilde{\sigma}_{S'}(r) = \tilde{\sigma}_S(r)\}$$

contains all solutions whose request tree structure is identical while the actual request paths may differ. Similarly, the neighborhood induced by the BOTDP

$$\mathcal{N}_{\text{BOTPD}}(S) = \{S' \in \mathcal{S}\colon \pi_\prec(S') = \pi_\prec(S)\}$$

contains all solutions mapping to the same permutation $\pi_\prec$. Both neighborhoods allow that a best neighbor can be found in polynomial time, however, on their own, both neighborhoods are impractical for two reasons: First, the neighborhoods are idempotent w.r.t. the best neighbors, i.e., searching for the best neighbor of a best neighbor will not provide an improving solution. Second, the neighborhoods are not connected because all neighbors exhibit the same sequences of child requests in case of the RPP and the same request permutation in case of the BOTDP, respectively. Nevertheless, these neighborhoods and the respective procedures to obtain best neighbors may be used for solution polishing in the final steps of heuristic procedures.

4.5 Tree-based construction methods

This section proposes two construction heuristics adapted for the PSCP from well-known construction heuristics for the ATSP and the *capacitated vehicle routing problem* (CVRP): the *modified Karp-Steele patching heuristic* of Glover et al. [48] and the *savings heuristic* of Clarke and Wright [28], respectively. Furthermore, a short description of the *Monte-Carlo insertion heuristic* proposed by Quilliot et al. [88] is provided to illustrate the variants studied in the computational study in Section 4.6.

4.5.1 Monte-Carlo insertion (MCI)

Quilliot et al. [88] propose a simple *Monte-Carlo insertion heuristic* (MCI). Starting from an empty BOT, the requests are inserted, one-by-one, in a random order. For each request, all insertion positions are evaluated, with and without drops. The cheapest insertion is performed.

To increase the algorithms efficiency, the original authors proposed precomputed neighborhoods to generate restricted sets of candidate drop nodes during the evaluation of insertion positions. To respect this proposal, we study three restrictions.

- **Full (F)**. All nodes are considered in the candidate set. There are no restrictions.
- **Only-explicit (E)**. Only explicit drop nodes and the implicit drop nodes implied by Proposition 4.1 are considered, i.e., the pickup and delivery nodes of the request that is to be inserted.
- **Neighborhood-2 (N)**. Each candidate set contains at least two neighbors which are calculated according to the requirements stated by Quilliot et al. [88]. The neighborhood of a node v consists of all other nodes v' with cost $c(v, v') \leqslant \mathcal{R}$ where $\mathcal{R}$ is the so-called *radius*. We choose the minimum radius, such that each node has at least two neighbors. Additionally, for all pairs of nodes v_i, v_j a *midpoint* $\text{mid}(v_i, v_j)$ is calculated. Candidate reload nodes are searched for in the neighborhood of midpoint $m = \text{mid}(m_1, m_2)$. The point m_1 corresponds to the midpoint between the prior reload node on the request path being inserted into and the pickup node of the request being inserted. Likewise, point m_2 corresponds to the midpoint between the delivery node of the request being inserted and the following reload node on the request path being inserted into.

The MCI algorithm may be run for a fixed number of iterations, each with a potentially different random permutation of the requests. The best obtained solution over all considered permutations is reported. Additionally, Quilliot et al. [88] describe a VND algorithm consisting of six individual neighborhoods, e.g., exchanging drop nodes and shifting segments in the tree. They use the VND algorithm to improve the solutions obtained by the MCI algorithm. Note that the proposed restrictions of the drop node candidate sets are also applicable to the VND neighborhoods that look up candidate drop nodes.

4.5.2 Modified Karp-Steele patching with drops (MKSD)

Patching heuristics for the ATSP start from a minimum-weight cycle cover of the nodes. As long as the cover contains more than a single cycle, two cycles are chosen and patched together resulting in a new cycle until only a single cycle remains. The calculation of the minimum-weight cycle cover is performed in polynomial time by calculating a minimum-weight perfect matching on a bipartite graph. The heuristic components are therefore limited

to (i) the selection of cycles to be patched, (ii) the selection of the arcs that are to be removed and inserted when patching two chosen cycles and (iii) the possible post-processing of the minimum-weight cycle cover.

In the *modified Karp-Steele patching heuristic* (MKS) of Glover et al. [48], all possible patching options are evaluated and the option with the smallest cost increase is applied. Algorithm 4.4 describes a modification of the MKS heuristic for the PSCP. The idea is to maintain a partial BOT containing the depot node v_0 and to either patch cycles into the partial BOT with or without drops, or to patch two cycles as in case of the ATSP.

Algorithm 4.4 MKSD

Input: PSCP instance

Output: Consistent BOT

First a minimum-weight cycle cover is calculated by solving a minimum-weight bipartite matching problem. Besides the requests R, a dummy request r_0 for the depot is introduced. Its nodes $v^-_{r_0}$ and $v^+_{r_0}$ may not be matched. The cycle containing the depot dummy request is transformed into a BOT rooted at the depot. Let C be the set of the remaining cycles.
As long as C is not empty, an iterative patching process is performed that reduces the cardinality of C by one in each iteration. Three possible patching operations are evaluated.

1. Patching a cycle $c_i \in C$ into the BOT by cutting a deadheading in c_i and inserting the resulting sequence into a request sequence of the BOT. (Figure 4.15b)

2. Patching a cycle $c_i \in C$ into the BOT via a drop node. The cycle c_i is cut on a deadheading and connected to a request path of a request in the BOT with a drop node. (Figure 4.15d)

3. Patching two cycles $c_i, c_j \in C$, $i \neq j$ together into a new cycle c'. (Figure 4.15c)

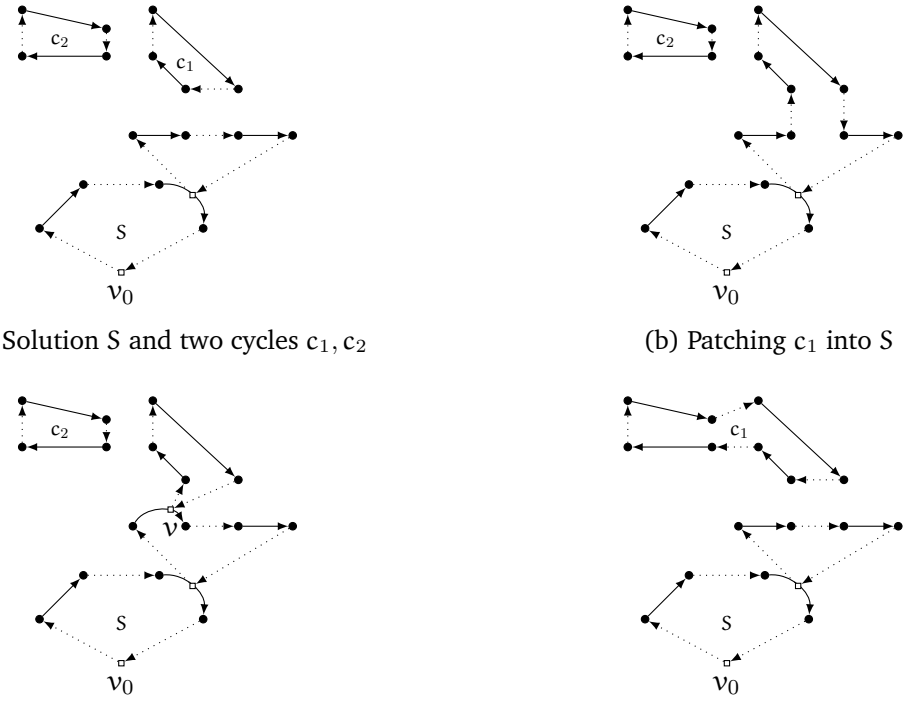
The cheapest patch is performed. Ties are resolved according to the order above. The resulting BOT is consistent, when the set of cycles C is empty.

The algorithmic framework outlined in Algorithm 4.4 leaves certain details unspecified. Based on the methods to calculate a minimum-weight cycle cover and the different patching options considered, we study the following variants.

– **Loops.** When calculating a minimum-weight cycle cover, self-loops may be allowed (Self) or forbidden (NoSelf). A self-loop is formed, when a delivery node v^+_r is matched to the pickup node v^-_r of the same request.

– **Contraction.** A cycle cover may be post-processed by a contraction heuristic as in the *contract-or-patch heuristic* (COP) of Glover et al. [48]. Given a parameter k, as long as there exists a cycle with less than k nodes, all cycles of length smaller or equal to k are cut on a deadheading and the resulting path is contracted into a single request node. Then, a new minimum-weight cycle cover is calculated until all cycles contain at least k original nodes. Afterwards, all nodes are expanded to their corresponding paths of original nodes. Note that the use of contraction excludes self-loops.

– **Patching options.** Different combinations of the proposed patching options are sufficient to derive consistent BOTs. In variant (A) only cycle-to-cycle patches and non-preemptive cycle-to-BOT patches are considered. This corresponds to the MKS heuristic for the ATSP. Variant (B) considers only cycle-to-BOT patches, with and without drops and variant (C) considers all three options.



(a) Solution S and two cycles c_1, c_2

(b) Patching c_1 into S

(c) Patching c_1 into S via drop v

(d) Patching c_2 into c_1

Figure 4.15: Example for patching with drops. (a) shows the initial situation with the BOT-structured solution S rooted at v_0 and two simple cycles c_1, c_2. (b), (c) and (d) illustrate the possible patching operations.

4.5.3 Savings with drops (SD)

The *savings heuristic* of Clarke and Wright [28] is a construction heuristic for the CVRP. First, each customer demand is served in its own tour. The savings for a pair of distinct tours corresponds to the cost saved when both tours are merged into a single tour by concatenation. The merge operations corresponding to the largest savings are performed iteratively until no further operations are possible, either due to the capacity constraints or because only a single tour remains.

Transferring the idea of the savings heuristic to the PSCP results in an algorithm similar to the patching heuristics described above. Instead of starting from a minimum-weight cycle cover, each request is served in its own tour starting and ending at the depot node. Then, $|R| - 1$ merge operations are performed until a single feasible tour is obtained. Each merge operation inserts one tour into another, with or without an additional drop node. Algorithm 4.5 provides an outline of the procedure and Figure 4.16 depicts both merge operations.

4.6 Computational study

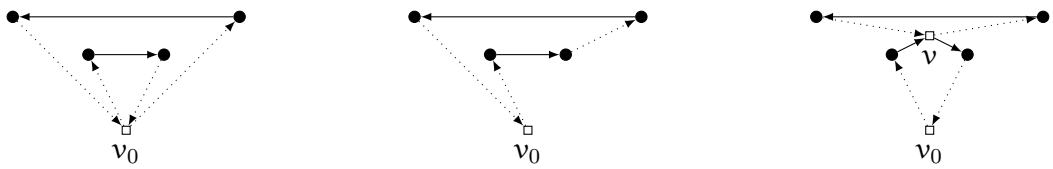
To compare the algorithms from the literature and the newly adapted algorithms described in Section 4.5, a computational study on a large set of instances is performed. Furthermore, the obtained solutions are analyzed w.r.t. their characteristics, e.g., the number of drops and the depth of the resulting BOTs. Finally, the best obtained preemptive solutions are compared to the corresponding optimal non-preemptive solutions to establish an intuition for the improvements made possible by preemption.

Algorithm 4.5 Savings with drops (SD)

Input: PSCP instance
Output: Consistent BOT
Initialize a pool of partial solutions $P := \{S_i : 1 \leqslant i \leqslant |R|\}$ in BOT representation. Each solution S_i is rooted at the depot v_0 and contains only a single request $\sigma_{S_i}(v_0) = \langle r_i \rangle$.
As long as the pool contains at least two solutions, the following steps are performed iteratively.

1. Calculate or update the savings for each pair of distinct partial solutions $S_i, S_j \in P$ by considering two possible merge operations.
   a) Inserting $\sigma_{S_i}(v_0)$ into a sequence $\sigma_{S_j}(v_k)$ for any drop node $v_k \in S_j$ or with roles of S_i and S_j interchanged. (Figure 4.16b)
   b) Replacing v_0 in S_i with a drop node v_k and inserting v_k into a sequence $\sigma_{S_j}(r_{k'})$ for any request $r_{k'} \in S_j$ or with roles of S_i and S_j interchanged. (Figure 4.16c)
2. Find the largest savings and perform the associated merge operation, thereby reducing the size of the pool $|P|$ by one.

The single remaining solution in the pool contains all requests, is feasible and corresponds to the resulting solution.



(a) Two solutions rooted at v_0 (b) Merge by insert (c) Merge via drop v

Figure 4.16: Example for savings with drops. (a) shows the initial situation with two solutions rooted at v_0. (b) and (c) illustrate the possible merge operations.

All algorithms have been implemented in C++ and were compiled using *GCC* version 8.1.0. The *network simplex* implementation of the *Lemon* graph library [33] (version 1.3.1) is used to calculate minimum-weight cycle covers. All experiments have been performed on an Intel Core i7-3770 3.4GHz machine with 64Bit Ubuntu Linux 16.04 and 16GB RAM.

In this study, the PSCP instances of Quilliot et al. [88] and the *pickup and delivery problem with time windows* (PDPTW) instances of Li and Lim [74] are used. The latter instances are transformed into PSCP instances by discarding all time window, service time and volume information, i.e., all requests are assumed to have unit demands. The instance sets are summarized in Table 4.2 with the minimum and maximum numbers of requests and nodes per set as well as the number of instances in each set.

The instances `rel` are designed in such a way that drops have a comparatively large impact on the total cost. Due to their artificial nature, these instances are excluded from some of the broader aggregations below, implied by the instance sets $\texttt{all}_1$ and $\texttt{all}_S$. The instances of Li and Lim [74] can be classified along the two dimensions *node distribution* and *planning horizon*. All nodes are sampled from the Euclidean plane and are either *clustered* (`lc`), *uniformly random* (`lr`) or *semi-clustered* (`lrc`), i.e., a mixture of both, clustered and uniformly random nodes. While the extent of the planning horizon becomes irrelevant without time windows, distances between corresponding pickup and delivery nodes are shorter for the shorter plan-

ning horizon and larger for the larger planning horizon w.r.t. the size of the space the nodes are sampled from. The sets corresponding to shorter and larger distances are indicated by suffixes 1 and 2, respectively.

Table 4.2: Instance sets

	requests $\|R\|$		nodes $\|V\|$			
name	min.	max.	min.	max.	#inst.	derived from
gr24	11	11	24	24	10	Quilliot et al. [88]
gr120	59	59	120	120	10	
hk48	23	23	48	48	10	
rel	15	141	31	283	14	
lc1	52	529	105	1059	59	Li and Lim [74]
lc2	51	511	103	1023	58	
lr1	50	527	101	1055	62	
lr2	50	506	101	1013	61	
lrc1	52	527	105	1055	58	
lrc2	51	507	103	1015	56	
all	11	529	24	1059	398	all aforementioned instance sets
$\texttt{all}_1$	11	529	24	1059	384	all\rel
$\texttt{all}_S$	11	59	24	120	30	gr24 $\cup$ gr120 $\cup$ hk48

The results are reported by their average gaps to the *best known solutions* (BKSs) and *lower bounds* (LBs) of the corresponding instances. The BKS correspond to the best solutions found during the experimental tests, preliminary tests and the development of the methods. The BKS for the *gr24*, *hk48* and *rel* instance sets are *optimal* and are reported in Quilliot et al. [88]. For optimal PSCP solutions, the LB corresponds to their costs. Otherwise the LB corresponds to the LB of Bordenave et al. [17] for the MSP, i.e., the cost of a minimum-weight cycle cover over all requests.

Additionally, obtained results are compared to optimal costs for the corresponding non-preemptive SCP instances. These optimal SCP costs were obtained by solving all instances with a branch-and-cut solver using CPLEX as well as the *Concorde TSP solver* (cf. Applegate et al. [8]).

4.6.1 Construction methods

The first experiment studies the performance of the proposed patching algorithms (MKSD) and the savings-based algorithms (SD) in comparison to the MCI algorithms from the literature with and without the VND of Quilliot et al. [88]. The algorithm configurations of the MCI algorithm are associated with a number of iterations, e.g., *MCI-F-10* corresponds to the MCI algorithm with full enumeration of drop nodes and 10 random permutations with the best solution reported. In contrast to the original authors of the MCI heuristic who reported results for 100 iterations, we study configurations with 10, 100 and 1000 iterations to gain an intuition about the impact of the number of iterations. All algorithm configurations were used to solve all instances. The costs of the resulting solutions and the wall-clock runtime required by the algorithms were recorded. The results are given in Table 4.3.

The average gaps to the BKS are given aggregated for each instance set individually, and in columns $\texttt{all}_1$ and $\texttt{all}_S$ aggregated over all instances of the respective combined instance sets. Columns *LB* report the average gaps to the LB. Column *OA* reports the average gaps

to the prior BKS costs reported in Quilliot et al. [88], negative values corresponds to an improvement on average. Columns *SCP* report the average gaps to the optimal SCP solutions. Columns *better* report the percentage of instances for which the obtained costs are equal to or better than the costs of the respective optimal SCP solutions. The average wall-clock time in column *avg. time* is aggregated over all instances. The algorithms *MCI-100* and *MCI-1-VND* correspond to the results reported by Quilliot et al. [88] for the MCI algorithm with 100 iterations and the VND algorithm applied to a solution obtained from a single iteration of the MCI algorithm, respectively.

The results confirm that more iterations of the MCI algorithm lead to better results on average. However, the cost difference between 10 and 100 iterations is generally larger than the difference between 100 and 1000 iterations while the computation time increases linear w.r.t. the number of iterations. Thus, choosing a specific number of iterations realizes a trade-off between the obtained solution quality and computation time. When the number of iterations is increased linearly, the computation time increases linearly as well while the increase in the obtained solution quality decreases. The results indicate that an iteration limit of 100 or slightly higher realizes a good trade-off.

The differences between the MCI configurations for the drop node candidate sets are rather small w.r.t. the cost of the solutions but large for the required computation time. MCI-F considers all potential drop nodes and is outperformed by MCI-N w.r.t. the computation time, as the latter does only consider a restricted set of candidate drop nodes. Likewise, MCI-E considers an even smaller set of candidate drop nodes, does not require a precomputed node neighborhood and is thus faster than MCI-N. Despite the full enumeration of drop nodes, MCI-F does not produce better results than configurations MCI-{N,E}. In fact, on average, configuration MCI-E performs best over instances $\texttt{all}_\texttt{S}$ and configuration MCI-N performs best over instances $\texttt{all}_1$, albeit the differences are small. The differences between the costs obtained by configurations MCI-N and MCI-E are marginal for instances $\texttt{all}_\texttt{S}$ and $\texttt{all}_1$. Only on instances `rel`, MCI-E performs better than MCI-N. The additional time required by MCI-N to calculate the drop node lookup neighborhood has a rather large impact on the overall computation time, when compared to MCI-E. In conclusion, among the MCI configurations, MCI-E is a good choice w.r.t. the costs obtained and the required computation time.

In comparison to the *MCI-100* results reported by the original authors of the MCI algorithm, all variants MCI-{F,N,E}-100 perform slightly worse on the instances `rel`, but better on the remaining instances $\texttt{all}_\texttt{S}$. We conclude that our proposed configurations of the algorithms are competitive w.r.t. the original algorithm in terms of the obtained solution quality.

Among the MKSD configurations, the non-preemptive MKSD-A algorithm performs best without self-loops while the other configurations MKSD-B and MKSD-C perform better with self-loops allowed. A possible reason is that self-loops enable the patching of drop cycles containing a single request, while without self-loops, each drop cycle contains at least two requests. Using contraction (COP2) performs better than self-loops for non-preemptive patching, but is otherwise worse for all other configurations. Contractions with larger values lead to even worse results and were excluded from the table. The best results are obtained by configuration MKSD-B with self-loops closely followed by MKSD-C with self-loops. Regarding the computation time all methods are rather efficient. Nevertheless, the use of contraction or self-loops consumes a small amount of additional computation time. Hence, MKSD-B with self-loops is identified to be the most promising patching based construction algorithm. The differences between the two savings based SD heuristics are marginal w.r.t. the quality of the obtained solutions, however, variant SD-E is considerably faster.

Comparing the best MCI, MKSD and SD configurations clearly shows that MKSD outperforms MCI and SD by a large margin in terms of the obtained average solution quality. This

Table 4.3: Results obtained by the construction algorithms for all instance sets.

	gap BKS [%]												gap LB [%]		gap OA [%]	gap SCP [%]		better [%]		
algorithm	rel	gr24	gr120	hk48	lc1	lc2	lr1	lr2	lrc1	lrc2	all_S	all_1	all_S	all_1	all_S	all_S	all_1	all_S	all_1	avg. time [s]
MCI-100	18.28	0.73	5.50	3.19	–	–	–	–	–	–	3.14	–	3.18	–	2.95	2.82	–	23.33	–	–
MCI-F-10	21.01	2.13	6.89	4.76	6.83	7.54	9.06	9.02	8.14	6.86	4.59	7.67	4.63	10.73	4.40	4.27	6.92	6.67	0.52	6.93
MCI-F-100	18.71	0.51	4.97	3.01	5.50	6.67	7.90	8.10	6.96	6.14	2.83	6.58	2.87	9.60	2.64	2.51	5.84	23.33	3.12	68.87
MCI-F-1000	16.07	0.20	4.60	2.14	4.65	6.02	7.16	7.56	6.28	5.61	2.31	5.93	2.35	8.91	2.13	2.00	5.19	33.33	5.21	687.15
MCI-N-10	22.03	2.17	6.89	4.64	6.66	7.53	8.99	8.94	8.10	6.84	4.57	7.61	4.61	10.66	4.38	4.24	6.86	6.67	0.52	3.47
MCI-N-100	19.79	0.49	5.01	3.02	5.42	6.65	7.81	8.06	6.88	6.14	2.84	6.53	2.88	9.55	2.65	2.53	5.80	23.33	2.86	5.17
MCI-N-1000	17.25	0.20	4.60	2.13	4.60	6.01	7.09	7.52	6.20	5.64	2.31	5.89	2.35	8.88	2.12	2.00	5.16	33.33	4.95	22.13
MCI-E-10	20.95	2.13	6.84	4.99	6.83	7.52	9.00	8.93	8.08	6.83	4.65	7.64	4.69	10.69	4.46	4.33	6.89	6.67	0.52	0.04
MCI-E-100	18.70	0.49	4.97	3.01	5.51	6.64	7.87	8.07	6.95	6.12	2.82	6.57	2.86	9.58	2.64	2.51	5.83	23.33	3.12	0.37
MCI-E-1000	15.91	0.20	4.56	2.16	4.65	5.99	7.14	7.49	6.25	5.64	2.30	5.91	2.34	8.90	2.12	1.99	5.17	33.33	5.21	3.65
MKSD-A-COP2	10.22	0.97	0.05	0.37	5.68	0.54	1.32	0.21	1.49	0.22	0.47	1.50	0.50	4.56	0.29	0.15	0.78	70.00	22.66	0.06
MKSD-A-NoSelf	10.22	0.90	0.05	0.23	5.68	0.52	1.26	0.21	1.45	0.21	0.39	1.47	0.43	4.54	0.21	0.08	0.75	83.33	30.21	0.02
MKSD-A-Self	10.22	1.36	0.06	0.24	5.93	0.68	1.63	0.27	1.77	0.26	0.56	1.67	0.59	4.75	0.38	0.24	0.95	66.67	9.90	0.03
MKSD-B-COP2	0.00	0.74	0.05	0.32	5.06	0.45	1.19	0.20	1.31	0.21	0.37	1.33	0.41	4.38	0.19	0.05	0.61	76.67	45.31	0.07
MKSD-B-NoSelf	0.00	0.16	0.02	0.11	4.78	0.37	0.97	0.17	1.11	0.18	0.10	1.17	0.14	4.22	-0.08	-0.21	0.46	100.00	79.43	0.04
MKSD-B-Self	0.00	0.31	0.00	0.00	4.28	0.14	0.20	0.02	0.26	0.01	0.10	0.76	0.14	3.79	-0.07	-0.21	0.06	93.33	84.64	0.09
MKSD-C-COP2	0.00	0.74	0.05	0.32	5.24	0.47	1.19	0.20	1.32	0.21	0.37	1.36	0.41	4.42	0.19	0.05	0.64	76.67	44.79	0.06
MKSD-C-NoSelf	0.00	0.16	0.02	0.11	5.11	0.39	0.98	0.17	1.14	0.18	0.10	1.23	0.14	4.29	-0.08	-0.21	0.52	100.00	77.60	0.03
MKSD-C-Self	0.00	0.44	0.00	0.00	5.07	0.19	0.23	0.02	0.28	0.01	0.15	0.90	0.18	3.96	-0.03	-0.16	0.20	93.33	83.07	0.07
SD-E	0.02	2.00	1.42	1.13	4.75	3.36	3.80	3.04	3.67	2.27	1.52	3.34	1.55	6.35	1.34	1.20	2.62	6.67	0.52	0.10
SD-F	0.02	2.00	1.42	1.13	4.76	3.36	3.80	3.04	3.67	2.28	1.52	3.34	1.55	6.36	1.34	1.20	2.62	6.67	0.52	4.94
MCI-1-VND	0.57	0.70	0.56	0.90	–	–	–	–	–	–	0.72	–	0.76	–	0.54	0.41	–	26.67	–	–
MCI-E-100-VND-F	0.67	0.07	0.25	0.57	1.34	1.12	1.40	0.84	1.43	0.60	0.30	1.06	0.33	3.97	0.12	-0.02	0.36	36.67	19.53	2578.05
MCI-E-100-VND-N	1.52	0.11	0.35	0.51	1.39	1.14	1.56	0.84	1.33	0.57	0.33	1.08	0.36	3.99	0.15	0.01	0.38	40.00	21.35	46.69
MCI-E-100-VND-E	0.67	0.34	0.22	0.53	1.40	1.06	1.46	0.89	1.30	0.57	0.36	1.06	0.40	3.97	0.19	0.05	0.36	50.00	22.40	14.19
MKSD-B-Self-VND-F	0.00	0.17	0.00	0.00	2.28	0.11	0.12	0.01	0.16	0.01	0.06	0.42	0.09	3.37	-0.12	-0.25	-0.28	96.67	90.62	423.00
MKSD-B-Self-VND-N	0.00	0.17	0.00	0.00	2.29	0.11	0.12	0.01	0.16	0.01	0.06	0.42	0.09	3.37	-0.12	-0.25	-0.28	96.67	90.62	10.45
MKSD-B-Self-VND-E	0.00	0.17	0.00	0.00	2.28	0.11	0.12	0.01	0.15	0.01	0.06	0.42	0.09	3.37	-0.12	-0.25	-0.28	96.67	90.62	1.40

is especially obvious when the percentage of instances is considered, for which the obtained costs are better than the optimal SCP solutions. Over instances $\texttt{all}_\texttt{S}$ and $\texttt{all}_1$, none of the MCI and SD algorithms exceed 34% and 6%, while MKSD-B with self-loops obtained better results for more than 93% and 84% of the instances, respectively. Additionally, MKSD-B with self-loops achieved the smallest average gap to the optimal SCP costs with 0.06% over instances $\texttt{all}_1$. For instances $\texttt{all}_\texttt{S}$, algorithms MKSD-B and MKSD-C with and without self-loops achieved an average net benefit w.r.t. the optimal SCP costs with gaps down to -0.21%. On top of these results, the best MKSD and SD methods also require a lot less computation time than the better MCI algorithms with 100 or more iterations.

Over instances $\texttt{all}_\texttt{S}$ and $\texttt{all}_1$, even the non-preemptive MKSD-A algorithm gave better results than the MCI algorithms. The MCI algorithms outperform the MKSD-A algorithm only on instance sets `gr24` and `lc1`. All MKSD algorithms obtain average gaps to the LB below 1% on instances $\texttt{all}_\texttt{S}$ and below 5% on instances $\texttt{all}_1$.

Post-processing with VND The results show that post-processing the constructed solutions with the VND proposed by the original authors of the MCI algorithm achieves noticeable improvements in general. The differences between the configurations MCI-E-100-VND-{F,N,E} are marginal, except for instances `rel`, where the neighborhood approach performs worse. Nevertheless, the major difference between the configurations is the computation time, which is clearly smaller for configuration VND-E than configurations VND-{F,N}. Apart from the computation time, the results obtained with the configurations MKSD-B-Self-VND-{F,N,E} are nearly identical.

We observe that the VND is not able to compensate for the differences between the initial solutions, i.e., on average, the initial solutions provided by MKSD-B-Self have higher quality than the solutions obtained by the MCI-E-100-VND configurations. In fact, all patching based construction algorithms including the non-preemptive MKSD-A outperform the originally proposed *MCI-1-VND* configuration and the patching algorithms MKSD-{B,C} with self-loops outperform the MCI-E-100-VND configurations, respectively.

The ratio of the computation times for MCI-E-100-VND-E and MKSD-B-Self-VND-E is larger than the ratio for MCI-E-100 and MKSD-B-Self, implying that starting from better solutions decreases the time spent in the VND algorithm. As such, using the MKSD-B algorithm to obtain initial solutions improves both, the average solution quality and the average computation time.

In conclusion, the proposed patching based algorithms provide improvements when compared to the state-of-the-art. In addition, the proposed restrictions of the drop node candidate sets based on Proposition 4.1 provide further speedups for the MCI and VND algorithms while maintaining a similar level of quality.

Solution characteristics Table 4.4 shows aggregated measures that characterize the solutions obtained by the different construction algorithms. As a solution can contain at most $|\mathsf{R}| - 1$ drops and therefore at most $|\mathsf{R}| - 1$ preempted requests, the number of drops and the number of preempted requests are normalized by $|\mathsf{R}|$ and displayed in columns *#drops* and *#preempted*. Columns *path length* and *depth* correspond to the absolute length of the request paths and the nesting depth of the solutions. The nesting depth of a solution corresponds to the maximum depth of the corresponding BOT, i.e., a non-preemptive solution has depth 0 and a solution with at least one drop has depth of at least 1. The subcolumns *diff.* provide the difference in percentage points to the measures calculated on the BKS of the corresponding instances.

The results show that all non-VND algorithms introduce insufficiently many drops on average and the VND algorithms introduce slightly too many when compared to the BKS.

Table 4.4: Solution characteristics for the constructive algorithms for all instance sets except `rel`

	#drops [%]			#preempted [%]			path length			depth		
algorithm	avg.	max.	diff.	avg.	max.	diff.	avg.	max.	diff.	avg.	max.	diff.
MCI-F-10	7.38	21.15	-2.17	6.14	18.18	-2.30	1.19	11	0.08	1.48	3	-0.24
MCI-F-100	7.44	22.64	-2.10	6.26	18.18	-2.18	1.17	10	0.06	1.41	3	-0.30
MCI-F-1000	7.60	20.37	-1.94	6.43	18.18	-2.01	1.18	11	0.07	1.39	3	-0.33
MCI-N-10	6.91	21.15	-2.63	5.85	18.18	-2.59	1.17	9	0.06	1.42	3	-0.29
MCI-N-100	7.00	22.64	-2.55	5.97	18.18	-2.48	1.15	7	0.04	1.39	4	-0.33
MCI-N-1000	7.15	18.87	-2.39	6.10	18.18	-2.35	1.17	8	0.06	1.42	3	-0.29
MCI-E-10	7.34	21.15	-2.21	6.12	18.18	-2.32	1.18	11	0.08	1.41	3	-0.31
MCI-E-100	7.31	22.64	-2.24	6.15	18.18	-2.30	1.17	10	0.06	1.34	3	-0.37
MCI-E-1000	7.53	18.87	-2.02	6.36	18.18	-2.09	1.18	12	0.07	1.36	3	-0.35
MKSD-B-COP2	1.59	9.09	-7.95	1.51	9.09	-6.94	1.04	6	-0.08	1.11	7	-0.60
MKSD-B-NoSelf	3.02	18.18	-6.53	2.86	18.18	-5.58	1.04	6	-0.07	1.46	10	-0.25
MKSD-B-Self	9.47	27.27	-0.07	8.41	22.64	-0.03	1.11	8	0.00	1.90	11	0.18
MKSD-C-COP2	1.39	9.09	-8.16	1.34	9.09	-7.11	1.04	5	-0.09	0.95	4	-0.76
MKSD-C-NoSelf	2.42	18.18	-7.13	2.31	18.18	-6.13	1.06	5	-0.06	1.18	6	-0.54
MKSD-C-Self	7.77	27.27	-1.77	6.96	22.22	-1.48	1.11	5	0.00	1.57	6	-0.15
SD-E	9.49	22.22	-0.05	8.61	21.74	0.16	1.09	5	-0.02	1.91	5	0.19
SD-F	9.52	22.22	-0.02	8.63	21.74	0.19	1.09	5	-0.02	1.92	5	0.20
MCI-E-100-VND-F	10.56	21.74	1.01	9.40	18.87	0.96	1.11	13	0.00	3.31	9	1.59
MCI-E-100-VND-N	10.34	24.53	0.80	9.23	20.75	0.79	1.11	8	0.00	3.18	9	1.47
MCI-E-100-VND-E	10.58	27.27	1.04	9.45	26.09	1.01	1.11	9	0.00	3.25	9	1.54
MKSD-B-Self-VND-F	9.74	27.27	0.20	8.59	22.22	0.15	1.11	9	0.00	1.96	11	0.24
MKSD-B-Self-VND-N	9.73	27.27	0.19	8.59	22.22	0.14	1.11	9	0.00	1.96	11	0.24
MKSD-B-Self-VND-E	9.73	27.27	0.19	8.59	22.22	0.14	1.11	9	0.00	1.95	11	0.24

However, the difference in the relative number of drop nodes does not correlate with the obtained solution quality as per Table 4.3. For the number of preempted requests the results are similar with the exception of the SD algorithms which, on average, preempt slightly more distinct requests compared to the BKS. The MCI-N algorithm introduces slightly fewer drops than the MCI-E and MCI-F variants. Analysis of the precomputed neighborhoods and midpoints used by the MCI-N algorithm shows that the majority of considered midpoints corresponds to a small set of central nodes, i.e., often the same drop node candidates are considered, leading to less drops overall.

Among the patching variants, those based on COP introduce the smallest number of drops, followed by those without self-loops. Those with self-loops allowed introduce a comparatively large number of drops. As the number of cycles is an upper bound on the number of drops introduced by the patching based algorithms, cycle covers with fewer but longer cycles are more likely to lead to fewer drops. Likewise, the savings based methods SD introduce a large number of drops, as each request is in its own BOT initially. The mentioned relationships are also reflected in the results reported for the average number of preempted requests.

The average lengths of the obtained request paths highlight the fundamental difference between the MCI and patching based algorithms, as the request paths of the former are slightly longer on average and clearly longer in terms of the maximum length. The reason is the sole local consideration of the MCI algorithms, i.e., each insertion is evaluated locally resulting in locally optimal drop nodes for the requests being inserted, but not necessarily for following requests and sequences of requests. As such, more requests require their own

Table 4.5: Improvements and computation times of the neighborhoods induced by RPP and BOTDP

		gap [%]		time [s]	
neighborhood	initial	avg.	max.	avg.	max.
$\mathcal{N}_{\text{BOTPD}}$	MCI-E-100	0.28	5.32	658.28	4228.26
	MKSD-B-Self	0.01	0.62	666.51	3871.42
$\mathcal{N}_{\text{RPP}}$	MCI-E-100	0.04	2.38	0.79	3.82
	MKSD-B-Self	0.01	0.62	0.36	3.45

specific drop nodes increasing the length of the request paths. This interpretation is further confirmed by the differences between the variants MCI-E and MCI-F compared to MCI-N, as the latter considers fewer drop nodes overall.

4.6.2 RPP and BOTDP based polishing

As claimed in Section 4.3, the neighborhoods $\mathcal{N}_{\text{RPP}}$ and $\mathcal{N}_{\text{BOTPD}}$ induced by the RPP and BOTDP solution methods, respectively, seem intuitively impractical. To examine the claim, both neighborhoods were applied to solutions generated with the MCI-E-100 and MSKD-B-Self construction algorithms. The improvements and required computation times are reported in Table 4.5 with column *gap* reporting the cost improvement in percentage of the total cost.

The results confirm the expected results, i.e., the average improvement is extremely small although cases of improvements above 5% were recorded. The computation required by the BOTDP is drastically larger than for the RPP, rendering it completely unusable for above small instances. In contrast, the RPP induced neighborhood can be searched a lot faster and may be used to polish the request paths of a final or intermediate solution, although the expected improvement is marginal. Hence, the RPP and BOTDP based polishing methods remain of mainly theoretical interest.

4.6.3 Benefits of preemption

The maximum theoretical improvement of preemption over non-preemption is a cost decrease of at most 50%. While this bound is tight, the instances realizing it are rather artificial. As such it is interesting, what improvements may be observed on more realistic instances. For this purpose, the BKS and LB for the PSCP are compared to the optimal solutions for the corresponding SCP instances. The results are reported in Table 4.6. Column *deadh.* reports the percentage of the costs induced by deadheadings in the optimal non-preemptive solutions. Column *gap LB* reports the average gap of the BKS to the cycle cover LB. Columns *impr. BKS* and *impr. LB* report the percentage of cost saved by the PSCP BKS and PSCP LB relative to the non-preemptive optimal SCP cost. That means, columns *impr. BKS* provide a lower bound and columns *impr. LB* provide an upper bound on the real improvements enabled through the use of preemption. The columns thereafter correspond to the measures introduced in Table 4.4.

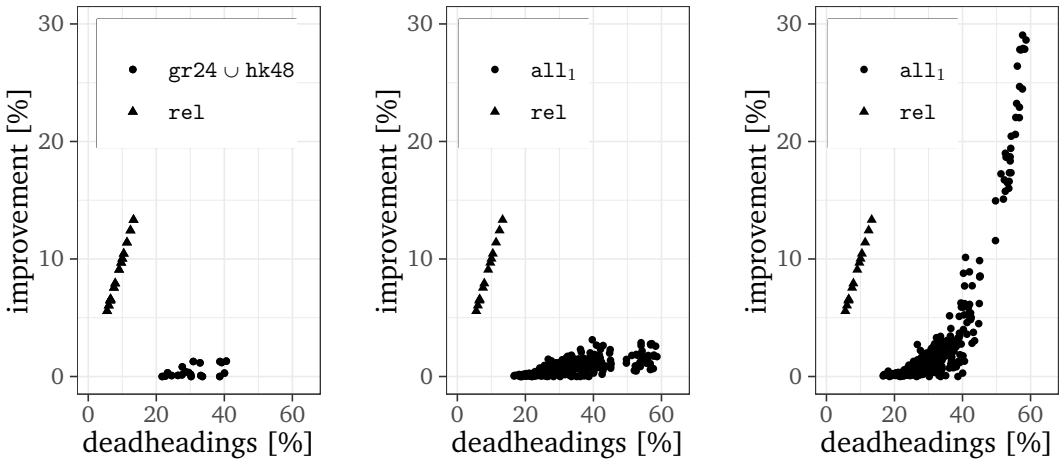
The results show that the average improvement by preemption is rather small, except for the instance set `rel`. Over all other instances, the average benefit is below 1% for the BKS and below 3.1% for the LB. The average improvements between the different instance sets differ

Table 4.6: BKS characteristics by instance set

			impr. BKS [%]			impr. LB [%]			#drops [%]			#preempted [%]			path length		depth	
instances	deadh. [%]	gap LB [%]	min.	avg.	max.	min.	avg.	max.	min.	avg.	max.	min.	avg.	max.	avg.	max.	avg.	max.
rel	8.78	0.00	5.57	8.79	13.33	5.57	8.79	13.33	26.67	31.07	32.62	14.18	19.42	26.67	1.69	6	3.50	5
gr24	35.77	0.00	0.00	0.68	1.31	0.00	0.68	1.31	0.00	10.00	18.18	0.00	9.09	18.18	1.12	2	0.80	1
gr120	19.48	0.11	0.00	0.03	0.11	0.00	0.15	0.46	0.00	1.36	3.39	0.00	1.36	3.39	1.00	1	0.50	1
hk48	26.22	0.00	0.00	0.22	0.82	0.00	0.22	0.82	0.00	6.52	13.04	0.00	6.09	13.04	1.12	2	0.80	1
lc1	47.75	14.92	0.43	1.27	2.87	1.76	13.19	29.05	8.57	14.72	24.53	7.84	12.50	20.75	1.18	8	2.17	6
lc2	31.66	0.58	0.00	0.38	1.08	0.30	0.96	2.78	0.00	7.27	13.86	0.00	6.69	12.87	1.08	6	1.83	4
lr1	32.26	1.27	0.19	1.08	3.12	0.97	2.31	6.14	5.66	13.31	20.19	3.77	11.85	18.27	1.13	5	2.23	7
lr2	22.29	0.14	0.00	0.19	1.47	0.06	0.32	2.72	0.00	4.14	15.69	0.00	3.84	13.73	1.07	3	1.16	2
lrc1	35.00	1.62	0.58	1.21	2.67	1.02	2.76	10.13	10.05	14.90	24.53	8.13	12.73	20.75	1.18	4	2.28	5
lrc2	24.88	0.13	0.00	0.20	0.97	0.05	0.33	1.47	0.00	4.54	11.76	0.00	4.37	11.76	1.04	2	1.12	2
all_S	27.16	0.04	0.00	0.31	1.31	0.00	0.35	1.31	0.00	5.96	18.18	0.00	5.51	18.18	1.10	2	0.70	1
all_1	31.90	2.87	0.00	0.70	3.12	0.00	3.09	29.05	0.00	9.54	24.53	0.00	8.44	20.75	1.11	8	1.72	7

noticeable, e.g., `lc1` with 1.27% and `lr2` with 0.19% for the BKS in comparison to 13.19% and 0.32% for the LB. The reason is given by the fact that preemption can only improve the deadheading component of the total solution cost, hence, a larger share of deadheadings may lead to larger improvement opportunities. In addition, for highly clustered instances like `lc1`, the cycle cover LB may be arbitrarily bad as the deadheadings connecting the different clusters are ignored. This relationship is highlighted in Figure 4.17 for all instances. Except for the `rel` instances, for 20% deadheadings in the SCP cost and below, the improvements of preemption are marginal. They increase as the percentages of deadheadings increase. Nevertheless, even w.r.t. the cycle cover LB the maximum possible improvement for any considered instance is below 30%.

In conclusion, when a non-preemptive routing exhibits a large percentage of deadheadings, preemption may be considered. On the other hand, for rather tight routings with small percentages of deadheadings, the complexity of preemption may outweigh its benefits.



(a) Opt. SCP and opt. PSCP (b) Opt. SCP and PSCP BKS (c) Opt. SCP and PSCP LB

Figure 4.17: The improvements between the optimal preemptive, preemptive BKS, preemptive LB and the optimal non-preemptive SCP costs by percentages of deadheadings in the optimal non-preemptive costs for instances $\texttt{all}_1$. The artificial nature of the `rel` instances is clearly visible.

The other solution characteristics like the percentage of preempted requests, the request path lengths and the depth of the tree exhibit similar relationships as those reported for the solutions obtained by the construction algorithms. The measures indicate that the improvements of preemption result from a lot of preemptions, each contributing a small share as opposed to a few preemptions with large impact. As the PSCP does not consider any penalty for preemptions, e.g., additional time or cost, the benefits of preemption may be even smaller in more realistic settings including these penalties.

4.7 Conclusion

In this chapter, the PSCP was reconsidered. Based on the existence of BOT structured solutions as established by Quilliot et al. [88], further structural properties have been derived and adaptations of well-known heuristics for the ATSP and CVRP have been presented and compared in a large computational study.

Using results derived for the PSP, we have shown that preemption may improve a non-preemptive solution by at most 50%. Similarly, explicit drop nodes may improve a solution

by at most 50% in case of asymmetric distances, by at most $33.\overline{3}\%$ in case of symmetric distances and not at all in case of line and circular structures.

Two polynomial-time computable subproblems have been identified. The RPP asks for a minimum-cost request path respecting a given sequence of immediate child requests. The BOTDP extends this concept to complete BOTs and asks for a minimum-cost solution satisfying a given input permutation of the requests. Based on the BOTDP we have shown that the search space of the PSCP can be limited to the permutations of the requests.

The modified Karp-Steele patching heuristic for the ATSP and the savings heuristic for the CVRP have been adapted for the PSCP. In contrast to the MCI heuristic both proposed heuristics are deterministic. The results of the computational study show that none of the proposed algorithms consistently outperforms the others. However, on average, the MKSD algorithm based on the patching heuristic clearly outperforms the other algorithms and improves upon the combination of the MCI and VND algorithms proposed in the literature. Nevertheless, the VND algorithm is able to further improve the solutions provided by the MKSD algorithm.

The comparison of the best known preemptive and optimal non-preemptive solutions indicates that the average improvement of preemption is rather small on realistic instances and increases as the percentage of deadheadings increases. Thus, preemption should be well-considered w.r.t. the properties of the problem instances to be solved, otherwise the complexities of preemption may outweigh its benefits.

Despite the simplicity of the PSCP in terms of its description, it remains difficult to consistently obtain solutions of high quality. Hence, possible future research directions for the PSCP include the evaluation of the BOT structure with respect to exact methods and the development of consistent heuristic improvement methods.

Chapter 5

Adaptively solving a multi-period vehicle and technician routing problem

In this chapter, we study a complex *vehicle routing problem* (VRP) that combines the delivery of vending machines from a central depot to specific customer locations with the installation of these machines by technicians at the customer locations after their delivery. The problem was the subject of the *VeRoLog Solver Challenge 2018–2019* (VSC2019). Parts of this chapter have already been published in the peer-reviewed research article Graf [51].

Contribution On the problem-specific level we contribute an efficient heuristic solution method that adapts to instance properties like size and objective function coefficients as well as to the available computing resources w.r.t. a given time limit. We provide an extensive computational study documenting the positive impact of the algorithm components and comparing our method to those proposed by other researchers. On a more abstract level, our method exemplifies how the *adaptive large neighborhood search* (ALNS) approach of Ropke and Pisinger [92] can be extended to facilitate a decomposition of a complex combined problem into subproblems and provide an adaptive way to allocate the computing resources to these subproblems.

Organization First we give a short description of the challenge and related problems in Section 5.1. A formal definition of the problem is given in Section 5.2. In Section 5.3 we describe the components of the solution method and their overall arrangement. In Sections 5.4 and 5.5 we consider the subproblems associated with the minimization of the maximum number of trucks on any day and the technician scheduling. The focus of Section 5.6 is the *adaptive layer* that guides the solution method's individual components. In Section 5.7 we perform experiments on benchmark instances provided by the challenge organizers and discuss the results. In Section 5.8 we briefly review the literature on other methods proposed for the problem at hand and compare our method with those of the other challenge participants on the final dataset provided by the challenge organizers. Section 5.9 summarizes the proposed solution method and the contributions of this chapter.

5.1 Introduction

Gromicho et al. [54] announced the VSC2019 of the EURO working group *Vehicle Routing and Logistics* (VeRoLog). Prior VeRoLog solver challenges have been held in the years 2014, 2015 and 2017. The goal of these challenges is to motivate the research on efficient solution methods for real-world inspired complex VRPs under various restrictions, including small but sensible time limits, i.e., alongside the development of suitable algorithms, their efficient implementation is likewise important.

The VSC2019 is divided into two challenges, the *all time best* and the *restricted resources challenge*. The former required the participants to upload solutions without any restrictions

on the processes that were used to derive these solutions. In contrast, to participate in the restricted resources challenge executable solver implementations had to be provided to the organizers. These executable solver implementations were used to solve a set of undisclosed instances with a restricted time limit on a single reference computer. The final ranking was calculated from the solutions reported in these runs. In the restricted resources challenge our solution method implementation performed best and achieved the first place among the eight finalists.

The problem studied in the VSC2019 is a multi-period VRP with deliveries from a central depot to customer locations and subsequent installations of the delivered items at these locations. The problem requires the routing of delivery trucks, the routing of technicians and a scheduling task due to labor restrictions constraining the number of consecutive working days for individual technicians.

The exact problem has, to the best of our knowledge, not been studied in the literature, as the combination and interaction of subproblems are rather specific. However, the subproblems and related problems have been studied more extensively in the past. The *service technician routing and scheduling problem* (STRSP) requires the routing and scheduling of technicians with varying skill sets to perform a given set of service tasks at customer locations. Focussing on different aspects, various variants of these problems are considered, e.g., Kovacs et al. [69] consider an STRSP with technician skill sets, time windows and the possibility to outsource jobs.

Beltrami and Bodin [13] introduce the *period vehicle routing problem* (PVRP) in the context of municipal waste collection that extends the *capacitated vehicle routing problem* (CVRP) by a planning horizon of a given number of periods (e.g., days) and customers that require service according to some schedule. As such, the PVRP requires additional decisions regarding the assignment of customer visits to specific periods, then, each period corresponds to a classic CVRP. A survey of the PVRP and its variants is provided in Francis et al. [38]. Archetti et al. [9] consider the *multi-period vehicle routing problem with due dates* (MVRPD), a PVRP variant similar to a subproblem of the problem at hand. In this variant, each customer is associated with a release and due period and must be assigned to a period inside this subset of periods.

We propose a solution method for the problem tackled in the VSC2019. The solution method merges an ALNS with a *variable neighborhood descent* (VND) procedure and integrates these components through an *adaptive layer* that guides the search. The adaptive layer is used to adjust the solution method to the instance to be solved, the computing environment and the time limit provided by orchestrating the embedded heuristics and balancing intensification and diversification during the search process.

5.2 Problem description

The considered problem requires the routing and scheduling of trucks and technicians to perform a set of delivery and installation requests $R = \{1, \dots, n\}$ in a given planning horizon $T = \{1, \dots, H\}$ of H days. Let V be a set of locations. Two locations $u, v \in V$ are associated with a symmetric distance $d_{uv} = d_{vu} \geqslant 0$ and the triangle inequality is assumed. Let $I = \{1, \dots, m\}$ be a set of *item types*, each item type $i \in I$ is associated with a weight q_i and a penalty value p_i. Each request $r \in R$ is associated with exactly one item type $i_r \in I$, the number of items $n_r \in \mathbb{N}_{>0}$ of type i_r, a delivery time window $[e_r, l_r] \subseteq \{1, \dots, H-1\}$ and a delivery location $v_r \in V$. The location $v^{dep} \in V$ is the *truck depot location* where an unlimited number of trucks is stationed. All trucks have the same capacity C and the same maximum daily distance D. Let W be a set of technicians. Each technician $w \in W$ has a home location $v_w^{dep} \in V$, a specific maximum daily number of installations C_w, and maximum daily distance

D_w. Each technician w has a certain skill set $I_w \subseteq I$ of item types they can install. Each request $r \in R$ must be contained exactly once in a truck tour for delivery and once in a technician tour for installation, respectively. The technician must have the corresponding skill w.r.t. the item type to perform the installation.

As the number of trucks is not part of the problem instance and thus unlimited, assume a solution with K truck tours Π_k, $1 \leqslant k \leqslant K$. Then, each *truck tour* $\Pi_k = \{\pi_{kj}\}_{j=1}^{\ell_k}$ is a set of ℓ_k *subtours*. Each of the subtours π_{kj} starts and ends at the depot v^{dep}. A subtour visits a subset of the customer locations in a specified order and delivers n_r items of type i_r to the customer at location v_r. The total weight of the items transported in a subtour must not exceed the truck's capacity

$$q_{\pi_{kj}} = \sum_{r \in \pi_{kj}} n_r q_{i_r} \leqslant C$$

and the distance of the whole truck tour d_{Π_k}, i.e., the sum of its subtours' distances $d_{\pi_{kj}}$, must not exceed the truck's maximum daily distance:

$$d_{\Pi_k} = \sum_{j=1}^{\ell_k} d_{\pi_{kj}} \leqslant D$$

Each technician $w \in W$ is associated with a set of *technician tours* $J_w = \{\sigma_{wj}\}_{j=1}^{\ell_w}$. A *technician tour* σ_{wj} starts and ends at the technician's location v_w^{dep} and corresponds to a sequence of requests, i.e., visits of customer locations. Each request corresponds to exactly one installation, the total number of installations in a tour must not exceed C_w and the total distance of each tour $\sigma_{wj} \in J_w$ must not exceed the technician's specific D_w.

All truck and technician tours are associated with exactly one day. Let $t_{\Pi_k} \in \{1, \dots, H-1\}$ denote the day of truck tour Π_k and let $t_{\sigma_w} \in \{2, \dots, H\}$ denote the day of technician tour σ_w. Truck tours must respect the delivery time windows $t_{\Pi_k} \in [e_r, l_r]$ of all requests $r \in \Pi_k$ they deliver. A technician can only install an item if the item has been delivered earlier, i.e., the delivery day $t_r^+ \in \{1, \dots, H-1\}$ of request r must precede $t_r^+ < t_r^-$ the installation day $t_r^- \in \{2, \dots, H\}$. A penalty depending on the number of items n_r and the penalty value p_{i_r} is incurred for each day that the installation is delayed beyond the earliest possible installation day $t_r^+ + 1$, i.e., given delivery and installation days, the penalty is incurred $t_r^- - t_r^+ - 1$ times.

While trucks are interchangeable and can perform tours on every day, the situation for the technicians is more complex due to the following *scheduling constraints*. A single technician w may perform at most one tour per day and must not perform tours on more than 5 consecutive days. If a technician performs tours on 5 consecutive days, then there cannot be any tours on the following two days. For 4 or less consecutive days, one day off is sufficient as shown in Example 5.1. Note that the first two days after the planning horizon $H+1, H+2$ are considered days off, i.e., 5 consecutive tours can be feasibly scheduled on days $\{H-4, \dots, H\}$ for $H \geqslant 5$.

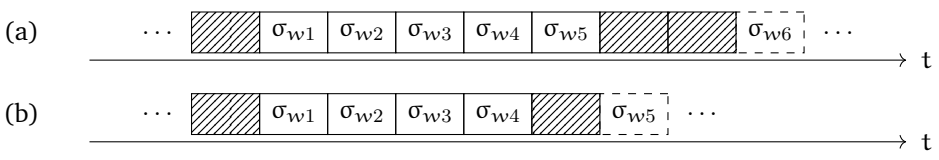


Figure 5.1: Feasible technician schedules

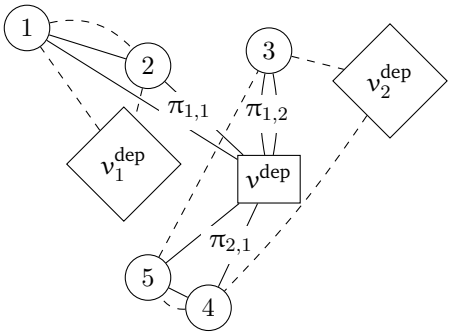
Example 5.1. *Two schedules (a) and (b) are shown in Figure 5.1. Schedule (a) contains a sequence of five consecutive tours $\sigma_{w1}, \dots, \sigma_{w5}$. Therefore, the technician cannot work days 6*

and 7 and the tour σ_{w6} cannot start before day 8. Schedule (b) contains a sequence of only four consecutive tours, allowing tour σ_{w5} to be scheduled after one day off on day 6.

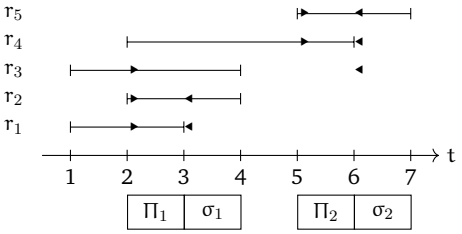
Finally, a solution $S = (\{\Pi_k : 1 \leqslant k \leqslant K\}, \{J_w : w \in W\})$ is a collection of $K \geqslant 0$ truck tours and up to $|W|$ sets of technician tours. A feasible solution contains each request exactly once in a truck tour and exactly once in a technician tour while respecting all time window, delivery-installation precedence, maximum distance, capacity and installation constraints. Example 5.2 shows a small solution with tours and a corresponding schedule.

Example 5.2. *Figure 5.2a shows a solution with five requests $R = \{1, \dots, 5\}$ located at nodes $\{1, \dots, 5\}$ and the truck depot at node v^{dep}. Two technicians $W = \{1, 2\}$ are located at locations v_1^{dep} and v_2^{dep}, respectively. Requests $\{1, 2, 3\}$ are delivered by truck tour $\Pi_1 = \{\pi_{1,1}, \pi_{1,2}\}$ in two subtours. Requests $\{1, 2\}$ are delivered in subtour $\pi_{1,1}$ and request 3 is delivered by itself in subtour $\pi_{1,2}$. Requests $\{4, 5\}$ are delivered by truck tour $\Pi_2 = \{\pi_{2,1}\}$ in a single subtour. The technicians each have a single tour $\sigma_{1,1}$ and $\sigma_{2,1}$, performing installations of requests $\{1, 2\}$ and $\{3, 4, 5\}$, respectively.*

Due to the time windows of requests $\{1, 2, 3\}$ the only feasible delivery day for tour Π_1 is $t_{\Pi_1} = 2$ as shown in Figure 5.2b. As requests $\{1, 2\}$ are delivered on day 2 and are both installed by technician 1 in a single tour, the installation day is $t_{\sigma_{1,1}} = 3$. Requests $\{4, 5\}$ are delivered on day $t_{\Pi_2} = 5$ due to the time window of request 5. Therefore, technician 2 can install requests $\{3, 4, 5\}$ at the earliest of day $t_{\sigma_{2,1}} = 6$. All requests except 3 are installed on their earliest installation days right after their respective delivery days, thus, only request 3 incurs a penalty for $t_3^- - t_3^+ - 1 = 3$ days.



(a) Truck (——) and technician (- - - -) tours, starting and ending at the truck (□) and technician (◇) depots, respectively. The deliveries and installations are performed at 5 customer (○) locations.



(b) Time windows (⊢——⊣) given by the instance, delivery days (▸), install days (◂) and the schedule as given by the solution.

Figure 5.2: Example solution with two truck and two technician tours

Objective function

The objective function and its parts are given in Equations (5.1)–(5.8). The penalty term in Equation (5.2) captures idle times between the requests' deliveries and installations. The total distances of truck and technician tours are captured in Equations (5.3) and (5.4). Equations (5.5) and (5.6) correspond to the number of truck tours and technician tours, respectively. Equations (5.7) and (5.8) define the maximum number of trucks on any single day and the total number of technicians with at least one tour. The terms are weighted by

a vector $\alpha = (\alpha_1, \ldots, \alpha_7)$ such that all objective function terms are mapped to a common scale.

$$Z_\alpha(S) = \alpha \cdot (z_p, z_{d_K}, z_{d_W}, z_{\#K}, z_{\#W}, z_K, z_W)^T \tag{5.1}$$

$$z_p = \sum_{r \in R} \left(t_r^- - t_r^+ - 1\right) \cdot p_{i_r} \cdot n_r \tag{5.2}$$

$$z_{d_K} = \sum_{k \in K} d_{\Pi_k} \tag{5.3}$$

$$z_{d_W} = \sum_{w \in W} d_{J_w} \tag{5.4}$$

$$z_{\#K} = K \tag{5.5}$$

$$z_{\#W} = \sum_{w \in W} |J_w| \tag{5.6}$$

$$z_K = \max_{t=1}^{H-1} |\{\Pi_k \colon 1 \leqslant k \leqslant K, t_{\Pi_k} = t\}| \tag{5.7}$$

$$z_W = |\{w \in W \colon J_w \neq \varnothing\}| \tag{5.8}$$

5.3 Solution method

To solve the given problem efficiently in a limited amount of time, the following solution method is proposed. It combines a VND with an ALNS and an adaptive layer that allows the dynamic adjustment of the method w.r.t. the given problem instance, a time limit and the computing environment. Due to the complexity of the problem with interactions between trucks and technicians, a decomposition is performed such that the improvements of the partial solutions associated with the trucks and the technicians are handled separately.

An overview of the method is provided in Figure 5.3. First, a set of simple preprocessing steps is performed: For each technician, the set of feasible requests according to the technician's skill set and maximum daily distance is calculated. If the set is empty, the technician is removed. Similarly, for each request, the feasible technicians are calculated. If there's only one technician that is able to perform a specific request, this technician is labeled *essential* and not associated with any technician cost as the technician will be present in every feasible solution.

First, an initial solution is constructed. Due to the set of workers W being finite and each worker having a daily maximum distance, finding a feasible solution for the technicians is already a computationally hard problem. In contrast, finding a feasible solution for the trucks is trivial, therefore the solution for the technicians is constructed first. We assume that each request $r \in R$ is delivered on its earliest delivery day e_r. This ensures maximally large time windows for the installation. Then, the *large neighborhood search* (LNS) for the technicians described below is used to iteratively construct a solution for the technicians until a feasible routing satisfying all installations is obtained. Finally, a minimum-cost insertion procedure is used to construct a solution for the trucks, such that the delivery time windows implied by the solution for the technicians are satisfied. The insertion procedure iteratively calculates the cheapest insertion positions for all requests that have not been added to the solution so far and performs the overall cheapest insertion.

Starting from the initial solution, the iterative method is run until the time limit is reached. Each iteration starts with a sequence of nested decisions leading to one of the improvement procedures $P_{\mathcal{K}}$, $P^\star_{\mathcal{K}}$, $P_{\mathcal{W}}$ or $P^\star_{\mathcal{W}}$. The *simple truck procedure* $P_{\mathcal{K}}$ targets the partial solution associated with the trucks, likewise the *simple technician procedure* $P_{\mathcal{W}}$ targets the partial

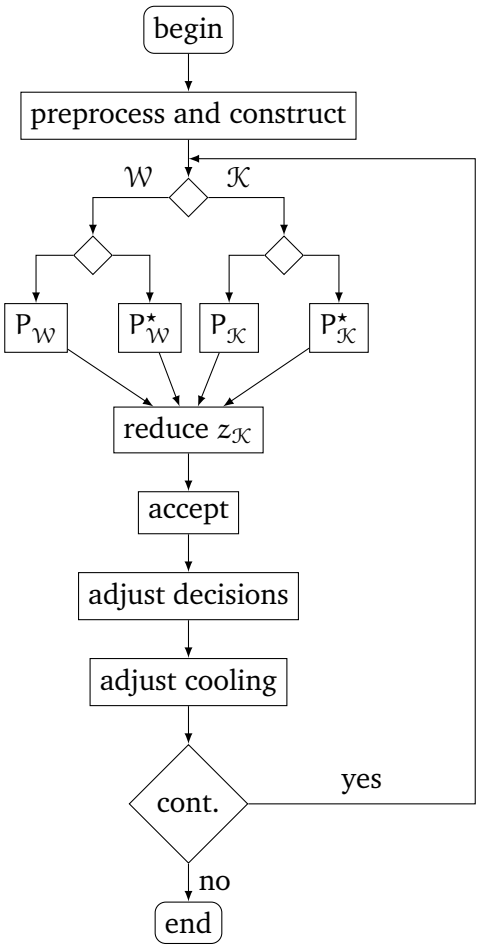


Figure 5.3: Overview

solution associated with the technicians. The *alternative procedures* $P^{\star}_{\mathcal{K}}$ and $P^{\star}_{\mathcal{W}}$ consist of two steps. The first step corresponds to the simple procedure and targets the partial solutions associated with trucks and technicians, respectively, but the time windows implied by the respective opposite partial solution are relaxed. The second step attempts to repair possible infeasibilities introduced by the relaxation of the time windows in the opposite partial solution.

The last step of the improvement aims to reduce the maximum number of trucks on any single day $z_{\mathcal{K}}$ by one through a matching based heuristic. The obtained candidate solution is compared to the incumbent solution and accepted according to a simulated annealing criterion. The observed improvement relative to the incumbent solution is used to score the decisions and to adjust the weights for subsequent iterations. Also, the parameters of the acceptance criterion, i.e., the current temperature as well as the cooling rate are adjusted according to the cooling schedule, the time limit and the observed runtime performance of the solution method.

5.3.1 Improvement procedures

The result of the hierarchical decision process is a specific improvement procedure. First, the *subproblem decision* is used to realize the decomposition of the problem, i.e., whether the truck or the technician solution is the primary target for improvement. Next, the *mode decision* guides the interaction between the partial truck and technician solutions during the

improvement process. The only interaction between trucks and technicians is based on the precedence constraint $t_r^+ < t_r^-$ of the delivery and installation days and the resulting penalty term. Two modes are considered, the *simple mode* requires that the current technician solution remains feasible if the truck solution is improved and vice versa. The *alternative mode* allows the improvement procedures to temporarily introduce infeasibility in the solution not associated with the primary target of the subproblem decision. To realize these modes, we introduce *latest delivery days* $\hat{t}_r^+ \in [e_r, l_r]$ and *earliest installation days* $\hat{t}_r^- \in [e_r + 1, l_r + 1]$ for all requests $r \in R$. Depending on the selected procedure and its mode these parameters are set to constrain the delivery or installation time windows of the requests in the embedded heuristics.

As both decisions are binary, the following four procedures are derived. $P_{\mathcal{K}}$ and $P^{\star}_{\mathcal{K}}$ primarily target the truck solution in simple and alternative mode. Similarly the procedures $P_{\mathcal{W}}$ and $P^{\star}_{\mathcal{W}}$ target the technician solution in simple and alternative mode.

- $P_{\mathcal{K}}$ (simple truck procedure). First, a large neighborhood move $\mathrm{LNS}_{\mathcal{K}}$ is performed. Afterwards a variable neighborhood descent $\mathrm{VND}_{\mathcal{K}}$ is run until a locally optimal solution is obtained. The latest delivery days $\hat{t}_r^+ := t_r^- - 1$ respect the current installation days, such that the technician solution remains feasible.

- $P_{\mathcal{W}}$ (simple technician procedure). This procedure is analogous to the procedure $P_{\mathcal{K}}$. A large neighborhood move $\mathrm{LNS}_{\mathcal{W}}$ and a variable neighborhood descent $\mathrm{VND}_{\mathcal{W}}$ are performed. Additionally, a scheduling heuristic is run after a locally optimal solution has been found. This heuristic may rearrange the tours associated with a technician to decrease the penalty term of the objective function. The earliest installation days $\hat{t}_r^- := t_r^+ + 1$ respect the current delivery days and the current partial truck solution is guaranteed to remain feasible.

- $P^{\star}_{\mathcal{K}}$ (alternative truck procedure). First, the simple truck procedure $P_{\mathcal{K}}$ is run with latest delivery days $\hat{t}_r^+ = l_r$, i.e., the delivery is restricted only by the delivery time windows $[e_r, l_r]$ given in the instance. Therefore, the locally optimal solution obtained after the $\mathrm{VND}_{\mathcal{K}}$ may violate the delivery-installation precedence constraints for a subset of requests $R' = \{r' \in R \colon t_{r'}^+ \geqslant t_{r'}^-\}$. The requests $r \in R'$ are removed from the partial technician solution and the simple technician procedure $P_{\mathcal{W}}$ with $\hat{t}_r^- := t_r^+ + 1$ is performed.

- $P^{\star}_{\mathcal{W}}$ (alternative technician procedure). In this procedure, the earliest installation days $\hat{t}_r^- := \max\{e_r + 1, t_r^+\}$ are set in such a way that the delivery installation precedence constraints are violated by at most one day, i.e., the installation of a request r may move onto the current delivery day t_r^+. The reason is that the delivery of a request is usually more constrained than the installation and a single application of the procedure should not render a major part of the current truck solution infeasible. As in the alternative truck procedure, infeasible requests are removed from the partial truck solution and the simple truck procedure $P_{\mathcal{K}}$ is run with $\hat{t}_r^+ := t_r^- - 1$.

Note that the simple truck procedure $P_{\mathcal{K}}$ will always result in feasible solutions as the number of available trucks is unlimited. All other procedures involve a possible modification of the technician solution and may not be able to obtain a feasible solution. In that case, the current iteration is aborted.

5.3.2 Large neighborhood search

The first step of all described procedures is a move in a large neighborhood. The goal is twofold. On the one hand, these moves should improve the current solution and on the other hand these moves act as a shaking procedure for the subsequent VND heuristics. The solution method uses multiple destroy and repair operators. The destroy operators are as follows.

- **Random.** Given a number $q \leqslant |R|$, q requests are randomly chosen and removed from the truck or technician solution. The number q is chosen randomly from a predefined interval.

- **Location based.** From the set of customer locations a single location is chosen randomly and all requests associated with that location are removed from the truck or technician solution.

- **Single day (trucks only).** From the set of days with at least a single truck scheduled, a random day is chosen and all trucks and requests scheduled on that day are removed from the truck solution.

- **Single technician (technicians only)**. From the set of active technicians, a single technician is chosen and all tours and requests associated with that technician are removed from the technician solution.

The repair operators reinsert the removed and currently unplanned requests into the remaining partial solution. All procedures are iterative and insert exactly one request in each iteration.

- **Parallel.** All insertion positions for all unplanned requests are evaluated and recorded. Then, a single insertion operation associated with exactly one position and request is chosen randomly with respect to a fixed *determinism parameter* as described in Ropke and Pisinger [92], i.e., an insertion operation with lower costs is chosen with a higher probability, yet even comparatively costly insertions may be performed. The insertion positions considered include the insertion into existing tours as well as the creation of a new tour.

- **Sequential.** In this procedure, the unplanned requests are inserted in a random order. The order is established anew each time the procedure is called. For the current request, all possible insertion positions, including the creation of new tours, are evaluated and recorded. Among these, an insertion is chosen randomly according to a fixed determinism parameter.

- **Regret-2.** This operator is based on the regret-k heuristic (cf. [84]). For each unplanned request all insertion positions are evaluated and recorded. Then, two cases are distinguished: (i) if there exists a non-empty set of unplanned requests with only a single insertion position, then one of these requests is chosen randomly and the insertion is performed. (ii) all unplanned requests allow for at least two insertions in which case the request is chosen that maximizes the cost difference between the best and the second best insertion position. Ties are not resolved, the first request that realizes the maximum is chosen.

5.3.3 Variable neighborhood descent

The solution obtained after the LNS move is polished by a local search heuristic combining well-known polynomial-size neighborhoods in a VND procedure. Although the neighborhoods and order of the neighborhoods are the same for trucks and technicians, some aspects like the handling of time windows are specifically tailored towards trucks and technicians, respectively, such that $\mathrm{VND}_{\mathcal{K}}$ for trucks and $\mathrm{VND}_{\mathcal{W}}$ for technicians are considered individually below.

1. **Intra-tour.** First *2-opt* is applied to a route until a local optimum is reached. Then, *Or-opt*, a subset of the *3-opt* neighborhood proposed in [81], is applied, also until a local optimum is reached.

2. **Relocate.** The relocate neighborhood considers moves whereby a request $r \in R$ is removed from a tour and inserted into another tour at a minimum-cost insertion position.

3. **Exchange.** This neighborhood removes two requests $r_1, r_2 \in R$ in two different tours and inserts each at a best position in the respective other tour.

The current neighborhood is searched exhaustively and all feasible, improving moves are recorded. If at least one such move exists, a move is chosen stochastically by using a neighborhood specific determinism parameter. Afterwards the process repeats from step 1. If the current neighborhood is empty, the process continues with the next neighborhood or terminates if the current solution is locally optimal w.r.t. all considered neighborhoods.

VND for trucks: $\mathrm{VND}_{\mathcal{K}}$

Each truck tour Π_k is restricted to the time window $[e_{\Pi_k}, \hat{t}^+_{\Pi_k}] := \bigcap_{r \in \Pi_k} [e_r, \hat{t}^+_r]$. For the neighborhood *relocate*, we consider a move of request $r \in R$ into the truck tour Π_k if $[e_r, \hat{t}^+_r] \cap [e_{\Pi_k}, \hat{t}^+_{\Pi_k}] \neq \emptyset$, i.e., the request can be inserted into the tour without violating time window constraints. In case that the request cannot be delivered on the currently scheduled day of the tour $t_{\Pi_k} \notin [e_r, \hat{t}^+_r]$, we move the tour to the latest possible day, i.e., we set $t_{\Pi_k} := \min\left\{\hat{t}^+_r, \hat{t}^+_{\Pi_k}\right\}$, as demonstrated in Figure 5.4. Contrary to that, the modification of scheduled days is not considered in the neighborhood *exchange*. That is, given two requests $r_i \in \Pi_i, r_j \in \Pi_j$, we calculate the tours $\Pi'_i = \Pi_i \backslash \{r_i\}$ and $\Pi'_j = \Pi_j \backslash \{r_j\}$ obtained by removing the requests from their respective tours. If $t_{\Pi'_i} \in [e_{r_j}, \hat{t}^+_{r_j}]$ and $t_{\Pi'_j} \in [e_{r_i}, \hat{t}^+_{r_i}]$, then the exchange (r_i, r_j) is considered.

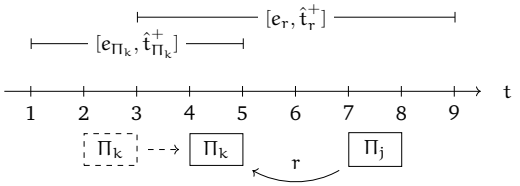


Figure 5.4: A move in the neighborhood *relocate*. As the current day of tour Π_k, $t_{\Pi_k} = 2$ is outside of request r's time window, the tour is moved to day $\min\{\hat{t}^+_r, \hat{t}^+_{\Pi_k}\} = 4$.

When a locally optimal truck solution is reached, we try to reduce the number of truck tours K by recombining the subtours of the truck tours. As all subtours are independent, only the maximum daily distance D and the time window constraints have to be considered.

Problem 5.1 (Subtour recombination)**.** *Given the set of all subtours*

$$\Pi = \{\pi_i \in \Pi_k : 1 \leqslant k \leqslant K\}$$

and for all subtours $\pi_i \in \Pi$*, their corresponding distances* d_{π_i} *and time windows*

$$[e_{\pi_i}, \hat{t}^+_{\pi_i}] \coloneqq \bigcap_{r \in \pi_i} [e_r, \hat{t}^+_r].$$

Consider all partitions of Π *into disjoint truck tours* $\Pi'_1, \ldots, \Pi'_{K'}$ *such that each truck tour* Π'_j *satisfies the maximum daily distance* $\sum_{\pi_i \in \Pi'_j} d_{\pi_i} \leqslant D$ *and can be scheduled on at least one day, i.e.,* $\bigcap_{\pi_i \in \Pi'_j} [e_{\pi_i}, \hat{t}^+_{\pi_i}] \neq \varnothing$*. Among these partitions, find one that minimizes the number of truck tours* K'*.*

Problem 5.1 is a special case of the *bin-packing problem with conflicts* (BPPC). The subtours correspond to *items*, their distances to *item weights*, the truck tours correspond to *bins* and the maximum daily distance corresponds to the *bin width*. Additionally, two subtours are in conflict iff their time windows do not overlap, i.e., they cannot be scheduled on a common day and thus, cannot be performed in a single truck tour.

Note that Problem 5.1 does not incorporate any notion of cost besides the number of resulting truck tours. However, moving a subtour to another day changes the delivery days of the requests served in that subtour. Especially when a subtour is moved to an earlier day, the penalty incurred by the associated requests may increase, possibly outweighing the savings achieved by decreasing the number of truck tours.

To resolve the issues resulting from penalty values and the conflicts between subtours, we consider the problem for each day individually. This ensures that the time window constraints are always satisfied as subtours that are already performed on a single day are always compatible, i.e., no conflicts need to be considered. Additionally, the scheduled days of the subtours are not changed, thus eliminating any impact on the associated penalty values. However, the opportunities to recombine subtours and reduce the number of trucks may be more limited as illustrated in Example 5.3. This reduced problem for each individual day is a *bin-packing problem* (BPP) with the same correspondences as above, just without the conflicts.

Example 5.3. *Consider the instance given in Figure 5.5a There are five subtours with time windows and distances. The maximum daily distance is* $D = 10$*.*

A feasible partition into truck tours is shown in Figure 5.5b with four truck tours, the first on day 1*, the next two on day* 2 *and the last on day* 3*. The subtours* $\{\pi_2, \pi_3, \pi_4\}$ *on day* 2 *may be combined into a single truck tour. However, subtours* $\{\pi_1, \pi_5\}$ *on days* 1 *and* 3 *cannot be combined into a single tour because their time windows do not overlap, i.e., subtours* $\{\pi_1, \pi_5\}$ *are in conflict.*

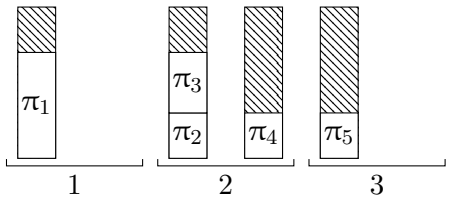
Combining subtours $\{\pi_1, \pi_5\}$ *with other subtours is still possible if subtours are allowed to move between days. This BPPC case is illustrated in Figure 5.5c, reducing the number of truck tours from* 4 *to* 2*. Note that subtours* $\{\pi_2, \pi_5\}$ *are moved to earlier days* 1 *and* 2*, respectively. As such, they may negatively impact the associated penalty values.*

If subtours are restricted to their currently scheduled days then only tours $\{\pi_2, \pi_3, \pi_4\}$ *are combined into a single truck tour and the number of truck tours is reduced from* 4 *to* 3*. This BPP case is illustrated in Figure 5.5d,*

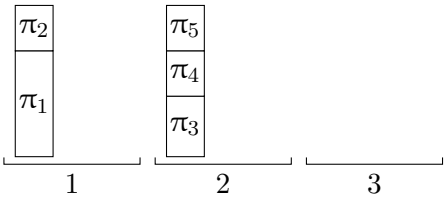
The problem is solved heuristically by applying the *best-fit decreasing* (BFD) heuristic for the BPP (cf. Garey et al. [44]) to each set of subtours whose truck tours are scheduled on the same day in the current input solution. The BFD heuristic traverses all given items by

i	e_{π_i}	$\hat{t}^{+}_{\pi_i}$	d_{π_i}	current t_{π_i}
1	1	1	7	1
2	1	2	3	2
3	2	2	4	2
4	2	3	3	2
5	2	3	3	3

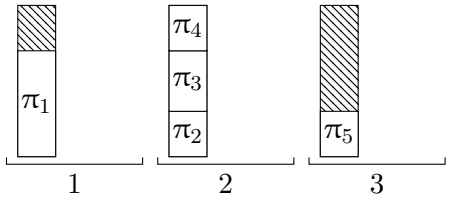
(a) Instance with five subtours and $D = 10$



(b) Partition into four truck tours



(c) Optimal BPPC partition



(d) Optimal BPP partition for each day

Figure 5.5: Example set of subtours and different combinations into truck tours. The height of each subtour's rectangle corresponds to its distance. The height of the rectangles including the shaded areas corresponds to the maximum daily distance D.

non-increasing weights and adds an item to the most-filled bin that has still enough space to accommodate the item.

VND for technicians: $\mathrm{VND}_{\mathcal{W}}$

The VND for technicians does not operate on scheduled days. Instead, each technician tour σ_{wi} is associated with the earliest possible day $\hat{t}^{-}_{\sigma_{wi}} = \max\left\{\hat{t}^{-}_{r} : r \in \sigma_{wi}\right\}$ on which tour σ_{wi} can be scheduled. In the *relocate* neighborhood, a move of a request r into a technician tour σ_{wi} is considered only if $\hat{t}^{-}_{r} \leqslant \hat{t}^{-}_{\sigma_{wi}}$ or in case of $\hat{t}^{-}_{r} > \hat{t}^{-}_{\sigma_{wi}}$ if the earliest possible day $\hat{t}^{-}_{\sigma_{wi}}$ of tour σ_{wi} can be feasibly moved to the later day $\hat{t}^{-}_{r}$. Moves for the neighborhood *exchange* between tours σ_{wi}, $\sigma_{w'j}$ and requests $r_i \in \sigma_{wi}$ and $r_j \in \sigma_{w'j}$ are considered only if $\hat{t}^{-}_{r_i} \leqslant \hat{t}^{-}_{\sigma_{w'j}\setminus\{r_j\}}$ and $\hat{t}^{-}_{r_j} \leqslant \hat{t}^{-}_{\sigma_{wi}\setminus\{r_i\}}$ hold.

When the VND has reached a locally optimal solution, actual schedules for the technicians have to be calculated, that is, the specific installation days for each tour have to be determined before the objective function can be evaluated w.r.t. penalties. The scheduling procedure is described in more detail in Section 5.5.

5.4 Minimization of the number of trucks

The objective function component $z_{\mathcal{K}}$ corresponds to the maximum number of trucks on any single day and is not considered explicitly by the above described procedures and operators. Instead a simple heuristic is used to address this part of the objective function directly by trying to reduce the maximum number of trucks by one without changing the routing and subtour combination decisions as detailed in Problem 5.2. As a feasible assignment of trucks to days with at most $z_{\mathcal{K}} - 1$ truck tours on any single day implies that at least one truck tour's delivery day is changed, the penalty component of the objective function is also influenced.

Problem 5.2 (Reduce $z_{\mathcal{K}}$)**.** *Given the set of truck tours* $\{\Pi_1, \dots, \Pi_K\}$ *with time windows* $[e_{\Pi_k}, \hat{t}^+_{\Pi_k}] \subseteq \{1, \dots, H-1\}$ *and penalties* $p_{\Pi_k} \coloneqq \sum_{r \in \Pi_k} p_r$.

Find a minimum-penalty assignment of truck tours Π_k *to delivery days* $t_{\Pi_k} \in [e_{\Pi_k}, \hat{t}^+_{\Pi_k}]$ *with at most* $z_{\mathcal{K}} - 1$ *truck tours on any single day.*

Problem 5.2 can be efficiently solved by a *minimum-cost network flow problem* (MCFP) formulation on a digraph $G = (U, A)$. The node set

$$U = \{u^-, u^+\} \cup \{\Pi_k \colon 1 \leqslant k \leqslant K\} \cup \{t \in \{1, \dots, H-1\}\}$$

comprises two artificial source and sink nodes u^-, u^+, a node for each truck tour Π_k and a node for each day $t \in \{1, \dots, H-1\}$. The arc set $A = A_1 \cup A_2 \cup A_3$ is the union of three arc sets

$$\begin{aligned} A_1 &= \{(u^-, \Pi_k) \colon 1 \leqslant k \leqslant K\} \\ A_2 &= \left\{(\Pi_k, t) \colon 1 \leqslant k \leqslant K, t \in [e_{\Pi_k}, \hat{t}^+_{\Pi_k}]\right\} \\ A_3 &= \{(t, u^+) \colon t \in \{1, \dots, H-1\}\} \end{aligned}$$

connecting the artificial source node to all nodes representing truck tours, all nodes representing truck tours to the nodes representing days in their current time window and all nodes representing days to the artificial sink node, respectively. In total, the digraph G has $|U| \leqslant K + H + 1$ nodes and $|A| \leqslant K + K(H-1) + H - 1$ arcs and is thus of pseudo-polynomial size w.r.t. the input size.

The arc weight c_{kt} for an arc $(\Pi_k, t) \in A_2$ from a node corresponding to truck tour Π_k to a node corresponding to day t is set to the penalty incurred when truck k is assigned to day t:

$$c_{kt} \coloneqq \sum_{r \in \Pi_k} (t_r^- - t - 1) p_{i_r} n_r$$

All other arcs have zero weight. The arc capacities $\mathrm{Cap}_{tu^+} \coloneqq z_{\mathcal{K}} - 1$ for an arc $(t, u^+) \in A_3$ from a node representing day t to the sink node u^+ are set such that at most $z_{\mathcal{K}} - 1$ trucks can be assigned to any day. All other arcs have unit capacity, i.e., $\mathrm{Cap}_a = 1$, $a \in A_1 \cup A_2$. A small instance and the corresponding MCFP network are illustrated in Example 5.4.

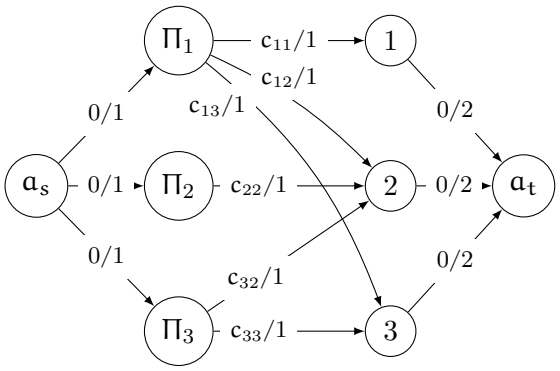
Example 5.4. *Consider the instance given in Figure 5.6a. All of the three truck tours are scheduled on the same day* $t_{\Pi_k} = 2$ *hence,* $z_{\mathcal{K}} = 3$*. The corresponding MCFP network is illustrated in Figure 5.6b. Tour* Π_1 *can be assigned to three days* $1, 2, 3$*, tour* Π_2 *only to day 2 and tour* Π_3 *to days* $2, 3$*. The capacity of the arcs connecting the nodes representing days to the artificial sink node is* $z_{\mathcal{K}} = 2$*, i.e., at most two trucks can be assigned to any day, reducing the maximum number of trucks on any single day by one. Generally,* $c_{kt} \leqslant c_{kt'}$ *holds for* $t \geqslant t'$*, thus the optimal solution is obtained with truck* Π_2 *assigned to day* 2 *and trucks* Π_1*,* Π_3 *assigned to day* 3*.*

A feasible assignment exists iff the associated MCFP allows for a flow of K. If such an assignment exists, the solution is transformed to reflect the assignment, otherwise the input solution is retained.

Suppose that all tours have pairwise distinct penalty values, then there is exactly one optimal distribution of the tours over the days. This exact approach may impact the diversity of the solutions produced in following iterations, especially if the *simple technician procedure* $P_{\mathcal{W}}$ is considered, because it is unable to change the current partial solution associated with the trucks. Additionally, solving the MCFP takes quite some time, which may not be beneficial when the impact of the maximum number of truck tours $z_{\mathcal{K}}$ is small.

k	e_{Π_k}	$\hat{t}^{+}_{\Pi_k}$	t_{Π_k}
1	1	3	2
2	2	2	2
3	2	3	2

(a) Example instance with $H = 4$ and $K = 3$. All trucks are scheduled on day $t_{\Pi_k} = 2$.



(b) MCFP with arc labels cost/capacity

Figure 5.6: Example construction of the MCFP network. The instance (a) is transformed into the MCFP network (b).

To alleviate both issues, the problem is relaxed by ignoring the arc weights c_a, $a \in A$ induced by the penalties. The resulting problem can be modeled as an *unweighted maximum bipartite matching* between the truck tours Π_k and at most $z_{\mathcal{K}} - 1$ nodes for each day t. This problem is then solved exactly with a greedy *augmenting path algorithm*, such that a feasible assignment is found iff it exists. Note that technically, the size of the matching problem's instance is pseudo-polynomial in $z_{\mathcal{K}}$. In cases where such an approach may seem inappropriate, e.g., if the number of truck tours or H, or both, are extraordinarily large, the MCFP approach described above remains a viable option.

To still account for the penalties, the adjacency lists of the nodes corresponding to truck tours are ordered in such a way, that the greedy algorithm considers arcs with lower penalties prior to those with higher penalties while searching for the next augmenting path. Nevertheless, the order of the tours Π_k is randomized, therefore, different solutions can be obtained for the same matching problem.

5.5 Technician scheduling

The operators used in the $\mathrm{LNS}_{\mathcal{W}}$ and $\mathrm{VNS}_{\mathcal{W}}$ heuristics improve the clustering and routing decisions regarding the technicians and requests, i.e., they change the sets of requests being performed together in a single tour and the sequence of requests in such a tour. In contrast, these operators do not deal with the assignment of tours to actual days and hence, do not generate the actual schedule. Instead, these heuristics only consider lower bounds for the penalties induced by the clustering decisions. Therefore, a scheduling procedure is required to obtain schedules for the technicians. As all technicians are independent, the following scheduling subproblem is solved individually for each technician.

Problem 5.3 (Technician scheduling)**.** *Given a horizon* $\{1, \ldots, H\}$ *and a set of tours* $J_w = \{\sigma_{wj}\}_{j=1}^{\ell_w}$ *associated with technician* $w \in W$*, earliest possible days* $\hat{t}^{-}_{\sigma_{wj}} := \max\left\{\hat{t}^{-}_{r} : r \in \sigma_{wj}\right\}$ *and penalty values* $p_{\sigma_{wj}} := \sum_{r \in \sigma_{wj}} p_r \geqslant 0$ *incurred for each day that a tour* σ_{wj} *is planned after its earliest possible day.*

Find a minimum-penalty schedule, such that (i) each tour is scheduled exactly once at its earliest possible day or later and (ii) working regulations are satisfied, i.e., whenever there is a block of five consecutive tours ending on day t*, the next tour cannot be scheduled before day* $t+3$*.*

In the following, we develop a simple *integer program* (IP) to model the problem, consider an efficient feasibility testing mechanism in the context of local search and insertion procedures, and propose a fast heuristic for a restricted problem.

5.5.1 Integer programming formulation

We model the scheduling problem for a fixed technician $w \in W$ as an IP using binary decision variables $x_{jt} \in \{0,1\}$, $j \in \{1,\dots,\ell_w\}$: $t \in \{\hat{t}^-_{\sigma_{wj}},\dots,H\}$ and polynomially many constraints w.r.t. the number of decision variables. The meaning of the variable assignment corresponds directly to the scheduling decision, i.e.,

$$x_{jt} = \begin{cases} 1, & \text{if tour } \sigma_{wj} \text{ is scheduled on day } t \\ 0, & \text{otherwise.} \end{cases}$$

Let $\mathcal{J} = \{1,\dots,\ell_w\}$ be the index set corresponding to the tours of technician w and $\mathcal{J}_t = \{j \in \mathcal{J}\colon \hat{t}^-_{\sigma_{wj}} \leqslant t\}$ be the indices of all tours that may be feasibly scheduled on day $t \in \{2,\dots,H\}$. Using these definitions, we provide the IP formulation (5.9)–(5.14).

$$\min \sum_{t=1}^{H} \sum_{j\in\mathcal{J}_t} (t \cdot x_{jt} - \hat{t}^-_{\sigma_{wj}}) p_{\sigma_{wj}} \tag{5.9}$$

s.t.
$$\sum_{t=\hat{t}^-_{\sigma_{wj}}}^{H} x_{jt} = 1 \qquad \forall j \in \{1,\dots,\ell_w\} \tag{5.10}$$

$$\sum_{j\in\mathcal{J}_t} x_{jt} \leqslant 1 \qquad \forall t \in \{1,\dots,H\} \tag{5.11}$$

$$\sum_{j\in\mathcal{J}_t} x_{jt} + \sum_{t'=t-5}^{t-1} \sum_{j\in\mathcal{J}_{t'}} x_{jt'} \leqslant 5 \qquad \forall t \in \{6,\dots,H\} \tag{5.12}$$

$$\sum_{j\in\mathcal{J}_t} x_{jt} + \sum_{t'=t-6}^{t-2} \sum_{j\in\mathcal{J}_{t'}} x_{jt'} \leqslant 5 \qquad \forall t \in \{7,\dots,H\} \tag{5.13}$$

$$x_{jt} \in \{0,1\} \qquad \forall j \in \{1,\dots,\ell_w\}\colon \forall t \in \{\hat{t}^-_{\sigma_{wj}},\dots,H\} \tag{5.14}$$

Expression (5.9) is a direct translation of the objective function minimizing the sum of penalties over all tours. Constraints (5.10) and (5.11) ensure that each tour is scheduled on exactly one day and that at most one tour is scheduled per day, respectively. Constraints (5.12) and (5.13) model the restrictions of consecutive days, ensuring that days $t+1$ and $t+2$ are days off, given that five tours are scheduled on days $\{t-4,\dots,t\}$.

To derive the actual schedule specified by the installation days $t_{\sigma_{wj}}$ of the tours, suppose a solution to the IP given by the decision variables x_{jt}. Then, the installation day of tour σ_{wj} is derived as follows: $t_{\sigma_{wj}} \coloneqq \sum_{t=\hat{t}^-_{\sigma_{wj}}}^{H} t \cdot x_{jt}$.

5.5.2 Testing feasibility

The number of tours that can be scheduled in a certain time frame depends on the number of days off required. Equation (5.15) gives the maximum number of tours for a time frame

of $m \geqslant 0$ consecutive days.

$$u_{\max}(m) = \begin{cases} 0 & \text{for } m = 0 \\ m - \left\lfloor \frac{m-1}{5} \right\rfloor & \text{otherwise} \end{cases} \tag{5.15}$$

Let $U_t = \left\{ \sigma_{wj} \in J_w \colon \hat{t}^-_{\sigma_{wj}} \geqslant t \right\}$ be the set of tours that need to be scheduled on days $\{t, \ldots, H\}$. A feasible schedule exists iff $|U_t| \leqslant u_{\max}(H - t + 1)$ for all $t \in \{1, \ldots, H\}$. As $u_{\max}$ is monotonically increasing, we observe that

$$|U_t| \leqslant u_{\max}(H - t + 1) \Rightarrow |U_t| \leqslant u_{\max}(H - (t - \delta) + 1)$$

holds for all $\delta \geqslant 0$. Therefore, we only need to test feasibility for the earliest possible days $\hat{t}^-_{\sigma_{wj}}$, $\sigma_{wj} \in J_w$ and the required time is polynomial in the number of tours.

As noted above, the local search and insertion heuristics operate on earliest possible days $\hat{t}^-_{\sigma_{wj}}$ instead of actual days $t_{\sigma_{wj}}$. Therefore, when evaluating local search moves and insertion operations, only the feasibility of the operation w.r.t. the scheduling constraints needs to be ensured. The number of evaluations of these operations exceeds the number of actual executions, usually by a rather large margin. As such, the goal is to design a feasibility checking procedure that does only incur a small runtime cost for evaluations while larger runtime costs are acceptable when an operation is actually performed.

In particular, two queries are required to evaluate, whether the scheduling instance remains feasible if either

1. a new tour at a specific earliest day is added to the technician or
2. the earliest possible day of a tour currently associated with the technician is modified.

Both cases can be checked in $\mathcal{O}(1)$ time by maintaining the set of so called *tight* days in $\mathcal{O}(H)$ time whenever a move or insertion is performed. A day t is *tight* iff $|U_t| = u_{\max}(H - t + 1)$, i.e., adding a new tour to any day in $\{t, \ldots, H\}$ will exceed the number of tours that can be scheduled feasibly. Let γ_t be the number of tight days in the set of days $\{1, \ldots, t\}$. A tour can be feasibly added on day t iff $\gamma_t = 0$. A tour can be moved from day t to a later day $t' > t$ if $\gamma_t = \gamma_{t'}$, i.e., the days $\{t + 1, \ldots, t'\}$ are not tight. Moving a tour to an earlier day $t' < t$ does not interfere with feasibility.

5.5.3 Greedy heuristic

To the best of our knowledge, the complexity status of the scheduling subproblem is unknown w.r.t. the minimization of the total penalty. As such, we do not describe a polynomial time algorithm to solve it but instead restrict the problem further by disallowing the scheduling of five consecutive days unless the five consecutive tours conclude the schedule.

We say a tour is *eligible* on day t if it can be feasibly scheduled on day t, i.e., $\hat{t}^-_{\sigma_{wj}} \leqslant t$. We denote by E_t the set of eligible tours on day t. A simple greedy heuristic for the restricted problem is given in Algorithm 5.1. The selection steps in lines 4 and 8 can be efficiently realized by a priori sorting all tours by non-decreasing earliest installation days $\hat{t}^-_{\sigma_{wj}}$ and maintaining a priority queue (e.g., a heap data structure) of all eligible tours that have not been scheduled so far. The sorting step requires $\mathcal{O}(\ell_w \log \ell_w)$ time, while each of the ℓ_w iterations incurs $\mathcal{O}(\log \ell_w)$ time for the lookup and update operations of the priority queue. The combined runtime complexity is $\mathcal{O}(\ell_w \log \ell_w)$.

It remains to show that the greedy heuristic generates a feasible plan whenever the input instance is feasibly solvable, although it only generates schedules satisfying the restricted

Algorithm 5.1 Greedy restricted

Input: Set of tours J_w, horizon $\{1, \dots, H\}$
Output: Days $t_{\sigma_{wj}}$ for each tour $\sigma_{wj} \in J_w$

$F \leftarrow \varnothing$ ▷ set of scheduled tours
$t \leftarrow 1$
while $|F| < \ell_w$ **do**
  $t \leftarrow \min\{t' \in \{t, \dots, H\} : |E_{t'} \backslash F| > 0\}$ ▷ next day with an eligible tour
  if $5 \leqslant t < H$ **and** four tours scheduled on days $\{t-4, \dots, t-1\}$ **then**
    $t \leftarrow t + 1$ ▷ insert day off
  end if
  $\sigma_{wj} \leftarrow$ tour with maximum penalty in $E_t \backslash F$
  $t_{\sigma_{wj}} \leftarrow t$ ▷ assign installation day
  $F \leftarrow F \cup \{\sigma_{wj}\}$
  $t \leftarrow t + 1$
end while

problem. We observe, that any feasibly solvable instance for the original problem is also feasibly solvable for the restricted problem.

Lemma 5.1. *If an instance of the original problem is feasible, then there exists a feasible schedule that contains at most one sequence of five consecutive tours and this sequence concludes the schedule.*

Proof. Assume a schedule with a sequence of five consecutive tours that does not conclude the schedule and whose last tour is scheduled on day t. Then days $t+1$ and $t+2$ are days off and the next sequence does not start before day $t+3$ (cf. Figure 5.7a). By delaying the last tour of the sequence by a single day, a sequence of four tours ends on day $t-1$, a single tour is scheduled on day $t+1$ and the next sequence does not start before day $t+3$ (cf. Figure 5.7b). This schedule is still feasible w.r.t. earliest possible days and days off, but is short of one sequence of five consecutive tours. Hence, iterating the above argument results in a feasible schedule with at most one sequence of five consecutive tours concluding the schedule. □

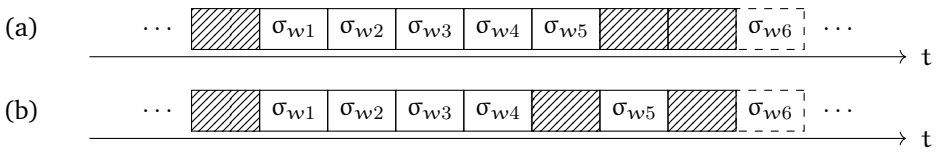


Figure 5.7: Breaking up a sequence of five consecutive tours without compromising feasibility.

From Lemma 5.1 it follows that all points discussed w.r.t. efficient feasibility checking still hold for the restricted problem and that the proposed greedy heuristic does generate feasible schedules whenever the given instance is feasible w.r.t. the original problem.

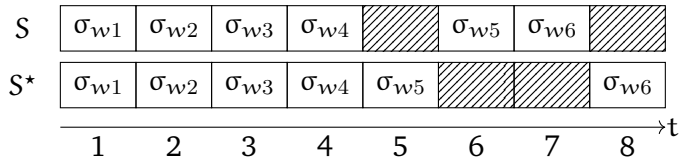
Quality of heuristic solutions

The greedy heuristic is rather simple and does not exhibit any approximation guarantees. First, the gap between the total penalties of optimal solutions to the original and restricted

problems may be arbitrarily large. Second, the gap between the heuristic and optimal solutions to the restricted problem may be arbitrarily large as well. Both cases are illustrated in Examples 5.5 and 5.6.

j	$\hat{t}^{-}_{\sigma_{wj}}$	$p_{\sigma_{wj}}$
1	1	α
2	2	α
3	3	α
4	4	α
5	5	α
6	7	1

(a) Instance with $H = 8$, $\alpha \gg 1$.



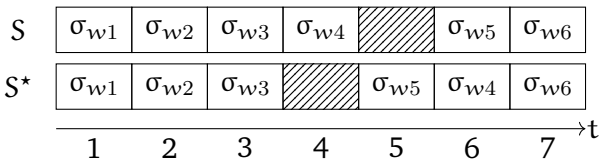
(b) Optimal solutions

Figure 5.8: The solutions S and S* shown in (b) are optimal w.r.t. the restricted and original problems for the instance given in (a). The total penalties are $p_S = \alpha$ and $p_{S^\star} = 1$.

Example 5.5. *Figure 5.8 shows an instance and two solutions S and S*. Although solution S is optimal for the restricted problem, it can be improved by moving tour σ_{w5} to its earliest possible day and delaying σ_{w6} by a single day. The penalty is reduced from α to 1. As α gets larger, the gap between both solutions gets likewise larger.*

j	$\hat{t}^{-}_{\sigma_{wj}}$	$p_{\sigma_{wj}}$
1	1	α
2	2	α
3	3	α
4	4	1
5	5	α
6	7	1

(a) Instance with $H = 7$, $\alpha \gg 2$.



(b) Heuristic and optimal solutions

Figure 5.9: The solutions S and S* shown in (b) are the heuristic and optimal solutions for the restricted problem instance given in (a). The total penalties are $p_S = \alpha$ and $p_{S^\star} = 2$.

Example 5.6. *Figure 5.9 shows an instance and two solutions S and S*, corresponding to the greedy heuristic solution and the optimal solution for the restricted problem, respectively. When the heuristic is due to schedule a tour on day $t = 4$, the only eligible tour is σ_{w4}. However, delaying σ_{w4} allows to schedule σ_{w5} on its earliest possible day. By introducing this additional delay, the total penalty is reduced from α to 2. As α gets larger, the gap between both solutions gets likewise larger.*

5.6 Adaptive layer

The adaptive layer is one of the key components of the solution method. Its purpose is the orchestration of the above described heuristics and the *acceptance criterion* such that the

initially stated goal of deriving best effort solutions with a limited budget of computational resources on a large variety of different instances is realized.

The adaptive layer in this solution method (i) adaptively decides for a subproblem, (ii) the mode and heuristics to improve the subproblem by and (iii) adjusts the parameters of the acceptance criterion such that the relationship between diversification and intensification of the search space is adequate w.r.t. the given time limit and computing environment.

All decisions are randomized but biased by probabilities associated with each decision outcome. These probabilities are adjusted during the solution process to adapt to the instance characteristics. In that way, our approach is a straightforward extension of the adaptive layer of the ALNS method proposed by Pisinger and Ropke [84]. Each embedded heuristic is associated with a weight or probability. According to these probabilities, a heuristic is chosen randomly and applied to the current solution. Then, depending on whether the application of the chosen heuristic leads to an improvement or not, the chosen heuristic's weight is adjusted. Over the course of the solution process, the adaptive layer adapts to the instance characteristics such that beneficial heuristics will be chosen and applied more often than those that lead to less or no improvements at all.

In Subsection 5.6.1 we describe the concrete realization of the heuristics and operator selection mechanism and discuss different alternatives along with their drawbacks. Subsequently, in Subsection 5.6.2 we give a description of the acceptance criterion and its adaptive cooling schedule.

5.6.1 Subproblem and heuristic selection

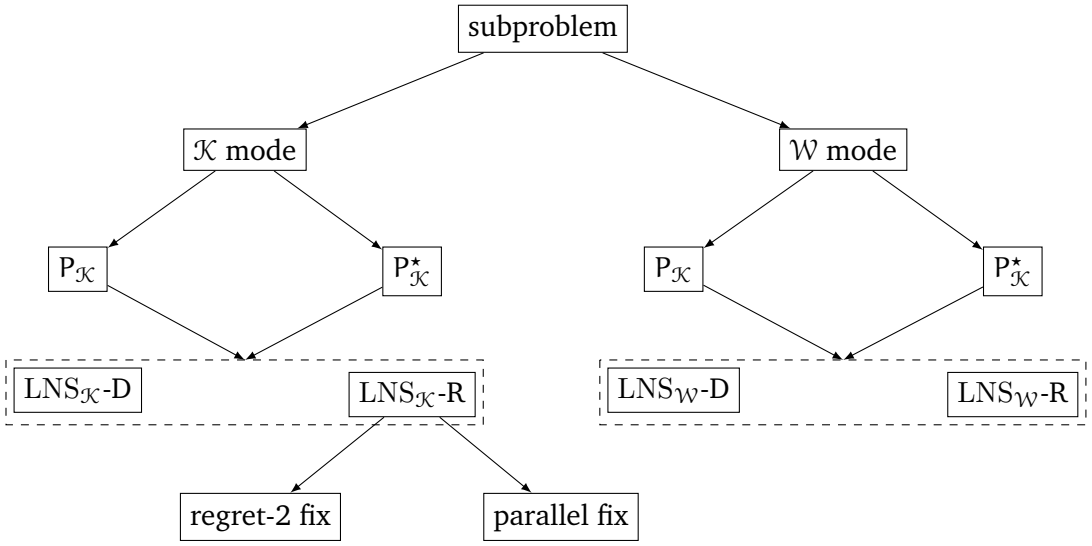


Figure 5.10: Hierarchy of decisions

All decisions are arranged in a tree-like hierarchy as shown in Figure 5.10. The first two decision layers correspond to the decomposition and mode dimensions of the procedures, i.e., whether to improve the truck or technician partial solution and whether to respect or ignore the opposite partial solution. In each procedure, the destroy and repair heuristics for the large neighborhood move are chosen. Depending on the chosen heuristics, further decisions may be evaluated. Given that the regret-2 or parallel minimum-cost insertion repair heuristics have been chosen in a truck procedure, then a parameter is chosen that either fixes trucks to their currently scheduled days or allows them to be moved inside their time windows.

At the end of each iteration, the candidate solution S' is compared to the current solution S and the improvement $\delta = \max\{0, z_a(S) - z_a(S')\}$ is evaluated. Each involved decision d is then scored with δ and accumulates these scores over a segment of a fixed number of n_{segment} iterations. After each segment, the weight of a decision is adjusted by an aging function with a decision specific aging parameter $\theta_d \in [0, 1] \subset \mathbb{R}$. Let $\omega_{d,i}$ be the weight of decision d and let $\phi_{d,i}$ be the sum of scores accumulated in the i-th segment. The update of the weights at the end of the segment is performed for each decision d according to the rule

$$\omega_{d,i+1} \leftarrow \theta_d \phi_{d,i} + (1 - \theta_d)\omega_{d,i}.$$

The decision specific aging parameters θ_d allow for the implementation of analogs to long- and short-term memory. Some decisions may adapt slowly and keep their weights for a longer period of time, while others adjust faster and might oscillate, as shown in the following Example 5.7.

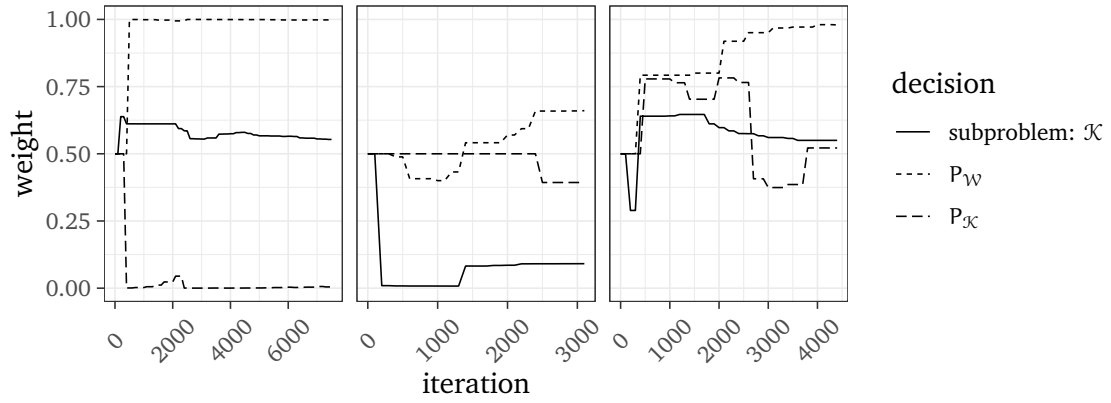


Figure 5.11: Weights for the subproblem and procedure decisions over 30 s for three different instances.

Example 5.7. *Figure 5.11 shows the weights of three decisions for three different instances and runs of 30 s. The solid trajectory corresponds to the probability of the truck subproblem being chosen in the subproblem decision. Similarly, the dashed trajectories correspond to the probabilities of the simple procedures* $P_{\mathcal{K}}$ *and* $P_{\mathcal{W}}$ *being chosen in the mode decisions. The aging factor for the subproblem decision is* $\theta_{\textit{subproblem}} = 0.01$ *and the factors for the mode decisions are* $\theta_{\mathcal{K}} = \theta_{\mathcal{W}} = 0.5$*, respectively.*

In the first instance, the mode decisions are rather imbalanced. The simple mode for technicians and the alternative mode for trucks are dominating their respective counterparts, while the subproblem decision is rather balanced. In the second instance, the subproblem decision heavily leans towards the technician subproblem while the mode decisions are balanced. In the third instance, the subproblem decision is balanced and adapts slowly, while the mode decision for the trucks adapts faster with large jumps.

Discussion of alternative approaches

Instead of keeping the subproblem decision in the adaptive layer, alternative approaches could have been used. The essential task at this point is to determine how to distribute the available computational budget over the two subproblems and their improvement. That is, how much of the budget should be allotted to the improvement of the partial solutions associated with the technicians or trucks, and when.

By *fixed allotment* we denote an a priori allotment of the computational budget to the subproblems, e.g. 50% to the technicians and 50% to the trucks. Such an approach is quite common for hierarchical or multi-stage decomposition approaches, e.g. in a different but similar complex problem, Kheiri et al. [67] allot 80% of the budget to a scheduling subproblem followed by a route improvement procedure in the remaining 20% of the budget. Similarly, Jagtenberg et al. [61] use a fixed order and first improve the technicians until no further improvements are made or a fixed portion of the budget has been used and allot the remaining budget to the improvement of the trucks w.r.t. the partial technician solution obtained before.

The drawbacks of the *fixed allotment* stem from the comparatively complex objective function. Suppose two heavily biased instances, e.g., those where either the technicians or the trucks completely dominate the cost and provide room for a lot of improvements. An allotment of 50% of the budget to each subproblem would essentially waste 50% of the budget. In fact, for any arbitrary but fixed allotment, instances can be derived such that a large share of the budget is wasted. As such, we abandoned such an approach.

By *objective function dependent allotment* we denote an approach that actively incorporates the objective function coefficients or the objective function value of a solution, or both. For instance, when the objective function terms corresponding to the partial truck solution clearly dominate the objective function terms corresponding to the partial technician solution, then more time could be allotted to the improvement of the trucks. The same approach could be applied when the objective function coefficients associated with the truck terms are much larger than those of the technicians, and vice versa.

The drawbacks of this approach become evident on instances where the objective function terms associated with either the trucks or technicians clearly dominate the respective other but do not leave much room for improvement. For example, suppose a solution with a partial truck solution outweighing the partial technician solution by a large margin. However, the truck partial solution is already near optimal as it is comparatively easy despite its large cost. In this case most of the remaining budget should be allotted to the improvement of the partial technician solution to realize the remaining improvement potential. As such, we abandoned such an approach as well.

In conclusion, the chosen adaptive mechanism overcomes the shortcomings of both, the *fixed* and the *objective function dependent allotments*, because only relative improvements are taken into consideration and w.r.t. the complete budget, a single heuristic application takes a comparatively small amount of the budget, such that there is enough time to adapt.

5.6.2 Acceptance criterion

Besides the allotment of the budget for the improvement of the subproblems, the budget needs to be balanced between search space exploration and intensification of promising regions. Like the ALNS approach of Pisinger and Ropke [84], we use a *simulated annealing* (SA) acceptance criterion. The cooling schedule should be designed such that it avoids (i) wasting time with intensification of mediocre solutions in case of a large remaining budget and (ii) wasting time with diversification in case of a small remaining budget. In other words, the cooling schedule should provide a balance of intensification and diversification that enables a best effort sampling of the search space w.r.t. the budget.

The central idea is to use a cooling schedule that will traverse a calculated temperature range $[\tau_\star, \tau_0]$ starting at temperature τ_0. The temperature range is calculated as a function

$$\tau_0 = f_0(S_0, \alpha)$$
$$\tau_\star = f_\star(S_0, \alpha)$$

of the initial solution S_0 and the objective function coefficients a. This provides the ability to adjust the temperature range to the total cost as well as the individual components of the objective function.

Assuming that the number of iterations η available to traverse the calculated temperature range $[\tau_\star, \tau_0]$ is known, then the cooling factor

$$\alpha(\tau_0, \tau_\star, \eta) = \alpha(\tau_{\text{start}}, \tau_{\text{target}}, \eta) = \sqrt[\eta]{\frac{\tau_{\text{target}}}{\tau_{\text{start}}}}$$

of the geometric cooling schedule $\tau_{i+1} = \alpha \cdot \tau_i$ can be calculated. However, in our case only the time limit is known. Therefore we measure the number of iterations per second and estimate the number of remaining iterations $\hat{\eta}$ regularly throughout the search process. Given the current temperature τ and the target temperature $\tau_\star$ we update the cooling factor by evaluating $\alpha(\tau, \tau_\star, \hat{\eta})$. This ensures that the temperature decreases monotonically and reaches a temperature rather close to the target temperature $\tau_\star$ when the time limit is reached.

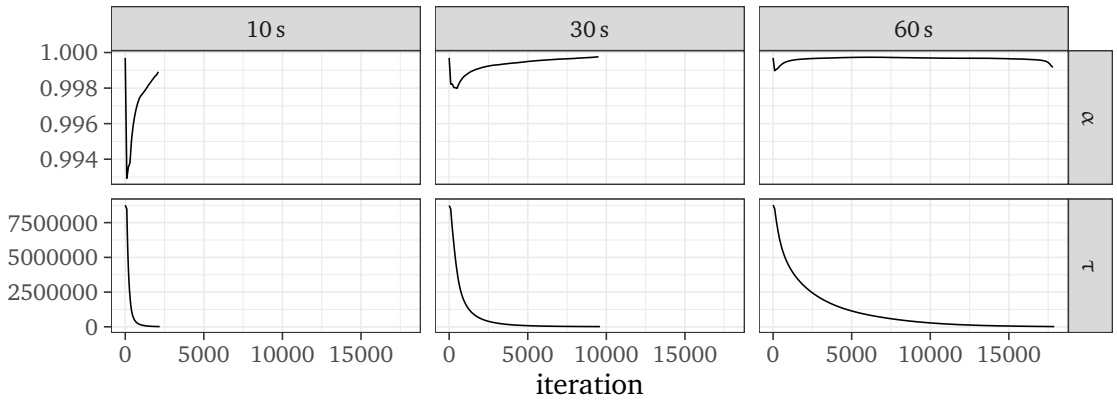


Figure 5.12: Cooling factor α and temperature τ for different time limits on the same instance.

Example 5.8. *The trajectories of the cooling factor and the temperature for three different time limits are shown in Figure 5.12. The range of temperatures traversed in the given time is the same for all time limits. The cooling factor α is dynamically adjusted to spread the traversal of the temperature range along the given period. As the cooling factor is initially set to a rather high value, it drops in the first few seconds of the runs. After that it increases again, as the number of iterations per second increases, but may also decrease as observed in the 60 s run towards the end.*

The cooling factor may increase or decrease during the search process, depending on the currently measured number of iterations per second that may vary due to the following influencing factors. At the beginning of the search process, the subproblem and mode decisions are balanced. After some adaptions, the simple or alternative procedures may have larger weights thereby increasing or decreasing the number of iterations per second as the alternative procedures usually take more computation time compared to the simple procedures. This effect may be reversed when the decisions balance again due to the aging function. Another aspect is the size and quality of the solution. As the search progresses, the number of routes will often decrease, increasing the number of iterations per second, as the LNS, VND, scheduling and associated bookkeeping procedures take less time. A third, rather distant aspect is the load of the computing environment, which may vary in multi-user settings.

5.7 Computational results

In this section, the proposed solution method and its components are evaluated. First the computational environment, the instances, the solver parameters and time limits are discussed. We then report results obtained in an experiment resembling the *restricted resources challenge*, i.e., the solution method is restricted to a single processor core and time limits derived from the sizes of the respective instances. Afterwards we evaluate individual components of the solution method. In the following Section 5.8 the solution method is compared to the solution methods of the other finalists of the restricted resources challenge.

The solution method was implemented in C++ and compiled with *GCC* version 8.1.0. Some effort has been put into enhancing the performance of the implementation, i.e., the method was coded such that (i) the number of allocations is rather small by reuse of buffers and memory objects, (ii) the total amount of memory required remains small even for larger instances and (iii) the majority of redundant calculations is cached, e.g., for unmodified routes. Unless otherwise noted, all experiments were performed on an Ubuntu Linux 18.04 Desktop with an Intel i7-3770 3.4GHz CPU and 16GB RAM.

The experiments were performed on the instance sets `early`, `late` and `hidden` provided by the challenge organizers [54] and can be obtained on the challenge's website[1]. The instance sets are summarized in Table 5.1 and contain instances ranging from 75 up to 1200 requests. The `early` instances were used in the *all time best challenge* while the `hidden` instances were used in the final evaluation of the *restricted resources challenge*. As this work mainly targets the restricted resources setting with a limited time budget, we calculate the time budget in seconds

$$\mathrm{b} = \lceil \beta \cdot (|\mathrm{R}| + 10) \rceil$$

in the same way as described for the challenge, as a function of the number of requests $|\mathrm{R}|$ and a *time limiting factor* β that is derived by an external benchmarking tool and captures the performance of a specific machine. For the aforementioned machine the factor $\beta \approx 0.58$ was established, such that $\beta = 0.5$ provides a reasonable lower bound for the time available, which will be used to evaluate the method.

Table 5.1: Instance sets

	H		$\|\mathrm{I}\|$		$\|V\|$		$\|W\|$		$\|\mathrm{R}\|$		
name	min.	max.	min.	max.	min.	max.	min.	max.	min.	max.	#inst.
`early`	15	55	3	7	54	317	25	125	150	900	25
`hidden`	15	55	3	7	54	317	25	125	150	900	25
`late`	15	55	3	7	30	418	25	125	75	1200	25

The solution method has a comparatively large number of parameters due to the set of procedures, operators, heuristics and the adaptive layer. A systematic investigation of the parameter space with respect to the performance of the method would require a large computational budget as (i) the number of replications needs to be sufficiently large to ensure appropriate statistical power and (ii) the time limit should be comparable to that of the challenge. For these reasons we forgo a systematic analysis and use the set of parameters that has been established over the course of the method's development and intermediate tests, which are shown in Table 5.2.

[1]VSC2019 website – instances: https://verolog2019.ortec.com/instances (last accessed: 2021-03-01)

Table 5.2: Solver parameters

parameter	value	description
$\gamma_{\mathcal{W},\text{relocate}}$	2.50	determinism parameter $\text{VND}_{\mathcal{W}}$ relocate
$\gamma_{\mathcal{W},\text{exchange}}$	3.50	determinism parameter $\text{VND}_{\mathcal{W}}$ exchange
$\gamma_{\mathcal{K},\text{relocate}}$	2.50	determinism parameter $\text{VND}_{\mathcal{K}}$ relocate
$\gamma_{\mathcal{K},\text{exchange}}$	3.50	determinism parameter $\text{VND}_{\mathcal{K}}$ exchange
$\gamma_{\mathcal{K},\text{sequential}}$	3.00	determinism parameter for truck sequential insertion
$\gamma_{\mathcal{K},\text{parallel}}$	2.50	determinism parameter for truck parallel insertion
$\gamma_{\mathcal{W},\text{parallel}}$	3.00	determinism parameter for technician parallel insertion
$\theta_{\text{subproblem}}$	0.01	aging factor for the subproblem decision
θ_d	0.50	aging factor for all other decisions
q	$[0.05\|R\|, 0.2\|R\|]$	interval for the number of requests to be removed in LNS destroy

$$f_0(S_0, a) = \frac{-\max\left\{2a_2, \ldots, 2a_7, \frac{1}{50}Z_a(S_0)\right\}}{\ln(0.5)} \tag{5.16}$$

$$f_\star(S_0, a) = \frac{-\max\left\{a_2, \ldots, a_7\right\}}{\ln(10^{-3})} \tag{5.17}$$

Equations (5.16) and (5.17) describe the functions used to derive the temperature range $[\tau_\star, \tau_0]$. These functions were derived during the development of the method w.r.t. the benchmark instances and are considered parameters. Function $f_0(S_0, a)$ incorporates both, the objective function value of the initial solution and the individual objective function coefficients in case that these dominate the total cost. Function $f_\star(S_0, a)$ does only incorporate the objective function coefficients and leads to a temperature accepting an increase in the largest objective function coefficient, e.g., one additional truck or truck tour, by a single unit with a probability of 10^{-3}.

5.7.1 Challenge experiment

In this experiment, we evaluate the solution method on all instances with a time limiting factor $\beta = 0.5$ similar to that in the restricted resources challenge and report results for all instances individually in Table 5.3. For each instance, 10 replications were performed. Columns *gap to BKS* report the minimum, average and maximum gap to the *best known solutions* (BKSs) for each instance. Column *best* corresponds to the best objective value over all 10 runs and column *BKS* reports the BKS value known to the author. The column *src* is marked with $*$ if the value reported in column *BKS* was obtained with our solution method at some point during the development or intermediate tests. Otherwise it was obtained by another challenge participant and shared with the author either in personal correspondence or through the challenge's website[2]. The last two columns *time* and $|R|$ report the the time limit and the number of requests.

[2]VSC2019 website – solutions: https://verolog2019.ortec.com/ (requires login, last accessed: 2021-03-01)

Table 5.3: Results for all instances with $\beta = 0.5$

	gap to BKS [%]							
instance	min.	avg.	max.	best	BKS	src	time b [s]	|R|
early01	0.00	0.18	0.59	3488033660	3487969810		80	150
early02	0.23	0.38	0.55	11174229530	11149038115		155	300
early03	0.33	0.46	0.62	180296655	179700885		230	450
early04	0.82	1.01	1.30	286542955	284205965		305	600
early05	1.21	1.65	2.15	2250780840	2223814105		380	750
early06	0.67	0.92	1.29	24323788785	24160989040		455	900
early07	0.02	0.23	0.36	45826660	45815700		80	150
early08	0.14	0.38	0.50	109951705	109798470		155	300
early09	1.81	2.29	3.16	18403475	18075485		230	450
early10	0.15	0.27	0.46	18527621330	18500638020		305	600
early11	2.38	3.08	3.79	29228535	28549460		380	750
early12	0.66	0.84	1.13	24090764105	23933097895		455	900
early13	0.02	0.16	0.30	582851885	582708670		80	150
early14	0.55	0.74	1.15	95303690	94780375		155	300
early15	0.04	0.09	0.15	1773559940	1772831110		230	450
early16	1.00	1.48	2.09	3320127835	3287392325		305	600
early17	0.14	0.24	0.35	3022396235	3018108020		380	750
early18	1.24	1.41	1.70	5193154610	5129752375		455	900
early19	1.30	2.22	3.17	9410600	9290203		80	150
early20	1.64	2.78	4.30	4841355	4763065	*	155	300
early21	0.82	1.67	2.50	1303452430	1292914150		230	450
early22	0.81	1.71	3.05	205125754	203485635		305	600
early23	1.95	2.91	4.01	56282850	55207660		380	750
early24	3.06	3.93	4.70	17869125	17337730		455	900
early25	0.57	0.77	0.94	67147715	66769325		80	150
hidden01	0.03	0.09	0.35	67316010	67292650	*	80	150
hidden02	0.88	2.41	3.71	872416710	864791960	*	155	300
hidden03	0.18	0.39	1.02	1353930150	1351465995	*	230	450
hidden04	1.76	3.28	4.17	5396485	5303105	*	305	600
hidden05	0.36	1.57	2.46	2402024544	2393395485	*	380	750
hidden06	0.40	0.71	0.93	33045125	32914360		455	900
hidden07	0.04	0.30	1.19	102088705	102045725	*	80	150
hidden08	0.06	0.17	0.32	728819050	728364305	*	155	300
hidden09	0.71	1.35	2.17	1697110962	1685175791	*	230	450
hidden10	0.49	1.06	1.71	31344765	31190970	*	305	600
hidden11	0.67	1.72	2.83	4059070475	4031918125	*	380	750
hidden12	0.16	0.26	0.38	2986458380	2981590210	*	455	900
hidden13	0.00	0.02	0.03	5238030267	5237773357	*	80	150
hidden14	0.14	0.21	0.32	1379477545	1377525235	*	155	300
hidden15	0.00	0.19	0.32	163481575	163478435	*	230	450
hidden16	0.94	1.58	2.03	53197430	52700705		305	600
hidden17	0.02	0.09	0.18	27293217890	27286839381	*	380	750
hidden18	0.62	0.85	1.20	52905170	52580595		455	900
hidden19	0.00	0.01	0.03	4379039855	4379008805	*	80	150
hidden20	0.34	1.27	2.22	125693035	125271615	*	155	300

	gap to BKS [%]							
instance	min.	avg.	max.	best	BKS	src	time b [s]	|R|
hidden21	0.62	1.04	1.25	33114585	32910875	*	230	450
hidden22	0.70	1.38	2.44	6689360	6642535		305	600
hidden23	0.14	0.37	0.65	22304557150	22272834550	*	380	750
hidden24	0.06	0.20	0.33	31330274560	31312199595	*	455	900
hidden25	0.00	0.01	0.01	549508710	549488300	*	80	150
late01	0.00	0.00	0.00	3843636295	3843609240	*	43	75
late02	0.70	1.51	4.42	897763770	891561525	*	155	300
late03	0.02	0.05	0.15	15535363935	15532540150	*	268	525
late04	0.20	0.29	0.36	2705485857	2700203485	*	380	750
late05	0.62	1.83	3.27	37427180	37196690	*	493	975
late06	0.05	0.12	0.26	5194729860	5192224880	*	605	1200
late07	0.00	0.00	0.00	267631560	267631530	*	43	75
late08	0.10	0.28	0.44	7118442740	7111546935	*	155	300
late09	0.06	1.34	2.07	1551475355	1550595930	*	268	525
late10	0.10	0.19	0.33	3073554830	3070477570	*	380	750
late11	0.19	0.27	0.37	33951659670	33888834870	*	493	975
late12	0.18	0.59	1.00	887941400	886355845	*	605	1200
late13	0.01	0.13	0.39	3463250	3462820	*	43	75
late14	0.26	0.44	0.58	96614850	96366140	*	155	300
late15	1.04	1.30	1.83	1556467308	1540403270	*	268	525
late16	0.12	0.52	0.90	21123618456	21099295832	*	380	750
late17	0.27	0.46	0.68	2957585830	2949662170	*	493	975
late18	1.60	1.89	2.37	37491730	36901090		605	1200
late19	0.01	0.03	0.04	29550000	29545765	*	43	75
late20	0.19	0.67	1.22	1990577285	1986794050	*	155	300
late21	0.81	1.91	3.00	148396440	147204150	*	268	525
late22	1.73	3.30	4.67	12762200	12545705	*	380	750
late23	1.31	1.88	2.60	262347197	258961642	*	493	975
late24	0.19	0.47	1.00	32635102890	32572300070	*	605	1200
late25	0.00	0.00	0.01	3072164955	3072105195	*	43	75

The maximum average gap of 3.3% is obtained for instance `late22` and the worst gap of 4.7% over all instances and runs is obtained for instance `early24`, i.e., all individual runs stay below 5% gap. The solution method performs well on all considered instances without peculiar outliers. This indicates that the adaptive layer and general arrangement of the procedures and heuristics are effective.

Regarding the sources of BKS values, we note that the `early` instances have been available throughout the *all time best challenge* and the development of the solution methods by the challenge participants. It is therefore likely that the BKS values for these instances are better than for the other instance sets. Of all the BKS values for the `early` instances, only a single value was obtained by our proposed solution method. The reasons are not completely clear, but likely options are (i) that we did not run our method for enough time or (ii) that our method, while obtaining good solutions fast, fails to obtain solutions of high quality. We expect that most of the BKS values reported for instance sets `late` and `hidden` will be improved in the future.

5.7.2 Influence of time limits

The time budget b is implicitly considered in the adaptive layer by estimating the number of remaining iterations from the current number of iterations per second and the remaining number of seconds. Therefore we want to study the solution method's performance under varying time budgets and investigate whether the proposed adaptive mechanism yields good results if the time budget is decreased. We consider time limiting factors $\beta \in \{1.0, 0.5, 0.25, 0.125\}$, i.e., we iteratively halve the time limit. We run 10 replications for each instance and time limit. Table 5.4 shows the obtained results. The columns under (a) report the average minimum gap, the columns (b) report the average mean gap and columns (c) report the average maximum gap obtained per instance and time limiting factors. The last row reports the worst gaps over all individual runs and instances.

Table 5.4: Results for different time limiting factors $\beta \in \{0.125, 0.25, 0.5, 1.0\}$

set	(a) avg. min gap [%]				(b) avg. mean gap [%]				(c) avg. max gap [%]			
	0.125	0.25	0.5	1	0.125	0.25	0.5	1	0.125	0.25	0.5	1
`early`	1.21	0.95	0.86	0.74	1.85	1.49	1.27	1.10	2.81	2.09	1.77	1.53
`hidden`	0.69	0.44	0.28	0.22	1.27	0.90	0.73	0.61	1.98	1.53	1.20	1.07
`late`	0.69	0.46	0.37	0.25	1.18	0.91	0.76	0.60	1.85	1.43	1.26	0.98
all	0.86	0.62	0.51	0.40	1.43	1.10	0.92	0.77	2.21	1.68	1.41	1.19
all									9.46	6.18	4.70	4.36

With increasing time limit the average best, mean and worst solution quality improves, as could be expected. However, the average worst gaps obtained over all instances do not increase drastically but stay below 3%. In fact, as shown in the last row, all of the individual 3000 runs resulted in gaps below 10% even for $\beta = 0.125$. We suspect these results due to the adaptive layer's time limit and runtime performance dependent cooling schedule that decreases the time spent with open exploration of the search space as the time limit is decreased.

5.7.3 Influence of embedded heuristics

The solution method is essentially an ensemble of procedures and specialized heuristics. This raises questions regarding the actual contributions of the individual heuristics to the overall performance of the method. In this experiment, we selectively exclude components from the solution method and compare the results. The considered components are (i) the adaption of procedure and mode weights (ADAPT), (ii) the bin packing of subtours into trucks (BPP), (iii) the embedded VND heuristics and (iv) the truck scheduling (TS). Each configuration was run on all 75 instances with 10 replications and a time limiting factor $\beta = 0.5$. For each instance, the gaps of the 10 runs are averaged and denoted by *gap*. The results are shown in Table 5.5. Columns *set* and *configuration* report the instance sets and method's configuration. Columns *gap* report the average and maximum gaps and column *worst run* reports the worst gap over all runs and instances.

The contributions of the bin packing (BPP) and the truck scheduling (TS) are rather small, yet noticeable, as on average, the baseline results are always slightly better. Regarding the observed worst runs, there is not much of a difference either between the baseline and the runs without BPP and TS. The contributions of the VND heuristics are comparatively large, especially when the maximum mean gaps and worst runs are considered, indicating that the VND heuristics are required to achieve a reasonably good performance on a certain subset

Table 5.5: Gaps for different configurations of the solution method with $\beta = 0.5$

		gap [%]		
set	configuration	avg.	max.	worst run [%]
all	*baseline*	0.95	3.82	5.14
	no BPP	1.16	4.58	5.49
	no TS	1.01	4.63	5.76
	no ADAPT	3.94	16.23	17.51
	no VND	2.75	19.55	34.01
	no ADAPT+VND	8.41	45.14	54.51
`early`	*baseline*	1.27	3.82	5.14
	no BPP	1.44	4.58	5.38
	no TS	1.39	4.63	5.76
	no ADAPT	4.63	14.03	15.52
	no VND	3.22	19.55	34.01
	no ADAPT+VND	9.58	45.14	54.51
`hidden`	*baseline*	0.80	2.83	4.04
	no BPP	1.01	4.05	5.44
	no TS	0.83	2.49	4.23
	no ADAPT	3.98	16.23	17.51
	no VND	2.29	7.13	8.33
	no ADAPT+VND	7.18	26.31	30.49
`late`	*baseline*	0.78	3.47	4.98
	no BPP	1.02	3.98	5.49
	no TS	0.81	3.09	4.95
	no ADAPT	3.22	11.08	12.19
	no VND	2.74	12.35	13.38
	no ADAPT+VND	8.46	25.83	28.36

of the instances. The worst average gaps are obtained if the adaptive behavior regarding the procedures and mode decisions is excluded, yet the worst averages are often smaller than the worst averages of the runs with VND excluded. This indicates that the adaptive behavior influences the performance on a larger number of instances, while the actual contribution of the VND depends on instance specifics. The frequencies of gaps in Figure 5.13 support this interpretation. With VND excluded, most of the gaps to *baseline* results are in the range from 0% to 3% with most observations around 0.5% while the gaps with ADAPT excluded are more spread out and shifted towards larger gaps with most observations around 2.5%. An analysis of the instances with the largest gaps obtained with excluded VND indicates that the VND performed well when the total costs are not completely dominated by the truck distance costs and especially when the number of truck tours is important. Excluding ADAPT and VND (ADAPT+VND) the results are drastically worse with individual worst gaps above 50% and the distribution of gaps is shifted even further towards larger gaps with most observations in the range of 1% to 9%.

Note, that for both, VND and ADAPT excluded, there are a few runs with improved results compared to the baseline, i.e., the adaptive behavior and the VND are not always beneficial.

Given these results, the bin-packing and the truck scheduling are not essential components of the solution method. They may still provide small gains justifying their inclusion. On the

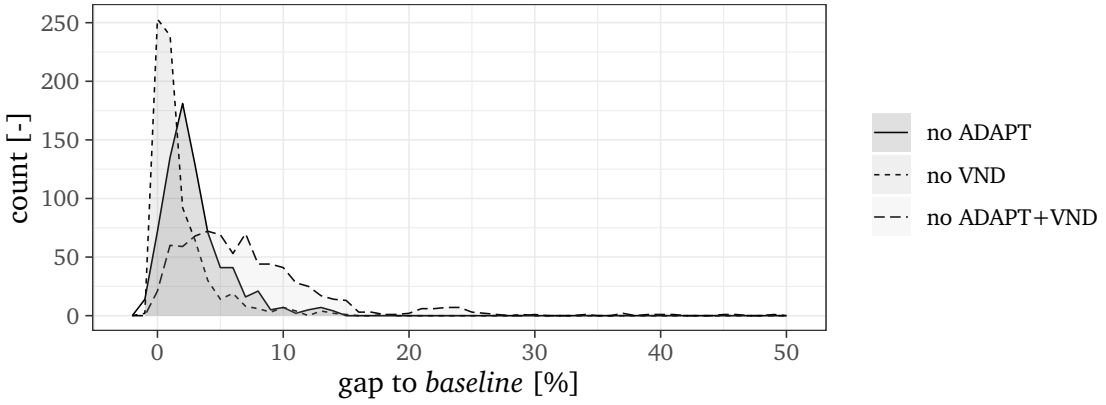


Figure 5.13: Frequencies of gaps to *baseline* results obtained in the runs without adaptive behavior, VND and both, respectively. The bin width is 1%.

other hand, the VND heuristics and the adaptive method are essential components with large contributions. While the VND heuristics provide large contributions for a specific subset of the instances, the adaptive method for procedures and modes contributes to the observed overall performance on a larger set of instances. We conclude that the observed baseline performance is an emergent property of both, the adaptive behavior and the VND.

5.7.4 Scheduling heuristic

The scheduling heuristic applied in the solution method does not fully exploit the opportunities provided by the scheduling constraints. In fact, as was shown, the gap between the heuristic and the optimal solutions may be arbitrarily large. In this experiment, we compare solutions obtained by the heuristic with optimal solutions on scheduling instances that arose during runs of the solution method on the `early` instances. In total, 707 824 unique scheduling instances from the 25 `early` instances were exported and all of these were solved with an IP solver using formulation (5.9)–(5.14) developed in Section 5.5.1.

The heuristic and optimal solutions differed only for 22365 (3.16%) of the scheduling instances. Aggregated over all instances, the average gap is 2.96%, with a maximum gap of 4357%. Aggregated over all instances with differing heuristic and optimal solutions, the average gap is 40.8%. The maximum gap of 4357% is quite large, but refers only to the cost of the scheduling problem without taking other costs into account. If the differences between the heuristic and the optimal scheduling solutions are examined as percentage of the BKS values of the corresponding complete instances, then the average and maximum percentages over all instances are 0.0004% and 0.4%, respectively. The average percentage over all instances with differing heuristic and optimal solutions is 0.01%.

Given these results, we conclude that the scheduling heuristic used in the solution method is sufficient for the considered instances and their objective functions' penalty scaling parameters α_1, as the penalty terms contribute far less than 10% to the overall cost on most instances. For the majority of instances, the heuristic provides optimal results, while rather large gaps are obtained in a few cases. Nevertheless, relative to the total cost of the complete solutions, these differences in the scheduling solutions are comparatively small.

5.8 Comparison with other solution methods

In this section, we review other solution methods for the same problem and compare them with our method. More precisely, we compare our solution method with those proposed by the other seven finalist teams of the restricted resources challenge. Additionally, we perform a direct comparison of our method with the method proposed by team MJG. To the best of our knowledge, there are currently four research articles proposing solution methods for the VSC2019 problem as listed in Table 5.6.

Table 5.6: Research articles

team	members	final rank	article
UOS	Graf, B.	1	Graf [51]
MJG	Geiger, M.J.	2	–
CokaCoders	Jagtenberg, C. et al.	3	Jagtenberg et al. [61]
AAVK	Kastrati, V. et al.	6	Kastrati et al. [64]
–	Kheiri, A. et al.	–	Kheiri et al. [66]

Jagtenberg et al. [61] propose a matheuristic that selects routes using an exact *set partitioning problem* (SPP) formulation while the routes themselves are generated heuristically. They decompose the problem between trucks and technicians and propose a SPP formulation for each. First, a set of candidate routes for the technicians is generated using a *greedy randomized adaptive search procedure* (GRASP) approach. Then the SPP is solved exactly to obtain an optimal selection of technician routes. Until a fixed time limit is reached or no further improvements are recorded, the selected technician routes are passed to a neighborhood operator that generates neighboring routes. These generated routes are added back to the set of possible routes and the SPP is solved again. Subsequently, a similar iterative approach is performed to generate a partial truck solution constrained by the installation days of the partial technician solution generated before.

Kheiri et al. [66] provide a *mixed integer programming* (MIP) formulation modeling the complete problem. Their modeling of the scheduling constraints corresponds to the model (5.9)–(5.14) given in Section 5.5.1. To solve larger problem instances they propose a *hyper-heuristic* (HH) that governs the selection and application of a set of 25 predefined *low level heuristics*. Their description of the low level heuristics indicates that the heuristics are randomized and do not take changes of the objective function into consideration during move evaluation. Instead, after a sequence of low level heuristics applications, the candidate solution is evaluated w.r.t. a current best solution. The complete approach is population based. At the start of each generation, a solution is drawn randomly from the population. Then, the low level heuristics are applied iteratively to the chosen solution and the result is compared with the globally best solution.

Although a research article is not available at the time of writing, a brief description of the method developed by team MJG is provided in the informal article [45]. From presentations given by Martin Josef Geiger of team MJG, we provide a high-level description of their solution method. The method utilizes a generalized neighborhood operating on tours for trucks and technicians. The current solution is improved by a variable neighborhood search like procedure that evaluates swaps between tours, evaluating moves w.r.t. all subsets, e.g., removing two requests from one tour and one request from another tour and swapping these subsets. Additionally, all routes are regularly improved using a 2-opt operator. Candidate solutions are accepted according to a *late acceptance hill-climbing* (LAHC) acceptance criterion, a discrete variant of SA. In contrast to our solution method, the solution method of team

MJG has a small number of parameters, essentially only the LAHC list length parameter.

Kastrati et al. [64] describe an *iterated local search* (ILS) approach based on seven problem specific neighborhoods and additional destroy and repair operators to facilitate appropriate shaking. Their neighborhoods operate on various levels of the solution representation, e.g., swapping or moving truck subtours between trucks or days and similar methods for technicians. In contrast to our method, some of their neighborhoods operate on complete requests, i.e., they operate on the delivery and the installation of a single request simultaneously. To generate initial solutions, they employ a greedy best-fit heuristic with integrated backtracking to handle cases in which the greedy heuristic alone would not provide a feasible solution. They report that this approach provided feasible initial solutions for all considered instance sets. Nevertheless the construction heuristic does not guarantee feasible solutions.

5.8.1 Challenge results

The results were obtained by the challenge organizers on a machine whose hard- and software specifications are unknown to us. Hence, the time limiting factor β remains unknown as well. The challenge organizers disclosed the raw results on the challenge's website[3].

The ranking procedure used by the challenge organizers (cf. [54]) is as follows: each participating team's solver is run on the 25 instances from the `hidden` set with nine runs per instance. For each team and instance, the best and worst runs are removed and the average cost for each team and instance is calculated over the remaining seven runs. Per instance, all teams are ranked $1, \ldots, 8$ w.r.t. increasing average costs, i.e., the best team on a specific instance is assigned rank 1, while the worst is assigned rank 8. Finally, the teams are ordered by their average rank, the team with the smallest average rank takes first place.

Based on the raw data provided by the challenge organizers, we applied the described ranking procedure. The results are shown in Table 5.7. Columns *avg. gap to BKS* give the minimum, average and maximum gaps to the BKSs over all 25 instances. Columns *ranks* give the minimum, average and maximum ranks for the individual instances, respectively. The average rank column implies the final ranking of the *restricted resources challenge* as reported in column *final rank*. Column *#best* gives the number of instances on which the team derived the best solution among all participating teams.

Table 5.7: Summarized results of the *restricted resources challenge*

	avg. gap to BKS [%]			ranks				
team	min.	avg.	max.	min.	avg.	max.	final rank	#best
AAVK	1.60	26.69	106.16	4	5.84	8	6	0
CokaCoders	0.38	18.79	149.22	2	4.60	8	3	0
justFall	1.63	27.07	101.66	4	6.24	8	8	0
MJG	0.11	1.45	3.90	1	2.04	4	2	5
orlab	0.38	18.91	53.32	2	4.80	8	4	0
TCSExplorer	2.08	8.70	21.11	3	5.28	8	5	0
UOS	0.01	0.88	3.13	1	1.20	2	1	20
wanderer	1.75	20.64	59.32	5	6.00	8	7	0

The average ranks show that team UOS (the author), team MJG and team CokaCoders achieved the 1st, 2nd and 3rd place with average ranks of 1.2, 2.04 and 4.6, respectively. The worst rank on any instance of our solution method is 2, and rank 4 for team MJG. All other

[3]VSC2019 website – solutions: https://verolog2019.ortec.com/downloads (last accessed: 2021-03-01)

teams ranked last on at least one instance. Correspondingly, our method achieved the best solution among all participants in 20 instances and was outperformed only by team MJG on five instances. Kheiri et al. [66] performed experiments under the rules of the *restricted resources challenge* themselves and compared their results to those published by the challenge organizers. According to their calculations, their HH approach would have ranked 6th in the final ranking between teams TCSExplorer and AAVK.

Considering the average gaps to the BKSs, the worst average gap of our solution method is 3.13%. In contrast the worst average gap for team CokaCoders is slightly above 149%. Nevertheless, all teams achieved an average gap of below 3% on at least one instance. Over all instances, the average gaps of teams UOS and MJG are relatively small and below 2%. The next best average gap is that of team TCSExplorer with 8.7%. All other teams are well above 15%.

We observe that all approaches worked well on some instances, most of the approaches seem to be rather focused on specific instance structures as can be seen by their large range of achieved average gaps, i.e., these approaches are really good on some instances, but not as good on other instances by a large margin.

Table 5.8: Results of the *restricted resources challenge* for individual instances

	avg. gap to BKS [%]							
instance	AAVK	CokaCoders	justFall	MJG	orlab	TCSExplorer	UOS	wanderer
`hidden01`	1.68	1.05	2.85	0.69	1.83	2.08	0.07	2.55
`hidden02`	89.21	14.77	49.48	2.43	50.22	11.77	2.37	45.08
`hidden03`	3.21	1.02	4.13	0.95	1.60	5.76	0.35	2.95
`hidden04`	15.47	149.22	67.13	1.51	21.01	12.79	3.13	34.30
`hidden05`	106.16	18.09	79.13	3.22	49.73	21.11	1.92	44.71
`hidden06`	2.22	6.13	3.68	0.11	1.21	2.68	0.60	2.69
`hidden07`	35.79	8.09	30.12	0.94	30.16	8.23	0.23	39.82
`hidden08`	3.02	1.15	3.92	0.62	1.20	8.23	0.15	3.93
`hidden09`	82.69	15.37	56.51	3.60	53.32	20.97	1.67	59.32
`hidden10`	57.47	41.60	61.55	3.90	38.17	16.13	1.32	54.52
`hidden11`	28.43	38.19	27.11	2.92	51.87	13.54	2.12	33.75
`hidden12`	3.89	2.48	3.55	0.31	1.72	6.86	0.30	2.67
`hidden13`	2.51	0.49	1.63	0.62	1.17	3.56	0.03	1.75
`hidden14`	1.90	0.66	2.66	0.90	1.01	4.84	0.22	1.97
`hidden15`	2.25	7.99	6.78	1.40	1.26	3.28	0.25	2.34
`hidden16`	20.99	30.65	18.66	0.28	35.61	12.43	1.59	27.23
`hidden17`	2.43	2.14	1.95	0.50	0.88	3.46	0.13	2.35
`hidden18`	5.28	27.13	11.41	0.14	8.02	4.86	0.84	5.97
`hidden19`	1.60	0.49	1.66	0.12	0.38	2.36	0.01	2.11
`hidden20`	83.92	16.70	69.90	3.68	33.77	12.94	1.50	53.96
`hidden21`	18.01	37.45	25.43	2.95	43.60	9.58	0.92	31.20
`hidden22`	13.95	38.06	40.23	0.51	18.85	5.69	1.63	20.89
`hidden23`	3.37	0.45	3.04	0.97	1.66	6.15	0.42	3.60
`hidden24`	2.87	0.38	2.53	0.83	0.79	5.18	0.20	2.02
`hidden25`	78.97	10.04	101.66	2.04	23.68	13.07	0.01	34.33

Table 5.8 reports the average gaps for each team and instance individually. We observe that for some instances all teams achieved similar gaps, e.g., instances `hidden01` and `hidden19` with all average gaps below 3%. In contrast, instances `hidden04` and `hidden05` gave average gaps ranging from 1.51% to 149.22% and 1.92% to 106.16%, respectively. Overall, the results

indicate that most solution approaches had difficulties on the same subset of instances, yet, the approaches of team UOS and team MJG provide good results more consistently.

5.8.2 Comparison with team MJG

In the following, we compare the solution methods of teams UOS and MJG, ranked 1st and 2nd in the challenge, in more detail and evaluate the influence of the time limit on the ranking over all instances. Due to software restrictions the experiments were performed on a comparable machine running 64bit Windows 10 on an Intel i7-6700 3.4GHz CPU and 16GB RAM. Our solution method was compiled with MSVC++ version 14.16.

Both solvers were run on the same machine on all instances with nine replications and the mean solution costs over the replications for each instance were calculated. We then compare the mean costs for each instance and all combinations of time limits and calculate the percentage of instances that our solver UOS achieved smaller average costs than solver MJG.

Table 5.9: Percentage of instances where solver UOS obtained smaller average costs than solver MJG for time limiting factors $\beta \in \{0.0625, 0.125, 0.25, 0.5\}$.

(a) all

UOS \ MJG	0.0625	0.125	0.25	0.5
0.0625	84.00	70.67	60.00	53.33
0.125	90.67	84.00	74.67	69.33
0.25	96.00	89.33	82.67	77.33
0.5	100.00	94.67	89.33	84.00

(b) $|\mathsf{R}| \geqslant 450$

UOS \ MJG	0.0625	0.125	0.25	0.5
0.0625	79.17	58.33	41.67	35.42
0.125	87.50	77.08	62.50	58.33
0.25	93.75	83.33	75.00	66.67
0.5	100.00	91.67	83.33	75.00

The results in Table 5.9a show that our solver achieved better results on the majority of instances across all combinations of time limits. Considering both solvers with equal time limits, the ratio varies slightly but is more or less the same. If only larger instances with $|\mathsf{R}| \geqslant 450$ are considered, then our solver performs worse but still achieves better results on the majority of instances for most combinations as shown in Table 5.9b. Only for the smallest time limit does it fall behind solver MJG. We conclude that our solver is comparatively fast and adapts well to the given time limit, especially on smaller instances.

5.9 Conclusions

In this chapter we proposed a combination of ALNS and VND to solve a complex combined multi-period, multi-depot and multi-capability vehicle routing and scheduling problem. A decomposition approach is applied to the problem, resulting in two main subproblems corresponding to the routing of trucks and the routing of technicians, respectively. Specialized heuristics are then in turn applied to improve the subproblems. We explicitly consider tight restrictions on the computational budget and derive an extension of the adaptive method used in the ALNS framework. This adaptive layer adjusts the solution method to the characteristics of the instances, the time limit and the computational environment.

The results show that the integration of the simple heuristic frameworks provides good results in a small amount of time. The proposed multi-level adaptive mechanism is promising regarding the subproblem selection and budgeting of the available computation time, especially regarding the heterogeneously parameterized objective functions. That means, the

adaptive layer tends to allocate more time to the improvement of the subproblem that dominates the objective function or provides more possibilities for improvement, respectively.

The solution method performs well on all considered instances, with worst individual gaps below 5% for sensible time limits and still below 10% if only a quarter of that time budget is used. The comparison with solvers of other participating teams in the challenge showed that our proposed solution method is able to find comparatively good solutions over the complete set of instances while some other approaches performed well on only a subset of the instances. A comparison with varying time limits furthermore showed that our method is comparatively efficient and obtains good results even for small time limits.

In conclusion, the provided solution method realizes a best effort approach regarding solution quality and a given time budget and finds good solutions fast. It may therefore be used as a seeding heuristic providing initial solutions to better, but slower heuristics.

Chapter 6

Conclusions

In the following we provide a summary of this thesis, including the tackled problems and the obtained results. Furthermore, we close with a discussion of future research directions in the context of large-scale *vehicle routing problems* (VRPs).

In Chapter 1 we highlighted the importance of research on large-scale VRPs and accompanying problems motivated by the challenges arising in the logistics sector. The logistics sector needs to adjust to increasing demands, larger operational areas, increased customer expectations and ever-changing legislative requirements. In that regard, optimization techniques help to improve their processes, from strategical decision making to day-to-day operations.

In Chapter 2 we provided background information on classic VRPs, local search, large neighborhoods and metaheuristics. Additionally we discussed experimental heuristics research and aspects to keep in mind when designing, performing and evaluating computational experiments.

In Chapter 3 we extended research on the exponential *multi-insertion neighborhood* (MIN) for the *vehicle routing problem with unit demands* (VRPU). Besides theoretical aspects of the neighborhood like its connectivity, diameter and the quality of local optima we have shown that finding an optimal set of mobile nodes is $\mathcal{NP}$-hard. To make use of the MIN in a heuristic solution method we proposed various heuristic node selection procedures and compared the resulting solution approach with two approaches from the literature in a computational study. The results indicate that the MIN based heuristic solution method is effective and efficient, especially when augmented with a *variable neighborhood descent* (VND) over smaller neighborhoods. On large-scale instances or with tight time limits, the MIN is a viable option to efficiently reach promising regions of the solution space starting from arbitrary initial solutions.

In Chapter 4 we continued research on tree-based construction heuristics for the *preemptive stacker crane problem* (PSCP). We merged results from the literature on the *preemptive swapping problem* (PSP) and the PSCP to derive theoretical bounds on the benefits of preemption and the benefits of additional nodes that are used for preemption only. In addition, we proposed an adaptation of the modified Karp-Steele patching algorithm for the PSCP and conducted extensive computational experiments showing that our method outperforms the state-of-the-art in both, quality and efficiency. We identified two polynomial-time solvable subproblems that turn sequences of requests into possibly preemptive solutions containing drop nodes. As such we observe that to solve the PSCP it is sufficient to find a specific permutation of the requests.

In Chapter 5 we considered the problem of the *VeRoLog Solver Challenge 2018–2019* (VSC2019) that requires the routing and scheduling of delivery vehicles and technicians with synchronization between deliveries and subsequent installations. We proposed a method combining *adaptive large neighborhood search* (ALNS), VND and problem specific procedures to minimize the number of trucks and truck tours. Additionally, a subproblem decomposition is performed such that the procedures either target the truck or the technician portions of a solution. The extension of ALNS's adaptive layer to the subproblem selection helps to allocate the available but scarce computational budget in such a way that the method performs

comparatively well across a wide range of different instances. As such, our method was able to obtain the first rank in the restricted resources challenge.

In conclusion, as the computing resources and processor speeds have increased over the past decades, so have the scale and complexity of the problems that arise in real-world settings. To efficiently solve complex and large-scale problems it continues to be necessary to make use of structural properties like unit demands or unit vehicle capacities that result from the specific scenarios under study and to allot the available computing resources in a best possible way. The results from Chapters 3 and 4 highlight how these properties can be used to derive efficient search procedures on the global and local scale. Similarly, the results of Chapter 5 highlight how complex combined problems can be tackled by decomposition and an adaptive allotment of the computing resources.

Future research directions

Effectively and efficiently tackling large-scale problem instances on the global scale requires additional research effort in multiple domains. These include (i) the study of widespread real-world special cases and their structural properties, (ii) the study of subdivided search processes that comprise different methods operating on different levels of granularity at different times of the search process and (iii) the study of problem decomposition in conjunction with parallelization and distributed solution methods.

While the study of generalized problems is intriguing because they allow to model and optimize a wide range of specific problems, their generality decreases the number of structural properties and assumptions that may be used to derive more efficient algorithms. Despite being special cases, certain problems occur comparatively often in practice and are thus worth additional research effort even if more generalized solution methods exist that perform well enough on smaller instances. For example, problems involving containerized road transportation with unit demands (e.g., *swap bodies* (SBs)) and vehicle capacities of one or two occur regularly in real-world settings.

State-of-the-art approaches on up to 500 nodes are not necessarily best suited for 5000 or 10000 nodes because they may not scale appropriately. From an algorithmic point of view further research effort should be spent on identifying more granular search strategies. For example, instead of providing one approach for the complete search process, the process may be divided into segments that employ different algorithms operating on different levels of granularity. While these approaches are mentioned in the literature, their overall coverage is comparatively small.

A large body of research is focused on sequential algorithms that perform well on conventional computer architectures. Nevertheless, modern parallelized and distributed hardware needs to be leveraged in large-scale practical settings. Parallelizing sequential algorithms is usually possible to some degree, however, different approaches should be investigated. These may be less efficient in a sequential setting but may scale better when run on parallelized and distributed hardware.

Bibliography

[1] Adenso-Díaz, B. and Laguna, M. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research* 54.1 (2006), pp. 99–114.

[2] Ahuja, R. K., Ergun, Ö., Orlin, J. B., and Punnen, A. P. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics* 123.1–3 (2002), pp. 75–102.

[3] Angel, E., Bampis, E., and Pascual, F. An exponential (matching based) neighborhood for the vehicle routing problem. *Journal of Combinatorial Optimization* 15.2 (2008), pp. 179–190.

[4] Anily, S., Gendreau, M., and Laporte, G. The preemptive swapping problem on a tree. *Networks* 58.2 (2011), pp. 83–94.

[5] Anily, S., Gendreau, M., and Laporte, G. The swapping problem on a line. *SIAM Journal on Computing* 29.1 (1999), pp. 327–335.

[6] Anily, S. and Hassin, R. The swapping problem. *Networks* 22.4 (1992), pp. 419–433.

[7] Ansótegui, C., Sellmann, M., and Tierney, K. A gender-based genetic algorithm for the automatic configuration of algorithms. *Principles and practice of constraint programming - CP 2009*. Springer Berlin Heidelberg, 2009, pp. 142–157.

[8] Applegate, D., Bixby, R., Chvatal, V., and Cook, W. *Concorde TSP solver*. 2006.

[9] Archetti, C., Jabali, O., and Speranza, M. G. Multi-period vehicle routing problem with due dates. *Computers & Operations Research* 61 (2015), pp. 122–134.

[10] Atallah, M. J. and Kosaraju, S. R. Efficient solutions to some transportation problems with applications to minimizing robot arm travel. *SIAM Journal on Computing* 17.5 (1988), pp. 849–869.

[11] Barr, R. S., Golden, B. L., Kelly, J. P., Resende, M. G. C., and Stewart, W. R. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics* 1.1 (1995), pp. 9–32.

[12] Bellmore, M. and Nemhauser, G. L. The traveling salesman problem: a survey. *Operations Research* 16.3 (1968), pp. 538–558.

[13] Beltrami, E. J. and Bodin, L. D. Networks and vehicle routing for municipal waste collection. *Networks* 4.1 (1974), pp. 65–94.

[14] Berbeglia, G., Cordeau, J.-F., Gribkovskaia, I., and Laporte, G. Static pickup and delivery problems: a classification scheme and survey. *TOP* 15.1 (2007), pp. 1–31.

[15] BIEK Bundesverband Paket & Expresslogistik. *KEP-Studie 2020 – Analyse des Marktes in Deutschland*. `https://www.biek.de/files/biek/downloads/papiere/BIEK_KEP-Studie_2020.pdf` (last accessed: 2021-03-01). 2020.

[16] Bordenave, C., Gendreau, M., and Laporte, G. A branch-and-cut algorithm for the preemptive swapping problem. *Networks* 59.4 (2011), pp. 387–399.

[17] Bordenave, C., Gendreau, M., and Laporte, G. Heuristics for the mixed swapping problem. *Computers & Operations Research* 37.1 (2010), pp. 108–114.

[18] Bourgeois, M., Laporte, G., and Semet, F. Heuristics for the black and white traveling salesman problem. *Computers & Operations Research* 30.1 (2003), pp. 75–85.

[19] Bowen, L. and Lupo, C. The performance cost of software-based security mitigations. *Proceedings of the ACM/SPEC international conference on performance engineering*. ACM, 2020.

[20] Brueggemann, T. and Hurink, J. Two very large-scale neighborhoods for single machine scheduling. *OR Spectrum* 29.3 (2007), pp. 513–533.

[21] Buckow, J.-N., Graf, B., and Knust, S. The exponential multi-insertion neighborhood for the vehicle routing problem with unit demands. *Computers & Operations Research* 120 (2020). DOI: 10.1016/j.cor.2020.104949.

[22] Burkard, R. and Deineko, V. Polynomially solvable cases of the traveling salesman problem and a new exponential neighborhood. *Computing* 54.3 (1995), pp. 191–211.

[23] Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Qu, R. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society* 64.12 (2013), pp. 1695–1724.

[24] Campos, V., Corberan, A., and Mota, E. Polyhedral results for a vehicle routing problem. *European Journal of Operational Research* 52.1 (1991), pp. 75–85.

[25] Chimani, M., Gutwenger, C., Jünger, M., Klau, G. W., Klein, K., and Mutzel, P. The open graph drawing framework (OGDF). *Handbook of graph drawing and visualization*. Ed. by R. Tamassia. CRC Press, 2014. Chap. 17.

[26] Christofides, N. *Worst-case analysis of a new heuristic for the travelling salesman problem*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[27] Christofides, N., Mingozzi, A., and Toth, P. The vehicle routing problem. *Combinatorial optimization*. Ed. by N. Christofides, A. Mingozzi, P. Toth, and C. Sandi. Wiley, 1979. Chap. 11, pp. 315–338.

[28] Clarke, G. and Wright, J. W. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research* 12.4 (1964), pp. 568–581.

[29] Cordeau, J.-F., Laporte, G., Savelsbergh, M. W. P., and Vigo, D. Vehicle routing. *Handbooks in operations research and management science*. Ed. by C. Barnhart and G. Laporte. Vol. 14. Elsevier, 2007. Chap. 6, pp. 367–428.

[30] Coy, S. P., Golden, B. L., Runger, G. C., and Wasil, E. A. Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics* 7.1 (2001), pp. 77–97.

[31] Dantzig, G. B. and Ramser, J. H. The truck dispatching problem. *Management Science* 6.1 (1959), pp. 80–91.

[32] Deineko, V. and Woeginger, G. A study of exponential neighborhoods for the travelling salesman problem and for the quadratic assignment problem. *Mathematical Programming* 87.3 (2000), pp. 519–542.

[33] Dezső, B., Jüttner, A., and Kovács, P. LEMON – an open source C++ graph template library. *Electronic Notes in Theoretical Computer Science* 264.5 (2011), pp. 23–45.

[34] Drake, J. H., Kheiri, A., Özcan, E., and Burke, E. K. Recent advances in selection hyper-heuristics. *European Journal of Operational Research* 285.2 (2020), pp. 405–428.

[35] Duarte, A., Sánchez-Oro, J., Mladenović, N., and Todosijević, R. Variable neighborhood descent. *Handbook of heuristics*. Ed. by R. Martí, P. M. Pardalos, and M. G. C. Resende. Cham: Springer International Publishing, 2018, pp. 341–367.

[36] Erdoğan, G. An open source spreadsheet solver for vehicle routing problems. *Computers & Operations Research* 84 (2017), pp. 62–72.

[37] Ergun, Ö., Orlin, J., and Steele-Feldman, A. Creating very large scale neighborhoods out of smaller ones by compounding moves. *Journal of Heuristics* 12.1-2 (2006), pp. 115–140.

[38] Francis, P. M., Smilowitz, K. R., and Tzur, M. The period vehicle routing problem and its extensions. *The vehicle routing problem: latest advances and new challenges*. Ed. by B. Golden, S. Raghavan, and E. Wasil. Boston, MA: Springer US, 2008, pp. 73–102.

[39] Franzin, A. and Stützle, T. Revisiting simulated annealing: a component-based analysis. *Computers & Operations Research* 104 (2019), pp. 191–206.

[40] Frederickson, G. N. and Guan, D. J. Nonpreemptive ensemble motion planning on a tree. *Journal of Algorithms* 15.1 (1993), pp. 29–60.

[41] Frederickson, G. N. and Guan, D. J. Preemptive ensemble motion planning on a tree. *SIAM Journal on Computing* 21.6 (1992), pp. 1130–1152.

[42] Frederickson, G. N., Hecht, M. S., and Kim, C. E. Approximation algorithms for some routing problems. *SIAM Journal on Computing* 7.2 (1978), pp. 178–193.

[43] Funke, B., Grünert, T., and Irnich, S. Local search for vehicle routing and scheduling problems: review and conceptual integration. *Journal of Heuristics* 11.4 (2005), pp. 267–306.

[44] Garey, M. R., Graham, R. L., and Ullman, J. D. Worst-case analysis of memory allocation algorithms. *Proceedings of the fourth annual ACM symposium on theory of computing - STOC '72*. ACM Press, 1972.

[45] Geiger, M. J. and Graf, B. VeRoLog Solver Challenge – 4. Implementierungswettbewerb der EURO Arbeitsgruppe Vehicle Routing and Logistics Optimization. *OR News* 67 (2019), pp. 57–57.

[46] M. Gendreau and J.-Y. Potvin, eds. *Handbook of Metaheuristics*. Springer US, 2010.

[47] Ghiani, G., Laporte, G., and Semet, F. The black and white traveling salesman problem. *Operations Research* 54.2 (2006), pp. 366–378.

[48] Glover, F., Gutin, G., Yeo, A., and Zverovich, A. Construction heuristics for the asymmetric TSP. *European Journal of Operational Research* 129.3 (2001), pp. 555–568.

[49] Golden, B. L., Wasil, E. A., Kelly, J. P., and Chao, I.-M. The impact of metaheuristics on solving the vehicle routing problem: algorithms, problem sets, and computational results. *Fleet management and logistics*. Ed. by T. G. Crainic and G. Laporte. Springer US, 1998, pp. 33–56.

[50] Golden, B. L., Assad, A. A., Wasil, E. A., and Baker, E. Experimentation in optimization. *European Journal of Operational Research* 27.1 (1986), pp. 1–16.

[51] Graf, B. Adaptive large variable neighborhood search for a multiperiod vehicle and technician routing problem. *Networks* 76.2 (2020), pp. 256–272.

[52] Graf, B. Preemptive stacker crane problem: extending tree-based properties and construction heuristics. *European Journal of Operational Research* 292.2 (2021), pp. 532–547.

[53] Greenberg, H. J. Computational testing: why, how and how much. *ORSA Journal on Computing* 2.1 (1990), pp. 94–97.

[54] Gromicho, J., 't Hof, P. van, and Vigo, D. The VeRoLog Solver Challenge 2019. *Journal on Vehicle Routing Algorithms* 2.1 (2019), pp. 109–111.

[55] Gutin, G. Exponential neighbourhood local search for the traveling salesman problem. *Computers & Operations Research* 26.4 (1999), pp. 313–320.

[56] Gutin, G. and Yeo, A. Small diameter neighbourhood graphs for the traveling salesman problem: at most four moves from tour to tour. *Computers & Operations Research* 26.4 (1999), pp. 321–327.

[57] Hooker, J. N. Needed: an empirical science of algorithms. *Operations Research* 42.2 (1994), pp. 201–212.

[58] Hooker, J. N. Testing heuristics: we have it all wrong. *Journal of Heuristics* 1.1 (1995), pp. 33–42.

[59] Hurink, J. An exponential neighborhood for a one-machine batching problem. *OR Spectrum* 21.4 (1999), pp. 461–476.

[60] Hutter, F., Hoos, H. H., and Stützle, T. Automatic algorithm configuration based on local search. *Proceedings of the 22nd national conference on artificial intelligence*. Vol. 2. AAAI'07. Vancouver, British Columbia, Canada: AAAI Press, 2007, pp. 1152–1157.

[61] Jagtenberg, C. J., Maclaren, O. J., Mason, A. J., Raith, A., Shen, K., and Sundvick, M. Columnwise neighborhood search: a novel set partitioning matheuristic and its application to the VeRoLog solver challenge 2019. *Networks* 76.2 (2020), pp. 273–293.

[62] Kaligosi, K. and Sanders, P. How branch mispredictions affect quicksort. *Lecture notes in computer science*. Springer Berlin Heidelberg, 2006, pp. 780–791.

[63] Karp, R. M. Reducibility among combinatorial problems. *Complexity of computer computations*. Springer US, 1972, pp. 85–103.

[64] Kastrati, V., Ahmeti, A., and Musliu, N. Solving vehicle routing and scheduling with delivery and installation of machines using ILS. *Proceedings of the 13th international conference on the practice and theory of automated timetabling*. Vol. I. 2020, pp. 207–223. ISBN: 978-0-9929984-3-1.

[65] Kerivin, H. L. M., Lacroix, M., and Mahjoub, A. R. Models for the single-vehicle preemptive pickup and delivery problem. *Journal of Combinatorial Optimization* 23.2 (2010), pp. 196–223.

[66] Kheiri, A., Ahmed, L., Boyacı, B., Gromicho, J., Mumford, C., Özcan, E., and Dirikoç, A. S. Exact and hyper-heuristic solutions for the distribution-installation problem from the VeRoLog 2019 challenge. *Networks* 76.2 (2020), pp. 294–319.

[67] Kheiri, A., Dragomir, A. G., Mueller, D., Gromicho, J., Jagtenberg, C., and Hoorn, J. J. van. Tackling a VRP challenge to redistribute scarce equipment within time windows using metaheuristic algorithms. *EURO Journal on Transportation and Logistics* 8.5 (2019), pp. 561–595.

[68] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science* 220.4598 (1983), pp. 671–680.

[69] Kovacs, A. A., Parragh, S. N., Doerner, K. F., and Hartl, R. F. Adaptive large neighborhood search for service technician routing and scheduling problems. *Journal of Scheduling* 15.5 (2012), pp. 579–600.

[70] Kovács, P. Minimum-cost flow algorithms: an experimental evaluation. *Optimization Methods and Software* 30.1 (2014), pp. 94–127.

[71] Kudva, G., Morin, T. L., and Pekny, J. F. A branch-and-cut algorithm for vehicle routing problems. *Annals of Operations Research* 50.1 (1994), pp. 37–59.

[72] Laporte, G. The vehicle routing problem: an overview of exact and approximate algorithms. *European Journal of Operational Research* 59.3 (1992), pp. 345–358.

[73] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, eds. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, 11, 1985. 484 pp.

[74] Li, H. and Lim, A. A metaheuristic for the pickup and delivery problem with time windows. *International Journal on Artificial Intelligence Tools* 12.02 (2003), pp. 173–186.

[75] Lin, B. W. and Rardin, R. L. Controlled experimental design for statistical comparison of integer programming algorithms. *Management Science* 25.12 (1979), pp. 1258–1271.

[76] Lin, S. Computer solutions of the traveling salesman problem. *Bell System Technical Journal* 44.10 (1965), pp. 2245–2269.

[77] R. Martí, P. Panos, and M. G. C. Resende, eds. *Handbook of Heuristics*. Springer International Publishing, 2016.

[78] Michiels, W., Aarts, E. H. L., and Korst, J. Theory of Local Search. *Handbook of heuristics*. Springer International Publishing, 2017, pp. 1–41.

[79] Miller, C. E., Tucker, A. W., and Zemlin, R. A. Integer programming formulation of traveling salesman problems. *Journal of the ACM* 7.4 (1960), pp. 326–329.

[80] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Producing wrong data without doing anything obviously wrong! *Proceeding of the 14th international conference on architectural support for programming languages and operating systems - ASPLOS '09*. ACM Press, 2009.

[81] Or, I. Travelling Salesman-Type Combinatorial Problems and Their Relation to the Logistics of Blood Banking. PhD thesis. Evanston, Illinois: Department of Industrial Engineering and Management Sciences, Northwestern University, 1976.

[82] Papadimitriou, C. H. The euclidean travelling salesman problem is NP-complete. *Theoretical Computer Science* 4.3 (1977), pp. 237–244.

[83] Parragh, S. N., Doerner, K. F., and Hartl, R. F. A survey on pickup and delivery problems. *Journal für Betriebswirtschaft* 58.2 (2008), pp. 81–117.

[84] Pisinger, D. and Ropke, S. A general heuristic for vehicle routing problems. *Computers & Operations Research* 34.8 (2007), pp. 2403–2435.

[85] Pisinger, D. and Ropke, S. Large neighborhood search. *Handbook of metaheuristics*. Springer US, 2010, pp. 399–419.

[86] Pitney Bowes Inc. *Pitney Bowes Parcel Shipping Index reports continued growth as global parcel volume exceeds 100 billion for first time ever*. Press release. `https://www.pitneybowes.com/au/newsroom/press-releases/pitney-bowes-parcel-shipping-index-reports-continued-growth-as-global-parcel.html` (last accessed: 2021-03-01). 2020.

[87] Punnen, A. The traveling salesman problem: new polynomial approximation algorithms and domination analysis. *Journal of Information and Optimization Sciences* 22.1 (2001), pp. 191–206.

[88] Quilliot, A., Lacroix, M., Toussaint, H., and Kerivin, H. Tree based heuristics for the preemptive asymmetric stacker crane problem. *Electronic Notes in Discrete Mathematics* 36 (2010), pp. 41–48.

[89] Ralphs, T. *Branch and cut for vehicle routing*. `https://www.coin-or.org/SYMPHONY/branchandcut/VRP/data/index.htm#V` (last accessed: 2021-03-01). 2003.

[90] Rardin, R. L. and Uzsoy, R. Experimental evaluation of heuristic optimization algorithms: a tutorial. *Journal of Heuristics* 7.3 (2001), pp. 261–304.

[91] Reinelt, G. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing* 3.4 (1991), pp. 376–384.

[92] Ropke, S. and Pisinger, D. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science* 40.4 (2006), pp. 455–472.

[93] Savelsbergh, M. W. P. and Sol, M. The general pickup and delivery problem. *Transportation Science* 29 (1995), pp. 17–29.

[94] Shaw, P. Using constraint programming and local search methods to solve vehicle routing problems. *Principles and practice of constraint programming — CP98*. Ed. by M. Maher and J.-F. Puget. Vol. 1520. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 417–431.

[95] Siek, J., Lumsdaine, A., and Lee, L.-Q. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[96] Sörensen, K. Metaheuristics – the metaphor exposed. *International Transactions in Operational Research* 22.1 (2013), pp. 3–18.

[97] Talluri, K. T. The four-day aircraft maintenance routing problem. *Transportation Science* 32.1 (1998), pp. 43–53.

[98] Toth, P. and Vigo, D. 1. An Overview of Vehicle Routing Problems. *The vehicle routing problem*. Society for Industrial and Applied Mathematics, 2002, pp. 1–26.

[99] Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., and Subramanian, A. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research* 257.3 (2017), pp. 845–858.

[100] Vidal, T. Node, edge, arc routing and turn penalties: multiple problems–one neighborhood extension. *Operations Research* 65.4 (2017), pp. 992–1010.

Acknowledgements

I thank my supervisor Prof. Dr. Sigrid Knust for her ongoing support, invaluable feedback and the opportunity to be part of her working group. I'm especially grateful for her guidance in navigating the scientific process with its presentations, conferences, reviews, journals, projects and grant proposals. Also for the many times she pointed out courses and events to develop my know-how and soft skills.

I thank Prof. Dr. Stefan Irnich for both agreeing to examine this thesis and for the constructive feedback.

I thank Jan-Niklas Buckow for the numerous discussions and the support throughout the years – I wish him all the best for his further studies and research. I thank my colleagues Sven Boge and Tobias Oelschlägel for their help and conversations on various topics from optimization to matters of day-to-day life. I enjoyed working with you!

I thank Friedhelm Hofmeyer for technical support and administrating the computer infrastructure and version control systems. I especially thank Friedhelm for the many times that we discussed movies, games and Laugengebäck. I thank my colleagues of the theoretical computer science group for the occasional chats, the snacks and their sofa.

For proofreading drafts I thank Jan-Niklas Buckow, Sven Boge, Mathias Menninghaus, Svantje Jung and Rainer Graf.

I thank my peers from the scientific community who I have met along the way on conferences and other events for the interesting talks and discussions. I thank the organizers and participants of the VeRoLog Solver Challenge 2019, especially Martin Josef Geiger, for the exciting challenge. I thank all the anonymous referees that provided helpful feedback for the three research articles that have been published during the preparation of this thesis. I thank the large number of open source contributors who have created all the programs, libraries, compilers and editors I use for research, programming and writing.