Applying Ant Colony Optimization to the Periodic Vehicle Routing Problem with Time Windows

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Dietmar Trummer

Matrikelnummer 9325754

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl
Mitwirkung: Dipl.-Ing. Dr. Sandro Pirkwieser

Wien, 24.09.2013

(Unterschrift Verfasser)   (Unterschrift Betreuung)

Applying Ant Colony Optimization to the Periodic Vehicle Routing Problem with Time Windows

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Dietmar Trummer

Registration Number 9325754

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl
Assistance: Dipl.-Ing. Dr. Sandro Pirkwieser

Vienna, 24.09.2013

_____   _____
(Signature of Author)   (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Dietmar Trummer
Neumayrgasse 26/37, 1160 Wien

    Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

_____   _____

(Ort, Datum)   (Unterschrift Verfasser)

Danksagung

Ich bedanke mich bei Professor Günther Raidl, der mir durch seine Betreuung dieser Arbeit den Abschluß meines Studiums ermöglicht hat.

Ganz herzlich möchte ich mich bei Sandro Pirkwieser bedanken, der mir durch seinen Rat und die Begleitung während der Erstellung der Arbeit Entwicklungsrichtungen aufzeigt, inhaltliche Impulse gab, und Diskrepanzen auflösen half.

Diese Arbeit widme ich meiner Mutter Heidi Trummer.

Abstract

The Periodic Vehicle Routing Problem with Time Windows (PVRPTW) is an extended, complex variant of the classical vehicle routing problem. On the one hand it differs from the latter by visiting a subset of the customers several times during a planning horizon spanning several days, where the selection of a visit day combination out of a set of viable ones for each such customer is part of the problem. On the other hand the customers have associated a time window in which the visit is allowed. The objective is to minimize the overall travel costs while respecting all constraints.

In this thesis we investigate the application of variants of the Ant Colony Optimization (ACO) metaheuristic to solve this highly constrained $\mathcal{NP}$-hard problem in combination with other techniques. For this purpose we apply ACO in two different ways: as heuristic solver for the pricing subproblem arising in a column generation approach for the linear programming-relaxed PVRPTW; and as an approximate problem solving method for the whole PVRPTW.

In the first approach we show that ACO can be used to speed up the column generation process. To achieve this ACO is used to solve the Elementary Shortest Path Problem with Resource Constraints (ESPPRC) that forms our pricing subproblem. The investigation results reflect that the application of ACO improves performance and quality of columns compared to an exact ESPPRC solver, although other applied metaheuristics produce the same effect. In fact we deduce that other components of the column generation algorithm, e.g. local search, have more influence on the solving performance than the choice of the metaheuristic.

For the second approach we present a new ACO algorithm: the cascaded ACO. The PVRPTW is decomposed in an upper level and a lower level problem which are both solved with specific ACO variants. The ACO for the upper level problem has to optimize the visit combinations, whereas the lower level ACO solves a Vehicle Routing Problem with Time Windows (VRPTW). Both ACO algorithms are optimized by introducing and combining several techniques from literature to improve performance. Additionally a method is shown that allows us to find semi-optimal settings for the various parameters of the ACO algorithms.

An extensive comparison of our results to results from previously published PVRPTW solution algorithms concludes the approach of using ACO as solver for the whole problem. Although, recently developed hybrid algorithms to solve the PVRPTW show better performance on large problem instances, our cascaded ACO outperforms the sole other ACO algorithm published so far.

Kurzfassung

Das Periodic Vehicle Routing Problem with Time Windows (PVRPTW) ist eine komplexe Erweiterung des klassischen Tourenplanungsproblems. Einerseits müssen hierbei die Kunden an mehreren Tagen innerhalb einer definierten Planungsperiode besucht werden, wobei die Auswahl der Kombination von Besuchstagen zum Problem gehört. Andererseits bestimmt jeder Kunde einen Zeitbereich, in welchem der Besuch stattfinden muss. Das Ziel ist die Minimierung der gesamten Tourenkosten bei Berücksichtigung aller Nebenbedingungen.

In dieser Diplomarbeit untersuchen wir die Anwendung verschiedener Varianten der Ant Colony Optimization (ACO) Metaheuristik, um dieses $\mathcal{NP}$-harte kombinatorische Optimierungsproblem zu lösen. Zu diesem Zweck wenden wir ACO auf zwei verschiedene Arten an: als heuristischen Lösungsalgorithmus für das Pricing Subproblem, welches bei einem Spaltengenerierungs-Ansatz zum Lösen des linearen Programmierungs-relaxierten PVRPTW auftritt; und als näherungsweisen Lösungsalgorithmus für das gesamte PVRPTW.

In der ersten Anwendung zeigen wir, wie durch Einsatz von ACO die Lösungszeit des Column Generation Prozesses verkürzt wird. Dazu wird ACO als Lösungsalgorithmus des Pricing Subproblems verwendet, welches wir als Elementary Shortest Path Problem with Resource Constraints (ESPPRC) identifiziert haben. Die Untersuchungsergebnisse spiegeln wieder, dass die Anwendung von ACO die Lösungsleistung bezüglich Laufzeit und Qualität der generierten Spalten verglichen mit einem exakten Lösungsansatz steigert. Allerdings konnten wir keinen Nachweis erbringen, dass ACO anderen Metaheuristiken hierbei vorzuziehen ist. Vielmehr schließen wir, dass andere algorithmische Komponenten, wie z.B. die lokale Suche, größeren Einfluss auf die Lösungsleistung besitzen, als die Wahl der Metaheuristik.

Für die zweite Anwendung stellen wir einen neuen ACO Algorithmus vor: cascaded ACO. Das PVRPTW wird in ein übergeordnetes "upper level" und ein untergeordnetes "lower level" Problem zerlegt, welche beide mit spezifischen ACO Varianten gelöst werden. ACO für das "upper level" Problem optimiert die Kombination von Besuchstagen, während ACO für das "lower level" Problem die sich ergebenden Vehicle Routing Problems with Time Windows (VRPTW) löst. Beide ACO Algorithmen wurden durch Einführung diverser Optimierungstechniken aus der Literatur angepasst. Zusätzlich wird eine Methode gezeigt, die es uns erlaubt hat, semi-optimale Einstellungen der zahlreichen Parameter der ACO Algorithmen zu finden.

Ein umfassender Vergleich unserer Resultate mit den Ergebnissen von bisher veröffentlichten PVRPTW Lösungsalgorithmen beschließt die Diskussion der Anwendung von ACO als Lösungsalgorithmus für das gesamte Problem. Obwohl kürzlich entwickelte hybride Algorithmen zur Lösung des PVRPTW eine bessere Lösungsleistung bei großen Probleminstanzen zeigen, konnte cascaded ACO den einzigen anderen bisher publizierten ACO Lösungsalgorithmus für das PVRPTW übertreffen.

Contents

x

CHAPTER 1

Introduction

1.1 Motivation

This thesis examines the application of a specific metaheuristic to solve a specialized form of routing problems.

Routing problems occur as real life problems in various situations. Logistics departments of carrier or other transport companies have to deal with the problem of minimizing costs when shipping goods. Communal garbage collection companies can save costs and resources with optimized routing for their garbage trucks. In a medical context, optimized blood or organ transportation can even save lives.

The archetype of routing problems is the *Travelling Salesman Problem* (TSP), where a (fictional) salesman has to visit a certain number of cities or customers in a round trip trying to minimize the total travel distance. Although the origins of the problem are unclear, it appeared in literature at the beginning of the 19th century. The first mathematical considerations were made in 1930 by Karl Menger [59]; in the 1950s it became increasingly relevant for the scientific community. Since then many optimization methods have been developed by researchers developing solving strategies for the TSP. Figure 1.1 shows the result of an exemplary application that solves the TSP using real road map information.

Many variations of the TSP jointly define the field of routing problems. When the visited cities or customers are divided among more than one travelling salesmen the problem is called the *multiple Travelling Salesman Problem* (mTSP) [81] [4]. If transportation capacity constraints for the travelling salesmen have to be taken into consideration, the problem becomes a *Vehicle Routing Problem* (VRP) [21] [92] that directly corresponds to the routing requirements of shipping companies servicing their customers with a single depot and multiple trucks. *Multi Depot Vehicle Routing Problems* (MDVRP) [58] consider more than one depot, and *Vehicle Routing Problems with Pickup and Delivery* (VRPPD) [72] generalize the idea by defining pick-up locations instead of depots and drop-off locations instead of customers that must be visited by a fleet of vehicles. If a planning period is introduced to the problem where the customers define specific service periodicities, it is called *Periodic Vehicle Routing Problem* (PVRP) [5] [38]. By intro-

Figure 1.1: Solution of the TSP for the nine regional capitals of Austria. Generated with `http://travellingsalesmanproblem.appspot.com/` using Google Maps™ mapping service

ducing time constraints to the problem that consider service times and service time windows for the customers as well as maximum travel durations for the vehicles it becomes a *Vehicle Routing Problem with Time Windows* (VRPTW) [88]. Both of these last named temporal considerations, planning period and time constraints, combine to make the *Periodic Vehicle Routing Problem with Time Windows* (PVRPTW) [14].

Many solution strategies have been developed to solve these routing problems using exact algorithms that compute optimal results as well as heuristic algorithms that generate sufficiently good results. One quite successful solution strategy for large TSP instances is using *Ant Colony Optimization* (ACO) [31]. This metaheuristic seems to be well suited for routing problems, since it simulates the natural behavior of real ant colonies when searching for short paths between food sources and their nests. In this thesis we apply ACO to the PVRPTW in two different ways:

- As method to solve the pricing problem of a column generation approach (see chapter 2.2)

- As overall method for the whole problem

1.2 Computational Complexity

The challenge with routing problems is that they belong to a class of problems with high complexity. In fact, current exact algorithms can generate optimal solutions with reasonable effort just for small problem instances.

Computational complexity theory [70] defines several complexity classes for computational problems. A commonly used machine model characterizing these classes is the Turing machine [93]. For our purpose we distinguish between deterministic and non-deterministic Turing machines. Whereas a deterministic Turing machine can perform just one step based on a given

2

state, a non-deterministic Turing machine can span several steps in parallel for each given state leading to a tree of succeeding states. Deterministic and non-deterministic Turing machines are equivalent in terms of problem solution power, but they differ in time behavior [86].

The following complexity classes are relevant for the optimization problems considered here:

- $\mathcal{P}$: complexity class of problems that can be solved by a deterministic Turing machine in polynomial time. Following Cobham-Edmonds thesis [11], problems belonging to this class are in practice efficiently solvable. $\mathcal{P}$ contains many important nontrivial problems, including the decision versions of linear programming, of the greatest common divisor problem, and of finding a maximum matching in a graph, as well as deciding if an integer is prime.

- $\mathcal{NP}$: complexity class of problems that can be solved by a non-deterministic Turing machine in polynomial time. This class obviously includes $\mathcal{P}$ since the deterministic Turing machine is a special case of the non-deterministic Turing machine.

- $\mathcal{NP}$-hard: complexity class of problems to which every problem in $\mathcal{NP}$ can be reduced in polynomial time. The resulting problems can also be in $\mathcal{NP}$, but not necessarily. Informally these problems can be viewed as the hardest problems of $\mathcal{NP}$ and problems that are even harder. This class contains many problems, including TSP and other routing problems, finding a minimum vertex cover in a graph, or the graph coloring problem. There are even $\mathcal{NP}$-hard problems that are not decidable, for example the halting problem.

- $\mathcal{NP}$-complete: complexity class of problems that are $\mathcal{NP}$-hard and in $\mathcal{NP}$. This class is very important since every $\mathcal{NP}$-complete problem represents the whole class in terms of complexity characterization, i.e. general findings on a single $\mathcal{NP}$-complete problem can be applied to every $\mathcal{NP}$-complete problem. A number of $\mathcal{NP}$-complete problems are known [54], including the boolean satisfiability problem, the knapsack problem, or the decision version of TSP.

Figure 1.2 depicts the correlation of the described complexity classes under the assumption that $\mathcal{P}$ and $\mathcal{NP}$ are not equivalent.

Since it has not yet been proven that the complexity class $\mathcal{P}$ is not equivalent to the complexity class $\mathcal{NP}$, it cannot be excluded that $\mathcal{P} = \mathcal{NP}$. The consequences would be enormous for practical computer science, due to the fact that the equivalence of $\mathcal{P}$ and $\mathcal{NP}$ implies the equivalence of the class of $\mathcal{NP}$-complete problems and $\mathcal{P}$! Therefore algorithms would exist that solve every $\mathcal{NP}$-complete problem in polynomial time. This would be of great value for the solution of many optimization problems, and a tremendous threat for cryptography.

Nevertheless, strong evidence exists that $\mathcal{P} \neq \mathcal{NP}$, since nobody has found an algorithm yet that solves any of the 3000 known $\mathcal{NP}$-complete problems in polynomial time. Therefore it is reasonable as well as practical to research the application of heuristics or metaheuristic algorithms to $\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems. Since routing problems belong to the class of $\mathcal{NP}$-hard problems with the decision version of the TSP belonging to $\mathcal{NP}$-complete problems [59], the use of metaheuristics is quite reasonable regarding complexity considerations.
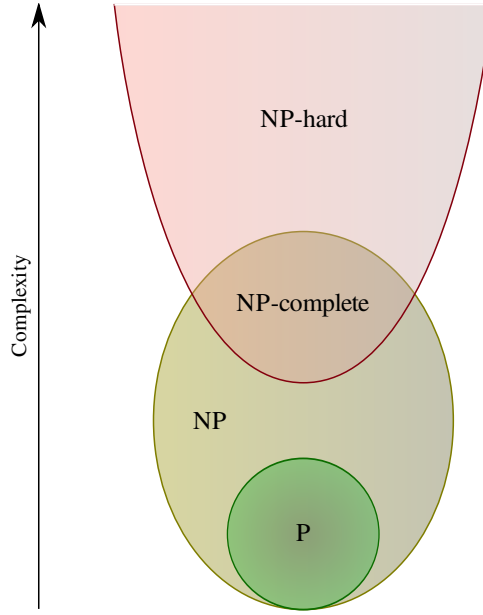
Figure 1.2: Euler diagram of problem complexity classes, under the assumption that $\mathcal{P} \neq \mathcal{NP}$

1.3 The Periodic Vehicle Routing Problem with Time Windows

The PVRPTW is primarily a generalized form of the TSP. Therefore it is defined similarly on a complete directed graph $G = (V, A)$, where $V = \{v_0, v_1, \ldots, v_n\}$ is the set of vertexes, and $A = \{a_{i,j} = (v_i, v_j) : v_i, v_j \in V, \ i \neq j\}$ is the set of arcs. The vertex v_0 represents the depot where each route has to start and end, the vertexes $v_1, \ldots, v_n$ represent the n customers that have to be visited. For each arc $a_{i,j}$ a non negative travel cost $c_{i,j}$ is defined.

For the generalization to the VRP we introduce a fleet of vehicles $H = \{h_1, \ldots, h_m\}$ that is based at the depot v_0. For each vehicle h_k a maximum carrying load Q_k is defined. With each customer v_i, $i \in \{1, \ldots, n\}$ a load demand q_i is associated. The m vehicles are not reused, i.e. each vehicle h_k has to start from the depot v_0 loaded with a maximum load of Q_k, then it services the assigned customers, and finally it returns to the depot where it ends the service.

The PVRP is the result of the next step of generalization, where a planning horizon $P = \{p_1, \ldots, p_t\}$ of t days is considered. Each customer v_i specifies a service frequency $f_i \in \{1, \ldots, t\}$, where 1 means that the customer has to be serviced on just one single day inside the planning horizon, and t means that it has to be serviced every day. Additionally, each customer v_i specifies a set of r_i visit combinations $R_i = \{C_{i,1}, \ldots, C_{i,r_i}\}$ where the visit combination $C_{i,x} \subseteq P$, $|C_{i,x}| = f_i \ \forall \ x \in \{1, \ldots, r_i\}$. A simple example illustrates this: Customer v specifies a service frequency $f_v = 2$ days for the planning horizon of $P = \{\mathsf{Mon, Tue, Wed, Thu, Fri, Sat}\}$ with $t = 6$ days. He specifies $r_v = 3$ different visit combinations $C_v = \{\{\mathsf{Mon, Thu}\}, \{\mathsf{Tue, Fri}\}, \{\mathsf{Wed, Sat}\}\}$. So the customer can be visited on days Mon and Thu, or Tue and Fri, or on Wed and Sat.

The consideration of time windows finalizes the generalization to the PVRPTW. Each cus-

tomer v_i specifies a service begin time window $[e_i, l_i]$ and a service duration d_i where $e_i, l_i, d_i \geq 0$. For each arc $a_{i,j}$ there is a non negative travel duration $z_{i,j}$. The servicing vehicle has to time the route in a way that it arrives at the customer v_i by l_i at the latest. If the vehicle arrives before e_i it has to wait till the service begins. Additionally there is a maximum route duration D_k for each vehicle h_k. The time window $[e_0, l_0]$ specifies the working time of the depot v_0, that is, the vehicles can leave at e_0 at the earliest and have to return at l_0 at the latest.

Table 1.1 provides an overview of the attributes that define the PVRPTW.

General	
n	number of customers
m	number of vehicles
P	planning horizon
t	number of days in P
Vertexes	
V	set of vertexes
v_0	depot
v_i	customer i, $i \geq 1$
q_i	load demand of customer i
f_i	service frequency of customer i
R_i	set of visit combinations of customer i
r_i	number of different visit combinations of customer i
$C_{i,x}$	the x-th visit combination of customer i
e_i	start of the service begin time window of customer i
l_i	end of the service begin time window of customer i
d_i	service duration at customer i
Vehicles	
H	fleet of vehicles
h_k	vehicle k
Q_k	maximum carrying load of vehicle k
D_k	maximum route duration of vehicle k
Arcs	
A	set of arcs
$a_{i,j}$	arc from vertex v_i to vertex v_j
$c_{i,j}$	travel costs from vertex v_i to vertex v_j
$z_{i,j}$	travel duration from vertex v_i to vertex v_j

Table 1.1: PVRPTW problem defining attributes

To solve the PVRPTW, one visit combination $C_{i,x}$ has to be selected from R_i for each customer v_i, and a maximum of m vehicle routes have to be found on the graph G for each day of the planning horizon P, such that the following rules apply:

- Each route has to start and end at the depot v_0.

- Each route has to start and end in the time window $[e_0, l_0]$.

- Each customer v_i belongs to exactly f_i routes.

- Each customer v_i is serviced on all days of the planning horizon that are part of the selected visit combination $C_{i,x}$.

- For each route the sum of the visited customers' load demands q_i does not exceed the maximum carrying load Q_k of the assigned vehicle h_k.

- For each route the total duration (travel durations $z_{i,j}$ + service durations d_i + waiting times at customers) does not exceed the maximum route duration D_k of the assigned vehicle h_k.

- The service for each customer v_i begins in the time window $[e_i, l_i]$.

- The total sum of travel costs $c_{i,j}$ over all routes is minimized.

1.4 Outline of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 explains combinatorial optimization and its challenges. It shows strategies for solving optimization problems exactly and describes in this context integer linear programming and its method set that is linear programming, branch-and bound, branch-and-cut, and branch-and-price. The technique of column generation is described to solve liner programming problems with a huge amount of variables. Furthermore, this chapter describes approximation strategies to solve optimization problems in an inexact manner. In this context we give an overview and classification of metaheuristics, examine their method set including neighborhood definition and local search, and describe the most popular metaheuristics. Consequently, the chapter outlines the possibilities to combine exact and approximate solution strategies.

Chapter 3 is devoted to previous research. An overview is given of history and state-of-the-art of column generation as well as metaheuristics applied to combinatorial optimization problems. Also recent work about hybridization of these two solution strategies is presented. Of course the outline focuses on routing problems in general and the PVRPTW in particular and emphases related work accordingly.

Chapter 4 describes in detail the application of ACO to the pricing subproblem of a column generation approach. Based on the set-covering formulation of the PVRPTW, it shows how to split the issue into a master problem and a pricing subproblem, whereas the restricted version of the master problem is solved via Simplex and the pricing subproblem is formulated as an *Elementary Shortest Path Problem with Resource Constraints* (ESPPRC). Here ACO is compared to other metaheuristics as well as to a pure exact solving strategy implemented with dynamic programming. Additionally the process of calibrating the parameters of ACO is described.

In chapter 5 ACO is applied to the whole PVRPTW. A new algorithm is developed that tries to focus on the exploratory strength of ACO. We call the algorithm *cascaded ACO* – it decomposes the problem into an optimization problem for visit combinations that is solved by an *upper level ACO* and a VRPTW that is solved by a *lower level ACO*. The parameter calibration focuses on the balance between exploitation of search history and problem knowledge and exploration

of search space, and tries to find a near-optimal setting regarding solution quality and algorithm runtime.

The computational results of the two application methods are presented in chapter 6. For that purpose a set of well-known problem instances is used to test the algorithms.

Chapter 7 concludes the thesis with an interpretation of the results. Further prospects and conceivable future work are discussed, including open issues that merit more detailed investigation.

Preliminaries

2.1 Combinatorial Optimization

Solving the PVRPTW and especially its base form the TSP is a typical combinatorial optimization task. In practice combinatorial optimization is one of the more difficult forms of mathematical optimization. It is characterized by a finite but often huge set of elements, with the goal being to find an optimal element regarding a cost function. Formally a combinatorial optimization problem [71] can be defined as $COP = (F, c)$, where F is a finite set of elements, and c is the cost function $c : F \longrightarrow \mathbb{R}$, where an element $f \in F$ has to be found for which $c(f) \leq c(x) \ \forall \ x \in F$.

For the PVRPTW F is the set of all feasible solutions that result from the combination of

- the selected customer visit combinations,

- the days of the planning horizon, and

- the maximum of m routes supported by the graph G

complying with the feasibility rules presented in section 1.3. This set is obviously finite, although it is huge since it follows in principle the growth of $n!$. The cost function c is defined by the total travel costs calculated by using the cost matrix $(c_{i,j})$.

A simple solution method, which is not practicable for real problem instances, is the total enumeration of the elements. However, there are strategies to solve combinatorial optimization problems more efficiently by not enumerating all but only the "relevant" elements. When a combinatorial optimization problem can be formulated in such a way that the optimizing cost function is expressed as a linear combination over an integer vector $x \in \mathbb{Z}^n$ and the feasibility restrictions are expressed as linear inequalities over x, we speak of an *Integer Linear Programming* (ILP) problem [87].

Definition 1 (Integer Linear Programming)**.** Let $A \in \mathbb{R}^{n \times m}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, then $\{\min c^T x \mid Ax \leq b, x \in \mathbb{Z}^n\}$[1] is called an integer linear programming or an integer programming problem.

In fact most of the practical relevant combinatorial optimization problems can be expressed as ILP problems. Unfortunately there are proofs that the general ILP is $\mathcal{NP}$-hard. However for special ILP classes polynomial-time or semi-polynomial-time algorithms have been found [87]. A special form of ILPs are *binary integer programming* (BIP) or *0/1 integer programming* problems, where the integrality constraint $x \in \mathbb{Z}^n$ is replaced with $x \in \{0, 1\}^n$. They are also classified as $\mathcal{NP}$-hard.

If it is possible to formulate a combinatorial optimization problem with a linear cost function and linear inequalities, but without the integrality constraint of x, then the formulation represents a *linear programming* (LP) problem [71].

Definition 2 (Linear Programming)**.** Let $A \in \mathbb{R}^{n \times m}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, then $\{\min c^T x \mid Ax \leq b, x \in \mathbb{R}^n\}$ is called a linear programming problem.

Most of the combinatorial optimization problems have been shown to have an ILP representation, like TSP and other routing problems, in fact all of the $\mathcal{NP}$-complete and most of the $\mathcal{NP}$-hard problems. Nevertheless, combinatorial optimization problems with an LP representation exist, such as finding a maximum matching in a graph, which makes them belong to the complexity class $\mathcal{P}$ that can be solved efficiently in general.

A mixed form of linear and integer linear programming problem formulation also exists, known as *mixed integer linear programming* (MILP). Here a part of the variables x have to be integral, the other part not. Like the general ILP the general MILP is also $\mathcal{NP}$-hard. Since the solution strategies are very similar we do not further differentiate between ILP and MILP.

Definition 3 (Mixed Integer Linear Programming)**.** Let $A_I \in \mathbb{R}^{n_I \times m}$, $A_N \in \mathbb{R}^{n_N \times m}$, $b \in \mathbb{R}^m$, $c_I \in \mathbb{R}^{n_I}$, $c_N \in \mathbb{R}^{n_N}$, $n = n_I + n_N$ then $\{\min c_I^T x_I + c_N^T x_N \mid A_I x_I + A_N x_N \leq b, x_I \in \mathbb{Z}^{n_I}, x_N \in \mathbb{R}^{n_N}\}$ is called a mixed integer linear programming problem.

Linear programming and integer linear programming are related, since an LP problem can be formulated by relaxing the integrality constraint of an ILP[2]. Figure 2.1 displays a graphical representation of the situation illustrated by example 1. The cost function $\min x - 4y$ is represented by the red dashed line, minimizing the cost function is indicated by the arrow that gives the direction for this optimization. The four inequalities divide the problem space into two half-spaces, indicated by the bounding line and an arrow. The resulting area of feasible values of the LP is shaded in yellow. By adding the integrality constraints $x, y \in \mathbb{Z}$ the problem becomes an ILP problem. The solution space of feasible values for the ILP consists of a finite set of points, displayed as green dots.

---

[1]Note that minimizing the cost function represents both possibilities of optimization, since maximizing can be transformed by changing the sign of c, i.e. $\max c^T x \equiv -\min(-c)^T x$. The same transformation can be applied for the inequalities to represent greater equal constraints: $A_i^T x \geq b_i \equiv (-A_i)^T x \leq -b_i$. In fact even equality constraints can be transformed to inequalities by replacing a linear equation with two linear inequalities with opposite sign: $A_i^T x = b_i \equiv A_i^T x \leq b_i, (-A_i)^T x \leq -b_i$

[2]Note that the integrality constraint of a BIP problem includes upper and lower bounds for the variables that can be covered by the relaxed problem: $x \in \{0, 1\}^n \longmapsto 0 \leq x \leq 1, x \in \mathbb{R}^n$
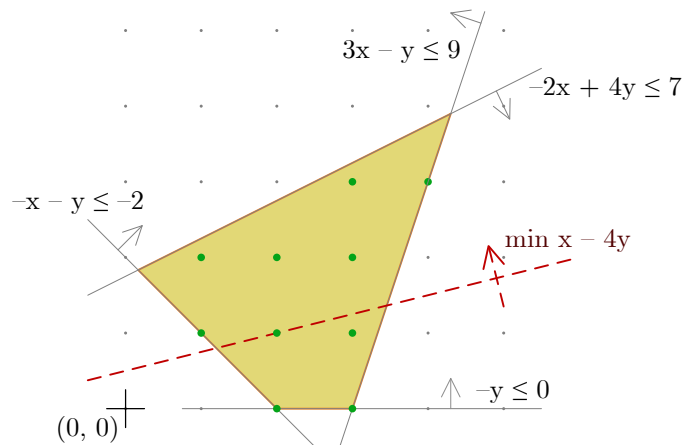
Figure 2.1: Graphical representation of an ILP problem and the related LP problem by relaxing the integrality constraints

Example 1. *An example of a linear programming problem with two variables x and y and four inequalities:*

$$\min x - 4y$$
$$3x - y \leq 9$$
$$-2x + 4y \leq 7$$
$$-x - y \leq -2$$
$$-y \leq 0$$

With the integrality constraint of x and y the problem becomes an integer linear programming problem:

$$x, y \in \mathbb{Z}$$

In general an LP problem with n variables and m inequalities can be interpreted as an n-dimensional space that is divided into m halfspaces by $(n - 1)$-dimensional hyperplanes. The intersection of these halfspaces forms a convex polyhedron that builds the space of feasible values. If the intersection is empty then the problem is not solvable. If the polyhedron is bounded then it is called a polytope[3] and there exists a solution for the LP problem. If the polyhedron is unbounded there may be a solution to the problem or the solution is not finite. Minimizing or maximizing a linear function over the convex polyhedron of feasible values corresponds to the search for an extreme point of the polyhedron that is always a vertex [87].

By adding the integrality constraints the solution space of the ILP is made discontinuous. The feasible points are located inside the convex hull spanned by the polyhedron. Therefore

---

[3]The terms *polyhedron* and *polytope* are not used consistently in the literature, especially regarding dimensionality, bound and unbound, or convex and not convex characteristics there exist different notions. The notion used here is taken from Schrijver [87]

the solution of the LP problem is a lower bound in the case of a minimizing optimization and an upper bound in the case of a maximizing optimization for the solution of the related ILP problem. But a simple rounding operation on the variables of the LP solution does not work, as can be easily seen in figure 2.1: The optimal solution for the LP problem of example 1 is $x = 4.3$, $y = 3.9$. Rounding to the nearest integer results in $x = 4$, $y = 4$, which is not a feasible solution; rounding down results in $x = 4$, $y = 3$, which is not an optimal solution. Even worse, there is no guarantee that an ILP problem is solvable when the integrality-relaxed LP problem has a solution. However, if the optimal solution of the LP-relaxed[4] problem is integral, then it is also the optimum for the ILP.

An important characteristic of an LP problem is that it is possible to formulate a symmetrical LP problem with the same optimal solution as long as a finite optimal solution exists. This problem is called the *dual* LP problem, the original problem is called the *primal* LP problem. To get the dual problem the variables of the primal problem are associated with constraint inequalities or equalities, the constraint inequalities or equalities are associated with dual variables, and the optimizing operator of the cost function changes from max to min or vice versa. The duality theorem for linear programming expresses this fact.

Theorem 1 (Duality Theorem for Linear Programming [41])**.** Let $A \in \mathbb{R}^{n \times m}, b \in \mathbb{R}^m, c \in \mathbb{R}^n$, then $\min\{c^T x \mid Ax \leq b, x \geq 0, x \in \mathbb{R}^n\} = \max\{b^T y \mid A^T y \leq c, y \leq 0, y \in \mathbb{R}^n\}$, as long as a solution exists and the optimum is finite.

This is just one form of several equivalent forms of the duality theorem, all of them dealing with different constraint operators or variable restrictions. Table 2.1 displays the rules for formulating the dual problem of a general primal problem. Example 2 shows the dual LP problem of the primal LP problem from example 1.

minimize	$\rightleftarrows$	maximize
constraint inequality $\leq$	$\rightleftarrows$	variable ≤ 0
constraint inequality $\geq$	$\rightleftarrows$	variable ≥ 0
constraint equality $=$	$\rightleftarrows$	unrestricted variable
variable ≤ 0	$\rightleftarrows$	constraint inequality $\geq$
variable ≥ 0	$\rightleftarrows$	constraint inequality $\leq$
unrestricted variable	$\rightleftarrows$	constraint equality $=$

Table 2.1: Ruleset for formulating dual LP problems from primal LP problems

Example 2. *The dual form of the linear programming problem of example 1. The two variables x and y of the primal problem are associated with two equality constraints, the four inequalities*

---

[4]Subsequently we call the LP problem that results from an ILP by removing the integrality constraints the LP-relaxed problem of the ILP

are associated with the four variables r, s, t, and u:

$$\max 9r + 7s - 2t$$
$$3r - 2s - t = 1$$
$$-r + 4s - t - u = -4$$
$$r, s, t, u \leq 0$$

The duality theorem implies an important fact for solving practice: Each feasible solution of a dual LP maximization problem is a lower bound for the optimal solution of the primal LP minimization and vice versa, as long as a finite optimum exists.

2.2 Exact Solution

Several algorithms have been developed to solve combinatorial optimization problems exactly, that is, to obtain an optimal solution. Since LP problems belong to the complexity class $\mathcal{P}$ they can be solved efficiently, but (unless $\mathcal{P} = \mathcal{NP}$) general ILP problems lack this advantage.

LP Solution Algorithms

Simplex

One of the most famous algorithms to solve LP problems was developed by George Dantzig: the *simplex algorithm* [19]. This algorithm works on an LP representation of the form $\min\{c^T x \mid Ax = b, x \geq 0\}$[5]. It uses the fact that a system of linear equalities can be transformed to the canonical form $Ix_B + \tilde{A}x_N = \tilde{b}$, where I is the identity matrix, when given a feasible solution. By doing so the set of variables x is divided into basic variables x_B and non-basic variables x_N. The algorithm iteratively swaps non-basic variables with basic variables and tries to reduce the cost function with each swap operation. The base structure for the algorithm is the simplex tableau $\begin{bmatrix} 1 & c_B^T & c_N^T & 0 \\ 0 & I & \tilde{A} & \tilde{b} \end{bmatrix}$ or $\begin{bmatrix} 1 & 0 & \tilde{c}^T & z \\ 0 & I & \tilde{A} & \tilde{b} \end{bmatrix}$ after applying some Gaussian elimination transformations for the first row that contains the cost function. Note that c_B^T and c_N^T are the coefficients of the cost function for basic and non-basic variables respectively. z contains the value of the cost function for the given feasible solution. Algorithm 2.1 shows this basic form of the general simplex algorithm.

Using the graphical representation of an LP problem the simplex algorithm can be interpreted as the traversal of the convex feasibility polytope from one vertex to the next by improving the value of the cost function. The basic feasible solution corresponds to an arbitrary vertex, the selection of a simplex tableau column and row corresponds to the selection of an edge of the polytope to the next vertex. The algorithm terminates at the optimum when no edge can be found that leads to a cost function improving vertex.

---

[5]Inequality constraints can be formulated as equalities by introducing slack variables, e.g. $A_i^T x \leq b_i \equiv A_i^T x - s_i = b_i, s_i \geq 0$; Unrestricted variables can be eliminated by replacing them with two restricted variables: $x_i = x_i^+ - x_i^-, x_i^+ \geq 0, x_i^- \geq 0$

Input: linear program $LP = \min\{c^T x \mid Ax = b, x \geq 0\}$
Output: optimal solution of LP

1 *Phase I:*
2 find a basic feasible solution S of LP
   `// if no basic feasible solution can be found LP has no`
   `   solution`
3 generate simplex tableau

4 *Phase II:*
5 **while** S *not optimal* **do**
   `// S is optimal if no cost function value reducing`
   `   non-basic variable exists`
6   select a simplex tableaux column from non-basic variables to reduce value of cost function
   `// if no row exists with a positive coefficient in the`
   `   selected column, the problem is unbounded and there is`
   `   no finite optimum`
7   select a simplex tableaux row to remove from basic variables
8   transform selected column to unit vector by Gaussian elimination, generating new solution S
9   rewrite simplex tableau
10 **end**

Algorithm 2.1: General Simplex Algorithm

In Phase I of the simplex algorithm a basic feasible solution has to be found to start with. This can be accomplished by formulating a new LP problem that is related to the original LP problem. For each equation an artificial variable y_j is added and the cost function is rewritten so that the optimal solution ensures that each artificial variable is 0: $\min\{\sum_j y_j \mid Ax + y = b, x \geq 0, y \geq 0\}$. Finding a basic feasible solution for this problem is trivial: $x = 0, y = b$. By applying the simplex algorithm to this problem an initial solution for the original problem can be found, unless the optimal value of $\min \sum_j y_j \neq 0$ in which case no feasible solution exists.

There are variations of the simplex algorithm that try to improve the behavior of the algorithm for specific types of LP problems. The dual simplex algorithm solves the dual formulation of the LP problem. This can have advantages in runtime when the number of constraints is large compared to the number of variables. Also for some LP problems it is trivial to determine a basic feasible solution for the dual LP problem which allows the omission of phase I of the simplex algorithm. For this it is important that the simplex tableau also generates the optimal solution of the dual problem in additional to the primal optimal solution which can be accomplished by using the Tucker tableau [45].

Another variant deals with LP problems that can be represented in quite sparse simplex tableaux: the revised simplex algorithm does not store the whole simplex tableau; instead it stores the necessary elements for the next simplex step and calculates the missing elements accordingly, by using LU decomposition of the simplex tableau and similar methods.

14

Other variations deal with special strategies for selection of non-basic variables and basic variables in the essential simplex step. For selection of a simplex tableau column of non-basic variables there exists the classic method proposed by Dantzig that uses the variable with the largest reducing cost coefficient. Other selection methods include steepest-edge pricing [37], devex pricing [50], partial pricing [68], or combinations of these. The selection of a simplex tableau row for elimination of a basic variable influences also the behavior of the simplex algorithm according algorithmic cycles that can occur in degenerated LP problems, which contain constraints that do not affect the feasible solution space. Lexicographic pivoting and Bland's rule [7] prevent cycling, whether a random row selection strategy makes cycling improbable but not impossible.

The simplex algorithm shows polynomial time behavior for "random" LP problems, which makes it quite efficient in practice [71]. Nevertheless, it is possible to construct LP problems where the simplex algorithm degenerates to exponential time behavior [56].

Finite Criss-Cross

The same principal time behavior is also shown by the *finite criss-cross algorithm*, another example of an exact LP problem solving algorithm. This algorithm, proposed by Chang, Terlaky [91] and Wang, is similar to the simplex algorithm since it uses also basis exchange operators to traverse the solution space to the optimal value. But unlike the simplex algorithm the criss-cross algorithm allows bases that do not correspond to vertexes of the feasibility polyhedron. Furthermore, the algorithm even allows infeasible bases to be traversed. To achieve this it uses the primal as well as the dual LP formulation and tries to find a feasible optimum by jumping from primal infeasible and/or dual infeasible bases to feasible bases. Thus, the algorithm has no need to perform a first phase as does the simplex algorithm, since it is not required to start with a feasible solution [39].

Ellipsoid

Another exact solution method for LP problems is the *ellipsoid algorithm* introduced by Khachijan [48], who applied previous work about non-linear optimization to linear programming and showed the polynomial time behavior of the ellipsoid algorithm for LP. The algorithm is based on a binary search over the optimizing cost function and a feasibility check of a set of inequalities. The feasibility check is performed on the polyhedron determined by the set $\{c^T x \geq z_k, Ax \leq b, x \geq 0\}$, where z_k is the cost function value for the k-th iteration of the binary search. The initial step is the construction of an ellipsoid whose volume contains the polyhedron. Then a hyperplane is generated to separate the polyhedron and the central point of the ellipsoid. The smallest possible ellipsoid is thereby constructed whose volume contains the intersection of the original ellipsoid and the halfspace containing the polyhedron. If after a precalculated number of iterations the central point is not inside the polyhedron, the polyhedron is empty and the set of inequalities is infeasible. Algorithm 2.2 describes the feasibility check by the ellipsoid algorithm. Since the binary search and feasibility check by the ellipsoid algorithm are polynomial in time behavior (for details see [48]), the solution algorithm for the LP problem is also polynomial.

15

Input: set of inequalities $S = \{c^T x \geq z_k, Ax \leq b, x \geq 0\}$
Output: "yes" if S is feasible, "no" if not and polyhedron of S is empty

1 set initial ellipsoid E_0 that contains polyhedron of S if not empty
2 calculate maximum number of iterations L_{max}
3 $l \leftarrow 0$
4 **repeat**
5 | **if** *central point of E_l inside polyhedron of S* **then return** *"yes"*
6 | $H \leftarrow$ hyperplane separating polyhedron of S from central point of E_l
7 | $E_{l+1} \leftarrow$ minimum volume ellipsoid containing $E_l \cap H^+$
        | `// ` H^+ ` is the halfspace separated by ` H ` that contains`
        | `    polyhedron of ` S ` if not empty`
8 | $l \leftarrow l + 1$
9 **until** $l = L_{max}$
10 **return** *"no"*

Algorithm 2.2: Ellipsoid Algorithm performing a feasibility check on a polyhedron

Although the ellipsoid algorithm is polynomial in time, it has hardly any practical relevance for LP problem-solving because the simplex algorithm performs better for real world problems. Nevertheless, it is significant for theoretical considerations since it proves that the LP problems belong to the complexity class $\mathcal{P}$.

Karmarkar

Other exact LP problem solving algorithms have been developed that solve each LP problem in polynomial time doing this more efficiently than the ellipsoid algorithm. One is *Karmarkar's algorithm* [53] which belongs to the class of interior point methods. It uses fast Fourier transforms to traverse the feasible solution space inside the polyhedron instead of traversing it on the surface walking from vertex to vertex. Since Karmarkar's algorithm competes with the simplex algorithm, other interior point methods such as primal-dual path-following interior point methods have been developed to enrich the class of exact LP problem solving algorithms.

ILP Solution Algorithms

Although the general ILP problem is $\mathcal{NP}$-hard and therefore cannot be solved efficiently (as long as $\mathcal{P} \neq \mathcal{NP}$), strategies have been developed to solve these problems exactly.

Cutting Plane

One of the first methods dealing with ILP problems was the *cutting plane algorithm*. Initially formulated for the TSP [20] it was generalized by Gomory for all ILPs [46]. The idea of the cutting plane algorithm is based on the LP problem solution: First the LP-relaxed problem has to be solved (e.g. with simplex algorithm) – the result is a lower bound (in the case of a minimizing problem) for the ILP solution. If the optimal solution is not integral then an

additional constraint has to be added that excludes this solution by making it infeasible, but leaves all other integral solutions inside the feasible space of the problem. This constraint equals a hyperplane (a cutting plane) that builds a new facet for the resulting polyhedron. The new LP-relaxed problem including the newly generated inequality is solved which continues the iterative process with a new optimal solution that is necessarily a tighter (higher in case of a minimizing problem) lower bound for the ILP solution. Algorithm 2.3 shows the general form of the cutting plane algorithm.

Input: ILP $\min\{c^T x \mid Ax = b, x \geq 0, x \in \mathbb{Z}\}$
Output: optimal solution of ILP

1 $x^{LP} \leftarrow$ solve LP $\min\{c^T x \mid Ax = b, x \geq 0\}$
2 $l \leftarrow 1$
3 **while** x^{LP} *not integral* **do**
4      $d_l^T x \leq e_l \leftarrow$ inequality separating x^{LP} from feasible integral solutions of ILP
5      $x^{LP} \leftarrow$ solve LP $\min\{c^T x \mid Ax = b, d_1^T x \leq e_1, \ldots, d_l^T x \leq e_l, x \geq 0\}$
6      $l \leftarrow l + 1$
7 **end**
8 **return** x^{LP}

Algorithm 2.3: General form of the Cutting Plane Algorithm

Finding a hyperplane that separates the LP-relaxed optimum from the rest of the feasible integral solutions is called the *separation problem*. Gomory presented a method based on the simplex tableau used when solving the LP-relaxed problem: after solving procedure the simplex tableau consists of rows in the form of $x_i + \sum_j \tilde{a}_{ij} x_j = \tilde{b}_i$, where x_i is a basic variable and x_j are non-basic variables. Separating integral and fractional parts leads to $x_i + \sum_j \lfloor \tilde{a}_{ij} \rfloor x_j - \lfloor \tilde{b}_i \rfloor = (\tilde{b}_i - \lfloor \tilde{b}_i \rfloor) - \sum_j (\tilde{a}_{ij} - \lfloor \tilde{a}_{ij} \rfloor) x_j < 1$. For any feasible integral x the left-hand side of the equation is integral implicating that the right-hand side has to be ≤ 0. On the other hand, for the non-integral optimal solution the right-hand side becomes $\tilde{b}_i - \lfloor \tilde{b}_i \rfloor$ that is > 0. Therefore an inequality can be formulated that separates all feasible integral x from the non-integral optimal solution: $(\tilde{b}_i - \lfloor \tilde{b}_i \rfloor) - \sum_j (\tilde{a}_{ij} - \lfloor \tilde{a}_{ij} \rfloor) x_j \leq 0$. This method can be applied to any kind of ILP problem, though in practice it leads to many iterations and numerical problems.

A geometrical interpretation of the cutting plane algorithm with a Gomory cut is displayed in figure 2.2 using example 1. The result of the LP-relaxed problem LP_1 is calculated. By using the simplex tableau generated while solving this problem, the Gomory cut is determined as $y \leq 3$. This inequality builds a hyperplane that separates LP_1 from the feasible integral solutions of the ILP, cutting off a part of the original polyhedron. In the figure it is shown as a blue dotted line defining a halfspace that cuts off the blue hatched area. For the next iteration the new constraint is added to the ILP problem, and again the LP-relaxed problem is solved. This leads to LP_2 that is a tighter lower bound for the ILP problem than LP_1. The procedure is repeated until an integral solution is found.

The method presented by Gomory to solve the separation problem is the most generic form for ILP problem solution, but it only uses weak cutting planes. In contrast strong cutting planes generate a facet of the polyhedron of the integral LP problem that is the tightest convex hull of all
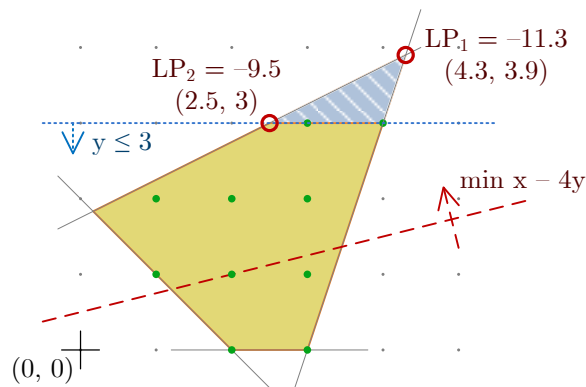
Figure 2.2: Geometrical interpretation of the cutting plane algorithm

feasible integer solutions. By researching the feasibility polyhedrons for specific ILP problems, more and better cutting planes have been found to enhance solving performance. These include for example the family of generalizing comb inequalities for the TSP [66], or lift-and-project or disjunctive inequalities [1], which made cutting planes for the last decades an important tool for successful ILP solving. Especially in combination with branch and bound (see below) these cutting planes showed considerable solving power.

There is an adjacent application for the cutting plane method: if a problem is formulated as LP with a huge set of constraints the solving algorithm can start with just a part of this set. Here the separation problem is to check if a constraint is violated that was not part of the starting set of constraints. By adding this constraint and solving the problem again a new solution is generated that can be checked against the remaining constraints. With this method it is possible to solve LP problems with an exponential number of constraints without enumerating them, as long as the separation problem can be solved efficiently! Typically, problem formulations with huge sets of constraints exist for hard combinatorial optimization problems that can be solved with the cutting plane method.

Because this method adds constraints to the LP problem that become visible in the simplex tableau as rows, the cutting plane is also called *row generation*.

Branch and Bound

A different approach for solving the ILP was used by Land and Doig when they formulated the *branch and bound algorithm* for integer programming [57]. Although it is also based on the solution of the LP-relaxed problem, it uses a divide and conquer principle to deal with the integrality constraints: after solving the LP-relaxed problem the LP with non-integral solution variables is split into two subproblems that are solved separately. The best solution for these two subproblems is the best solution of the whole problem. To solve a subproblem the same method is applied, leading to a recursive algorithm that traverses a decision tree. A leaf of the tree is found when the solution of the LP-relaxed subproblem is integral.

The process of splitting problems into subproblems is called branching and can be accom-

plished quite easily with ILP problems, by simply adding a constraint to each of the subproblems which differentiates for a non-integral solution variable between the lower and the upper integer: for the first subproblem add the constraint $x_i \leq \lfloor x_i^{LP} \rfloor$, for the second the constraint $x_i \geq \lceil x_i^{LP} \rceil$, where x_i^{LP} is the optimal non-integral solution value of x_i calculated by solving the LP-relaxed problem.

For the bounding part of the algorithm a lower and upper bound for the problem have to be calculated. In the case of a minimizing problem, if a subproblem's lower bound is $\geq$ than the problem's upper bound the branching of the subproblem can be omitted because the subproblem cannot have a better solution. To obtain a lower bound the value of the cost function for the LP-relaxed problem solution is used. To obtain an upper bound each feasible i.e. integral solution can be used. The branch and bound algorithm for ILP solving is shown in Algorithm 2.4.

Data: global value: $upper \leftarrow \infty$ (upper bound)
Data: global vector: x^{ILP} (optimal solution of ILP)
Input: ILP $\min\{c^T x \mid Ax = b, x \geq 0, x \in \mathbb{Z}\}$
Output: optimal solution of ILP

1   $x^{LP} \leftarrow$ solve LP $\min\{c^T x \mid Ax = b, x \geq 0\}$
2   **if** *LP is not feasible* **then**
    | // ILP is also not feasible
3   **else if** x^{LP} *integral* **then**
    | // feasible ILP solution
4 | **if** $c^T x^{LP} < upper$ **then**
5 | | $upper \leftarrow c^T x^{LP}$
6 | | $x^{ILP} \leftarrow x^{LP}$
7 | **end**
8   **else if** $c^T x^{LP} \geq upper$ **then**
    | // omit because of bounds
9   **else**
    | // branch into two subproblems
10 | select x_i with not integral value x_i^{LP}
11 | recursive Branch and Bound with ILP $\cup \{x_i \leq \lfloor x_i^{LP} \rfloor\}$
12 | recursive Branch and Bound with ILP $\cup \{x_i \geq \lceil x_i^{LP} \rceil\}$
13 **end**

Algorithm 2.4: Branch and Bound Algorithm for ILP

The branch and bound decision tree used to solve example 1 is displayed in figure 2.3. Z^{LP} is the value of the cost function for the optimal solution of the LP-relaxed problem in each branch and bound node. The tree is traversed with a left-to-right depth-first search strategy. Therefore, the integral solution $x^{LP} = 3, y^{LP} = 3$ defines an upper bound of -9 for the rest of the tree traversal. This bound enables the algorithm to omit a further branch for the LP subproblem with an optimal cost function value of -9 which is the lower bound of this subproblem, so no better solution can be found in this branch. Other decision tree traversal strategies (e.g. best-first search, or other) can lead to a different behavior regarding number of branches and progression
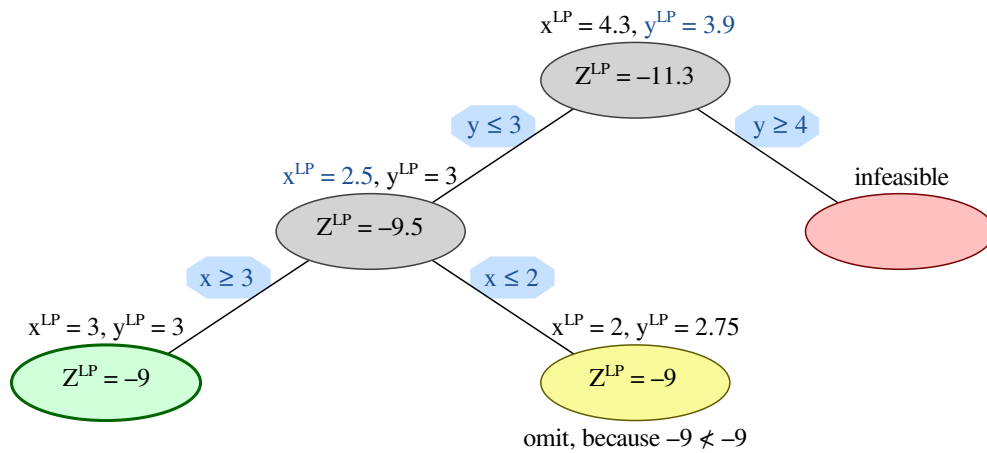
of the upper bound.

$x^{LP} = 4.3, y^{LP} = 3.9$

$Z^{LP} = -11.3$

$y \leq 3$ $y \geq 4$

$x^{LP} = 2.5, y^{LP} = 3$ infeasible

$Z^{LP} = -9.5$

$x \geq 3$ $x \leq 2$

$x^{LP} = 3, y^{LP} = 3$ $x^{LP} = 2, y^{LP} = 2.75$

$Z^{LP} = -9$ $Z^{LP} = -9$

omit, because $-9 \nless -9$

Figure 2.3: A Decision Tree of the Branch and Bound algorithm

A geometrical interpretation of a branch operation is shown in figure 2.4. The optimal solution for the actual LP-relaxed problem is marked with a red circle $x^{LP} = 2.5, y^{LP} = 3$. The branch is performed using the variable x, the resulting two LP problems have each added an inequality specifying that $x \leq 2$ for LP$_I$, or $x \geq 3$ for LP$_{II}$.
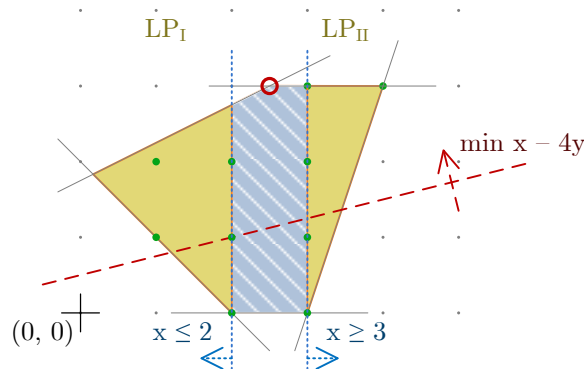

Figure 2.4: Geometrical interpretation of a Branch operation of the Branch and Bound algorithm

Branch and Cut

Branch and cut is a hybrid of branch and bound and cutting plane algorithms. Early combinations of these two algorithms proposed to solve the LP-relaxed problem then find strong cutting planes and finally perform a branch and bound including these additional planes; this type of algorithm was called cut and branch [17]. For this approach all original constraints have to be part of the problem when entering the branch and bound phase.

With the development of branch and cut [69] this disadvantage could be avoided. Here the cutting plane algorithm is applied at each branch of the branch and bound decision tree. Algorithm 2.5 shows the general branch-and-cut procedure. Variants of the algorithm distinguish between local cuts and global cuts that can be applied on the whole decision tree and therefore speed up the cutting plane step of distinct subtrees.

Data: global value: $upper \leftarrow \infty$ (upper bound)
Data: global vector: x^{ILP} (optimal solution of ILP)
Input: ILP $\min\{c^T x \mid Ax = b, x \geq 0, x \in \mathbb{Z}\}$
Output: optimal solution of ILP

1 $\acute{\text{LP}} \leftarrow \min\{c^T x \mid \acute{A}x = \acute{b}, x \geq 0\}$ LP with reduced constraints
2 $x^{\acute{LP}} \leftarrow$ solve $\acute{\text{LP}} \min\{c^T x \mid \acute{A}x = \acute{b}, x \geq 0\}$
3 **if** $\acute{LP}$ *is not feasible* **then**
   | // ILP is also not feasible
4 **else**
5 | $CP \leftarrow \{\}$
6 | **repeat**
7 | | $CP \leftarrow CP \cup$ inequality separating $x^{\acute{LP}}$ from feasible integral solutions of ILP
8 | | $x^{\acute{LP}} \leftarrow$ solve $\acute{\text{LP}} \min\{c^T x \mid \acute{A}x = \acute{b}, x \geq 0\} \cup CP$
9 | **until** $x^{\acute{LP}}$ *is feasible in non-reduced LP* $\{Ax = b, x \geq 0\}$ *(or later)*
10 | **if** $x^{\acute{LP}}$ *integral* **then**
   | | // feasible ILP solution
11 | | **if** $c^T x^{\acute{LP}} < upper$ **then**
12 | | | $upper \leftarrow c^T x^{\acute{LP}}$
13 | | | $x^{ILP} \leftarrow x^{\acute{LP}}$
14 | | **end**
15 | **else if** $c^T x^{\acute{LP}} \geq upper$ **then**
   | | // omit because of bounds
16 | **else**
   | | // branch into two subproblems
17 | | select x_i with not integral value $x_i^{\acute{LP}}$
18 | | recursive Branch and Cut with ILP $\cup CP \cup \{x_i \leq \lfloor x_i^{\acute{LP}} \rfloor\}$
19 | | recursive Branch and Cut with ILP $\cup CP \cup \{x_i \geq \lceil x_i^{\acute{LP}} \rceil\}$
20 | **end**
21 **end**

Algorithm 2.5: Branch and Cut Algorithm for ILP

Column Generation

There are problem formulations with a huge set of variables. Dantzig and Wolfe even proposed a method to generate such formulations for LP problems, known as Dantzig-Wolfe decomposition [22]. The *column generation algorithm* was developed to solve such problems by applying the same principles as the simplex algorithm does: since the non-basic variables of a feasible solution are 0 and only the basic variables build the solution, the non-basic variables do not need to be enumerated. Therefore it is just necessary for a simplex step to find a non-basic variable that improves the value of the cost function, that is, a variable with negative reduced costs for a minimizing problem or positive reduced costs for a maximizing problem. This is called the *pricing problem* [26].

Let us consider the LP problem $\{\min c^T x \mid Ax \leq b, x \geq 0\}$ with its dual LP problem $\{\max b^T y \mid A^T y \leq c, y \leq 0\}$. We call the LP with a reduced set of variables in the context of column generation the *restricted master problem* (RMP) $\grave{\text{LP}}$ $\{\min \grave{c}^T \grave{x} \mid \grave{A}\grave{x} \leq b, \grave{x} \geq 0\}$ with its dual RMP $\{\max b^T y \mid \grave{A}^T y \leq \grave{c}, y \leq 0\}$. For a given optimal solution $\grave{x}^*$ of the RMP the optimal solution of the dual RMP is y^*. If y^* is feasible in the dual non-restricted LP then it is also optimal in the dual non-restricted LP problem, because the cost functions of the dual LP and the dual RMP are equal and the set of constraints is merely a subset. But if y^* is optimal for the dual non-restricted LP, then $\grave{x}^*$ is optimal for the non-restricted LP! Therefore the pricing problem to find a variable with negative reduced costs is equivalent to the problem of finding a violated constraint in the dual LP problem. If no such violated constraint can be found, the solution is optimal. Such a constraint can be found by determining a column j where $a_{.j}{}^T y^* \not\leq c_j$, whereas $a_{.j}$ is the j-th column of A. This pricing problem for the general column generation method can also be expressed as the requirement to find a $j \mid \sum_i a_{ij} y_i^* > c_j$. Algorithm 2.6 shows the general column generation method.

Input: LP $\min\{c^T x \mid Ax = b, x \geq 0, x \in \mathbb{R}\}$
Output: optimal solution of LP

1 $\grave{\text{LP}} \leftarrow \min\{\grave{c}^T x \mid \grave{A}x = b, x \geq 0\}$ LP with reduced variables
2 $J^+ \leftarrow \{\}; c^+ \leftarrow \{\}; A^+ \leftarrow \{\}$
3 $y^* \leftarrow$ solve dual $\grave{\text{LP}}$ $\max\{b^T y \mid \grave{A}^T y \leq \grave{c}\}$
4 **while** *cost reducing column j exists so that $\sum_i a_{ij} y_i^* > c_j$* **do**
5 $\quad \mid \quad J^+ \leftarrow J^+ \cup j; c^+ \leftarrow c^+ \cup c_j; A^+ \leftarrow A^+ \cup a_{.j}$
6 $\quad \mid \quad y^* \leftarrow$ solve dual $\grave{\text{LP}}$ $\max\{b^T y \mid \grave{A}^T y \leq \grave{c}\} \cup \{a_{.j}^T y \leq c_j\}, j \in J^+$
7 **end**
8 **return** *solve $\grave{\text{LP}}$* $\min\{\grave{c}^T x + \sum_{j \in J} c_j x \mid \grave{A}x + \sum_{j \in J} a_{.j} x = b, x \geq 0\}$
Algorithm 2.6: General Column Generation Algorithm for LP

Variations of this algorithm add not just one but several variables with negative reduced costs in each iteration. Also non-basic variables could be removed from the actual solution. Nevertheless, the key to an efficient column generation application is a well-chosen formulation of the problem. By doing so the pricing subproblem can become a well-known combinatorial optimization problem which can be solved by efficient problem specific techniques instead of

applying general LP or ILP algorithms.

The column generation approach is symmetrical to the row generation approach of the cutting plane algorithm for the dual problem. Therefore column and row generation can be interpreted as dual solution strategies to solve an LP problem.

Although, column generation is a method to solve LP problems (similar to the row generation approach of cutting plane), it becomes an important part of the next exact ILP solution technique.

Branch and Price

Branch and price is a hybrid of branch and bound and column generation algorithms. The algorithm uses the LP-relaxed problem, but in contrast to branch and cut that starts with a reduced set of constraints, branch and price starts with a reduced set of variables. Here column generation is applied at each node of the branch and bound decision tree.

For branch and price it is very important to choose an appropriate branching strategy. The problem for classic branch and bound formulations with a huge set of variables is that fixing a single variable to its lower and upper integer often leads to very imbalanced decision trees. Therefore problem-specific branching schemas have been developed, often based on the initial set of variables that allow efficient branch and price application [3].

Branch and Cut and Price[6]

For the last two decades principles have been investigated to combine branch and cut with branch and price. Problem specific applications were developed that use cutting planes as well as column generation at each node of the branch and bound decision tree, e.g. [2], [84] or [60]. Nevertheless, combining row and column generation techniques raises various challenges due to the dynamic nature of the relaxations, and the fact that adding constraints or variables can destroy the structure of the pricing or separation problem respectively [80].

2.3   Metaheuristics

In addition to the exact solving algorithms just reviewed there are many strategies to solve combinatorial optimization problems approximately. These strategies include problem-specific heuristic approaches as well as generic approximate solution schemas that are called metaheuristics [43]. The challenge in using metaheuristics is not to develop a fitting solution strategy from scratch, but to apply the metaheuristic principles to the combinatorial optimization problem.

Metaheuristics are therefore high level strategies for efficient exploration of search spaces. They can be characterized by the fact that they are approximate and usually non-deterministic, and they are not problem-specific. The search process is guided in such a way that, on the one hand, the whole search space is explored and, on the other, the accumulated search experience is exploited. Therefore, a metaheuristic has to find a balance between *diversification* and *intensification* of its search strategy [8], to identify efficiently regions with high quality solutions.

---

[6]The name for the branch and cut and price method varies in the literature. Alternatives include: *branch, cut and price* and *branch, price and cut* [27]

The guided search procedure of a metaheuristic usually uses some kind of neighborhood structure to explore the search space in a local search. Formally a neighborhood structure can be defined as [8]:

Definition 4 (Neighborhood Structure). A neighborhood structure is a function $N : S \longrightarrow 2^S$ that assigns to every $s \in S$ a set of neighbors $N(s) \subseteq S$, where $N(s)$ is called the neighborhood of s, and S is the search space which corresponds to the finite set of elements of the combinatorial optimization problem.

Based on that definition a local minimum with respect to a neighborhood structure N can be defined as a solution $s^* \in S$, whereas for all $s \in N(s^*) : f(s^*) \leq f(s)$. The function $f : S \longrightarrow \mathbb{R}$ corresponds to the cost function of the combinatorial optimization problem.

The basis for most metaheuristics is a local search step which finds a new local minimum from a given solution. Algorithm 2.7 shows a general local search algorithm[7].

Input: Solution s
Output: local minimum solution of s

1 **repeat**
2     $\acute{s} \longleftarrow$ select one element from $N(s)$, where $f(\acute{s}) < f(s)$
3   **if** *no such element exists* **then return** s // local minimum
4     $s \longleftarrow \acute{s}$
5 **until** *termination criterion*
6 **return** s // local minimum for all visited solutions

Algorithm 2.7: General Local Search Algorithm

Aspects of the following criteria can be used to characterize the different metaheuristics:

- Nature inspired vs. not nature inspired: a fairly intuitive but not very relevant characterization criteria is the original source of inspiration. Simulated annealing, genetic algorithms, and ant colony optimization try to take advantage of natural phenomena for efficient solving of combinatorial optimization problems.

- Population based search vs. single point search: If just a single solution is manipulated at each iteration the metaheuristic follows a single point search strategy. On the other hand, if multiple solutions are used concurrently then the search is based on a population of "individuals". In genetic algorithms the population is manipulated using the genetic operations, and in ant colony optimization a colony of ants uses pheromone trails to guide their search procedure.

- Trajectory method vs. discontinuous method: metaheuristics can be distinguished by whether their search procedure follows a single search trajectory, or whether large jumps are allowed that result in a discontinuous search pattern. Therefore, trajectory methods

---

[7]Actually, even local search itself is referred as one of the oldest and simplest metaheuristics, also known as hill climbing, iterative improvement, etc.

evolve by traversing the search space using the neighborhood of the current solution. Guided local search performs a local search and dynamically modifies the objective function to gradually move away from local minima. Tabu search remembers previously generated solutions to avoid them during the local search, and simulated annealing uses a random sampling of the current solution's neighborhood to come to a new solution. In iterated local search, variable neighborhood search, greedy randomized adaptive search procedure, genetic algorithms, or ant colony optimization starting points for a subsequent local search are generated corresponding to jumps in the search space.

- Dynamic vs. static objective function: metaheuristics can be classified according to whether they modify the evaluation of the points in the solution space dynamically by changing the objective function, or by using a static objective function. Guided local search modifies the objective function dynamically to avoid local minima. Also the tabu list of previous solutions in a tabu search can be interpreted as applying terms dynamically to the objective function with infinitely high values.

- Multiple vs. single neighborhood structures: the number of different neighborhood structures used in the metaheuristic can be used for classification. A typical representative of a multiple neighborhood strategy is the variable neighborhood search. In addition, the perturbation operation of the iterated local search to leave local optima can be interpreted as an operation in a second neighborhood structure. In genetic algorithms the mutation operation is also interpreted as movement in a second neighborhood structure.

- Memory usage method vs. memoryless method: an important criterion for classifying metaheuristics is the use of the search history by storing the search experience in memory. Tabu search and ant colony optimization explicitly use memory to store previously visited points in the search space or a pheromone structure representing the search paths of predecessor ants. Also the population of genetic algorithms can be interpreted as memory for the search experience.

Table 2.2 summarizes the classification of the selected relevant metaheuristics in its standard form [6], where ● means that this aspect is present, ◗ that this aspect is partially present, and ○ that this aspect does not characterize the metaheuristic. The metaheuristics are described in the remainder of this chapter closely following Talbi [90].

	ILS	GLS	TS	VNS	GRASP	SA	GA	ACO
Nature inspired	○	○	○	○	○	●	●	●
Population based	○	○	○	○	○	○	●	●
Trajectory method	○	●	●	○	○	●	○	○
Dynamic objective function	○	●	◗	○	○	○	○	○
Multiple neighborhood structures	●	○	○	●	○	○	◗	○
Memory usage	○	○	●	○	○	○	●	●

Table 2.2: Classification of Metaheuristics

Iterated Local Search

The *iterated local search* (ILS) metaheuristic calculates from an initial solution a local minimum by applying a local search. Since the quality of the local minimum depends mainly on the initial solution, the current local minimum is used to generate a new initial solution that can be optimized to a new local minimum. Therefore, the local search is iterated to explore a larger part of the search space [62].

Algorithm 2.8 shows the general ILS procedure. During each iteration a perturbation operation is applied to the current local minimum solution. This operation has to jump far enough in the search space to reach another local minimum from the subsequent local search. The palette of perturbation operations ranges from a simple randomized generation of new start solutions, to search history aware techniques that find new regions of attraction. After the local minimum has been found for the perturbed solution, it is compared to the current local minimum and consequently adopted or rejected for the next iteration. Also here different acceptance criteria are used: it is possible to accept any new solution, to accept solutions based on the comparison of the cost function values, or even to include the search history for the acceptance decision.

Input: Solution s
Output: local optimal solution of s by ILS

1 $s^* \longleftarrow$ local search of s
2 **repeat**
3     $s_p \longleftarrow$ perturbation of s^*
4     $s_p^* \longleftarrow$ local search of s_p
5     $s^* \longleftarrow$ accept or reject s_p^*
6 **until** *termination criterion*
7 **return** s^*

Algorithm 2.8: General Iterated Local Search Algorithm

A graphical interpretation of the basic operations of the ILS algorithm is shown in figure 2.5.

Guided Local Search

The *guided local search* (GLS) metaheuristic uses a different strategy to overcome local minima. It dynamically changes the evaluation of the cost function according to the current local minimum [96].

To achieve this a set of solution features FT is defined. Each feature is associated with a cost and a penalty. The cost should reflect the influence of the presence of this feature to the cost function; the penalty represents the importance of the feature and is determined during the algorithm. For routing problems the feature can be associated with the presence of an arc, the costs correspond to the arc-costs. For a given local optimum solution s^* a utility is calculated for each feature $ft \in s^*$: $u_{ft}(s^*) = \frac{c_{ft}}{1 + p_{ft}}$, where c_{ft} is the associated cost and p_{ft} the associated penalty. The feature with the highest utility is then penalized by increasing the associated penalty. The cost function of the problem is modified for s^*, so that $\hat{f}(s^*) = f(s^*) + \lambda \sum_{ft \in s^*} p_{ft}$, where λ represents a weight factor. Algorithm 2.9 shows the general GLS procedure.
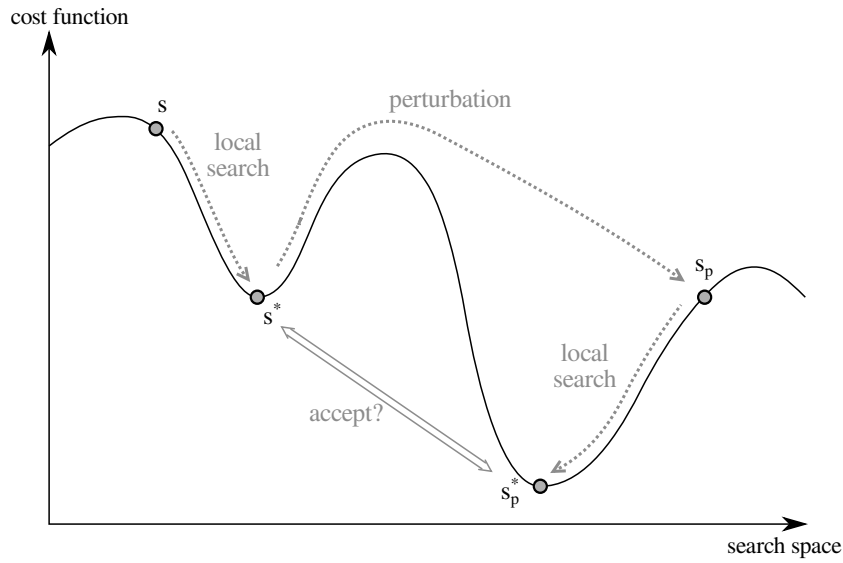
Figure 2.5: Graphical Interpretation of the basic operations of ILS

Input: Solution s, weight λ
Output: best solution found by GLS

1 $p_{ft} \longleftarrow 0 \; \forall \; ft \in FT$
2 **repeat**
3 $\quad$ $s^* \longleftarrow$ local search of s using cost function $\hat{f}(s) = f(s) + \lambda \sum_{ft \in s} p_{ft}$
4 $\quad$ $u_{max} \longleftarrow 0$
5 $\quad$ **foreach** $ft \in s^*$ **do**
6 $\quad\quad$ **if** $\frac{c_{ft}}{1+p_{ft}} > u_{max}$ **then** $ft_{max} \longleftarrow ft; u_{max} \longleftarrow \frac{c_{ft}}{1+p_{ft}}$
7 $\quad$ **end**
8 $\quad$ $p_{ft_{max}} \longleftarrow p_{ft_{max}} + 1$
9 $\quad$ $s \longleftarrow s^*$
10 **until** *termination criterion*
11 **return** s^*

Algorithm 2.9: General Guided Local Search Algorithm

Figure 2.6 shows the graphical interpretation of the GLS algorithm. The penalization operator "raises" the cost function around the solution s^*. Therefore subsequent local search operations can find a new local optimum not influenced by the penalty.

Tabu Search

The next metaheuristic uses memory to escape from the valley surrounding a local minimum. It was developed concurrently as *tabu search* [44] (TS) and as *steepest ascent mildest descent heuristic* [49]. The idea is to allow the local search algorithm to decrease the cost function
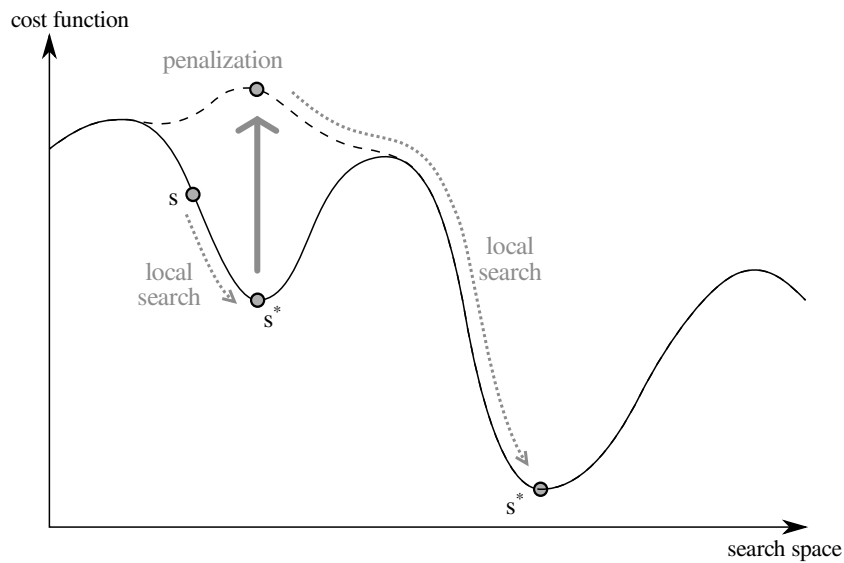
Figure 2.6: Graphical Interpretation of the basic operations of GLS

value if no improvement is possible. To avoid circularity a memory structure that is known is the tabu list stores solutions that have been visited previously. Instead of storing solutions it is also possible to store moves that cannot be undone, for example for a routing problem an added vertex could be marked as non-removable by adding it to the tabu list. An aspiration criterion has to decide if the tabu move is really forbidden because it could lead to a new best solution in a later state of the algorithm.

The tabu list itself can be interpreted as short term memory because it is limited and "forgets" tabu elements over time. Also other memories can be introduced into the algorithm like a medium term memory that stores best solutions to intensify searching, or a long term memory that stores information of unexplored regions in the search space to diversify searching. Algorithm 2.10 shows the basic tabu search algorithm without additional memories.

Input: Solution s
Output: best solution found by TS

1 $TL \longleftarrow \{\}$ `// tabu list`
2 **repeat**
3     $\acute{s} \longleftarrow$ select one element from $N(s) \backslash (\text{aspirated } TL)$, where $f(\acute{s}) < f(s)$
4   **if** *no such element exists* **then** $\acute{s} \longleftarrow$ select one element from $N(s)$
5   update TL with s or the move $s \mapsto \acute{s}$
6     $s \longleftarrow \acute{s}$
7 **until** *termination criterion*
8 **return** s

Algorithm 2.10: Basic Tabu Search Algorithm

A graphical interpretation of the basic TS procedure is shown in figure 2.7. The tabu list enables the algorithm to climb up the slope of the valley surrounding the local minimum without falling back.
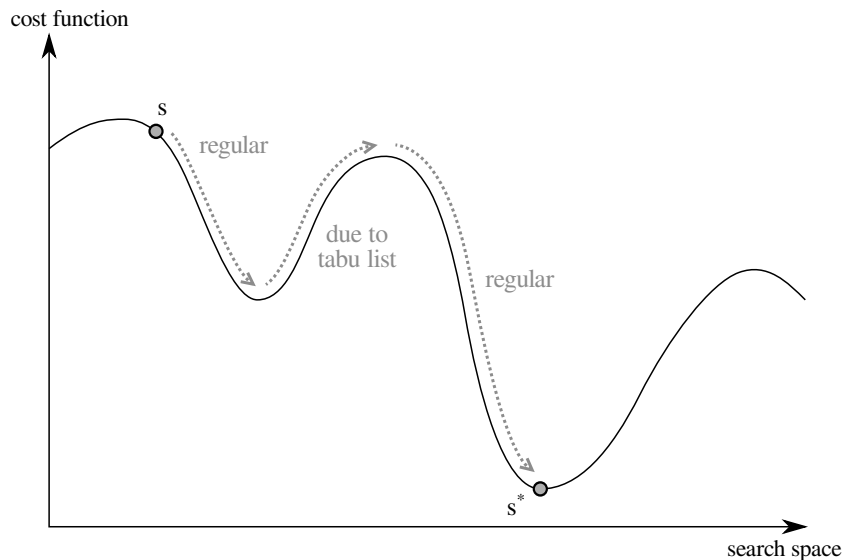

Figure 2.7: Graphical Interpretation of the basic TS

Variable Neighborhood Search

The basic idea of the *variable neighborhood search* (VNS) metaheuristic is to use multiple neighborhood structures to avoid being caught in a local minimum of a single neighborhood structure [64]. The core of the VNS is the variable neighborhood descent, a deterministic search algorithm that changes the neighborhood structure when a local minimum is reached.

First the neighborhood structures have to be defined; then the algorithm starts with the first neighborhood structure and descends to a local minimum. Now the second neighborhood structure is activated and a local minimum is sought for both the second and the first neighborhood structure, before the algorithm switches to the third neighborhood structure. Algorithm 2.11 shows the variable neighborhood descent.

Figure 2.8 shows a graphical interpretation of the basic variable neighborhood decent operations. The figure contains three neighborhood structures. Starting from solution s_1 a local search with respect to neighborhood structure 1 is performed leading to solution s_1^*. Then the algorithm switches to neighborhood structure 2, which means that the following local search is now performed in the new neighborhood structure, and so on. Note that $s_1^* = s_2$ and $s_2^* = s_3$, because the algorithm is just adjusted to a new neighborhood structure, and this operation does not modify the solution.

The general VNS algorithm is a non-deterministic variant of the variable neighborhood descent. It consists of three steps in each iteration. The first step is the shaking operation which randomly generates a new solution $\acute{s}$ using the current neighborhood structure. The next step

Input: Solution s
Output: best solution found by variable neighborhood descent

1 $l = 1$ // first neighborhood structure
2 **while** $l \leq l_{max}$ **do**   // search using all neighborhood structures
3     $\acute{s} \longleftarrow$ select one element from $N_l(s)$, where $f_{N_l}(\acute{s}) < f_{N_l}(s)$
4   **if** *no such element exists* **then**
5        $l \longleftarrow l + 1$ // switching to next neighborhood structure
6   **else**
7        $s \longleftarrow \acute{s}$
8        $l \longleftarrow 1$
9   **end**
10 **end**
11 **return** s

Algorithm 2.11: Variable Neighborhood Descent Algorithm

performs a local search starting from $\acute{s}$ that generates $\acute{s}^*$ replacing the current solution s, if $f(\acute{s}^*) < f(s)$. The last step is the move operation, which proceeds to the next neighborhood structure if no better solution was found, or which starts again with the first neighborhood structure if the solution was improved.
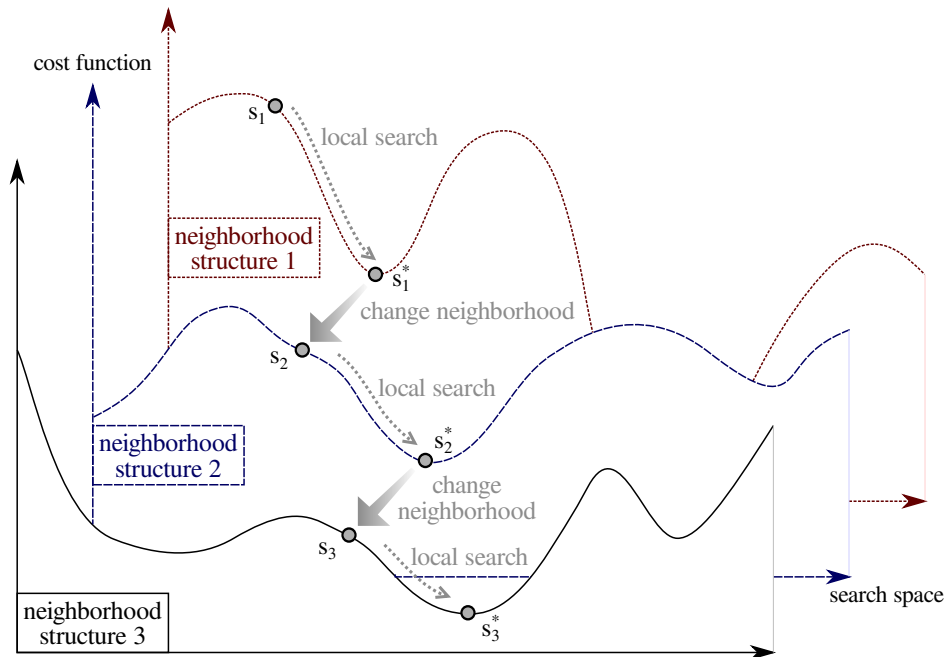

Figure 2.8: Graphical Interpretation of the basic variable neighborhood descent operations

Greedy Randomized Adaptive Search Procedure

The *greedy randomized adaptive search procedure* (GRASP) is an iterative greedy heuristic consisting of two basic steps [36]: a randomized greedy algorithm that generates a feasible solution, and a subsequent local search. The algorithm contains no memory, even the solution of the previous iteration is not remembered, so only the best solution detected after a certain number of independent iterations is returned.

To generate a feasible solution the randomized greedy algorithm has to construct the solution in keeping with a greedy heuristic. For each construction step only solution elements come into consideration that promise the best improvement for the current solution state. For example, these solution elements could be the arcs with the lowest costs in a routing problem; they are stored in a restricted candidate list. One element is selected randomly from this list and added to the current solution state as long as the resulting solution is feasible. Algorithm 2.12 shows the GRASP containing a randomized greedy algorithm and the local search.

Input:
Output: best solution found by greedy randomized adaptive search procedure

1   $s_{best} \longleftarrow$ null
2   **repeat**
       // randomized greedy algorithm
3      $s \longleftarrow \{\}$
4   **repeat**
5        $RCL \longleftarrow$ restricted candidate list of solution elements
6        $e \longleftarrow$ random solution element of RCL
7   **if** $s \cup e$ *is feasible* **then** $s \longleftarrow s \cup e$
8   **until** s *is a complete solution*
       // local search
9      $s^* \longleftarrow$ local search of s
10   **if** s_{best} *is null or* $f(s_{best}) > f(s^*)$ **then** $s_{best} \longleftarrow s^*$
11   **until** *termination criterion*
12   **return** s_{best}

Algorithm 2.12: Greedy Randomized Adaptive Search Procedure

Simulated Annealing

Simulated annealing (SA) is a metaheuristic inspired by nature which simulates the effect of heating and slowly cooling crystalline structures like metal to reach a state of minimal molecular energy. The principle was applied to combinatorial optimization in the early 1980s [55]. The idea of this stochastic algorithm is to accept degradation of the current solution based on a probabilistic function where the probability of acceptance decreases over time.

The probability function P depends on a parameter T, the actual "temperature", where $P(f(s), f(\acute{s}), T) = \mathrm{e}^{-\frac{f(\acute{s})-f(s)}{T}}$, and s and $\acute{s}$ are the current and new solution. T is decreased over time to decrease the probability of accepting worse solutions. Improving solutions are

always accepted. Algorithm 2.13 shows the general simulated annealing algorithm for combinatorial optimization. The algorithm allows the design of an acceptance function for degraded solutions, as well as a cooling schedule consisting of a starting temperature T_{max} and a temperature decreasing operation. Additionally, the equilibrium condition has to be designed to allow the algorithm to iterate using a constant temperature.

Input: Solution s
Output: best solution found by simulated annealing

1 $T \longleftarrow T_{max}$ `// starting temperature`
2 repeat
3   **repeat**
4 $\acute{s} \longleftarrow$ select randomly one element from $N(s)$
5 $P \longleftarrow e^{-\frac{f(\acute{s}) - f(s)}{T}}$
6   **if** $f(\acute{s}) < f(s)$ *or accepting $\acute{s}$ with probability P* **then**
7 $s \longleftarrow \acute{s}$
8   **end**
9   **until** *equilibrium condition*
10 $T \longleftarrow$ decreased temperature
11 until *termination criterion*
12 return s

Algorithm 2.13: General Simulated Annealing Algorithm

Genetic Algorithms

Another nature-inspired metaheuristic is represented by *genetic algorithms* (GA). Originally developed to understand natural processes [51] they have been adapted to solve combinatorial optimization problems [23]. GAs are population based metaheuristics. They are based upon a set of chromosomes, where a chromosome is an encoding of a feasible solution in the problem's search space. A chromosome consists of a string of codes also called genes that represent the coded solution elements.

GAs consist of three operations as shown in algorithm 2.14: the selection of parent chromosomes, the reproductive crossover operator to generate offspring, and a mutation operator to diversify the population. Selection depends on the fitness values of the chromosomes. An increase in the fitness of a chromosome favors its selection for reproduction. Typical selection methods are roulette wheel selection or tournament selection. Usually the fitness of a chromosome corresponds to the cost function value of its encoded solution. The crossover operator is a simple swap operation on parts of the chromosome's strings of genes. The role of crossover is to pass on characteristics of the parents to the offspring. Depending on the encoding schema the offspring sometimes has to be "repaired" to represent a valid solution. The mutation operator applies small random changes on the gene string of arbitrarily selected chromosomes. This serves to keep the gene pool of the population diverse enough to escape local minima.

Input:
Output: best solution found by genetic algorithm

1 $P \longleftarrow$ set of random chromosomes   `// initial population`
2 $s_{best} \longleftarrow$ best solution of decoded P
3 repeat
4 | $p_1, p_2 \longleftarrow$ select two individuals from P probabilistically based on their fitness $f(p)$
5 | $\acute{p}_1, \acute{p}_2 \longleftarrow$ crossover chromosomes of p_1 and p_2
6 | $P \longleftarrow P \backslash \{p_1, p_2\} \cup \{\acute{p}_1, \acute{p}_2\}$ `// replace parents with offspring`
7 | $P \longleftarrow$ mutate randomly chromosomes in P
8 | $s_{best} \longleftarrow$ best solution of decoded $P \cup s_{best}$
9 until *termination criterion*
10 return s_{best}

Algorithm 2.14: Basic genetic algorithms procedure

Ant Colony Optimization

Nature inspired algorithms that try to imitate collective behavior of species living in colonies are called swarm intelligence algorithms. The *ant colony optimization* (ACO) metaheuristic is modeled on the cooperative behavior of real ants.

The inspiration was taken from the mechanisms which allow ant colonies to find the shortest paths to food sources. Ants do not directly communicate to each other for this process; they modify the environment by releasing pheromones to the ground. Ants searching for food preferentially follow paths with higher pheromone concentration than paths with lower pheromone concentration. Observable is that speed and rate of pheromone release is almost constant over all individuals. Shorter paths are simply "rewarded" by the fact that ants following those paths return earlier to the nest, which increases the concentration of those paths more quickly. Therefore a simple ruleset for the individuals is necessary to generate a successful cooperative solution strategy – a phenomenon that is characteristic for swarm intelligence.

In experiments with Argentine ants these mechanisms where demonstrated on an experimental setup called the double bridge experiment [24] [47]. Figure 2.9 shows a schematic of a variant of those experimental setups. It was demonstrated that although the ants choose their path in the beginning randomly over the shorter or the longer bridge after some time the shorter bridge becomes the favorite alternative. This is because the pheromone concentration rises faster for the shorter path. In fact, the longer bridge was abandoned completely during the experiment.

There are additional effects that are mimicked by ACO algorithms: pheromones vaporize over time, and the pheromone amount released during the return to the nest depends on the quality of the food source.

The ACO metaheuristic was first proposed by Marco Dorigo 1992 in his PhD thesis [28]. The basis of his work is the *ant system* which simulates most of the principles described. The idea was developed further leading to the *ant colony system* introduced by Dorigo and Gambardella 1997 [30]. Another improvement was the $\mathcal{MAX}$–$\mathcal{MIN}$ *ant system* proposed by Stützle and Hoos 2000 [89]. However, all these developments make use of the basic principles inspired by real ant behavior. ACO metaheuristics were applied successfully to a huge range of
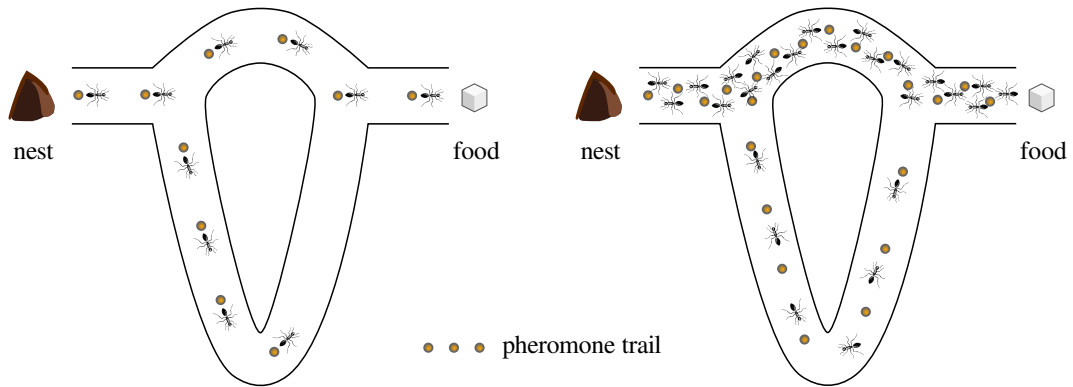
Figure 2.9: Double Bridge Experiment

problems, such as routing, scheduling, or assignment problems [32]. Nevertheless the principles of ACO can be illustrated best with problem formulations that require the solution for the shortest path in a graph.

The basic ACO framework is shown in algorithm 2.15. It has to be noted that an artificial ant is just an agent, constructing solutions and modifying as well as being influenced by the environment. $\mathcal{T}$ is the memory of the ACO algorithm: it stores the pheromone trails and is changed by evaporation and reinforcement processes.

Input:
Output: best solution found by ant colony optimization

1 $s_{best} \longleftarrow$ null
2 $\mathcal{T} \longleftarrow$ initialize pheromone trails
3 **repeat**
4      $A \longleftarrow \{\}$ // solutions of ant colony
5   **for** *each ant* **do**
6          $s \longleftarrow$ construct solution using $\mathcal{T}$
7          $s^* \longleftarrow$ local search of s // optional
8   **if** s_{best} *is null or* $f(s_{best}) > f(s^*)$ **then** $s_{best} \longleftarrow s^*$
9          $A \longleftarrow A \cup s$
10   **end**
       // pheromone update
11     $\mathcal{T} \longleftarrow$ evaporate pheromones
12     $\mathcal{T} \longleftarrow$ reinforce pheromone trails using A and s_{best}
13 **until** *termination criterion*
14 **return** s_{best}

Algorithm 2.15: Basic ant colony optimization framework

The construction process of an ant can be considered as a greedy heuristic procedure in a probabilistic manner. It is influenced by two factors: the problem dependent heuristic informa-

tion, and the pheromone trails that memorize the behavior of previous ants. The construction starts with an empty partial solution. At each step the solution is extended by adding a solution component from the set of candidates, where only elements are considered that generate feasible solutions. The choice of a solution component is probabilistic, and is biased by the pheromone trails. The probability of choosing a solution component is calculated differently for the various ACO algorithms, but they follow in principle the following formula:

$$P_e = \frac{\tau_e{}^\alpha \times \eta_e{}^\beta}{\sum\limits_{c \in CL} \tau_c{}^\alpha \times \eta_c{}^\beta}, \quad \forall e \in CL$$

where τ_e is the pheromone value associated with the solution element e and η_e is a value that represents the attractiveness of this element based on the problem dependent heuristic. This could be, for example, the reciprocal of the distance of a city for the TSP. $\alpha \in [0, \infty[$ and $\beta \in [0, \infty[$ are parameters which determine the influence of the pheromone values and problem dependent heuristic respectively. If $\alpha = 0$ then the construction process becomes a stochastic greedy algorithm, if $\beta = 0$ only the pheromone trails guide the construction. Algorithm 2.16 shows the construction process.

Input: pheromone trails $\mathcal{T}$
Output: solution constructed by an artificial ant

1 $s \longleftarrow \{\}$
2 **repeat**
3 $\quad CL \longleftarrow$ candidate list of allowed solution elements
4 $\quad P_{CL} \longleftarrow$ evaluate probabilities of candidates $P_e = \frac{\tau_e{}^\alpha \times \eta_e{}^\beta}{\sum_{c \in CL} \tau_c{}^\alpha \times \eta_c{}^\beta}, \ \forall e \in CL$
5 $\quad e \longleftarrow$ stochastic choice of solution element $\in CL$ based on P_{CL}
6 **until** *s is a complete solution*
7 **return** s

Algorithm 2.16: Construction of a solution by an artificial ant

After constructing a feasible solution it can be enhanced with a local search. Although this step is optional, it is usually included in state-of-the-art ACO implementations [29].

The last step of an ACO algorithm is the updating of the pheromone values. Its primary function is to increase those values that promise high quality solutions by exploiting the search history. A secondary function is the global reduction of pheromone values, on the one hand to "forget" solution elements no longer in use, and on the other hand to avoid a too rapid convergence toward suboptimal regions. This evaporation procedure is performed as follows:

$$\tau_c = (1 - \rho)\tau_c, \quad \forall \tau_c \in \mathcal{T}$$

where $\rho \in [0, 1]$ is the evaporation rate. The reinforcement procedure to increase pheromone values is simply

$$\tau_c = \tau_c + \Delta \tau_c, \quad \forall \tau_c \in \mathcal{T}$$

where the characteristic of the ACO algorithm is manifested in the term $\Delta\tau_c$. For the original algorithm using the *ant system* the reinforcement is just influenced by the artificial ants of the current iteration, therefore $\Delta\tau_c = \sum_{s_a \in A} \Delta\tau_{c,a}$, where $\Delta\tau_{c,a} = 1/L_a$ if solution element c is part of solution s constructed by ant a, 0 otherwise. L_a corresponds to the total costs of solution s_a, it could be for example the tour length of this solution for the TSP. This schema ensures that those pheromone values are increased more rapidly that represent solution elements which are part of better solutions. An *ant colony system* modifies the reinforcement rule by introducing a local update schema. Here each artificial ant modifies immediately the pheromone trails by $\tau_c = (1-\varphi)\tau_c + \varphi\tau^0$ if solution element c is part of the ants solution. τ^0 is the initial pheromone value and $\varphi \in [0,1]$ is the pheromone decay coefficient. This leads to a diversification of the search, since pheromone values are decreased for solution elements that are used by the actual artificial ant, which lowers the attractiveness of these elements for subsequent ants. To balance the algorithm additionally an intensification operation is performed in the global reinforcement procedure. Here $\Delta\tau_c = 1/L_{best}$ if solution element c is part of the solution s_{best}. The $\mathcal{MAX}$–$\mathcal{MIN}$ *ant system* uses the same global reinforcement procedure, but it additionally limits the values of the pheromones to $[\tau_{min}, \tau_{max}]$. No local pheromone update is made, and the initial pheromone values are all set to τ_{max}. With this rule the pheromone values are only decreased by evaporation; if a solution element is part of the best solution it eludes this process. There are more variants of pheromone update rules, for example not only the best solution, but also the n best solutions – the so called elitist ants – can bias the pheromone values. Or instead of using the best-so-far solution s_{best} the iteration-best solution or even a combination of both is used.

One critical factor for successfully applying ACO is the careful initialization of the numerous parameters of the algorithm. Table 2.3 shows the main parameters of the basic ACO algorithm. To find an efficient set of parameter values, elaborate research on the optimization problem is necessary. Fortunately lot of previous work gives guidance and presents experience on successful application.

Parameter	Role	Practical range
α	Influence of pheromone values	$]0, 20]$
β	Influence of problem dependent heuristic	$]0, 20]$
ρ	Evaporation rate of pheromones	$]0, 0.2]$
φ	Pheromone decay coefficient	$[0, 1[$
n	Number of artificial ants	$[1, 100]$

Table 2.3: Main parameters of the basic ACO algorithm

2.4 Hybridization

The previous chapters indicate that there are many algorithms that provide exact as well as approximate solutions for combinatorial optimization problems. Especially over the last decade new strategies have been developed that combine those methods and algorithms. These new strategies can be named hybrid metaheuristics, since they combine mainly specific metaheuristics with other or even the same metaheuristic algorithms, or with exact solution algorithms.

To categorize these hybrids the following aspects can be used that characterize hybrid metaheuristics [79]:

- What is hybridized: Metaheuristics with metaheuristics, with exact methods, or with other algorithms (problem specific and/or heuristic)

- Level of hybridization: high-level with weak coupling vs. low-level with strong coupling

- Control strategy: integrative or collaborative hybridization

- Order of execution: sequential vs. parallel or intertwined execution

To give an overview of the relevant classes of hybrid metaheuristics we use the classification schema proposed by Puchinger and Raidl [78]. The schema was originally presented for combinations of metaheuristics with exact algorithms only (figure 2.11), for this overview we have adopted it to also classify combinations of metaheuristics with metaheuristics (figure 2.10).

Combining metaheuristics with metaheuristics
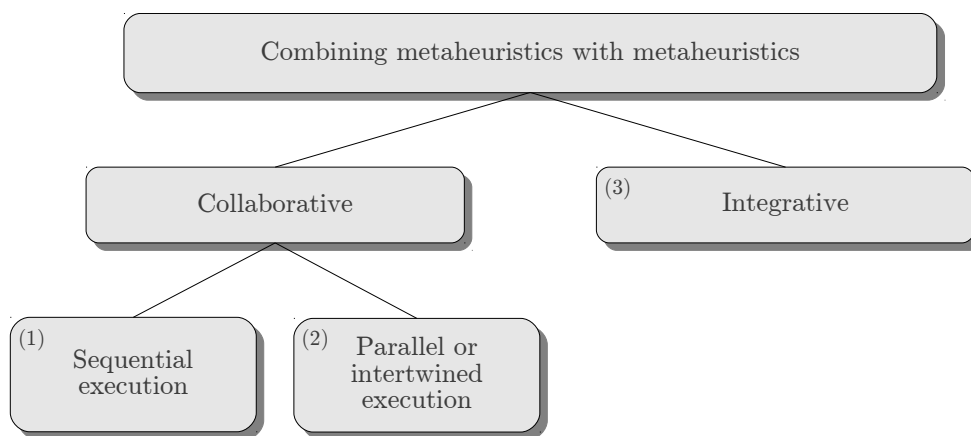
Figure 2.10: Classification of metaheuristics with metaheuristics combinations

Strategies that combine metaheuristics with metaheuristics can be sub-divided into three classes: (1) collaborative combinations with sequential execution, (2) collaborative combinations with parallel or intertwined execution, and (3) integrative combinations.

Hybrids that belong to the first class (1) typically collaborate on a high level. The subsequent metaheuristic algorithms have to wait for the results of preceding algorithms. For example the initial solution of a single point search metaheuristic can be generated by a preceding metaheuristic algorithm. Similarly the initial population for a population based metaheuristic can also be constructed by other metaheuristic algorithms. For some problems with complex search spaces it is reasonable to use a population based metaheuristic to quickly find a high-performance region, and then switch to a more specialized search metaheuristic to "fine tune" the solution.

For the second class (2) the metaheuristic algorithms work in parallel or at least partly parallel. Many strategies in this class apply a homogeneous approach to run multiple instances of the same metaheuristic in parallel to increase performance and exploration force. This requires some kind of communication framework to exchange information among the algorithms. A quite prominent example is the island model for GA. Here several GAs run in parallel, and individuals can move under certain circumstances from one GA to another. This hybridization model was also applied to various other metaheuristics [90].

The last class (3) addresses integrative combinations of metaheuristics. These approaches distinguish themselves by embedding a metaheuristic in another metaheuristic algorithm that becomes the master algorithm for this incorporation. This can be done by replacing an operation of the master algorithm with the embedded metaheuristic, or by simply adding the functionality of the embedded metaheuristic to the master algorithm. For example, the local search operation of most metaheuristics can be replaced by a more explorative search algorithm. Moreover, it is possible to modify single operations of population based metaheuristics by adding a metaheuristic algorithm. For instance the mutation operation of a GA could be changed to additionally perform a metaheuristic search operation. The problem with this approach is a premature convergence of the population, that is why it should be used with care.

Combining metaheuristics with exact algorithms[8]
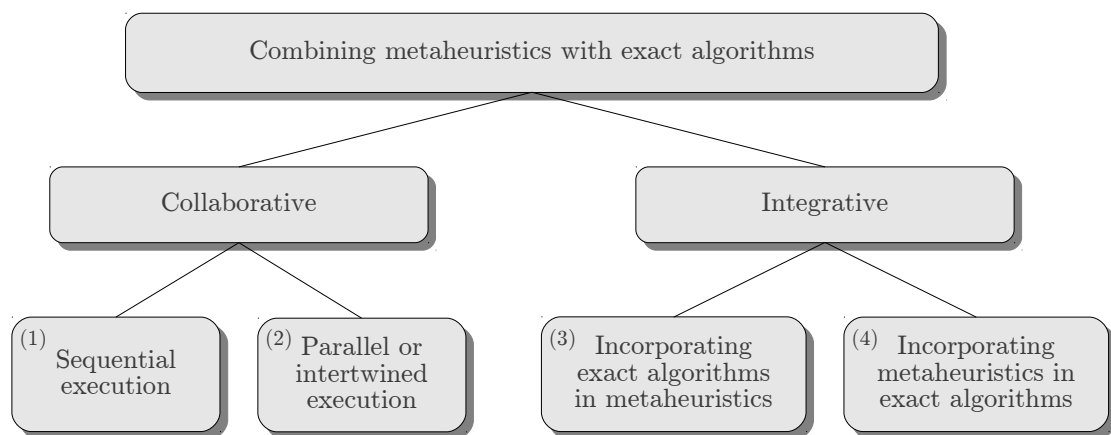

Figure 2.11: Classification of metaheuristics with exact algorithms combinations

There are two main motivations for hybridizing metaheuristics and exact algorithms. On the one hand exact algorithms may improve their performance by combining them with metaheuristics. On the other hand metaheuristics may find solutions of better quality by combining them with exact algorithms.

To classify these combinations we distinguish (1) collaborative combinations with sequential execution, (2) collaborative combinations with parallel or intertwined execution, (3) integra-

---

[8]The hybridization of metaheuristics and exact algorithms is also referred as *matheuristics*, based on the conjunction of the terms *mathematical programming* and *metaheuristics*

tive combinations where exact algorithms are incorporated in metaheuristics, and (4) integrative combinations where metaheuristics are incorporated in exact algorithms.

The first class (1) of cooperation comprehends hybrids, where metaheuristics and exact algorithms are used in sequence to provide some kind of information to the other. Information that can be provided to metaheuristics by exact algorithms are partial solutions that can be completed by the metaheuristic, a lower bound for the problem, optimal solutions for relaxed problems that can be exploited by the metaheuristic, and reduced problems in the form of simplified objective functions. Information that can be provided to exact algorithms by metaheuristics provides a good upper bound for a subsequent bounding algorithm, and an initial feasible solution to omit phase I of a subsequent LP-relaxed simplex algorithm. Also multiple metaheuristic calls can be used as a column generator for a set of diverse solutions.

Combinations of the second class (2) are characterized by a parallel cooperative execution of metaheuristics and exact algorithms. Only a few strategies belong to this class. A parallel cooperation between a branch and bound algorithm and a metaheuristic is possible where the metaheuristic delivers regularly upper bounds to increase the pruning performance of the branch and bound. Also other forms of cooperation are proposed where asynchronous teams work on the target problem, on subproblems, or on reduced and relaxed problems. The agents use a shared memory to exchange information and especially discovered solutions.

Strategies of the third class (3) comprise cooperation techniques where a metaheuristic works as a master algorithm for an embedded exact algorithm. For example, this exact algorithm can solve the relaxed problem and its dual to guide operations in the metaheuristic such as neighborhood search, recombination, mutation or repair actions. It is also possible to perform a neighborhood search exactly when choosing the neighborhood structures appropriately. This method is also known as very large scale neighborhood search. Exact algorithms can also be used in merging solutions and finding best combinations, for example to be intercorporated into the crossover operator of a GA. Finally, exact algorithms can be used during the decoding of incomplete solution representations. Therefore, problems can be coded in a way that leaves a part of the problem for the decoding of the result, and solves the rest of the problem in the metaheuristic.

The last class (4) consists of hybrids which use exact algorithms as master algorithm and embed metaheuristics into it. Here the metaheuristic can be used to generate solutions and upper bounds, especially for embedding in a branch and bound environment. Also metaheuristics can be used to speed up branch and cut and branch and price algorithms by working as a generator for cutting planes or as a column generator. Especially the column generation approach is reported as promising for speeding up the whole optimization process [83].

A survey on existing approaches for combining metaheuristics and exact algorithms is given in [78].

3

Related Work

The PVRPTW was introduced by Cordeau et al. in 2001 [14] as an important generalization of the VRPTW. The highly constrained routing problems were solved in this paper with a tabu search algorithm that allowed intermediate infeasible solutions. This method was improved by the authors in 2004 [15] by presenting an enhanced version of their tabu search algorithm that uses slack times to delay the start of a vehicle which makes more routes feasible. The concept of forward slack times to minimize route durations for the VRPTW was introduced by Savelsberg [85].

About Solving the PVRPTW

Beside the aforementioned tabu search method by the originators of the PVRPTW only a few works deal with solving this variant of vehicle routing problems. Only in the last year has the scientific community devoted more attention this problem.

In 2008 Pirkwieser and Raidl [73] introduced a variable neighborhood search algorithm for the PVRPTW. This method was improved in 2009 by the authors with two enhancements: in the first enhancement [75] the VNS was hybridized with an exact method that solved one part of the problem with branch and bound. The exact solver was fed with routes of feasible solutions from the VNS and returned optimal visit combinations using a collaborative intertwined hybridization schema. The second enhancement [76] introduced a multiple VNS technique where several VNS instances run concurrently and exchange at defined points in time information of the best solution found so far. These multiple VNS instances are similarly hybridized with a branch and bound algorithm as explained above.

A column generation approach was proposed by Pirkwieser and Raidl in 2009 [74] to obtain strong lower bounds. Therefore, a set-covering formulation was introduced for the PVRPTW, and the LP-relaxed problem was solved using an exact LP solver. The pricing subproblem was determined by the formulation as an elementary shortest path problem with resource constraints. For its solution a dynamic programming approach was described as one method, and a GRASP metaheuristic was implemented as a complementary method to increase performance.

A different hybridization technique for the PVRPTW was proposed by Pirkwieser and Raidl in 2010 [77]. To the method already mentioned in [75] and [76] an algorithm is described where the solutions derived by column generation for the LP-relaxed problem of the set-covering formulation of the PVRPTW are used to initialize the chromosomes of an evolutionary algorithm. This type of hybridization can be considered as a collaborative sequential combination of an exact algorithm with a metaheuristic.

Yu and Yang [97] published in 2011 the application of an improved ant colony optimization algorithm to the PVRPTW. The authors use a pheromone structure that stores pheromones for each day and arc of the problem. They present two crossover operations to implement a local search improvement for solutions generated by single ants. Each ant updates the pheromone structure, where the pheromone increment depends on the relative solution quality and a punishment coefficient that penalizes infeasible solutions violating the fleet constraint.

A recent paper of Nguyen et al. [67] reports on the successful implementation of a hybrid genetic algorithm for solving the PVRPTW which uses the tabu search metaheuristic proposed by Cordeau et al. [14] and the variable neighborhood search metaheuristic proposed by Pirkwieser and Raidl [73] to improve offspring chromosomes in the GA population. With this hybridization approach two metaheuristics are embedded into a GA in an integrative manner.

Cordeau and Maischberger also recently [16] proposed a hybrid of iterated local search and tabu search for solving various VRP variants. The hybrid adds a perturbation operation to a tabu search metaheuristic and is implemented as a parallel algorithm that uses a multi-start environment and shares knowledge of the solutions at predefined times. The algorithm was tested on the PVRPTW and a wide range of other VRP variants.

The most recent work on the PVRPTW comes from Vidal et al. [95]. The authors propose a hybrid genetic algorithm similar to the algorithm introduced by Nguyen et al. but improving it by adding adaptive diversity management. The whole algorithm works with two populations, one for feasible and one for infeasible solutions that are allowed to violate capacity, duration, or time window constraints. Diversity management consists of dynamic penalty adaption for infeasible solutions, a survivor selection strategy, and an explicit quantification of individual diversity contribution. The algorithm presented is designed to solve variants of routing problems with time windows, including the PVRPTW.

About Column Generation Approaches

In 1992 Desrochers et al. [25] wrote an article that can be regarded as a breakthrough for exact solving vehicle routing problems by applying a column generation approach. In fact, the authors used column generation for the VRPTW to solve the LP-relaxed problem that was formulated as a set-covering model. They showed that this technique provided excellent lower bounds for embedding it into a branch and bound algorithm. The master problem was solved with the simplex algorithm, and the column generating subproblem was formulated as the shortest path problem with resource constraints solved by dynamic programming.

Demonstrating the application of column generation to the VRPTW was also the aim of the work of Danna and Le Pape published in 2005 [18]. They embedded their column generation algorithm in a modified variant of branch and bound which utilizes additional local search operations before branching. In contrast to Desrochers they solved the subproblem as an elementary

shortest path problem with resource constraints. For that a modified dynamic programming approach was introduced.

In 2007 Mourgaya and Vanderbeck [65] applied column generation to the PVRP. They used a tactical planning model with two optimization criteria: workload balancing and regionalization. The model was reformulated with Dantzig-Wolfe decomposition to be efficiently solved with column generation. The pricing subproblem was formulated as the linearized form of the quadratic knapsack problem that was solved using a greedy heuristic. The whole problem was solved iteratively by solving the LP-relaxed restricted problem with column generation, then applying a rounding heuristic that produces a feasible solution for the original problem which is returned to the solving procedure of the LP-relaxed restricted problem.

A VRP variation with a combination of pickup and delivery and time windows was the focus of an article by Ropke and Cordeau in 2009 [84]. They introduced a branch and cut and price algorithm for the pickup and delivery problem with time windows. For the calculation of the lower bound column generation was applied with a set partitioning formulation of the problem. Two different subproblem variants were compared: an elementary shortest path problem with resource constraints, pickup and delivery; and a shortest path problem with resource constraints, pickup and delivery. Finally the authors recommend using the first variant – at least for the considered problem instances – since it allows stronger lower bounds and it seems equally hard to solve.

In 2009 Pirkwieser and Raidl [74] proposed the aforementioned column generation approach for the PVRPTW.

About Ant Colony Optimization Approaches

Starting with the application of ACO to the TSP at its first introduction by Dorigo et al. [31], a lot of publications have been written that use this metaheuristic to solve a wide range of combinatorial optimization problems, including many variations of routing problems.

In 1999 Gambardella et al. [42] introduced MACS-VRPTW, a multiple ant colony system for the VRPTW. Two concurrently running ant colony systems perform two different optimization tasks: one system is responsible for the minimization of the total costs, whereas the other system reduces the number of vehicles used. The two systems communicate by updating the pheromones of the other system when a new best solution is found.

In 2002 Dörner et al. [33] presented a new construction technique for ACO solving the VRP. The authors applied the savings algorithm proposed by Clarke and Wright [10] to the construction step of their ant system based VRP solver. For this purpose the routes are not constructed by adding customers to an incomplete route using a probabilistic nearest neighborhood heuristic; instead routes are merged based on the savings of the mergers in a probabilistic manner.

An application of ACO to the PVRP was published by Matos and Oliveira in 2004 [63]. The authors propose a two phased approach. In the first phase a savings based ant system algorithm creates routes for a modified problem where the customers are duplicated by their service frequency. In this way they generate a large scale VRP solved by ACO. In the second phase the routes generated by the first phase are analyzed to determine if they can be serviced on the same day regarding multiple customer visits. A graph coloring problem is constructed and an exchange mechanism is implemented to assign the routes to the days in the planning horizon

according to the visit combinations of the customers. Finally a VRP is solved for each single day using the ant system based ACO of the first phase.

A decomposition approach for large VRPs was proposed by Reimann et al. in 2004 [82]. Here the VRP is decomposed into several TSPs that are solved separately. The iterative algorithm uses a savings based ant system as the core component and consists of the following steps during one iteration: the whole VRP is solved with ACO to find a good set of routes. These routes create separation clusters for the problem and each customer is assigned to a cluster after this step. Then for each cluster a TSP is solved with ACO. Finally, the resulting routes are joined to create a total feasible solution that modifies the pheromones of the ACO for the whole VRP problem.

In 2006 Zhang et al. [98] applied ant colony system based ACO to the vehicle routing problem with time windows and re-used vehicles. The authors combine pheromone update strategies of the ant colony system, $\mathcal{MAX}$–$\mathcal{MIN}$ ant system and a rank based version of the ant system for their approach. Additionally they propose a construction heuristic for the single ants which prefer to visit customers first, that have earlier starts of service begin time, shorter service durations, and earlier ends of service begin time.

Füllerer et. al presented in 2009 [40] a hybrid ACO algorithm to solve the two-dimensional loading vehicle routing problem, a problem that combines the loading of freight into the vehicles, and the routing of the vehicles to satisfy the customers' demands. A savings based ant system is used to optimize the routes. The construction step is modified to be additionally biased by an indicator that is proportional to the vehicles area consumed by the goods for the routes to be merged. To calculate these areas as well as to check the feasibility of packing multiple algorithms such as packing heuristics or branch and bound are combined and performed for each iteration. Intermediate infeasible solutions with respect to capacity and area utilization are allowed and penalized with an accordingly modified cost function.

A recently developed hybrid ACO algorithm was applied to the TSP with time windows by López-Ibáñez et al. in 2009 [61]. This hybrid combines ACO with beam search and is therefore called Beam-ACO. Here a beam search algorithm replaces the standard construction step of ACO. The algorithm is restarted each time it reaches convergence which is determined by the pheromone values distances to the upper and lower limits. The pheromone update schema is biased by the actual ant's solution, the best solution since restart, and the best solution found so far. The beam search is a probabilistic tree search algorithm that relies on accurate bounding information that can be calculated computationally inexpensively. This is presented as a reason why Beam-ACO outperforms previous methods for the TSP with time windows.

ACO for Pricing Problem

This chapter shows the application of ACO as a solution to the pricing subproblem for a column generation approach which solves the linear relaxation of the PVRPTW. The aim is to demonstrate that with ACO strong lower bounds can be found with competitive performance.

4.1 Formulation of the PVRPTW

Here we present a common formulation of the PVRPTW as MILP [34]. Consider the attributes introduced in chapter 1.3 for the problem. Let $x_{p,k,i,j}$ be variables that indicate if the arc $a_{i,j} \in A$ is part of the route of vehicle $h_k \in H$ of the fleet of vehicles on day $p \in P$ of the planning horizon. Therefore, if $x_{p,k,i,j} = 1$ then vehicle h_k travels on day p from vertex v_i to vertex v_j, if $x_{p,k,i,j} = 0$ then not. Further let the variables $y_{i,r}$ indicate, whether for customer v_i the visit combination $r \in R_i$ is selected ($y_{i,r} = 1$), or not ($y_{i,r} = 0$). Finally, let the variables $s_{p,k,i}$ be the service start time of vehicle h_k at customer v_i on day p, and $w_{p,k}$ the wait time of vehicle h_k on day p before it leaves the depot v_0.

Let further $V_C = V \setminus \{v_0\}$ be the set of customer vertexes without the depot, and $\pi_{i,r,p}$ a constant that indicates whether visit combination $r \in R_i$ of customer v_i contains day $p \in P$ ($\pi_{i,r,p} = 1$), or not ($\pi_{i,r,p} = 0$). Lastly, let $M \gg 0$ be a sufficiently high constant value. Using these variables we can formulate the PVRPTW as:

$$\min \sum_{p \in P} \sum_{h_k \in H} \sum_{v_i \in V} \sum_{v_j \in V} c_{i,j} x_{p,k,i,j} \tag{4.1}$$

subject to

$$\sum_{r \in R_i} y_{i,r} = 1 \qquad\qquad \forall v_i \in V_C \quad (4.2)$$

$$\sum_{r \in R_i} \pi_{i,r,p} y_{i,r} - \sum_{h_k \in H} \sum_{v_j \in V} x_{p,k,i,j} = 0 \qquad\qquad \forall p \in P, \forall v_i \in V_C \quad (4.3)$$

$$\sum_{h_k \in H} \sum_{v_j \in V_C} x_{p,k,0,j} \leq m \qquad\qquad \forall p \in P \quad (4.4)$$

$$\sum_{h_k \in H} \sum_{v_i \in V_C} x_{p,k,i,0} - \sum_{h_k \in H} \sum_{v_j \in V_C} x_{p,k,0,j} = 0 \qquad\qquad \forall p \in P \quad (4.5)$$

$$\sum_{v_j \in V} x_{p,k,j,i} - \sum_{v_j \in V} x_{p,k,i,j} = 0 \qquad\qquad \forall p \in P, \forall h_k \in H, \forall v_i \in V_C \quad (4.6)$$

$$\sum_{v_j \in V} x_{p,k,j,i} + \sum_{v_j \in V} x_{p,k,i,j} \leq 2 \qquad\qquad \forall p \in P, \forall h_k \in H, \forall v_i \in V_C \quad (4.7)$$

$$\sum_{v_i \in V_C} \sum_{v_j \in V} q_i x_{p,k,i,j} \leq Q_k \qquad\qquad \forall P \in P, \forall h_k \in H \quad (4.8)$$

$$e_0 + w_{p,k} + z_{0,j} - s_{p,k,j} + M x_{p,k,0,j} \leq M \qquad \forall p \in P, \forall h_k \in H, \forall v_j \in V_C \quad (4.9)$$

$$s_{p,k,i} + d_i + z_{i,j} - s_{p,k,j} + M x_{p,k,i,j} \leq M \qquad \forall p \in P, \forall h_k \in H, \forall v_i \in V_C, \quad (4.10)$$
$$\forall v_j \in V_C$$

$$s_{p,k,i} + d_i + z_{i,0} + M x_{p,k,i,0} \leq l_0 + M \qquad \forall p \in P, \forall h_k \in H, \forall v_i \in V_C \quad (4.11)$$

$$s_{p,k,i} + d_i + z_{i,0} - e_0 - w_{p,k} + M x_{p,k,i,0} \leq D_k + M \qquad \forall p \in P, \forall h_k \in H, \forall v_i \in V_C \quad (4.12)$$

$$s_{p,k,i} - e_i \sum_{v_j \in V} x_{p,k,i,j} \geq 0 \qquad\qquad \forall p \in P, \forall h_k \in H, \forall v_i \in V_C \quad (4.13)$$

$$s_{p,k,i} - l_i \sum_{v_j \in V} x_{p,k,i,j} \leq 0 \qquad\qquad \forall p \in P, \forall h_k \in H, \forall v_i \in V_C \quad (4.14)$$

$$x_{p,k,i,j} \in \{0,1\} \qquad\qquad \forall p \in P, \forall h_k \in H, \forall v_i \in V, \quad (4.15)$$
$$\forall v_j \in V$$

$$y_{i,r} \in \{0,1\} \qquad\qquad \forall v_i \in V_C, \forall r \in R_i \quad (4.16)$$

$$s_{p,k,i} \geq 0 \qquad\qquad \forall p \in P, \forall h_k \in H, \forall v_i \in V_C \quad (4.17)$$

$$w_{p,k} \geq 0 \qquad\qquad \forall p \in P, h_k \in H \quad (4.18)$$

In this *vehicle flow formulation* of the PVRPTW the objective function 4.1 minimizes the total travel costs over all days, all vehicles and all arcs. The equations 4.2 are the *cover constraints* which guarantee that for each customer one visit combination is selected. The visit combinations are linked with the vehicle routes by the *visit constraints* of equations 4.3 which make sure that the customers are visited on the days of the selected visit combination. Inequalities 4.4 restrict the number of vehicles used per day to the fleet size. We reference them as *fleet constraints*. Equations 4.5 ensure that each vehicle that leaves the depot has to return to the depot on the same day. They compose the *flow constraints* together with equations 4.6 and inequalities 4.7

which guarantee for each customer the incidence of no arcs if the customer is not on the vehicle route, or exactly two arcs if the customer is on the vehicle route. It is to be noted that the elimination of sub-routes of a vehicle route is fulfilled with the time constraints below. Inequalities 4.8 are the *capacity constraints* that limit the load for each vehicle. Inequalities 4.9 – 4.11 are the *time constraints*. For each vehicle route the service start times per customer as well as the depot wait time are brought into relation with the working start time of the depot 4.9, with the subsequent following service start times 4.10, and with the working end time of the depot 4.11 using the travel and service durations. The high constant value M is used in these inequalities to apply the time constraints only for relevant arcs, that is, if $x_{p,k,i,j} = 1$. For $x_{p,k,i,j} = 0$ the inequalities are always true irrespective of the other variables values. The travel time of a vehicle is limited by the *duration constraints* 4.12. Inequalities 4.13 and 4.14 comprise the *time window constraints*.

To solve this MILP with column generation the LP-relaxed problem has to be solved. But this quite natural formulation of the problem has the disadvantage of having a weak LP-relaxation. That is, the optimal solution of the LP-relaxed problem is a weak and therefore a bad lower bound for the MILP [92] [34].

By applying a generalized Dantzig-Wolfe decomposition – as similarly demonstrated by Cordeau et al. for the VRPTW [12] – the PVRPTW can be reformulated, so that its LP-relaxation delivers a much stronger lower bound for the MILP. Let Ω_k be the set of all feasible routes based on the complete directed graph $G = (V, A)$ beginning and ending in depot v_0 with respect to flow, capacity, time, duration, and time window constraints 4.5 – 4.14 for the vehicle h_k[1]. To indicate that customer v_i is part of route $\omega \in \Omega_k$, let $\phi_{i,\omega} = 1$, else $\phi_{i,\omega} = 0$. Further, let $x_{p,k,\omega}$, be the number of times that route $\omega \in \Omega_k$ is selected on day $p \in P$ for vehicle $h_k \in H$, and γ_ω the costs of route ω: $\gamma_\omega = \sum_{a_{i,j} \in \omega} c_{i,j}$. Therefore the *set partitioning formulation* of the PVRPTW is:

$$\min \sum_{p \in P} \sum_{h_k \in H} \sum_{\omega \in \Omega_k} \gamma_\omega x_{p,k,\omega} \tag{4.19}$$

subject to

$$\sum_{r \in R_i} y_{i,r} = 1 \qquad \forall v_i \in V_C \tag{4.20}$$

$$\sum_{h_k \in H} \sum_{\omega \in \Omega_k} \phi_{i,\omega} x_{p,k,\omega} - \sum_{r \in R_i} \pi_{i,r,p} y_{i,r} = 0 \qquad \forall p \in P, \forall v_i \in V_C \tag{4.21}$$

$$\sum_{h_k \in H} \sum_{\omega \in \Omega_k} x_{p,k,\omega} \leq m \qquad \forall p \in P \tag{4.22}$$

$$x_{p,k,\omega} \in \{0, 1\} \qquad \forall p \in P, \forall h_k \in H, \forall \omega \in \Omega \tag{4.23}$$

$$y_{i,r} \in \{0, 1\} \qquad \forall v_i \in V_C, \forall r \in R_i \tag{4.24}$$

---

[1]Since capacity and duration constraints depend on the vehicle's attributes, the set of feasible routes may be different for different vehicles

The objective function 4.19 of this formulation minimizes the total route costs over all days, all vehicles and all feasible routes. Equations 4.20 are the unmodified cover constraints. For the visit constraints 4.21 $\phi_{i,\omega}$ had to be substituted. Inequalities 4.22 cover the fleet constraints.

Assumptions

We assume a homogeneous set of vehicles, i.e. the maximum carrying load Q_k is equal for all vehicles and the maximum route duration D_k is equal for all vehicles: $Q = Q_k \: \forall \: h_k \in H$ and $D = D_k \: \forall \: h_k \in H$.

We further assume that the travel costs satisfy the triangle inequality: $c_{i,j} \leq c_{i,k} + c_{k,j}$ $\forall \: v_i, v_j, v_k \in V$.

Formulating the Restricted Master Problem

To get a RMP feasible for column generation, we apply some transformation steps to the set partitioning formulation of the PVRPTW.

Since we want to compute a lower bound for the ILP we have to LP-relax the problem to get a linear program that can be solved by e.g. simplex. Therefore the binary integrality constraints 4.23 and 4.24 are relaxed.

Further it is reasonable to relax the set partitioning formulation to a set covering formulation[2]. Bramel and Simchi-Levis [9] showed the effectiveness of the set covering formulation for the VRPTW regarding column generation. Therefore, we also apply the relaxation for the column generation approach for the PVRPTW [74]. The cover constraints 4.20 can be easily relaxed to a ≥ 1 inequality, since it is obvious that for each solution containing a customer with two or more selected visit combinations there is a better or equally good solution with only one select visit combination. Due to the satisfied triangle inequality the visit constraints 4.21 can also be relaxed [92]. The advantage of the set covering model is that the solution space of the dual problem is reduced, due to the fact that equality constraints in the primal problem become unrestricted variables in the dual problem, and inequality constraints become restricted variables.

Because of the assumption of a homogeneous vehicle fleet, we can simplify the formulation and eliminate the vehicles from the objective as well as the constraints. Therefore, we introduce the variable $x_{p,\omega}$ that is the number of times the route $\omega \in \Omega$ is selected at day $p \in P$. Ω is the set of all feasible routes starting and ending in the depot v_0; because of the homogeneity of the vehicle fleet, $\Omega = \Omega_k \: \forall \: h_k \in H$.

The set of variables $x_{p,\omega}$ is huge, that is the reason why we restrict the master problem to a small set of variables which is enlarged iteratively until all relevant variables are contained. Therefore we start with t initial small subsets of feasible routes $\dot{\Omega}_p \subset \Omega \: \forall \: p \in P$, one for each day of the planning horizon. This enables us to distinguish the further generated columns by the

---

[2]The general form of a set partitioning problem is expressed as $\min c^T x$ subject to $Ax = 1, x \in \{0,1\}^n$, and for the set covering problem it is $\min c^T x$ subject to $Ax \geq 1, x \in \{0,1\}^n$

day of the planning period. We formulate the RMP for the column generation approach as:

$$\min \sum_{p \in P} \sum_{\omega \in \dot{\Omega}_p} \gamma_\omega x_{p,\omega} \tag{4.25}$$

subject to

$$\sum_{r \in R_i} y_{i,r} \geq 1 \qquad\qquad \forall v_i \in V_C \tag{4.26}$$

$$\sum_{\omega \in \dot{\Omega}_p} \phi_{i,\omega} x_{p,\omega} - \sum_{r \in R_i} \pi_{i,r,p} y_{i,r} \geq 0 \qquad\qquad \forall p \in P, \forall v_i \in V_C \tag{4.27}$$

$$\sum_{\omega \in \dot{\Omega}_p} x_{p,\omega} \leq m \qquad\qquad \forall p \in P \tag{4.28}$$

$$x_{p,\omega} \geq 0 \qquad\qquad \forall p \in P, \forall \omega \in \dot{\Omega}_p \tag{4.29}$$

$$y_{i,r} \geq 0 \qquad\qquad \forall v_i \in V_C, \forall r \in R_i \tag{4.30}$$

In this formulation of the RMP the objective function 4.25 minimizes the total route costs over all days and feasible routes. Inequalities 4.26 comprise the cover constraints guaranteeing that at least one visit combination is selected per customer. The visit constraints 4.27 ensure that the customers are visited at least once during the days of the selected visit combination. The inequalities 4.28 referred to as fleet constraints restrict the vehicles used per day to the fleet size.

The RMP in this form has a manageable number of constraints $tn + n + t$ by using $t|\dot{\Omega}| + \sum_{v_i \in V_C} r_i$ variables.

Formulating the Pricing Subproblem

To obtain additional variables for the RMP that have the potential to improve the cost function, a pricing subproblem has to be solved that generates these variables. Therefore, the problem is to find feasible vehicle routes $\omega \in \Omega$ with negative reduced costs when injecting them into the RMP. This problem is referred to as the *shortest path problem with resource constraints* (SPPRC) [52], namely with capacity constraints, duration constraints, and time window constraints.

Consider the dual problem of the RMP and let $\chi_{p,i}$ be the dual variables for the visit constraints 4.27 and ψ_p the dual variables for the fleet constraints 4.28. Based on the preliminaries from chapter 2.2 the task of the subproblem is to find columns – that is vehicle routes $\omega \in \Omega$ –, whereas $\sum_{p \in P} \sum_{v_i \in V_C} \phi_{i,\omega} \chi_{p,i}^* + \sum_{p \in P} \psi_p^* > \sum_{p \in P} \gamma_\omega$. $\chi_{p,i}^*$ and ψ_p^* are the dual variables' values for the optimal solution of the RMP. By reformulation and substitution (see the similar method by Cordeau et al. for the VRPTW [12]) we obtain $\sum_{p \in P} \sum_{v_i \in V_C} \sum_{v_j \in V} (c_{i,j} - \chi_{p,i}^*) x_{\omega,i,j} + \sum_{p \in P} \sum_{v_j \in V} (c_{0,j} - \psi_p^*) x_{\omega,0,j} < 0$, where $x_{\omega,i,j}$ indicates whether the arc $a_{i,j}$ is part of ω or not. Based on that we can solve a subproblem for each day $p \in P$ by using the reduced costs

$$\bar{c}_{p,i,j} = \begin{cases} c_{i,j} - \chi_{p,i}^* & \text{if } v_i \in V_C \\ c_{i,j} - \psi_p^* & \text{if } v_i = v_0 \end{cases}$$

After applying the reduction, the costs of an arc can certainly become negative and, in addition, the triangle inequality is no longer ensured to be satisfied. Therefore solving the SPPRC enforces shortest paths with cycles, especially if the resource constraints are not very tight. Since the optimal vehicle routes of the PVRPTW will not contain cycles, the SPPRC used as the subproblem may generate columns that will never be used in the optimal solution. That is the reason why we restrict ourselves to the *elementary shortest path problem with resource constraints* (ESPPRC) [18]. By using the ESPPRC as the subproblem better lower bounds are expected. Unfortunately, this renders the subproblem $\mathcal{NP}$-hard. Nevertheless there are algorithms and techniques to solve the ESPPRC as the pricing subproblem of a column generation approach with good performance [74].

To demonstrate the task of the pricing subproblem figure 4.1 shows a small example of a graph with one depot vertex v_0 and three customer vertexes v_1, v_2 and v_3. The numbers at the arcs represent the reduced costs $\bar{c}_{p,i,j}$ of a certain day $p \in P$. The intervals in square brackets denote the time windows $[e_i, l_i]$ for each vertex. For simplicity let us assume that the travel times between vertexes is 1: $z_{i,j} = 1 \; \forall \; a_{i,j} \in A$, and the service duration at each customer is also 1: $d_i = 1 \; \forall \; v_i \in V_C$. Let it further be assumed that the maximum route duration $D = 5$ and the maximum carrying load does not matter. Due to duration and time window constraints seven feasible vehicle routes exist (without proof). Three of them result in total to a negative reduced cost: $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_0$ with negative reduced costs of -23; $v_0 \rightarrow v_1 \rightarrow v_3 \rightarrow v_0$ with negative reduced costs of -10; and $v_0 \rightarrow v_2 \rightarrow v_0$ with negative reduced costs -13. The "shortest" path is therefore to be calculated as the vehicle route with the highest negative reduced costs. In this example it is -23 and the route is marked with green dashed arrows.
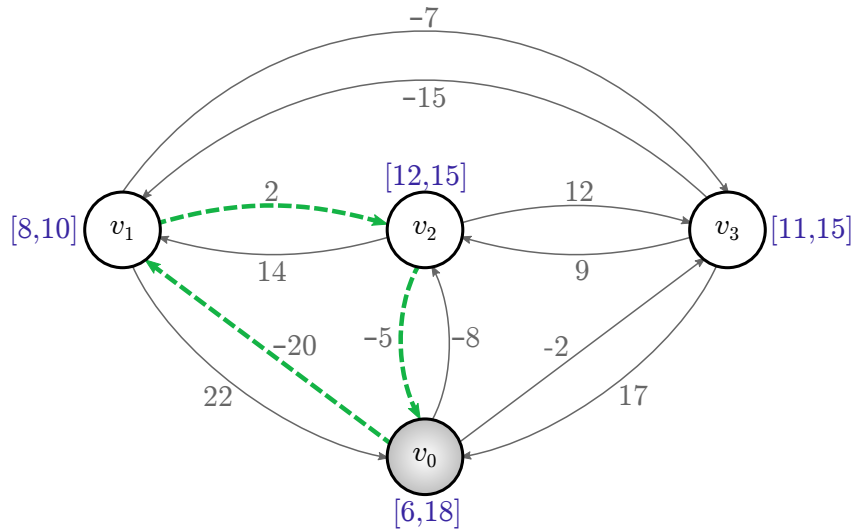

Figure 4.1: Demonstration of the ESPPRC as pricing subproblem

An ESPPRC pricing subproblem for the RMP presented above has to be solved for each day $p \in P$. Let $x_{i,j}$ indicate whether arc $a_{i,j}$ is part of the elementary path, let s_i be the service start time at customer $v_i \in V_C$, and let w be the wait time before the path traversing vehicle leaves

the depot. Then the pricing subproblem can be formulated as:

$$\min \sum_{v_i \in V} \sum_{v_j \in V} \bar{c}_{p,i,j} x_{i,j} \tag{4.31}$$

subject to

$$\sum_{v_i \in V_C} x_{i,0} - \sum_{v_j \in V_C} x_{0,j} = 0 \tag{4.32}$$

$$\sum_{v_j \in V} x_{j,i} - \sum_{v_j \in V} x_{i,j} = 0 \qquad \forall v_i \in V_C \tag{4.33}$$

$$\sum_{v_j \in V} x_{j,i} + \sum_{v_j \in V} x_{i,j} \le 2 \qquad \forall v_i \in V_C \tag{4.34}$$

$$\sum_{v_i \in V_C} \sum_{v_j \in V} q_i x_{i,j} \le Q \tag{4.35}$$

$$e_0 + w + z_{0,j} - s_j + M x_{0,j} \le M \qquad \forall v_j \in V_C \tag{4.36}$$

$$s_i + d_i + z_{i,j} - s_j + M x_{i,j} \le M \qquad \forall v_i \in V_C, \forall v_j \in V_C \tag{4.37}$$

$$s_i + d_i + z_{i,0} + M x_{i,0} \le l_0 + M \qquad \forall v_i \in V_C \tag{4.38}$$

$$s_i + d_i + z_{i,0} - e_0 - w + M x_{i,0} \le D + M \qquad \forall v_i \in V_C \tag{4.39}$$

$$s_i - e_i \sum_{v_j \in V} x_{i,j} \ge 0 \qquad \forall v_i \in V_C \tag{4.40}$$

$$s_i - l_i \sum_{v_j \in V} x_{i,j} \le 0 \qquad \forall v_i \in V_C \tag{4.41}$$

$$x_{i,j} \in \{0,1\} \qquad \forall v_i \in V, \forall v_j \in V \tag{4.42}$$

$$s_i \ge 0 \qquad \forall v_i \in V_C \tag{4.43}$$

$$w \ge 0 \tag{4.44}$$

The objective function 4.31 of the ESPPRC as the pricing subproblem of the PVRPTW minimizes the total reduced path costs over the arcs along the paths from depot v_0 to depot v_0 with at least one customer in between. The equations 4.32 – 4.34 define the flow constraints of the ESPPRC: the path that leaves the depot has to return to the depot (4.32), and each customer has to be visited exactly once by a path or it is not part of the path (4.33 and 4.34). Inequality 4.35 denotes the capacity constraint. The time constraints 4.36 – 4.38 ensure that, on the one hand, the path contains no subpaths. On the other hand, they define relations between the service start times at the customer, the working time interval of the depot, and the wait time. The duration constraint is covered in inequalities 4.39, and the time window constraints in inequalities 4.40 and 4.41.

To solve the pricing subproblem in an iterative column generation context it is neither necessary to solve the ESPPRC to optimality for all iterations, nor has the result of the pricing problem to be a single path with negative reduced costs. Of course several paths or even all paths with negative reduced costs can be injected into the RMP. With columns of appropriate

quality this will speed up the convergence of the column generation algorithm with the price of a faster growing RMP regarding variables.

Furthermore, the optimal solutions for each day $p \in P$ only have to be determined in the "last" iteration of the column generation algorithm . If for all days the optimal elementary shortest path has no negative reduced costs, the column generation is finished and therefore the LP-relaxed problem of the PVRPTW is optimally solved providing a lower bound for the problem. Since the ESPPRC is $\mathcal{NP}$-hard it even seems feasible to solve the pricing subproblem for several iterations approximately as long as the algorithm provides paths with high quality for the RMP. This was demonstrated by Pirkwieser and Raidl [74] for the GRASP metaheuristic; we want to show that ACO is also able to improve the pricing subproblem solution for such highly constrained problems as the PVRPTW.

4.2   Design Decisions

In this section the main algorithm for the column generation approach and its components are presented. Algorithmic design issues are illustrated and design decisions explained.

Column Generation Algorithm for the PVRPTW

Algorithm 4.1 shows the overall column generation algorithm for the PVRPTW used by our approach. First the problem instance is loaded into memory. Then there it is possible to provide an initial feasible solution for this problem instance (e.g. calculated by any heuristic). This solution defines the initial sets of variables $\grave{\Omega}_p$ for the RMP. If no initial solution is provided then the algorithm introduces highly penalized slack variables to the RMP which is constructed with cover, visit and fleet constraints based on the PVRPTW problem instance. The number of negative reduced cost columns generated by the exact ESPPRC solver is initialized for each day to $undefined$, and the first day is set to active. The initial state of the algorithm is to start with the approximate ESPPRC solver.

Then the loop for the column generation iterations starts until for each day of the planning period the number of negative reduced cost columns returned by the exact ESPPRC solver is 0. The first step of an iteration is to solve the RMP with an exact simplex based LP problem solver. By using the optimal solution of the RMP the values of the dual variables for the visit and fleet constraints can be obtained. These are used to define the ESPPRC pricing subproblem for the actual day by calculating the reduced costs $\bar{c}_{p,i,j}$ for each arc $a_{i,j}$. Next the algorithm decides, based on its state, whether it should use the approximate ESPPRC solver or the exact ESPPRC solver. The result is in each case a set of negative reduced cost columns. If this set is not empty the columns found are injected into the RMP. The next day is selected and a new state of the algorithm is determined before the next iteration starts.

Initial Solution vs. Slack Variables

The general column generation algorithm needs a set of columns $\grave{\Omega}$ to start with. For example, this set can be obtained by taking any feasible solution from a heuristic or metaheuristic

1 $PVRPTW \longleftarrow$ load PVRPTW problem instance

2 $\forall p \in P : \dot{\Omega}_p \longleftarrow$ load initial feasible solution, if provided

3 $RMP \longleftarrow$ construct RMP with cover, visit and fleet constraints using $PVRPTW$ and
$\dot{\Omega}_p$ `// if no initial solution provided add slack variables`

4 $NRC_p^{exact} \longleftarrow undefined \,\forall\, p \in P$

5 $p \longleftarrow$ first day $\in P$

6 $state \longleftarrow approx$

7 repeat

8 $RMP^* \longleftarrow$ `SolveLinear(`RMP`)`

9 $\chi_p \longleftarrow$ dual variables for visit constraints of RMP^*

10 $\psi_p \longleftarrow$ dual variable for fleet constraint of RMP^*

11 $ESPPRC \longleftarrow$ `GenerateESPPRC(`$PVRPTW, \chi_p, \psi_p$`)`   `// here the`
                                           `// reduced costs are calculated`

12   **if** $state \neq exact$ **then**

13 $ESPPRC^* \longleftarrow$ `SolveApprox(`$ESPPRC$`)`

14 $\Omega^+ \longleftarrow$ set of negative reduced costs columns of $ESPPRC^*$

15   **else**

16 $ESPPRC^* \longleftarrow$ `SolveExact(`$ESPPRC$`)`

17 $\Omega^+ \longleftarrow$ set of negative reduced costs columns of $ESPPRC^*$

18 $NRC_p^{exact} \longleftarrow |\Omega^+|$

19   **end**

20   **if** $|\Omega^+| > 0$ **then**

21 $RMP \longleftarrow$ `InjectColumns(`RMP, Ω^+`)`

22   **if** $state \neq exact$ **then** $NRC_p^{exact} \longleftarrow undefined \,\forall\, p \in P$

23   **end**

24 $p \longleftarrow$ choose next day

25 $state \longleftarrow$ `DetermineNextState(`$state, NRC^{exact}, |\Omega^+|$`)`

26 until $NRC_p^{exact} = 0 \,\forall p \in P$

27 return RMP^*

<div align="center">

Algorithm 4.1: Column generation algorithm for PVRPTW

</div>

algorithm that solves the PVRPTW. This solution may be interpreted as the initial solution upon which the column generation algorithm improves until reaching the final solution.

Instead of providing an initial solution it is also possible to introduce slack variables to the RMP. These slack variables work as an initial set of "pseudo-routes". Since they do not represent any real route of the solution they have to be eliminated during the iterations of the algorithm. This is achieved by penalizing these variables in the objective function. Therefore, adding "real" routes as result columns of the ESPPRC solver will supersede the slack variables over time. If slack variables stay active until the column generation algorithm has finished, then the LP-relaxed problem is not feasible also implying that the PVRPTW is not solvable. The

initial formulation of the RMP with slack variables is shown here:

$$\min \sum_{p \in P} \sum_{v_i \in V_C} x_{p,i}^{slack} K \tag{4.45}$$

subject to

$$\sum_{r \in R_i} y_{i,r} \geq 1 \qquad \forall v_i \in V_C \tag{4.46}$$

$$x_{p,i}^{slack} - \sum_{r \in R_i} \pi_{i,r,p} y_{i,r} \geq 0 \qquad \forall p \in P, \forall v_i \in V_C \tag{4.47}$$

$$0 \leq m \qquad \forall p \in P \tag{4.48}$$

$$y_{i,r} \geq 0 \qquad \forall v_i \in V_C, \forall r \in R_i \tag{4.49}$$

$$x_{p,i}^{slack} \geq 0 \qquad \forall p \in P, \forall v_i \in V_C \tag{4.50}$$

The initial RMP with slack variables contains no regular route variables. This can be interpreted as $\grave{\Omega} = \{\}$. Therefore, slack variables $x_{p,i}^{slack}$ for each day $p \in P$ and each customer $v_i \in V_C$ are introduced which enable solving the RMP. The objective function 4.45 contains initially only slack variables which are penalized by the penalty value K. The cover constraints 4.46 stay unmodified; in the visit constraints 4.47 the regular route variables are replaced by slack variables. The fleet constraints 4.48 are degraded to a simple tautology.

Figure 4.2 shows a graph that displays a typical example of the behavior of the column generation algorithm with slack variables. It shows the development of the objective function value over the iterations. Additionally the number of active slack variables is displayed as blue round marks. The algorithm starts with all 235 slack variables active. They are rapidly replaced by columns of real routes, explaining the steep decrease of the objective value. From iteration 20 onwards all slack variables are inactive. After 64 iterations no more negative reduced costs columns could be found and the algorithm returns the lower bound for this problem as 2682.76.

For comparison figure 4.3 shows the behavior of the column generation algorithm without slack variables. Instead an initial solution was provided for the same problem letting the algorithm start with an objective value of 3694.9. A typical behavior can be identified by the progress of the objective value: it takes some iterations if started with an initial solution until the added columns improve the objective function. It can be observed that the better the initial solution, the more effective the initial effort in enhancing the column pool of the RMP.

Solving the RMP

Since the RMP is an ordinary LP problem it can be solved by any exact LP solver which is able to provide the optimal dual variables. We used the CPLEX® Optimizer by IBM ILOG®, an optimization software for mathematical programming. The CPLEX Optimizer includes among other modules a high performance LP solver based on the simplex algorithm.
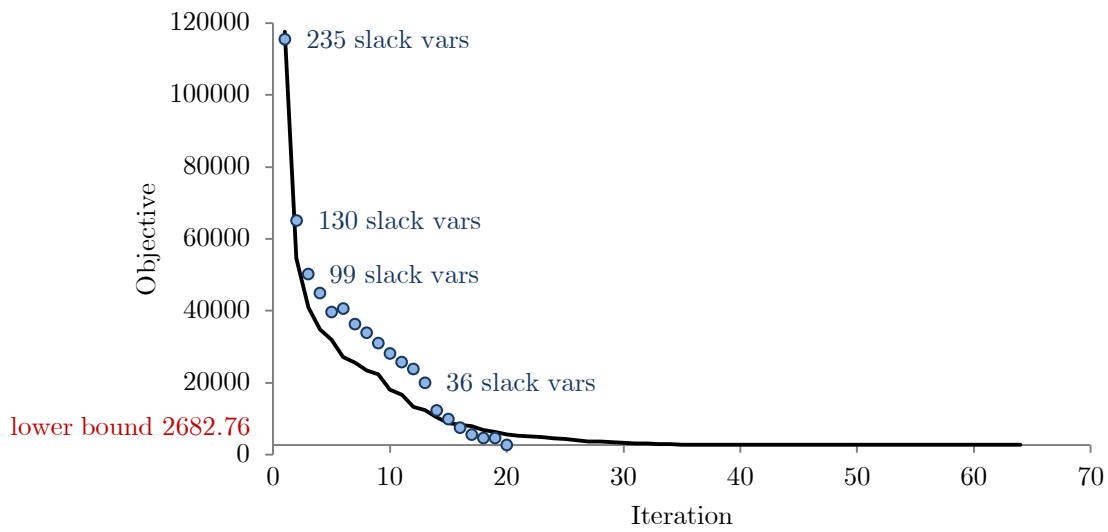
Figure 4.2: Example of the behavior of the column generation algorithm with slack variables
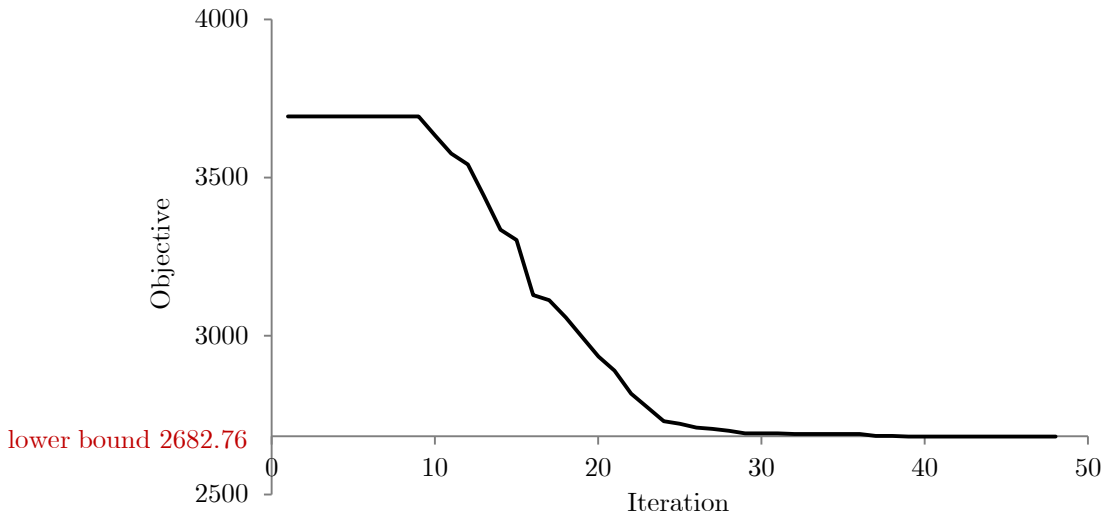

Figure 4.3: Example of the behavior of the column generation algorithm with initial solution

Solving the ESPPRC Approximately

We implemented several methods to solve the ESPPRC approximately to develop some kind of benchmark and to be able to identify the factors influencing good performance. The basis for all of these methods is a set of simple path changing operations. On the one hand, they allow the generation of paths and, on the other, they build the central components for local search operations.

We implemented five different fairly simple path changing operations. The result of such an operation has to be a feasible path in respect of capacity, duration, and time window constraints. Figure 4.4 shows the operations graphically. Let us denote n_p the number of customer vertexes
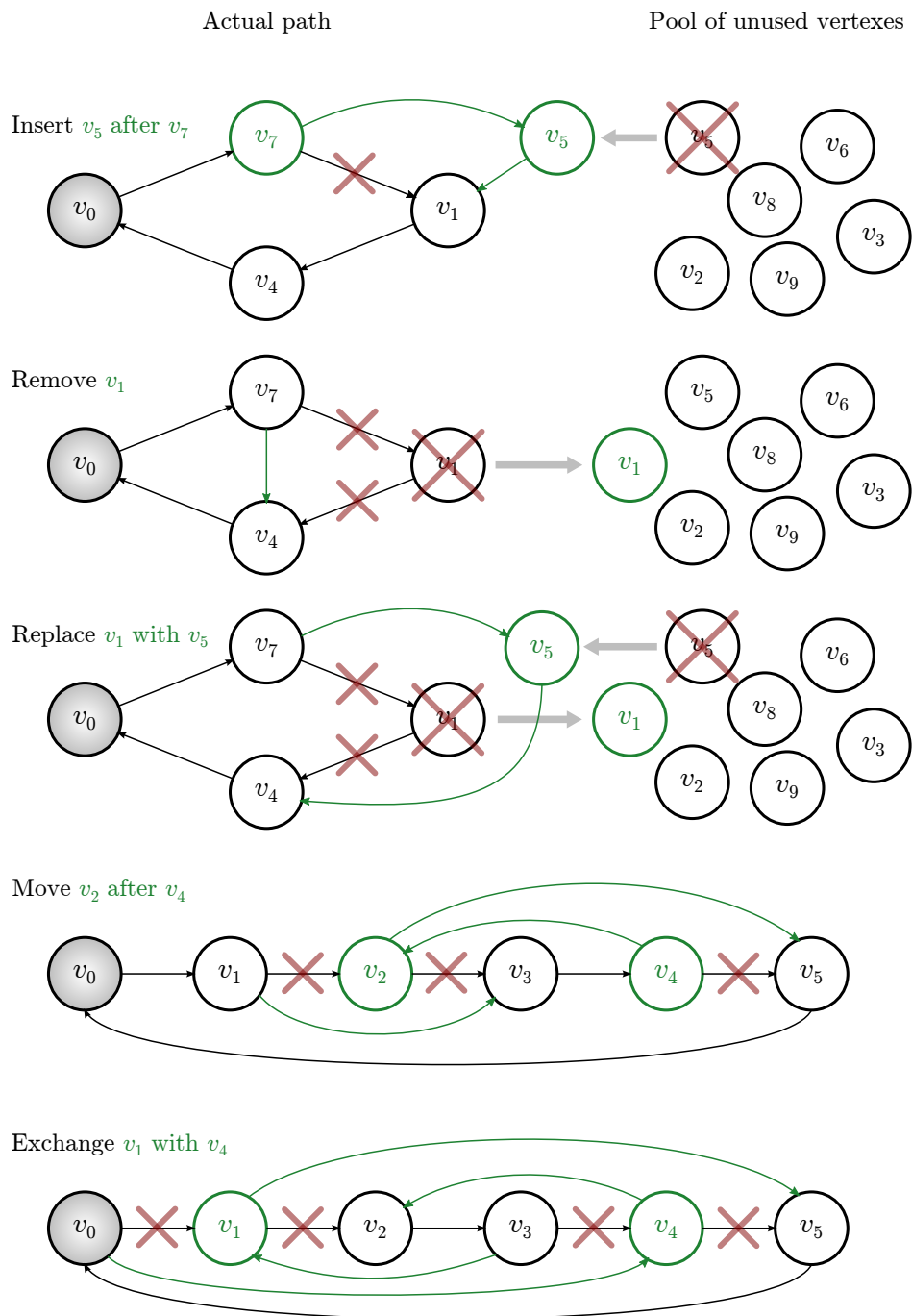
Figure 4.4: Schematic of the five different path changing operations used

that are part of the actual path, $n - n_p$ is therefore the size of the pool of customer vertexes not contained in the actual path.

- *Insert* operation: the insert operation takes a customer vertex from the pool of unused vertexes and inserts it after an arbitrary vertex of the actual path. There are $(n-n_p)(1+n_p)$ possible insert operations based on an actual path that have to be checked for feasibility.

- *Remove* operation: this simplest of all operations removes an arbitrary customer vertex from the actual path allowing n_p possible remove operations.

- *Replace* operation: this operation replaces a customer vertex from the actual path with a vertex from the pool of unused vertexes. Based on an actual path there are $n_p(n - n_p)$ possible replace operations which have to be checked for feasibility.

- *Move* operation: the move operation is a path local operation. It takes a customer vertex of the actual path and moves it to an arbitrary position in the path. There are $(n_p - 1)^2$ different move operations that have to be checked for feasibility regarding duration and time window constraints.

- *Exchange* operation: the last operation is also a path local operation that exchanges the positions of a customer vertex of the actual path with another arbitrary selected customer vertex. The $\frac{1}{2}n_p(n_p - 1)$ different exchange operations have also to be checked for feasibility regarding duration and time window constraints.

As already mentioned these path changing operations are the core operations of the local search procedures we used for the approximate ESPPRC solvers. We decided to apply a *randomized best improvement* strategy for the local search as shown in algorithm 4.2: based on an actual path the local search iterates until no further improvement can be applied. For each iteration the algorithm chooses a randomized order of the described path change operations with equal probabilities. The actual path is replaced with a path where the best improvement is applied regarding the actual path change operation. If there is no improvement for the actual operation the algorithm switches to the next operation in the randomized list.

In addition to ACO we implemented three other approximate ESPPRC solvers for benchmarking purposes. The **ILS** starts with an empty path $v_0 \rightarrow v_0$ and applies the aforementioned local search procedure in each iteration. The perturbation operation is implemented by applying 10 feasible random path change operations (random in respect of type of the operation as well as of instance of operation). New paths are accepted always, since no comparison of the perturbed local best solution with the previous best solution is performed. The solution of each iteration is checked as to whether it has negative reduced costs, in which case it is collected and finally returned to the column generation algorithm.

The **VNS** also starts with an empty path. The neighborhood structures are defined by the path changing operations: the first neighborhood consists of insert and remove operations, the next neighborhood of move operations, the third of replace operations, and the last of exchange operations. In our implementation only the shaking operator is influenced by the actual neighborhood, the local search is always performed with the previously described algorithm using all

Input: ESPPRC path s
Output: ESPPRC path with local maximal negative reduced costs

1 **repeat**
2 $Op \longleftarrow$ random permutation of operations Insert, Remove, Replace, Move, Exchange
3 $o \longleftarrow 1$
4 $improved \longleftarrow false$
5   **repeat**
6   **if** *exists improvement of s with operation Op_o* **then**
7 $s \longleftarrow$ perform best improvement with operation Op_o on s
8 $improved \longleftarrow true$
9   **end**
10 $o \longleftarrow o + 1$
11   **until** *improved or $o > 5$*
12 **until** *not improved*
13 **return** s

Algorithm 4.2: Local search algorithm for the approximate ESPPRC solvers

path changing operations (this showed a higher performance than letting the VNS local search depend on the actual neighborhood). Additionally the intensity of the shake operation also depends on the actual neighborhood: the higher the neighborhood structure, the more stochastically random path change operations in the actual neighborhood are performed. If the costs of the new path are negative, then the new path is accepted for the next iteration if the costs are lower than the costs of the previously accepted path; otherwise the next neighborhood structure is selected. All negative reduced costs paths are returned to the column generation algorithm.

The **GRASP** depends on a greedy randomized heuristic to create a path. For that the algorithm starts with an empty path and adds customer vertexes stochastically to the last position before returning to the depot until no further customer vertex can be added. To make the heuristic greedy only those customer vertexes are taken into consideration that improve the path costs. These customer vertexes are stored for each step in the restricted candidate list (RCL), and the candidate to be added to the path is selected randomly. When no customer vertexes can be found for the RCL, the construction step is finished and the solution is improved by applying the previously described local search. Each solution is returned to the columns generation algorithm, since they all have negative reduced costs.

All of the implemented approximate pricing subproblem solvers only return distinct columns. This means that before adding a vehicle route to the actual set of negative reduced cost columns the algorithms check if the route is already contained. Nevertheless, calls from different column generation iterations may produce vehicle routes that have already been submitted in a previous iteration.

We loop all of the implemented approximate ESPPRC subproblem solvers for 1000 iterations. This value is a trade-off between a short solution time and enough time for convergence. The short solution time is important since the ESPPRC subproblem has to be solved several times during column generation. The convergence is needed to find good solutions even in later

calls of the subproblem solvers when the search space is very limited in respect of existing negative reduced cost columns. To enforce convergence the approximate subproblem solvers are granted 4000 more iterations if no negative reduced cost column has been found during the first 1000 iterations. Actually this method balances the effort for RMP solving and ESPPRC solving for the whole column generation process.

Solving the ESPPRC Exactly

For the exact solving of the ESPPRC as the pricing subproblem we used the algorithm presented by Pirkwieser and Raidl [74]. The algorithm applies a dynamic programming approach and is based on the improved label correcting algorithm introduced by Feillet et al [35]. The proposed tripartite dominance rules are applied which makes the exact ESPPRC solver in the early stages of the column generation procedure a semi-heuristic algorithm that does not find all negative reduced cost columns. Nevertheless, because of the cascaded application of the dominance rules it is ensured that in the last iteration of the column generation algorithm no negative reduced cost columns exist. We used the variant of the algorithm where we stopped the column generating procedure in each iteration after 1000 columns have been produced, which is called "forced early stop" in [74].

Column Injection

The columns generated by the ESPPRC pricing subproblem solvers have to be injected into the RMP. Since the subproblem is always solved for a single day, it is self-evident that the columns for this day should be to the set of columns of the RMP. To this end one coefficient per negative reduced cost columns has to be added to the objective function, one to the cover constraints, and one to the fleet constraints.

Fortunately columns can be added not only to the single day, but also to several or even all days of the planning horizon. Although the ESPPRC was solved for a single day, the result is a feasible route for all days. This is because the subproblem's constraints do not depend on the day of the planning period, only the subproblem's objective function is influenced by it. This allows us to inject the negative reduced cost columns generated for a single day to the whole planning period by adding t coefficients to the objective function, cover constraints and fleet constraints. The impact is of course a faster growth of columns $\hat{\Omega}$ which might slow down the solving of the RMP. On the other hand, the columns of one day may be reasonable also for other days, which might reduce the overall iterations of the column generation algorithm.

Figure 4.5 shows the typical behavior of the column generation algorithm regarding variable injection. The dashed lines display the scenario when the columns are only added for the single day of subproblem solving. The solid lines show the behavior when adding the columns for all days of the planning horizon. This diagram displays the decreasing value of the objective function, as well as the increasing size of the set of columns for the two scenarios. Although the number of columns behaves as expected and rises faster when adding the routes to all days, the objective function converges much faster to the optimal value of 2881.01. In this example the problem instance has a planning horizon of four days. Even though the columns are quadrupled when injecting them for all days into the RMP, because of the faster convergence
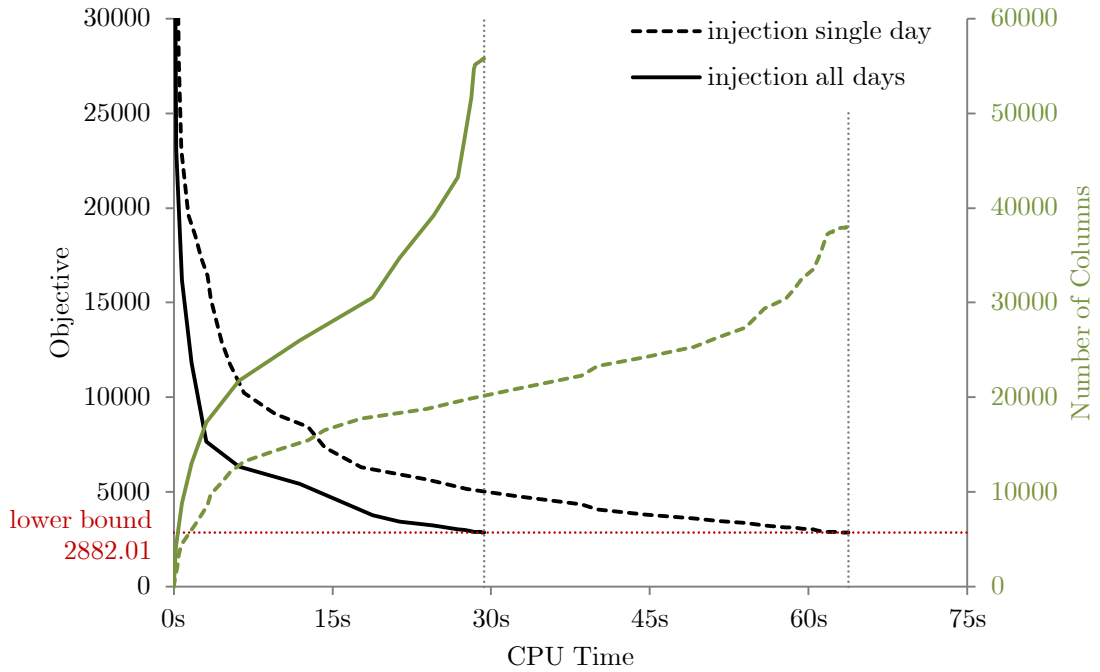
Figure 4.5: Example of the behavior of the column generation algorithm with different variable injection techniques

the final number of columns is just about 1.5 times as high as for the single day injection scenario (≈ 60000 vs. ≈ 40000 columns). Numerous experiments have confirmed this behavior for several PVRPTW problem instances. Therefore we inject negative reduced costs columns generated by the ESPPRC solvers always for all days of the planning period.

Determining the Algorithm State

The last step of an iteration of the column generation algorithm for the PVRPTW is the determination of the state for the next iteration. As Pirkwieser and Raidl [74] showed a quite successful technique is to start with the approximate ESPPRC solver and then switch to the exact ESPPRC solver for the rest of the iterations. We apply the same strategy but adapt the rule for switching to the exact solver. Since we want to focus on the approximate ESPPRC solver we emphasize this phase of the algorithm. The exact solver is activated when the heuristic stalls, which is identified by a reduced decrease of the objective value. Figure 4.6 shows the behavior of the algorithm. As long as the objective value is improved relevantly the algorithm stays in the "approximate state". When the improvement of the objective value falls below a specific decline the algorithm switches to the "exact state". The rule for switching the state is: if the delta of the objective value from one iteration to the next $\Delta Z^{RMP} = Z_{it}^{RMP} - Z_{it+1}^{RMP} < \kappa Z_{it}^{RMP}$ for all days of the planning period in a row, then switch from the approximate ESPPRC solver to the exact ESPPRC solver. κ is a small constant value that was determined experimentally to 0.0001.

Unfortunately the rule does not work if the column generation algorithm was started with
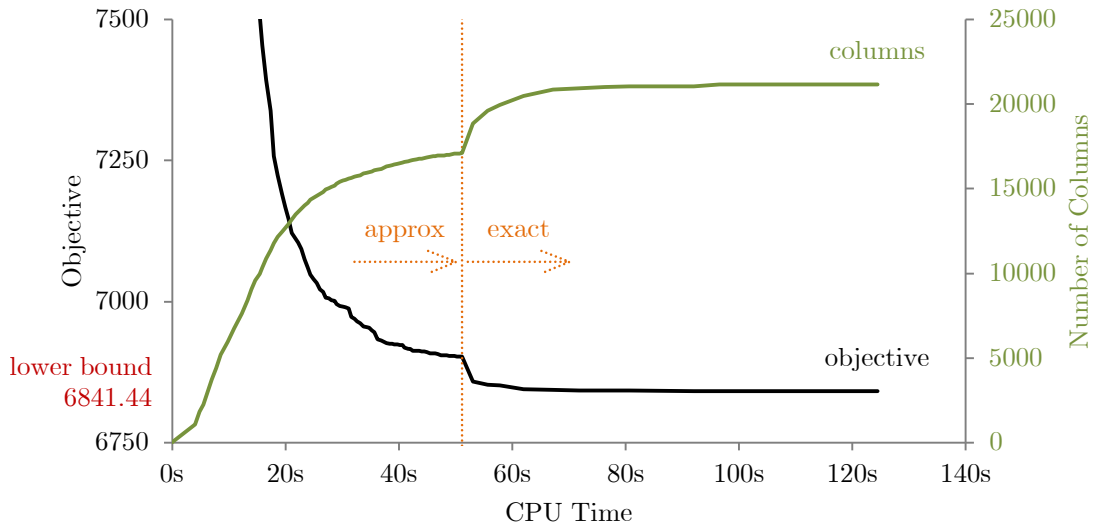
Figure 4.6: Example of the column generation algorithm started with slack variables switching from approximate to exact ESPPRC solver
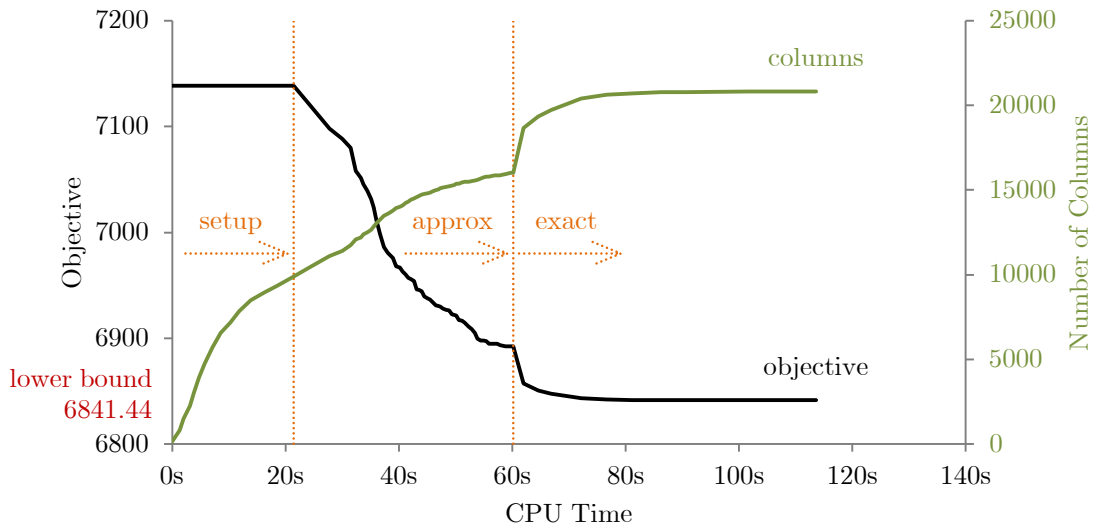

Figure 4.7: Example of the column generation algorithm started with an initial solution switching from approximate to exact ESPPRC solver

an initial solution, because the pricing subproblem has to deliver several columns in numerous iterations before a real improvement of the already good objective function can start. Figure 4.7 shows the plateau of the objective value at the beginning of the algorithm runtime. Therefore we expand the rule for switching to the exact ESPPRC solver: this state change is only executed if the algorithm had improved the objective value before, that is, if $\Delta Z^{RMP} = Z_{it}^{RMP} - Z_{it+1}^{RMP} > \kappa Z_{it}^{RMP}$ for a single iteration.

This behavior of the column generation algorithm started with an initial solution can be

interpreted as a three-state machine: first the algorithm is in a setup state where the column pool is enriched without improving the objective value. Then it enters the approximate state where the approximate algorithm determines most promising columns with low computational effort which reduce the objective value. And lastly it enters the exact state where the exact subproblem solver generates the missing columns to determine the optimal objective.

4.3 Ant Colony Optimization for the ESPPRC as Pricing Subproblem

Our ACO approach uses a mixture of the savings based ant system [33] and ant colony system [30] to fulfill the preliminaries of an efficient algorithm that delivers good results early. This is important due to the limitations of the pricing subproblem solving process: finding good solutions within 1000 up to 5000 iterations which allow the column generation algorithm to converge to the lower bound rapidly.

The components of our ACO approach are derived from well-known ACO applications and consist of:

- Ant colony: we decided to allow a quite small ant colony to explore the search space concurrently based on the actual pheromone trails and construction heuristic. After constructing the paths for all ants of the colony the best path is selected to modify the pheromone trails for future generations.

- Path construction: the path construction process for an ant follows the classic construction algorithms for ACO. It is therefore a probabilistic greedy heuristic biased by the pheromone trails of past ant generations. The construction process adds customer vertexes at the end of the path as long as the resulting path is feasible. The probability of one construction step is determined by the pheromones $\tau_{i,j}$ and the heuristic component $\eta_{i,j}$ using the common ACO formula

$$P_j = \begin{cases} \dfrac{\tau_{i,j}{}^\alpha \times \eta_{i,j}{}^\beta}{\sum\limits_{k \in V_C^{\text{feasible}}} \tau_{i,k}{}^\alpha \times \eta_{i,k}{}^\beta} & \forall v_j \in V_C^{\text{feasible}} \\[2em] 0 & \forall v_j \notin V_C^{\text{feasible}} \end{cases}$$

where i denotes the index of the customer v_i part of the actual path that is visited last before returning to the depot: $v_0 \to \cdots \to v_i \to v_0$. V_C^{feasible} is the set of customer vertexes that are not part of the actual path and that can be added to the path after the customer v_i without violating capacity, duration, and time window constraints. Based on these probabilities the construction step is determined by roulette wheel selection.

- Heuristic component: the calculation of the heuristic component $\eta_{i,j}$ was inspired by the savings algorithm introduced by Clarke and Wright [10] as proposed by Dörner et al. [33]. The savings based heuristic does not only take into account the cost $\bar{c}_{i,j}$ when appending the customer vertex v_j to an actual path ending with vertex v_i. Moreover it calculates the savings for merging the trivial path $v_0 \to v_j \to v_0$ with the actual path. Since the trivial

path is not necessarily a negative reduced cost column and therefore not an option as a result, we modified the heuristic formula in a way so that it is based on the cost's delta. Figure 4.8 shows the situation when appending a customer vertex v_j to an actual path with last vertex v_i: the heuristic pressure to add customer vertex v_j is higher, the lower the cost $\bar{c}_{i,j}$, the lower $\bar{c}_{j,0}$, and the higher $\bar{c}_{i,0}$. In contrary to the unmodified version of the savings based heuristic we do not take into consideration the cost $\bar{c}_{0,j}$. This makes our heuristic less a savings based heuristic than a delta cost based heuristic:

$$ \eta_{i,j} = \frac{1}{\bar{c}_{i,j} + \bar{c}_{j,0} - \bar{c}_{i,0} - \bar{c}_{\text{norm}} + 1} \qquad \forall v_i \in V, \forall v_j \in V $$

Notice that the reduced costs $\bar{c}$ may be negative and the triangle inequality is not valid for the subproblem! Therefore, we normalize the delta costs with the term $\bar{c}_{\text{norm}} = \min_{v_i \in V, v_j \in V_C} \bar{c}_{i,j} + \bar{c}_{j,0} - \bar{c}_{i,0}$ which can be calculated in advance.

- Local search: after constructing a path for each ant of the colony we apply a local search to the best path found using the algorithm described before.

- Pheromone initialization: each feasible customer vertex should be eligible for selection in the probabilistic construction step. Therefore, all pheromone values have to be initialized to a value > 0. This initial pheromone value is based upon the difference between the highest and lowest reduced costs of the ESPPRC and is calculated as

$$ \tau_{\text{min}} = \frac{1}{\bar{c}_{\text{max}} - \bar{c}_{\text{min}} + 1} $$

where $\bar{c}_{\text{min}}$ and $\bar{c}_{\text{max}}$ can be calculated in advance as $\min_{v_i, v_j \in V} \bar{c}_{i,j}$ and $\max_{v_i, v_j \in V} \bar{c}_{i,j}$.

- Pheromone update strategy: after each iteration the pheromones are updated. The update strategy takes into account an evaporation element, the path of the best ant of the current iteration, and the path of the best ant over all iterations so far. The new pheromone values for the next iteration are calculated as

$$ \tau_{i,j} = \tau_{\text{min}} + (1 - \rho)(\tau_{i,j} - \tau_{\text{min}}) + F_{i,j}^{\text{best}} \Delta \tau^{\text{best}} + F_{i,j}^{\text{elitist}} \Delta \tau^{\text{elitist}} \quad \forall v_i \in V, \forall v_j \in V $$

whereas $F_{i,j}^{\text{best}} = 1$, if arc $a_{i,j}$ is part of the best path of the actual iteration, and $F_{i,j}^{\text{elitist}} = 1$, if arc $a_{i,j}$ is part of the best path so far, otherwise they are 0. The pheromone modifying terms $\Delta \tau^{\text{best}}$ and $\Delta \tau^{\text{elitist}}$ depend on the total negative reduced cost of the best path of the current iteration and the best path over all iterations respectively. The lower the cost, the higher is the pheromone modifying effect of the paths. They can be calculated as

$$ \Delta \tau^{\text{best}} = \frac{1}{\sum_{v_i \in V} \sum_{v_j \in V} F_{i,j}^{\text{best}} (\bar{c}_{i,j} - \bar{c}_{\text{min}}) + 1} $$

$$ \Delta \tau^{\text{elitist}} = \frac{1}{\sum_{v_i \in V} \sum_{v_j \in V} F_{i,j}^{\text{elitist}} (\bar{c}_{i,j} - \bar{c}_{\text{min}}) + 1} $$
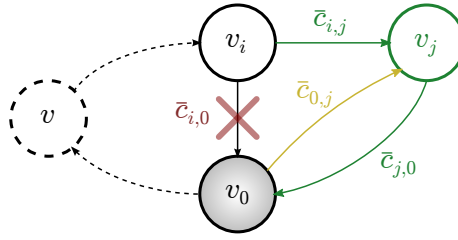
Figure 4.8: Situation for the construction heuristic when adding the customer vertex v_j to an actual path

Algorithm 4.3 composes these components and shows the ACO algorithm for approximate ESPPRC pricing subproblem solving. It has to be noted that the actual iteration limit it_{max} is divided by the size of the ant colony. This ensures that this population based metaheuristic can be compared with the other implemented simple point search algorithms.

ACO Choice of Parameters

Our ACO approach depends on four parameters: the *colony size* defines the number of ants that concurrently construct paths during one iteration. α expresses the influence of the pheromone values τ on the construction procedure, whereas β expresses the influence of the heuristic element η on the construction procedure. And lastly ρ defines the evaporation rate of the pheromone values.

To find a parameter setting that enables a solution of various problem instances with good performance we tested numerous parameter value combinations on a representative set of problem instances. As performance indicator we defined the spent CPU time of the column generation algorithm to find the optimal lower bound. This includes the CPU times for solving the RMP, for the exact ESPPRC subproblem solver, and of course for the ACO ESPPRC subproblem solver. The set of problem instances consisted of the Pirkwieser/Solomon instances p4r103 and p6c101 as well as the Pirkwieser/Cordeau instances 3a, $9a_{r1}$, $3b_{r1}$, and $8b_{r1}$ (for discussion about the instances please consult chapter 6.1).

The CPU time was normalized by dividing it through the average CPU time for all experiments performed on the same problem instance. The outcome of this is a comparable CPU time indicator which is distributed around 1. Values < 1 indicate that the column generation of those experiments was shorter than the average CPU time; values > 1 indicate that the experiments took longer than the average.

We performed ten experiments for each parameter setting and each problem instance. The parameters were set as following: colony size $\in \{1, 3, 10\}$, $\alpha \in \{0, 1, 2, 5\}$, $\beta \in \{0, 1, 2, 5\}$, and $\rho \in \{0, 0.01, 0.1\}$, which results in a total of $6_{\text{instances}} \times 10_{\text{runs}} \times 3_{\text{colony size}} \times 4_{\alpha} \times 4_{\beta} \times 3_{\rho} = 8640$ experiments.

To measure the different parameter settings we looked at each parameter separately and visualized the distribution of the CPU time indicator for all experiments clustered by the specific parameter values with a box plot, showing the minimal and maximal value, the lower and upper quartile, as well as the median of the CPU time indicator. Figure 4.9 shows four charts, one

Input: ESPPRC with reduced costs $\bar{c}$
Output: negative reduced costs paths

1 calculate $\bar{c}_{\min}$, $\bar{c}_{\max}$, $\bar{c}_{\text{norm}}$, and $\tau_{\min}$ from ESPPRC
2 initialize pheromones $\tau_{i,j} \longleftarrow \tau_{\min} \; \forall v_i, v_j \in V$
3 $NRC \longleftarrow \{\}$ `// negative reduced costs paths`
4 $s_{elitist} \longleftarrow \{\}$ `// best ant's path so far`
5 $it_{max} \longleftarrow 1000/\text{colony size}$
6 **for** $it \longleftarrow 1$ **to** it_{max} **do**
7 $s_{best} \longleftarrow \{\}$ `// iteration best ants path`
8   **for** *each ant in colony* **do**
9 $s \longleftarrow \{(v_0 \rightarrow v_0)\}$ `// initial path`
10 $v_i \longleftarrow v_0$
11   **while** $V_C^{feasible} \neq \{\}$ **do**
12   calculate $P_j \; \forall v_j \in V_C^{\text{feasible}}$ based on $\tau_{i,j}$, $\bar{c}_{i,j}$, $\bar{c}_{j,0}$, $\bar{c}_{i,0}$, and $\bar{c}_{\text{norm}}$
13 $v_j \longleftarrow$ select next vertex by roulette wheel selection based on P_j
14 $s \longleftarrow s \backslash \{(v_i \rightarrow v_0)\} \cup \{(v_i \rightarrow v_j), (v_j \rightarrow v_0)\}$ `// add vertex`
15 $v_i \longleftarrow v_j$
16   **end**
17   **if** $s_{best} = \{\}$ *or costs of* $s <$ *costs of* s_{best} **then** $s_{best} \longleftarrow s$
18   **end**
19 $s_{best} \longleftarrow$ local search of s_{best}
20   **if** $s_{elitist} = \{\}$ *or costs of* $s_{best} <$ *costs of* $s_{elitist}$ **then** $s_{elitist} \longleftarrow s_{best}$
21   calculate $\tau_{i,j} \; \forall v_i, v_j \in V$ based on $\tau_{i,j}$, $\tau_{\min}$, s_{best}, and $s_{elitist}$
                                                    `// pheromone update`
22   **if** *costs of* $s_{best} < 0$ **then**
23   **if** $s_{best} \notin NRC$ **then** $NRC \longleftarrow NRC \cup \{s_{best}\}$
24   **end**
25   **if** $it = it_{max}$ *and* $NRC = \{\}$ **then** $it_{max} \longleftarrow 5000/\text{colony size}$
26 **end**
27 **return** NRC

Algorithm 4.3: ACO algorithm for the ESPPRC pricing subproblem

for each parameter. The first chart shows the parameter *colony size*. The 8640 experiments are clustered by the colony sizes of 1, 3 and 10 ants. The distribution of the CPU time indicator for the 2880 experiments per colony size is shown in the corresponding box plot. The other three charts show the distributions for α, β and ρ respectively.

The charts show that for our algorithm a colony size of 3 ants has on average a better performance than a colony size of 1 or 10 ants. This is indicated by a lower median and also by lower quartile values. Interestingly the α parameter had no noticeable influence on the average performance of our experiments; the same is true for the parameter ρ. For the parameter β it seems that higher values show a worse performance.

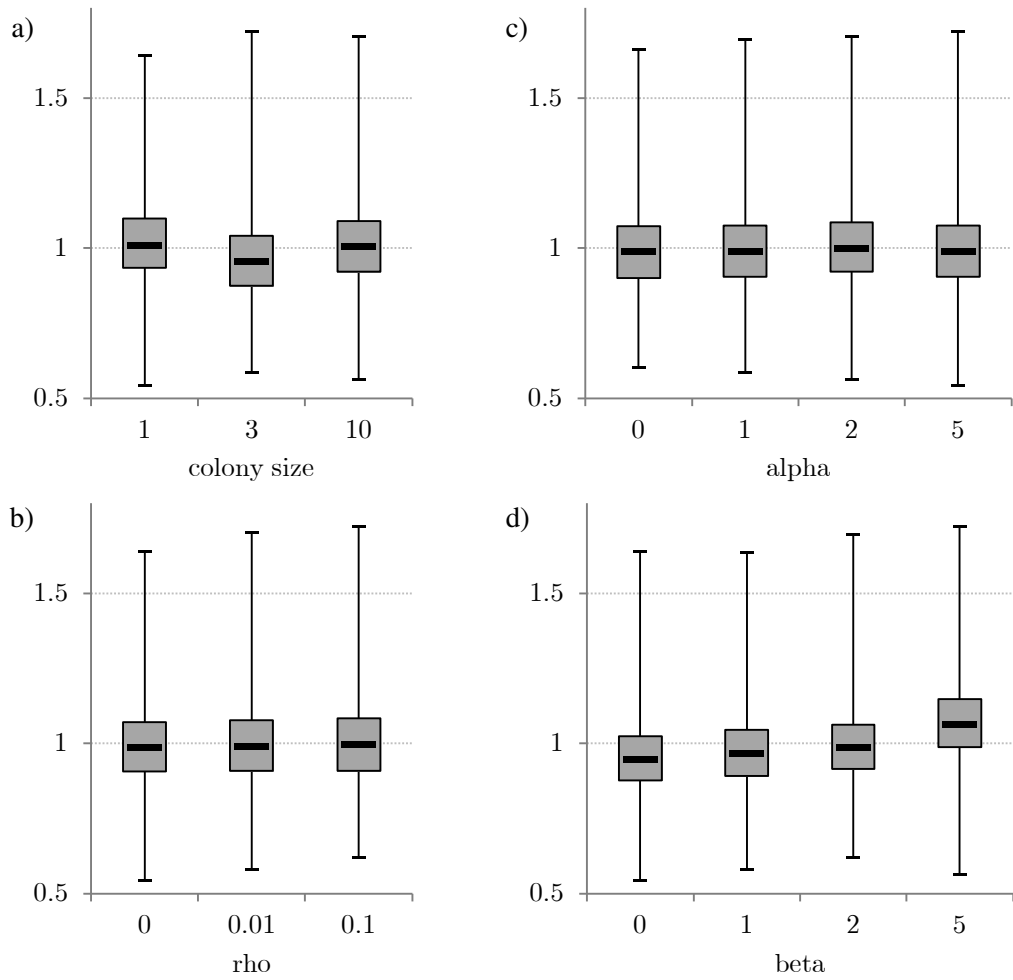Based on these charts we set the parameter values for our ACO algorithm to solve the ESP-

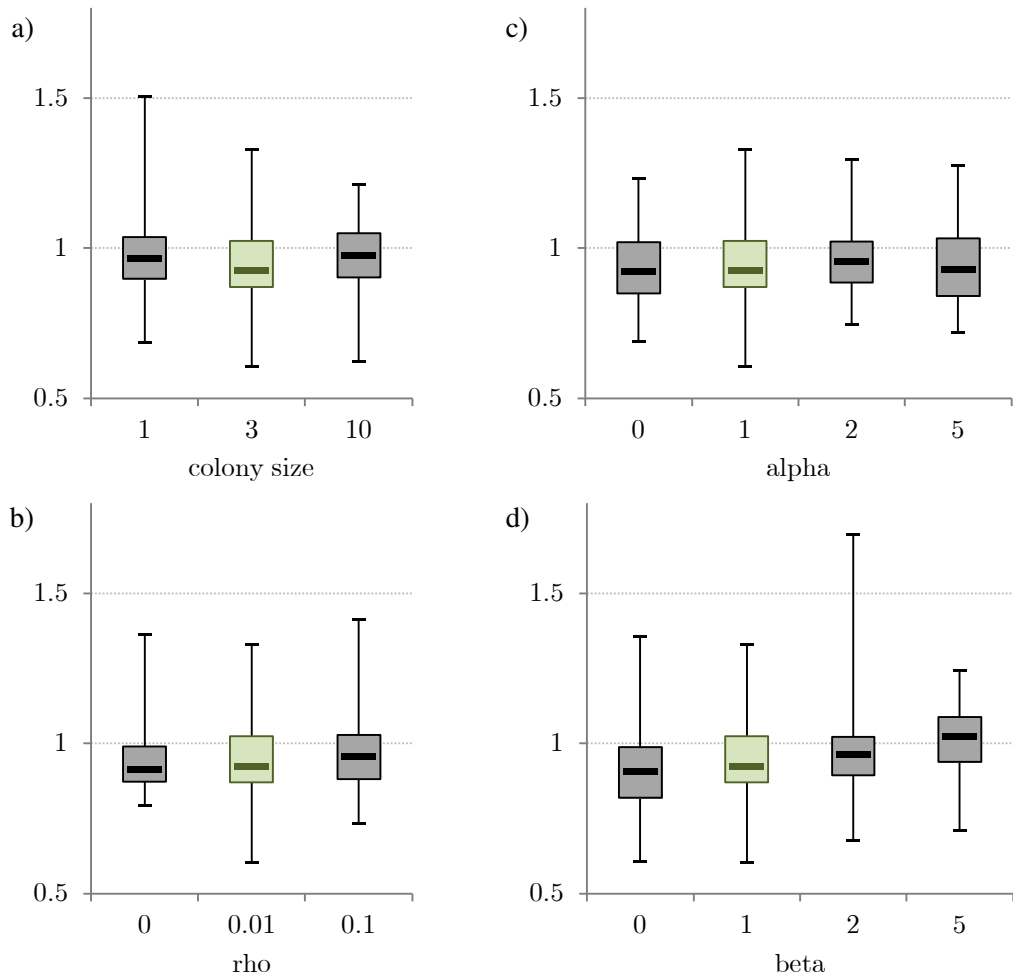**Figure 4.9:** Box plots showing the distribution of the CPU time indicator over all experiments for the ACO parameters a) *colony size*, b) ρ, c) α, and d) β

PRC pricing subproblem as shown in table 4.1.

Parameter	Value
colony size	3
α	1
β	1
ρ	0.01

Table 4.1: Parameter settings for the ACO algorithm

To verify the parameter setting we visualized the distribution of the CPU time indicator for the experiments made with the selected parameter values. We expect noticeably lower distri-

bution values than the average, especially the median should be below 1. Figure 4.10 shows the distribution based on the 60 experiments with selected parameter setting as green box plot confirming our expectations. Additionally each chart of this figure shows the distribution of the other parameter values for each parameter when fixing the values of the complementary parameters to the determined best parameter setting. A Wilcoxon rank-sum test[3] showed that the experiments with a colony size of 3 ants had a significantly better performance than with 1 or 10 ants (P-value < 0.05 one-tailed). The same is true for experiments with $\beta = 0$ or $\beta = 1$ compared with $\beta = 5$. For the other comparisons of parameter settings no significant evidence of a performance gain or loss could be found.

4.4 Implementation

The column generation approach to solve the linear relaxation of the PVRPTW was implemented in C++. The following class diagram 4.11 shows an overview of the implementation structure.

---

[3]in the literature also called the Mann-Whitney U test or Mann-Whitney-Wilcoxon test

Figure 4.10: Box plots showing the distribution of the CPU time indicator over the experiments with best parameter setting for the ACO parameters a) *colony size* with $\alpha = 1$, $\beta = 1$ and $\rho = 0.01$, b) ρ with *colony size* = 3, $\alpha = 1$ and $\beta = 1$, c) α with *colony size* = 3, $\beta = 1$ and $\rho = 0.01$, and d) β with *colony size* = 3, $\alpha = 1$ and $\rho = 0.01$

```
  ┌─────────────────────┐        ┌──────────────────────────────┐
  │     CplexModel      │        │       ColumnGenerator        │ 1
  ├─────────────────────┤        ├──────────────────────────────┤
  │+objective[1]        │        │-allRoutes[*]                 │
  │+variables[*]        │◁───────│+solveLinearRelaxation()      │◇───┐
  │+constraints[*]      │        │-createEspprc()               │    │
  │+solve()             │        └──────────────────────────────┘    │
  └─────────────────────┘                                            │ *
```

CplexModel
+objective[1]
+variables[*]
+constraints[*]
+solve()

ColumnGenerator   1
-allRoutes[*]
+solveLinearRelaxation()
-createEspprc()

*

Route
-depotLeavingTime[1]
-customers[*]

ProblemInstance
-costs[*]
-durations[*]
-noVehicles[1]
-maxVehicleLoad[1]
-maxRouteDuration[1]
+load()

1

«creates»

*

1

Espprc
-reducedCosts[*]
-durations[*]
-customerDemands[*]
-customerServiceTimes[*]
-customerTimeWindows[*]
-maxVehicleLoad[1]
-maxRouteDuration[1]
+negativeReducedCostsExact()
+negativeReducedCostsILS()
+negativeReducedCostsVNS()
+negativeReducedCostsGRASP()
+negativeReducedCostsACO()
-localSearch()
-bestInsert()
-bestRemove()
-bestReplace()
-bestMove()
-bestExchange()

*

Customer
-serviceFrequency[1]
-visitCombinations[*]
-demand[1]
-serviceTime[1]
-timeWindow[1]

Figure 4.11: Class diagram of the implemented column generation approach

ACO for whole Problem

In this chapter an ACO algorithm is presented to solve the whole PVRPTW heuristically. The algorithm focuses on the core elements of ACO and tries to apply them to the challenges of this highly constrained variant of vehicle routing problems.

5.1 Cascaded Ant Colony Optimization

The difficulty in applying the ACO metaheuristic to the whole PVRPTW is to find a method for optimization that meets two distinct aims: on the one hand, the optimal visit combinations for all customers have to be found; on the other, the best routes have to be determined satisfying temporal, capacity and fleet constraints. These two aims are hierarchically connected since an optimal routing depends on the selected visit combinations. Based on the structure of this problem we propose a hierarchical application of ACO to the PVRPTW: the *cascaded ant colony optimization* (cascaded ACO).

Algorithmic Principle

Our algorithm follows a simple decomposition principle. For a given combination of customers' visit combinations the PVRPTW is nothing more that t VRPTWs, one for each day of the planning period P. Therefore the PVRPTW can be interpreted as multiple VRPTWs, one for each day and combination of visit combinations. Thus it can be decomposed by enumerating all combinations of visit combinations and solving the resulting VRPTWs. The best solution composed of the relevant VRPTWs solutions determines the best routes as well as the optimal combination of visit combinations. Of course the size of the set of all combinations of visit combinations grows exponentially with the number of customers (as long as they have two or more visit combinations to select); therefore enumeration is not efficiently possible.

This is the reason why we apply ACO as an approximate solution strategy to find a good combination of visit combinations. We call this ACO the *upper level ACO* – it generates iteratively combinations of visit combinations based on pheromones that have been influenced by the

quality of previously generated solutions. To evaluate the quality of a solution it needs to solve the daily VRPTWs resulting from the selected visit combinations. These VRPTWs are solved by the *lower level ACO* which tries to find good VRPTW solutions in a short time.

Figure 5.1 illustrates the decomposition principle of our approach. Each ant of the upper level ACO searches the search space of the customers' visit combinations, the ants of the lower level ACO find a solution for the VRPTWs that result from the selected visit combinations of the actual upper level ant. Notice that each combination of visit combinations generates t independent VRPTWs, one for each day of the planning horizon.
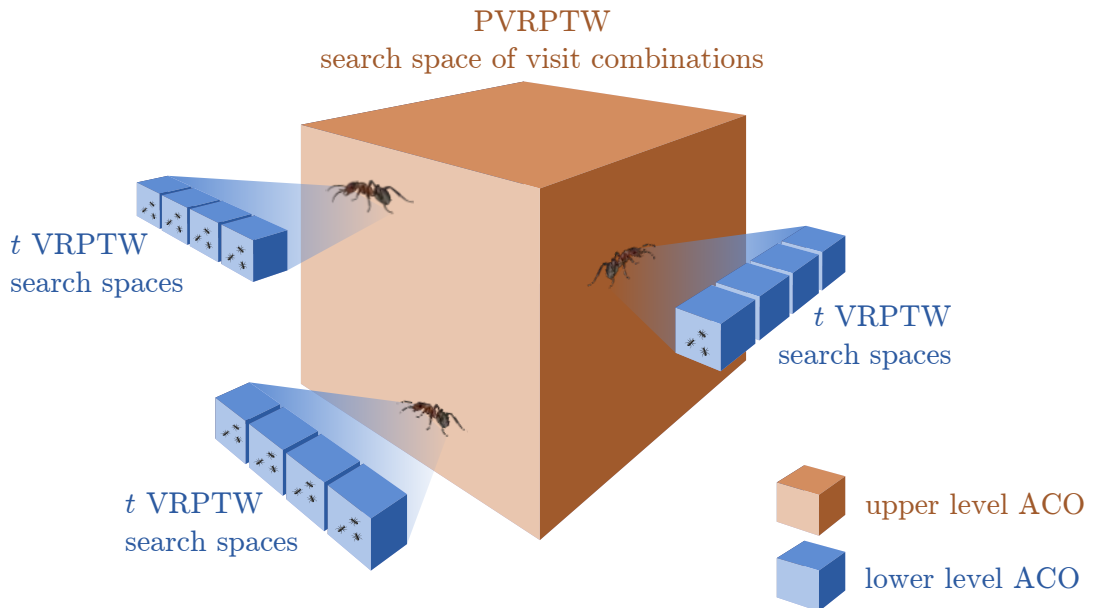
Figure 5.1: Decomposition principle of cascaded ACO

A related decomposition principle can be found in other approaches to solving periodic routing problems, e.g. by Cordeau et al. for the PVRP [13] and the PVRPTW [14]. But unlike these approaches we keep the solving procedures for the lower level VRPTW completely independent from the upper level PVRPTW solver and can therefore apply independent solution algorithms. A quite similar decomposition approach is found in Vianna et al. for the PVRP [94], where the chromosomes of a hybrid genetic algorithm represent the selected visit combinations and a fast heuristic solves the resulting VRPs for solution quality evaluation.

General Design Decisions

Algorithmic Structure

The two distinct ACO algorithms face different requirements. The upper level ACO is started once and has to converge continuously in a good solution. For each iteration exploitation of the search history and exploration of promising regions of the search space have to be balanced carefully. A premature convergence towards a local minimum has direct impact on the total

solution quality. A search pattern that focuses too much in exploring all regions of the search space leads to long runtimes. The computational effort for a single iteration of the upper level ACO is very high, since t VRPTWs have to be solved to evaluate the solution quality of this iteration. Therefore we decided to implement an ACO variant that makes use of each ant of the construction process. We used for that a slightly modified version of the ACO schema presented by Matos and Oliveira [63].

On the other hand, the lower level ACO is started numerous times with totally different but also with similar or even equal search space configurations. The main focus is to converge quickly since the runtime of the lower level ACO is a major performance factor for the overall cascaded ACO algorithm. Moreover, VRPTW solutions generated in spite of an early termination should nevertheless indicate if the actual combination of visit combinations has the potential to bear a good PVRPTW solution. The exploration drive of the lower level ACO does not has to be that explicit, because multiple starts triggered by different iterations of the upper level ACO induce an implicit exploration force. To cover these requirements we used the savings ant system introduced by Dörner et al. [33] which promises an early convergence based on a strong heuristic component.

A further challenge for the design of cascaded ACO was the fact that due to the short runtime of the VRPTW solving lower level ACO the final solution quality is not as good as it could be. Therefore we experimented with different algorithmic structures to improve the single VRPTW solution qualities for the final PVRPTW solution. Tests with a continuously increasing number of iterations for the lower level ACO, starting with a small number of iterations at the beginning and finishing with a reasonably high number of iterations at the end of cascaded ACO produced no satisfying results. In fact we measured the same runtime and quality when setting the number of iterations for the lower level ACO to a constant value between the starting and finishing number. Therefore we decided to introduce a finalizing intensification step to cascaded ACO. In this step the best PVRPTW solution is further improved by restarting the lower level ACO for each day of the planning period, initialized by the single VRPTW solutions of the PVRPTW solution. The lower level ACO gets for that a reasonably high number of iterations to improve the VRPTW solutions further.

The structure of our cascaded ACO algorithm is shown in figure 5.2. During the regular construction phase of the algorithm the upper level ACO constructs iteratively combinations of visit combinations which are evaluated by calling the lower level ACO for each day of the planning period. A local search step improves the combination of visit combinations based on the actual solution, and the pheromone update step concludes an iteration of the upper level ACO. The lower level ACO works as a nested iterative procedure that constructs vehicle routes which are improved by a local search procedure. Here too an iteration is completed with a pheromone update step. To support a reasonable algorithm runtime we decided to use a constant value k_U for the number of iterations of the upper level ACO. The number of iterations for the lower level ACO depends on a constant value k_L and is linearly scaled by the number of customers of the PVRPTW. This schema balances the computational effort for upper level and lower level solution generation.

The finalizing intensification phase can be interpreted as one additional iteration of the upper level ACO without construction of the combination of visit combinations. Instead the best solu-
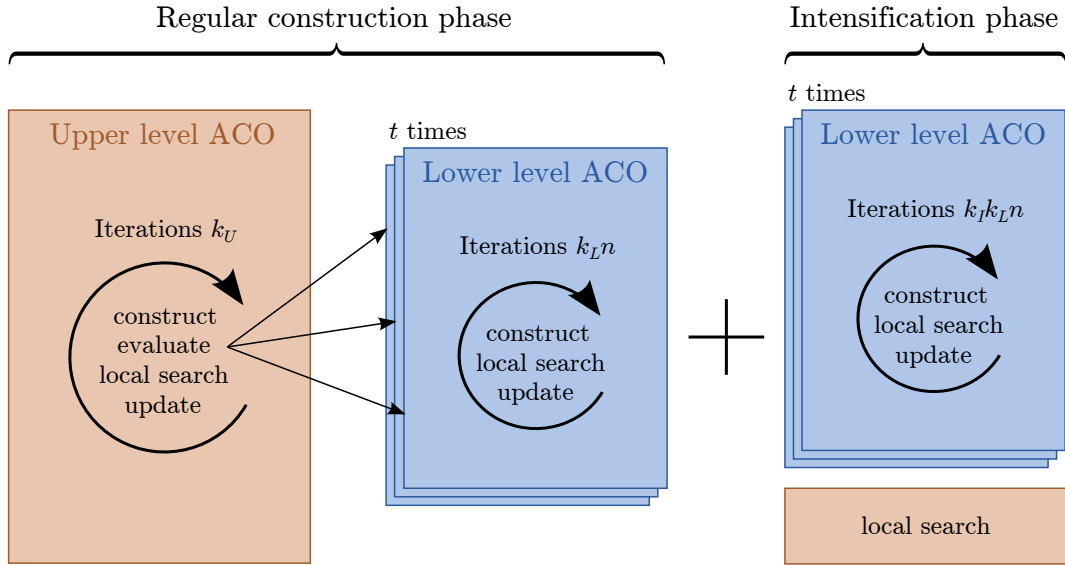
Figure 5.2: Overall structure of cascaded ACO

tion found during the regular construction phase is improved by starting the lower level ACO for each day with a number of iterations which is multiplied by the factor k_I. A concluding upper level local search to improve the visit combinations generates the final PVRPTW solution.

The total number of VRPTW solution generating lower level ACO iterations can therefore be calculated as $k_U t k_L n + t k_I k_L n$. The constant values have been determined experimentally for reasonable algorithm runtimes as $k_U = 1000$, $k_L = 5$, and $k_I = 20$.

Figure 5.3 shows an example of the behavior of cascaded ACO. The diagram for the PVRPTW solving progress displays for all 1000 (k_U) iterations of the upper level ACO the development of the solution quality. The black bold line represents the best solution found so far, whereas the gray dotted line shows the solution quality of the current iteration. To determine the PVRPTW solution cost the lower level ACO is called four times because in this example the problem instance defines a planning period of four days. For a single exemplary iteration the upper part of the figure shows the behavior of the four lower level ACO calls. Based on the constructed combination of visit combinations for that iteration four VRPTWs are solved using the fast converging lower level ACO. Since the problem instance contains 100 customers each solving procedure runs for 500 ($k_L n$) iterations. The combination of the best VRPTW solutions of all four days produces the PVRPTW solution which is improved further by an upper level local search resulting in the solution quality for the upper level ACO iteration.

Feasibility and Penalties

Cascaded ACO does accept just feasible intermediate PVRPTW as well as VRPTW solutions regarding visit, capacity, duration, and time window constraints. The fleet constraint may be violated for a solution but it is penalized by a high constant value for each extra vehicle needed. The penalty $c_P^{VRPTW} = k_P \max(\text{routes}(s^{VRPTW}) - m, 0)$ is added to the costs when evaluating a
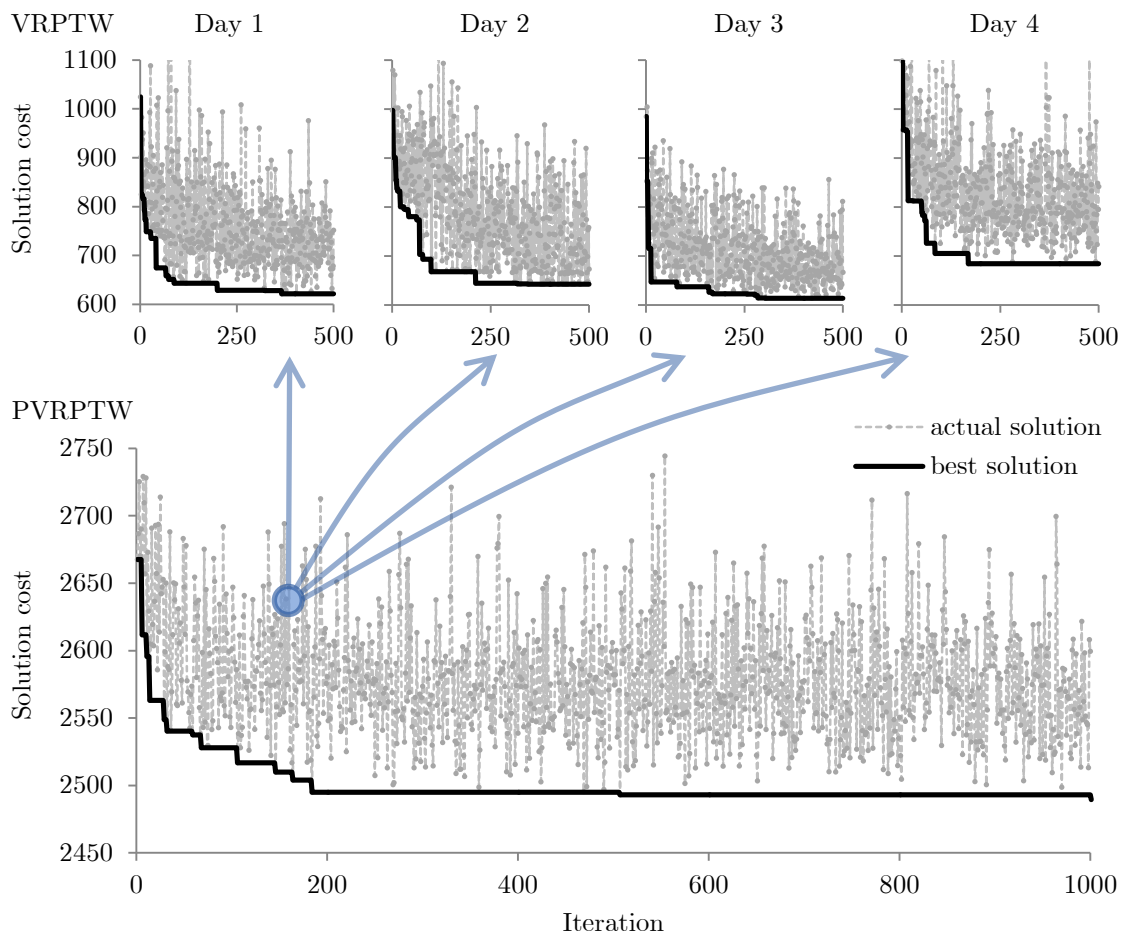
Figure 5.3: Example of the behavior of the cascaded ACO algorithm

lower level VRPTW solution, the penalty $c_P^{PVRPTW} = k_P \max(\text{vehicles}(s^{PVRPTW}) - m, 0)$ is added to the costs when evaluating an upper level PVRPTW solution, whereas the function routes() determines the number of routes a VRPTW solution contains, and the function vehicles() determines the vehicles needed for a PVRPTW solution.

Independent Solvers

It has to be noted that in principle the algorithmic structure does not need to implement the lower level solver as an ACO algorithm. Any fast converging VRPTW solving algorithm can substitute the lower level ACO. Even for the intensification phase a distinct VRPTW solver could be used which focuses less on fast convergence and more on the solution quality.

5.2 Upper Level ACO

The upper level ACO is implemented as an ordinary ACO algorithm based on a pheromone structure as memory, a probabilistic construction step to determine a combination of visit combinations, a local search step to improve the constructed solution, and a pheromone update step to allow exploitation of the search history. But since the construction of a combination of visit combinations does not construct a full PVRPTW solution the lower level ACO is needed to generate such a solution.

Algorithm 5.1 shows the upper level ACO consisting of the regular construction phase and the intensification phase. After initializing the pheromone structure the algorithm iterates for k_U iterations[1]. Each iteration starts with the probabilistic construction of a combination of visit combinations based on the pheromone values of previous iterations. For each day of the planning period the set of customers is built which has to be visited on that day regarding the selected combination of visit combinations. This set defines a VRPTW for that day which is solved by the lower level ACO. The solutions of the VRPTWs for each day are combined to a PVRPTW solution that is improved further by a local search step that optimizes the visit combinations of the current solution. Based on the current and best solution found so far the pheromones are updated. Please notice that no ant colony is used for an iteration, since each ant modifies the pheromone values. After the regular construction phase intensification is performed by reading the combination of visit combinations of the best solution. This determines for each day of the planning horizon the set of customers that have to be visited. By starting the lower level ACOs with an initial VRPTW solution for the corresponding day the algorithm improves the current best solution further. The combination of the improved VRPTW solutions results in an improved PVRPTW solution which is finalized by a local search call.

Pheromone Structure

The purpose of the pheromone structure of the upper level ACO is to memorize combinations of visit combinations of previous iterations. Therefore we propose a simple two-dimensional hierarchical structure: the first dimension represents the customer $v_i \in V_C$; the second dimension represents the visit combination of the customer $C_{i,x} \in R_i$. The size of the structure is $\sum_{v_i \in V_C} r_i$ elements. Each element contains the pheromone value of a specific customer's visit combination.

Construction Step

The construction step has to construct a complete PVRPTW solution. On the one hand, it consists of the probabilistic construction of a combination of visit combinations and, on the other, of the evaluation of it by generating solutions for the resulting VRPTWs.

For the generation of a combination of visit combinations the classic ACO construction schema is applied: the solution elements are the single customer's visit combinations. To select a solution element the algorithm chooses probabilistically a visit combination of the set of visit

---

[1]We use the subscript $_U$ to mark elements belonging to the upper level ACO and the subscript $_L$ for elements of the lower level ACO

Input: PVRPTW
Output: best solution found by cascaded ACO
```
// regular construction phase
```
1 $\tau_U \longleftarrow$ initialize pheromones
2 $s_{Ubest} \longleftarrow$ null
3 **repeat**
4    $vc \longleftarrow$ create visit combinations using τ_U
5   **foreach** *day* $p \in P$ **do**
6      $V_{Cp} \longleftarrow$ set of customers to be visited on day p regarding vc, $V_{Cp} \subseteq V_C$
7      $s_{Lp} \longleftarrow$ `LowerLevelACO(`k_L`, `*PVRPTW*`, `V_{Cp}`)` `// solving the VRPTW`
8   **end**
9    $s_U \longleftarrow$ compose solution as $\bigcup_{p \in P} s_{Lp}$
10   $s_U^* \longleftarrow$ `LocalSearch`$_U$`(`s_U`)`
11   **if** s_{Ubest} *is null or costs of* $s_{Ubest} >$ *costs of* s_U^* **then** $s_{Ubest} \longleftarrow s_U^*$ $\tau_U \longleftarrow$ update pheromones using s_U^* and s_{Ubest}
12 **until** *termination criterion* k_U *iterations*
```
// intensification phase
```
13 **foreach** *day* $p \in P$ **do**
14   $V_{Cp} \longleftarrow$ set of customers visited on day p regarding visit combinations of s_{Ubest}
15   $s_{Lp} \longleftarrow$ VRPTW solution for day p of s_{Ubest}
16   $s_{Lp} \longleftarrow$ `LowerLevelACO(`$k_I k_L$`, `*PVRPTW*`, `V_{Cp}`, `s_{Lp}`)`
17 **end**
18 $s_U \longleftarrow$ compose solution as $\bigcup_{p \in P} s_{Lp}$
19 $s_U^* \longleftarrow$ `LocalSearch`$_U$`(`s_U`)`
20 **return** s_U^*

Algorithm 5.1: Upper level ACO algorithm

combinations for a customer. The probability to choose a visit combination $C_{i,x}$ of a customer v_i is determined by

$$P_{i,x} = \frac{\tau_{U\,i,x}{}^{\alpha_U}}{\sum\limits_{y \in R_i} \tau_{U\,i,y}{}^{\alpha_U}} \qquad\qquad \forall v_i \in V_C, \forall C_{i,x} \in R_i$$

Please notice the missing factor for a heuristic component $\eta_U{}^{\beta_U}$ that differentiates the construction schema implemented here from a classic ACO construction schema. The difficulty is to find an appropriate factor η_U that distinguishes between good and bad visit combinations for a customer. Experiments with heuristic factors that try to equally distribute customers over the days of the planning period did not markedly improve the solution quality; therefore we decided to remove the heuristic component entirely. Nevertheless, this might be a topic for further improvement of cascaded ACO.

Unlike the construction step for classic routing elements the solution elements for the upper level ACO are independent and consist ordinarily of a limited number of choices. Therefore, the sequence of customers for construction of the combination of visit combinations does not

influence the result. The customers are simply looped and the visit combination is determined by roulette wheel selection based on the probabilities $P_{i,x}$.

The selected visit combinations determine the customers that have to be visited on the different days of the planning period. The lower level ACO then generates a VRPTW solution for each day; the combination of these solutions comprises a PVRPTW solution that is used for evaluation of the selected combination of visit combinations.

Local Search

The upper level ACO uses a local search step to improve the PVRPTW solution generated by the construction step. For this purpose the solution is iteratively modified until no further improvement is possible. One iteration tries to perform a single visit combination exchange move. For that a customer is selected and the algorithm tries to change the visit combination of this customer to obtain a better solution. Actually, the local search operator tests all alternative visit combinations; if an improvement is found the best visit combination exchange move is performed; if not the next customer is tested. This local search procedure is performed using a randomized best exchange schema to optimality, i.e. the sequence of customers is determined by random, the operator always selects the best exchange alternative, and the local search procedure stops if no improvement could be performed for all customers.

Algorithm 5.2 shows the local search procedure for the upper level ACO. The outermost loop iterates until no customer's visit combination could be improved. The next inner loop tests all customers in random sequence. For each customer the best delta cost Δc_{best} is determined by looping the visit combinations. For a visit combination all days are checked if the customer has to be removed from or inserted into the days. If the customer is neither removed from nor inserted to a day, the position of the customer at that day is not changed. For each remove or insert operation the delta costs are calculated. This implies that the best insert position has to be found in the case of an insert operation. The sum of all delta costs for a visit combination defines the quality of the visit combination exchange move. The best exchange move is determined by the visit combination with the highest negative delta costs.

To illustrate a local search operation an example is shown in figure 5.4. The first row displays the solution before the local search operation was performed. The PVRPTW instance consists of nine customers, three vehicles, and a planning horizon of four days. Customer v_1 is selected for improvement. Its current visit combination determines that customer v_1 has to be visited on day 1 and day 3, but the local search operator identified a better visit combination: day 2 and day 4. Therefore customer v_1 is removed from the routes of the days 1 and 3 and is inserted to the best position (that is the best route as well as the best position inside this route) of the days 2 and 4. This is displayed in the second row of the figure.

Actually the local search operator goes one step further: it applies additionally a route local *node insertion move* operator to the routes where the customer has to be removed and the routes where it has to be inserted for further improvement. For explanation of the node insertion move please see section Local Search of chapter 5.3.

Since solutions are allowed that violate the fleet constraint, the local search procedure serves also as an important factor to generate feasible solutions: Removing the last node of a route is the same as reducing the number of vehicles by one for that specific day. If the number of

Input: PVRPTW solution s_U

Output: PVRPTW solution with local minimal costs

1 repeat

2 $\widetilde{V_C} \longleftarrow$ random permutation of customers from PVRPTW

3 $\tilde{i} \longleftarrow 1$

4 $improved \longleftarrow false$

5   **repeat**

6 $v_i \longleftarrow$ customer at position $\tilde{i}$ of $\widetilde{V_C}$

7 $\Delta c_{best} \longleftarrow 0; C_{i,best} \longleftarrow null$

8   **foreach** *visit combination $C_{i,x} \in R_i$* **do**

9 $\Delta c \longleftarrow 0$

10   **foreach** *day $p \in P$* **do**

11   **if** *customer has to be removed from day p* **then** $\Delta c \longleftarrow \Delta c +$ remove cost

12   **if** *customer has to be inserted to day p* **then** $\Delta c \longleftarrow \Delta c +$ insert cost

13   **end**

14   **if** $\Delta c < \Delta c_{best}$ **then** $\Delta c_{best} \longleftarrow \Delta c; C_{i,best} \longleftarrow C_{i,x}$

15   **end**

16   **if** $\Delta c_{best} < 0$ **then**

17 $s_U \longleftarrow$ perform remove and insert operations on s_U changing to visit combination $C_{i,best}$

18 $improved \longleftarrow true$

19   **end**

20 $\tilde{i} \longleftarrow \tilde{i} + 1$

21   **until** $improved$ or $\tilde{i} > |V_C|$

22 until *not improved*

23 return s_U

Algorithm 5.2: Upper level local search procedure

vehicles exceeds the fleet size such a move is highly rewarded because the penalty is reduced or even removed from the new solution. This automatically produces pressure to generate feasible solutions regarding the fleet constraint.

Pheromone Update

The update schema for the upper level ACOs pheromones was inspired by the update schema of Matos and Oliveira [63]. It uses evaporation, the best solution found so far, and the solution of the current iteration to modify the pheromone values for the next iteration. The pheromones are updated each iteration using the formula

$$\begin{aligned} \tau_{U i,x} = \tau_{U min} &+ (1 - \rho_U)(\tau_{U i,x} - \tau_{U min}) \\ &+ F_{i,x}^{\text{best}} \Delta \tau_U{}^{\text{best}} + F_{i,x}^{\text{actual}} \Delta \tau_U{}^{\text{actual}} \qquad\qquad \forall v_i \in V_C, \forall C_{i,x} \in R_i \end{aligned}$$

Day 1	Day 2	Day 3	Day 4

PVRPTW solution with selected visit combination of customer v_1: Day 1 + Day 3:

PVRPTW solution with selected visit combination of customer v_1: Day 2 + Day 4:
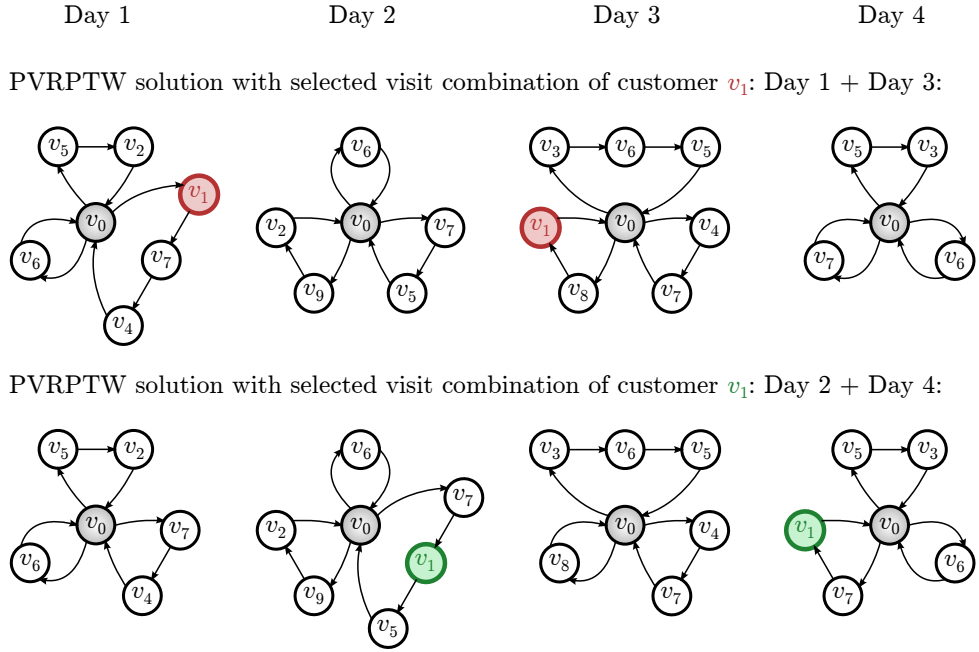
Figure 5.4: Example of a single local search operation for the upper level ACO

where $F_{i,x}^{\text{best}}$ is 1 if visit combination $C_{i,x}$ was selected for customer v_i in the best solution found so far, otherwise it is 0. Respectively, $F_{i,x}^{\text{actual}}$ indicates if visit combination $C_{i,x}$ was selected for customer v_i in the solution of the actual iteration. The upper level ACO uses a constant value to increase pheromones by the best solution:

$$\Delta\tau_U{}^{\text{best}} = \sigma_U$$

The increment by the current solution depends on the quality of the solution related to the quality of the best solution found so far:

$$\Delta\tau_U{}^{\text{actual}} = \omega_U \frac{c^{\text{best}}}{c^{\text{actual}}}$$

whereas c^{actual} represents the total costs of the actual PVRPTW solution and c^{best} the costs of the best solution.

The pheromone structure is initialized at the beginning of the upper level ACO by setting all pheromones to the value $\tau_{U\,min}$. This is also the lower limit of the pheromone values ensured by the update schema. $\tau_{U\,min}$ is set to the constant value 1.

Parameters

The upper level ACO contains the following parameters that have to be setup for an algorithm execution with good performance regarding quality of solution and algorithm runtime:

Parameter	Description
α_U	Emphasis of the pheromone values for selection of visit combinations
ρ_U	Evaporation factor for pheromone values
σ_U	Increment value for pheromones based on the best solution
ω_U	Increment value for pheromones based on the current solution

Table 5.1: Parameters of the upper level ACO

5.3 Lower Level ACO

The challenge for the lower level ACO is to solve a VRPTW in a short time, i.e. with few iterations. We therefore implemented a savings based ant system algorithm as introduced by Dörner et al. [33] for the VRP, slightly adapting it to intensify the support for early convergence.

The savings based ant system combines the exploratory properties of ACO with the savings heuristic proposed by Clarke and Wright [10] that assumes starting with n vehicles, each servicing just one single customer. Then iteratively the two vehicle routes are merged that produce the "maximum saving" of costs which is determined by the term $\delta^{saving} = c_{i,0} + c_{0,j} - c_{i,j}$, where v_i is the last node of the first vehicle route and v_j the first node of the second vehicle route.

To speed up the construction procedure and to explicitly balance exploitation and exploration of the lower level ACO we added the pseudo-random-proportional state transition rule that was originally introduced by Dorigo and Gambardella [30] for the ant colony system.

The savings based ant system uses a colony of ants which concurrently explore the search space. Each ant constructs a VRPTW solution which is further improved by a randomized local search procedure based on an inter-route node insertion move operator.

The best solution found so far and the top ants of the actual ant colony bias the update schema for the pheromones.

An overview of the lower level ACO is shown in algorithm 5.3. After initialization of the pheromones the algorithm initializes the best solution found so far. If an initial VRPTW solution was passed as a parameter to the lower level ACO to intensify an already generated solution, the best solution is initialized with that solution, otherwise the best solution is set to $null$. Then the algorithm enters the main loop. The iterations are given as input parameter enabling the upper level ACO to call the lower level ACO during the regular construction phase (using less iterations) as well as the intensification phase (using more iterations). The real loop count is upscaled by the number of customers and downscaled by the colony size. The downscale is performed since we prefer to loop over ants rather than iterations. In the main loop a sorted attractiveness matrix is generated that is the basis of the construction step for each ant of the colony. The ants generate solutions by performing a probabilistic construction step followed by a local search step. After each ant of the colony has created a solution the pheromones are updated. The best solution of all ants is returned.

Input: iterations, PVRPTW, V_C set of customers, optional s_{Linit} initial VRPTW solution
Output: best VRPTW solution found by lower level ACO

1 $\tau_L \longleftarrow$ initialize pheromones
2 $s_{Lbest} \longleftarrow s_{Linit}$
3 $it_{max} \longleftarrow n \times$ iterations / colony size
4 **repeat**
5 $\xi_L \longleftarrow$ calculate attractiveness for each customer $\in V_C$ using τ_L
6 $S_L^{colony} \longleftarrow \{\}$ `// set of colony solutions`
7   **foreach** *ant in colony* **do**
8 $s_L \longleftarrow$ construct solution using ξ_L
9 $s_L^* \longleftarrow$ `LocalSearch`$_L$`(`s_L`)`
10   **if** s_{Lbest} *is null or costs of* $s_{Lbest} >$ *costs of* s_L^* **then** $s_{Lbest} \longleftarrow s_L^*$
                  $S_L^{colony} \longleftarrow S_L^{colony} \cup s_L^*$
11   **end**
12 $\tau_U \longleftarrow$ update pheromones using s_{Lbest} and top solutions of S_L^{colony}
13 **until** *termination criterion* it_{max} *iterations*
14 **return** s_{Lbest}

<div align="center">Algorithm 5.3: Lower level ACO algorithm</div>

Construction Step

Before an ant can construct a solution an attractiveness matrix is generated that is used by the whole colony. The attractiveness of a solution element, that is, an arc $a_{i,j}$, is calculated as

$$\xi_{L i,j} = \tau_{L i,j}^{\alpha_L} \times \eta_{L i,j}^{\beta_L} \qquad\qquad \forall v_i \in V, \forall v_j \in V_C$$

where $\tau_{L i,j}$ represents the pheromone value of the arc biased by previously generated solutions and $\eta_{L i,j}$ is the heuristic component. α_L and β_L are the parameters that balance the influence of pheromones and problem dependent heuristic. For the savings based ant system the heuristic component is the savings term of the savings heuristic $\eta_{L i,j} = c_{i,0} + c_{0,j} - c_{i,j}$.

The attractiveness matrix consists of $n + 1$ attractiveness vectors, one for each vertex (including the depot). The cells of a vector represent the attractiveness values of all arcs leaving the vertex which is similar to the attractiveness of a customer to follow the vertex in a route. These arcs are sorted in decreasing order based on the attractiveness value. This ensures that during the roulette wheel selection of the construction step of an ant the solution elements with higher probability appear before solution elements with lower probability. It even allows us to skip less probable arcs and accelerate the construction step.

After generation of the attractiveness matrix the ant colony generates the VRPTW solutions. For that purpose each ant starts with an empty route, and the customer vertexes are appended probabilistically. We implemented the *pseudo-random-proportional state transition rule* to determine how a new customer vertex is appended.

The basis of the pseudo-random-proportional state transition rule is the preset probability value $q_{0 L} \in [0, 1]$. $q_{0 L}$ determines the probability that the next vertex to select is the best option

of vertexes based on the attractiveness. Therefore, every time a vertex has to be appended to a route a random value $q \in [0, 1[$ is generated. If $q < q_{0L}$ the attractiveness vector of the last customer in the actual route is scanned for the arc with the highest attractiveness. The scanning is necessary because only feasible solutions are allowed. Nevertheless it is efficient because the attractiveness vector is sorted. If $q \geq q_{0L}$ the next vertex to select is determined probabilistically by roulette wheel selection based on the attractiveness vector. The parameter q_0 directly influences the balance between exploitation and exploration: a high value of q_{0L} intensifies the exploitation force; a low value of q_{0L} favors the biased exploration strategy of the classic ant system.

If the next customer vertex to select for a route that ends with vertex v_i has to be determined by roulette wheel selection, the probability of a vertex v_j is calculated as

$$
P_j = \begin{cases} \dfrac{\xi_{Li,j}}{\sum\limits_{k \in \overline{V_C}^{\text{feasible}}} \xi_{Li,k}} & \forall v_j \in \overline{V_C}^{\text{feasible}} \\ 0 & \forall v_j \notin \overline{V_C}^{\text{feasible}} \end{cases}
$$

where $\overline{V_C}^{\text{feasible}}$ is the reduced set of customer vertexes that can be reached by the vertex v_i in the current route without violating any constraints. The set is reduced by letting only the top μ_L customer vertexes participate in the roulette wheel selection. The parameter μ_L defines the size of the neighborhood for the roulette wheel selection of the construction step. From a performance perspective this reduces the algorithm's runtime since the attractiveness vectors are sorted which allows a part of the customer vertexes to be skipped.

By combining the pseudo-random-proportional state transition rule and neighborhood reduction the parameter q_{0L} can also be interpreted as the probability that the neighborhood μ_L is set to 1. Therefore, q_{0L} and μ_L both regulate the balance between exploitation and exploration of the lower level ACO.

If $\overline{V_C}^{\text{feasible}}$ becomes empty but there are still customers in V_C that have not been serviced yet, the current route is closed and a new empty route is started. The construction step stops when all customer vertexes have been assigned to a route of the VRPTW solution.

Local Search

When the construction step of an ant is finished it has generated a feasible VRPTW solution regarding capacity, duration, and time window constraints. The fleet constraint may be violated, because the construction step cannot prevent solutions that need too many vehicles. The local search procedure of the lower level ACO has two purposes: first it has to improve the quality of the VRPTW solution regarding solution costs. Second it should encourage the solution to become feasible regarding the fleet constraint.

The basic operation of the lower level ACOs local search algorithm is the single *node insertion move* (NIM). Experiments with other moves such as the single *node exchange move*, the *Lin 2-exchange move*, the *Lin 3-exchange move*, and the *Or insertion move* did not improve performance regarding quality and runtime sufficiently or at all. Therefore, and because we wanted to focus more on the properties of the ACO than of the local search procedure, we decided to stay with the simple NIM.

We use the NIM in two variants: the route local NIM moves an arbitrary customer vertex inside a single route to a new position; the inter-route NIM removes a customer from an arbitrary route and inserts it into a new route. Figure 5.5 shows an example of the two NIM variants.
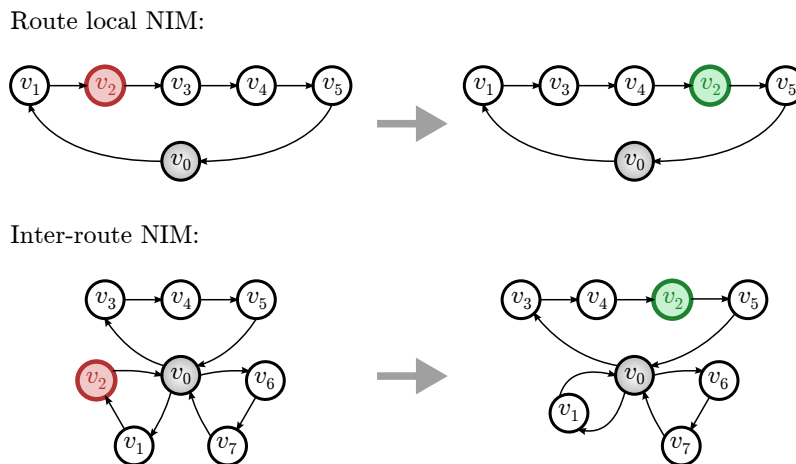
Route local NIM:

Inter-route NIM:

Figure 5.5: Node insertion moves for the local search of the lower level ACO

There are $|V_C|(|V_C| - 1)$ different NIMs to check for improvement during the local search to reach a local minimum. But since we experienced that for larger problem instances ($n > 100$) the runtime of the lower level ACOs local search becomes the dominant part of the whole cascaded ACO we decided not to implement it to reach a local minimum. Instead we reduced the runtime complexity of our local search implementation by linearization through randomization. To achieve this, a local search operation takes two random routes – a source route and a destination route – from the VRPTW solution. If the routes are identical then route local NIMs are performed to optimality. If they differ the first inter-route NIM between the two routes that improves the solution costs is performed; then the source and the destination route are further improved by applying route local NIMs to optimality. The local search operations are repeated n times before the local search algorithm terminates.

Additionally we defined the parameter $r_{0L} \in [0, 1]$. r_{0L} determines the probability for vehicle reduction: if the VRPTW solution is not feasible regarding the fleet constraint, then with a probability of r_{0L} the source route is not selected randomly. Instead the route with the fewest customers serviced is selected as source route, and the first inter-route NIM is performed even when it does not improve solution quality but increases solution costs. This rule supports the reduction of vehicles since it encourages a reduction in the route size (in terms of number of customers) of the smallest route until it is reduced to zero.

The local search procedure for the lower level ACO is shown in algorithm 5.4. In the main loop the random value r is compared with the parameter r_{0L}. If $r < r_{0L}$ the algorithm tries to reduce the number of routes by removing a customer from the route with the fewest customers and inserting it into a random route. Otherwise two random routes are selected for inter-route NIM. To accept an inter-route NIM the costs of the solution have to be reduced ($\Delta c < 0$) or the algorithm tries currently to reduce the number of vehicles (routes(s_L) > m and $r < r_{0L}$).

Input: VRPTW solution s_L
Output: potentially improved VRPTW solution

1 repeat
2 $r \longleftarrow$ random value between 0 and 1
3   **if** *routes$(s_L) > m$ and $r < r_{0L}$* **then**
4 $route_S \longleftarrow$ route with fewest serviced customers of s_L
5   **else**
6 $route_S \longleftarrow$ random route of s_L
7   **end**
8 $route_D \longleftarrow$ random route of s_L
9   **if** *$route_S = route_D$* **then**
10 $s_L \longleftarrow s_L$ with $route_S$ improved by route local NIMs to optimality
11   **else**
12   **foreach** $v_i \in route_S \backslash \{v_0\}$ **do**
13   **foreach** $v_j \in route_D$ **do**
14 $\Delta c \longleftarrow$ remove cost v_i from $route_S +$ insert cost v_i after v_j in $route_D$
15   **if** *$\Delta c < 0$ or (routes$(s_L) > m$ and $r < r_{0L}$)* **then**
16 $route_S \longleftarrow route_S$ with v_i removed
17 $route_D \longleftarrow route_D$ with v_i inserted after v_j
18 $s_L \longleftarrow s_L$ with $route_S$ improved by route local NIMs to optimality
                 and $route_D$ improved by route local NIMs to optimality
19   **break** *foreach loops*
20   **end**
21   **end**
22   **end**
23   **end**
24 until *termination criterion n iterations*
25 return s_L

Algorithm 5.4: Lower level local search procedure

Pheromone Update

The pheromones of the lower level ACO are updated after an ant colony has created their VRPTW solutions. For that the update schema takes into consideration an evaporation effect as well as the best solution found so far and the σ_L best solutions of the current ant colony – we call them the top solutions of the ant colony. Let ρ_L be the evaporation factor of the lower level ACO then the pheromones are updated as

$$\tau_{Li,j} = (1 - \rho_L)^{\text{colony size}} \tau_{Li,j} + F_{i,j}^{\text{best}} \Delta \tau_L^{\text{best}} + \sum_{k=1...\sigma_L} F_{i,j}^{\text{top}_k} \Delta \tau_L^{\text{top}_k} \quad \forall v_i \in V, v_j \in V_C$$

$F_{i,j}^{\text{best}}$ is 1 if the arc $a_{i,j}$ is part of the best solution, otherwise it is 0. $F_{i,j}^{\text{top}_k}$ is 1 if the arc $a_{i,j}$ is part of the top solution with rank k, otherwise it is 0. The best of the top solutions has rank 1, the worst of the top solutions rank σ_L.

The pheromone increment for the best solution is determined as a constant value based on the number of top ants:

$$\Delta\tau_L^{\text{best}} = \sigma_L + 1$$

whereas the pheromone increment for the top solutions depends on the rank of the solution scaled by the solutions quality:

$$\Delta\tau_L^{\text{top}_k} = (\sigma_L + 1 - k)\frac{c^{\text{best}}}{c^{\text{top}_k}}$$

where c^{best} represents the VRPTW solution costs of the best solution found so far and c^{top_k} the costs of the top solution with rank k.

Notice that we scale the evaporation factor by the colony size. This also allows for few iterations to produce an evaporation effect. With this evaporation scaling, pheromone values of arcs that are never part of a best or top solution decrease independently of the colony size.

The pheromones are initialized at the start of the lower level ACO with the value 1. If a pheromone decreases to a value lower then τ_{Lmin} the pheromone value is reset to τ_{Lmin}. This prevents arcs from disappearing as candidates for a solution, although we determined τ_{Lmin} experimentally to the very low value of 10^{-6}. Therefore, the lower limit of the pheromones is only relevant during the intensification phase of the cascaded ACO, because the few iterations of the lower level ACO during the regular construction phase prevent the pheromone values from reaching τ_{Lmin} (even with relatively high evaporation rates ρ_L).

Parameters

The lower level ACO contains the following parameters that have to be setup for an algorithm execution with good performance regarding quality of solution and algorithm runtime:

Parameter	Description
$colony\ size_L$	Size of the ant colony
q_{0L}	Probability for the pseudo-random-proportional state transition rule
α_L	Influence of pheromone values for construction step
β_L	Influence of savings based heuristic for construction step
μ_L	Neighborhood size for construction step
r_{0L}	Probability of reducing the number of vehicles for local search
ρ_L	Evaporation factor for pheromone values
σ_L	Number of top ants in the ant colony

Table 5.2: Parameters of the lower level ACO

5.4 Intensification

As already mentioned the intensification phase is a simple "restart" of the lower level ACO initialized with a predefined solution, followed by a final upper level local search step. However,

intensification is reasonable especially for larger instances ($n > 100$). Due to the very limited iterations of the lower level ACO during the regular construction phase, the algorithm to solve the VRPTWs of large instances cannot converge quickly enough to generate an excellent solution.

Although for intensification the lower level ACO starts with a "clean" pheromone matrix, the preset VRPTW solution as best solution found so far rapidly generates pheromone values that favor the surrounding search space around this VRPTW solution. This leads quickly to a pheromone memory similar to that of the original lower level ACO call, thus supporting a successful intensification.

With the constants for iteration determination that we selected ($k_U = 1000$, $k_I = 20$) the intensification phase does not take more than approximately 2% of the total cascaded ACO runtime. But since the intensification causes a twenty-fold increase in the lower level ACO's time to converge to a better solution than during the regular construction phase, we measured improvements of several percent due to intensification for some of the larger instances.

We also experimented with an intermediate intensification during the regular construction phase: each time the lower level ACO finds a new best solution we started an intensification of that solution to bias the pheromones accordingly. But as it turned out, this leads to worse results. This can be explained by the insight that the pheromones are indeed biased towards a better solution than found during the regular lower level ACO call, but subsequent regular lower level ACO calls cannot reach the quality of an intensified solution since they get too little time. Therefore an intermediate intensification leads to a very early convergence towards a suboptimal solution.

5.5 Parameters for Cascaded ACO

There are a lot of parameters that influence the solving behavior of the cascaded ACO. Actually, there are too many parameters to determine an optimal setting by combining considered parameter values of all parameters.

To find a good parameter setting we wanted to satisfy two criteria: quality of the solutions should be optimized, and algorithm runtime should be minimized. By iteratively checking parameter values and combinations regarding these criteria we determined a good parameter setup.

By testing this setup a sensitivity analysis for the parameters was performed. For this a representative set of test instances was selected – representative regarding size, distribution of customer vertexes, and narrowness of time windows. This set contains the Solomon/Pirkwieser instances p4r101, p4rc104, r6c102, and p6rc105 as well as the Cordeau instances 1a, 3a, 8a, 4b, and 7b (for discussion about test instances see chapter 6.1).

For a specific parameter a set of parameter values was defined that surrounds the parameter value of the previously determined good parameter setup. Then cascaded ACO was called 20 times with each parameter value of the currently selected parameter. This was repeated for all parameters. The result is a distribution of solution qualities and algorithm runtimes for each test instance and parameter value of a parameter. To compare solution qualities independently of the test instance the solution costs were normalized: the result of each experiment was divided by the average result of all experiments with the same test instance. Therefore the solution quality is distributed around 1.

The distribution of the solution qualities for the parameter values are displayed as box plots for each parameter separately, which at least permits us to verify the chosen parameter setup. It also indicates the algorithm's sensitivity to quality and runtime regarding each parameter. The box plot contains minimal and maximal value, lower and upper quartile, and the median of the solution qualities. Additionally the average algorithm runtime is displayed as a line.

In the following diagrams the chosen parameter values are displayed in green. Each tested parameter value represents 180 experiments (9 test instances, 20 experiments each). Parameter values that are marked with a star (*) indicate a setting where one or more experiments produced an infeasible solution. The blue line shows the average relative runtime of the parameter values. The axis for the relative solution quality is shown on the left ($95\% - 105\%$ scaled to the median of the experiments with the chosen parameter setting); the axis for the relative algorithm runtime is on the right ($0.5 - 3.5$ scaled to the average relative algorithm runtime with chosen parameter setting).

Parameters of the upper level ACO

Figure 5.6 shows the results for the parameters contained in the upper level ACO. α_U (5.6 a) specifies the influence of the pheromone values for the upper level construction process. A value of 1 is the obvious choice for this parameter. If α_U is set to 0 the upper level construction process becomes a pure random search due to the missing heuristic component. Higher values increase the sensitivity of the construction process to small differences of the pheromones, but the experiments showed an unstable behavior allowing infeasible solutions to be generated (for $\alpha_U = 2$ five of 180 experiments, for $\alpha_U = 5$ even ten of 180 experiments generated infeasible solutions). A Wilcoxon rank-sum test clearly confirms (P-value < 0.05 one-tailed) the choice of $\alpha_U = 1$ as a superior value.

For the evaporation factor of the upper level ACO ρ_U (5.6 b) we decided to take the value 0.1. Here a value of 1 allowed the upper level construction process becoming a random search - confirming the results of the setting of α_U to 0. The Wilcoxon rank-sum test showed that the experiments with $\rho_U = 0.1$ generated significantly better solutions than with other tested values.

The values for the pheromone increment parameters σ_U (5.6 d) and ω_U (5.6 c) were set to 10 and 3 respectively. A σ_U value that is too high may generate infeasible solutions (for $\sigma_U = 50$ one of 180 experiments resulted in an infeasible solution). Wilcoxon rank-sum confirmed that $\sigma_U = 10$ performs better than other σ_U values, except for $\sigma_U = 5$ which seems to be an equally good choice. On the other hand, the influence of the parameter ω_U is marginal for the performance of cascaded ACO.

No parameter of the upper level ACO significantly influences the overall runtime of the cascaded ACO algorithm.

Parameters of the lower level ACO

Results for the parameters of the lower level ACO are shown in figures 5.7 and 5.8. The colony size of the lower level ACO (5.7 b) has an influence on the solution quality as well as on the runtime of the algorithm. A Wilcox rank-sum test verified that a colony size of $n/3$ or $n/10$ generates significantly better solutions than with colony sizes of n, $n/30$, or even 1 ant. Since
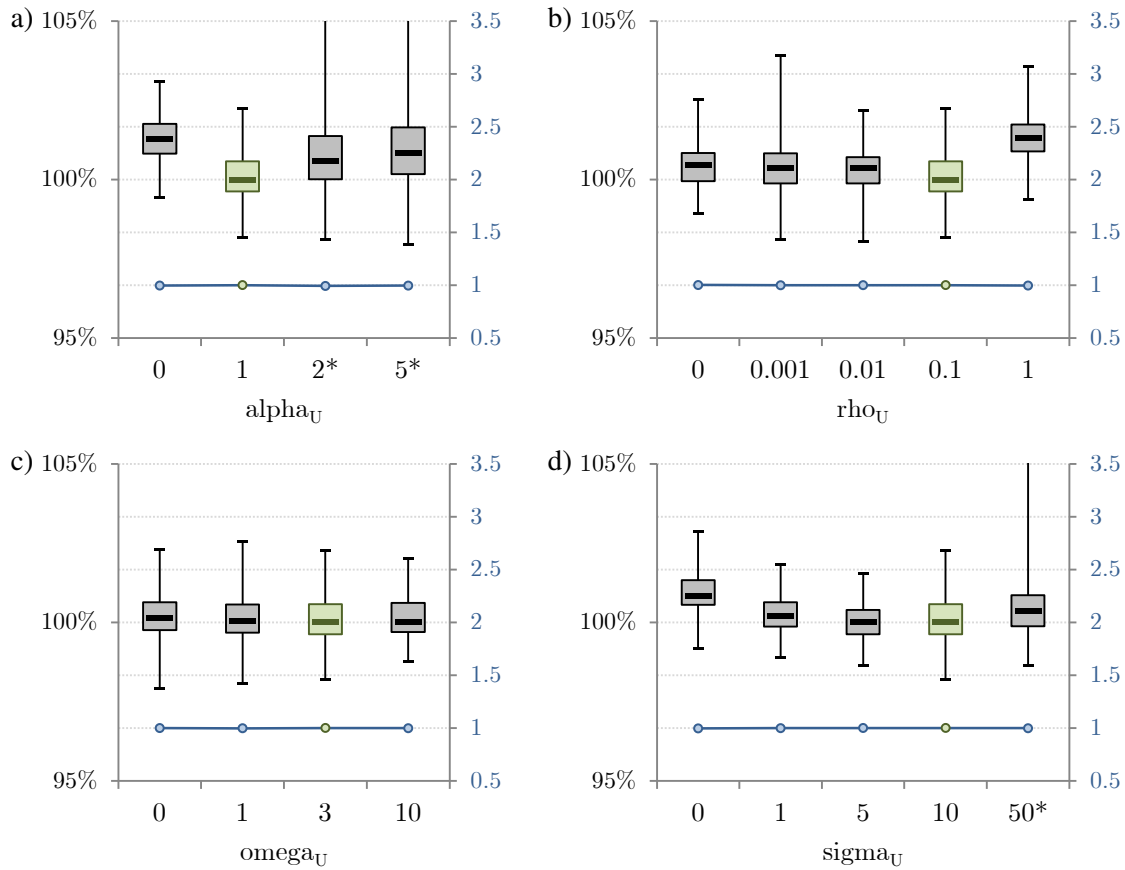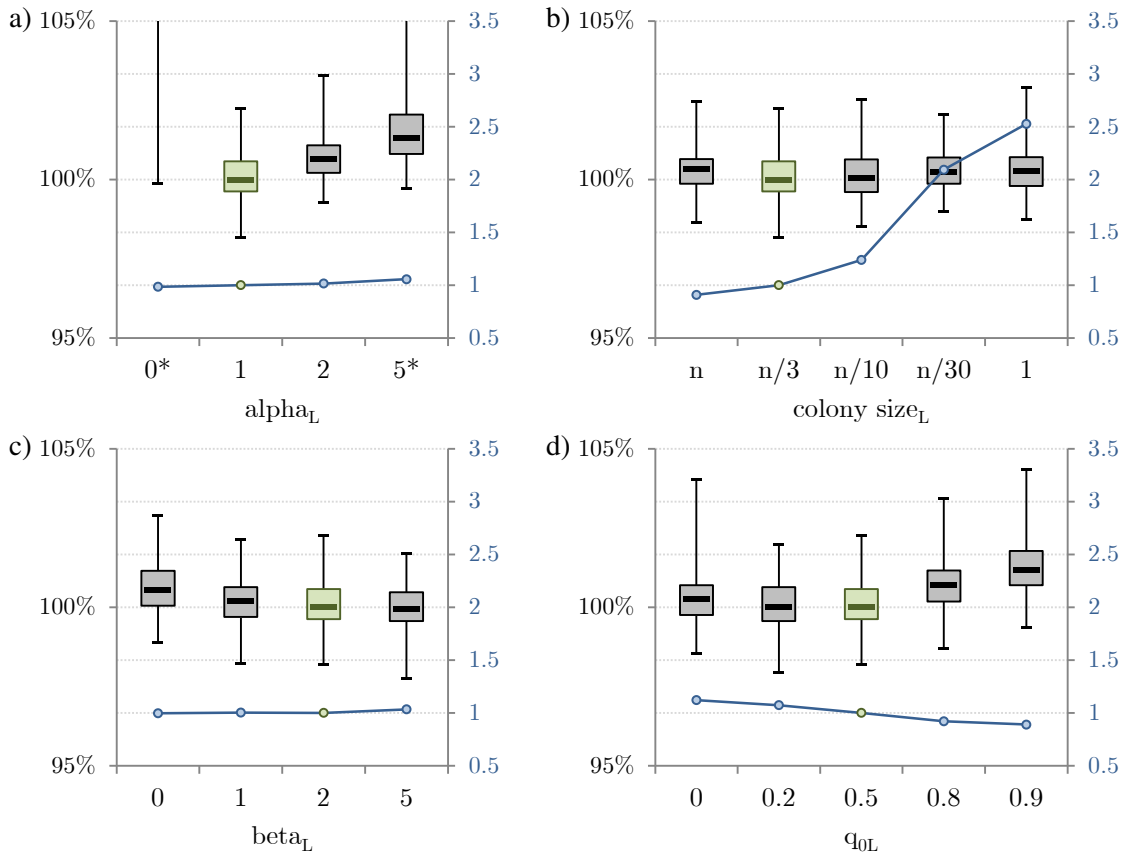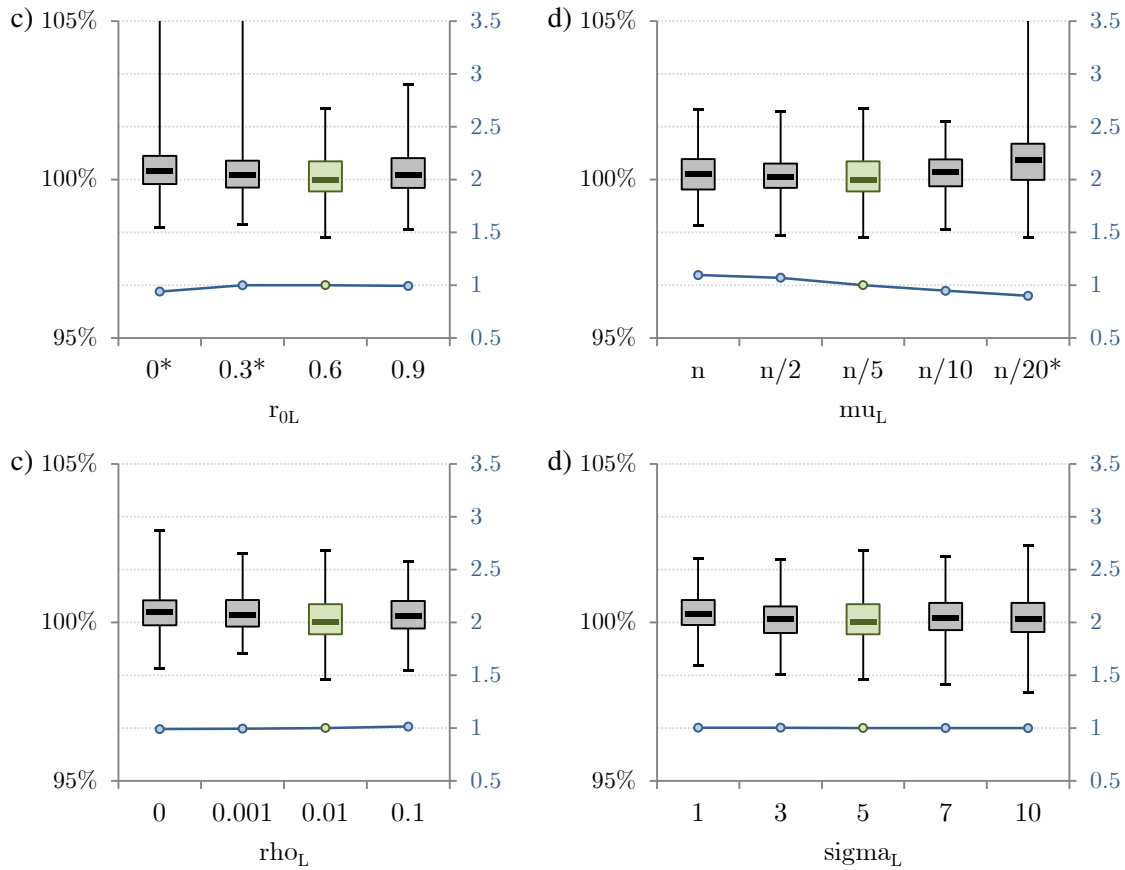
Figure 5.6: Box plots showing relative solution quality as well as runtime of the parameters for the upper level ACO using the selected parameter setup. Sensitivity of parameters a) α_U, b) ρ_U, c) ω_U, and d) σ_U

larger colony sizes show a better runtime behavior the choice of $n/3$ is justified. The increasing runtime for smaller colonies can be explained by the decreasing effect of the attractiveness matrix, which has to be rebuilt more often for small colonies.

The parameter α_L (5.7 a) and β_L (5.7 c) determine the influence of the pheromones and of the savings based heuristic to the construction step of the lower level ACO. As proved by several examples in the literature (e.g. Dorigo et al. [31]) $\alpha_L = 1$ shows the most stable algorithm performance with the best results, particularly since a pure heuristic search with $\alpha_L = 0$ generates approximately 25% infeasible solutions (49 of 180 experiments), and a too high $alpha_L$ value of 5 also tends to generate infeasible solutions (one of 180 experiments). Based on these findings β_L was set to 2, although $\beta_L = 5$ would also show similar solution qualities but with a slightly increased runtime. Both choices showed significantly better solution qualities than $\beta_L = 0$ or $\beta_L = 1$ based on a Wilcoxon rank-sum test.

A rising probability of the pseudo-random proportional state transition rule q_{0L} (5.7 d) obviously decreases the algorithm runtime, because with higher q_{0L} values the computational costly

Figure 5.7: Box plots showing relative solution quality as well as runtime of the parameters for the lower level ACO using the selected parameter setup. Sensitivity of parameters a) α_L, b) *colony size*$_L$, c) β_L, and d) q_{0L}

roulette wheel selection for the construction step is more likely skipped. Interestingly ignoring the pseudo-random proportional state transition rule ($q_{0L} = 0$) shows significantly worse results than enforcing exploitation to 50% of the construction decisions ($q_{0L} = 0.5$). The same is true for a too strong exploratory force ($q_{0L} = 0.8$ or 0.9). Therefore we have chosen to use $q_{0L} = 0.5$.

Similar to q_{0L} the parameter μ_L (5.8 b) balances exploitation and exploration of the lower level ACO. For the same reasons a decreasing neighborhood also decreases the algorithm runtime. But too small neighborhoods may generate infeasible solutions (for $\mu_L = n/20$ one of 180 experiments). An effective compromise between good runtime and stability of the algorithm seems to be the value $\mu_L = 0.5$, although this parameter value does not show significantly better solutions than with other values (except for $\mu_L = n/20$).

The parameter r_{0L} (5.8 a) determines the probability of the local search procedure of the lower level ACO to try to reduce the number of vehicles. The tests showed that by using a too low value for r_{0L} the cascaded ACO indeed becomes instable in terms of feasibility. For

Figure 5.8: Box plots showing relative solution quality as well as runtime of the parameters for the lower level ACO using the selected parameter setup. Sensitivity of parameters a) r_{0L}, b) μ_L, c) ρ_L, and d) σ_L

$r_{0L} = 0$ two of 180 experiments, for $r_{0L} = 0.3$ one of 180 experiments generated an infeasible solution. We decided to use the value 0.6.

The last two parameters determine pheromone evaporation ρ_L (5.8 c) and pheromone incrementation σ_U (5.8 d). For ρ_L we identified the value 0.99 to generate significantly better solutions than with the other tested values. σ_L specifies the number of top ants of an ant colony. The experiments showed that the solutions generated with 5 top ants were significantly better than with only 1 top ant regarding a Wilcoxon rank-sum test. Nevertheless the other tested values seem to perform quite similarly. Therefore we decided to rely on literature (Dörner et al. [33]) and selected $\sigma = 5$.

Summary

The choice of parameter values significantly influences solution quality as well as the runtime of the cascaded ACO. An unbalanced parameter setup can even generate infeasible solutions:

settings that do not allow the algorithm to converge and degrade it to a random search procedure ($\alpha_L = 0$, $r_{0L} = 0$ or 0.3), as well as settings that focus too much on exploitation of memory or heuristics that let the algorithm converge too quickly and do not allow it to escape from local minima appropriately ($\alpha_U = 2$ or 5, $\sigma_U = 50$, $\alpha_L = 5$, $\mu_L = n/20$) produced such infeasible solutions.

The following table summarizes the parameter settings we used for the cascaded ACO:

Parameter	Value
α_U	1
ρ_U	0.9
σ_U	10
ω_U	3
$colony\ size_L$	$n/3$
q_{0L}	0.5
α_L	1
β_L	2
μ_L	$n/5$
r_{0L}	0.6
ρ_L	0.99
σ_L	5

Table 5.3: Parameter settings for the cascaded ACO algorithm

5.6 Implementation

The cascaded ACO to solve the whole PVRPTW was implemented in C++. The following class diagram 5.9 shows an overview of the implementation structure.

```
┌─────────────────────────┐                      ┌──────────────────────────┐
│        Solution         │                      │      UpperLevelACO       │
├─────────────────────────┤    «creates»         ├──────────────────────────┤
│ -costs[1]               │◁─────────────        │                          │
│ -penalty[1]             │                      ├──────────────────────────┤
│ -visitCombinations[*]   │                      │ +solvePVRPTW()           │
├─────────────────────────┤                      │ -createLowerLevelACO()   │
│ +localSearch()          │                      └──────────────────────────┘
└─────────────────────────┘

            ◇ 1

            │

            * │

┌─────────────────────────┐                      ┌──────────────────────────┐
│        MultiPath        │    «creates»         │      LowerLevelACO       │
├─────────────────────────┤                      ├──────────────────────────┤
│ -costs[1]               │◁─────────────        │ -allowedCustomers[*]     │
│ -penalty[1]             │                      ├──────────────────────────┤
├─────────────────────────┤                      │ +solveVRPTW()            │
│ +removeNode()           │                      └──────────────────────────┘
│ +insertNode()           │
│ +interRouteNIM()        │
└─────────────────────────┘
                                                 ┌──────────────────────────┐
            ◇ 1                                  │      ProblemInstance     │
                                                 ├──────────────────────────┤
            │                                    │ -costs[*]                │
                                                 │ -durations[*]            │
            * │                                  │ -noVehicles[1]           │
                                                 │ -maxVehicleLoad[1]       │
┌─────────────────────────┐                      │ -maxRouteDuration[1]     │
│          Path           │                      ├──────────────────────────┤
├─────────────────────────┤                      │ +load()                  │
│ -costs[1]               │                      └──────────────────────────┘
│ -load[1]                │
│ -depotLeavingTime[1]    │                                  ◇ 1
│ -customers[*]           │
├─────────────────────────┤                                  │
│ +removeNode()           │
│ +insertNode()           │                                  * │
│ +localNIM()             │
└─────────────────────────┘                      ┌──────────────────────────┐
                                                 │        Customer          │
                                                 ├──────────────────────────┤
                                                 │ -serviceFrequency[1]     │
                                                 │ -visitCombinations[*]    │
                                                 │ -demand[1]               │
                                                 │ -serviceTime[1]          │
                                                 │ -timeWindow[1]           │
                                                 ├──────────────────────────┤
                                                 │                          │
                                                 └──────────────────────────┘
```

Figure 5.9: Class diagram of the implemented cascaded ACO

Computational Results

6.1 Problem Instances

The presented algorithms were tested with different well known problem instances to allow comparison of the methods used. In the literature there are PVRPTW instances from Cordeau et al. [14], Pirkwieser and Raidl [75] that were developed from VRPTW instances of Solomon [88], and new large scale PVRPTW instances from Vidal et al. [95] that were constructed by following the generation procedure of Cordeau et al. [14].

To test the column generation approach we used a modified set of instances from Cordeau et al. and the set of instances from Pirkwieser and Raidl, both containing problems with a planning period of four and six days. The set of instances from Cordeau et al. was modified to reduce the size of some problems by randomly removing customers and decreasing fleet size. This "reduced" set of Cordeau instances was introduced by Pirkwieser and Raidl [74] – we will refer to this set of instances as *Pirkwieser/Cordeau instances*. The second set of instances was derived by Pirkwieser and Raidl from VRPTW instances introduced by Solomon. For this set visit combinations were assigned evenly to the customers of the problem instances at random – we will call these instances the *Pirkwieser/Solomon instances*.

To test the cascaded ACO we used the original set of instances introduced by Cordeau et al. and we refer to it as the *Cordeau instances*. As a second test set we used the Pirkwieser/Solomon instances.

Tables 6.1 and 6.2 show some of the main characteristics of the original Cordeau instances and the reduced set of Pirkwieser/Cordeau instances. In addition to the name of the instance, the number of customer vertexes (n), fleet size (m), the number of days of the planning horizon (t), and the average frequency over all customers ($\text{avg}(f_i)$) is displayed. Additionally the tables contain the maximum carrying load of the vehicles (Q) and the average demand over all customers ($\text{avg}(q_i)$). Finally, maximal route duration of the vehicles (D), average service duration ($\text{avg}(d_i)$), average distance[1] to the depot ($\text{avg}(z_{i0})$), and average time window size relative to the

---

[1] travel costs and travel duration between two vertexes v_i and v_j is calculated as Euclidean distance between the

Instance	n	m	t	$\mathrm{avg}(f_i)$	Q	$\mathrm{avg}(q_i)$	D	$\mathrm{avg}(d_i)$	$\mathrm{avg}(z_{i0})$	$\mathrm{avg}(window_i)$
1a	48	3	4	2.0	200	13.7	500	11.5	47.0	14.7%
2a	96	6	4	2.0	195	12.7	480	13.4	43.7	13.5%
3a	144	9	4	2.0	190	12.4	460	12.5	55.5	13.1%
4a	192	12	4	2.0	185	12.9	440	13.0	49.5	13.3%
5a	240	15	4	2.0	180	14.0	420	13.0	41.7	13.6%
6a	288	18	4	2.0	175	12.7	400	13.3	48.5	13.6%
7a	72	5	6	3.0	200	13.2	500	14.0	51.3	13.4%
8a	144	10	6	3.0	190	13.9	475	13.2	51.4	13.2%
9a	216	15	6	3.0	180	12.7	450	11.9	53.0	13.6%
10a	288	20	6	3.0	170	13.4	425	12.8	50.5	13.5%
1b	48	3	4	2.0	200	13.7	500	11.5	47.0	28.0%
2b	96	6	4	2.0	195	12.7	480	13.4	43.7	25.9%
3b	144	9	4	2.0	190	12.4	460	12.5	55.5	27.0%
4b	192	12	4	2.0	185	12.9	440	13.0	49.5	27.5%
5b	240	15	4	2.0	180	14.0	420	13.0	41.7	26.9%
6b	288	18	4	2.0	175	12.7	400	13.3	48.5	26.9%
7b	72	4	6	3.0	200	13.2	500	14.0	51.3	27.4%
8b	144	8	6	3.0	190	13.9	475	13.2	51.4	27.0%
9b	216	12	6	3.0	180	12.7	450	11.9	53.0	27.0%
10b	288	16	6	3.0	170	13.4	425	12.8	50.5	26.9%

Table 6.1: Cordeau instances

Instance	n	m	t	$\mathrm{avg}(f_i)$	Q	$\mathrm{avg}(q_i)$	D	$\mathrm{avg}(d_i)$	$\mathrm{avg}(z_{i0})$	$\mathrm{avg}(window_i)$
1a	48	3	4	2.0	200	13.7	500	11.5	47.0	14.7%
2a	96	6	4	2.0	195	12.7	480	13.4	43.7	13.5%
3a	144	9	4	2.0	190	12.4	460	12.5	55.5	13.1%
4a$_{r1}$	160	10	4	2.0	185	12.6	440	12.8	49.1	13.3%
7a	72	5	6	3.0	200	13.2	500	14.0	51.3	13.4%
9a$_{r1}$	96	7	6	3.0	180	13.1	450	11.8	54.5	13.6%
9a$_{r2}$	120	8	6	3.0	180	12.3	450	12.0	51.8	13.6%
8a	144	10	6	3.0	190	13.9	475	13.2	51.4	13.2%
2b$_{r1}$	32	2	4	2.0	195	13.1	480	12.2	47.7	26.2%
1b	48	3	4	2.0	200	13.7	500	11.5	47.0	28.0%
2b$_{r2}$	64	4	4	2.0	195	12.5	480	14.0	41.7	25.8%
3b$_{r1}$	72	4	4	2.0	190	12.5	460	12.1	52.4	27.0%
7b$_{r1}$	24	2	6	3.0	200	12.5	500	11.3	55.2	28.0%
8b$_{r1}$	36	2	6	3.0	190	12.9	475	11.2	50.6	26.8%
7b$_{r2}$	48	3	6	3.0	200	13.5	500	15.4	49.4	27.2%
8b$_{r2}$	60	3	6	3.0	190	14.4	475	13.8	53.2	27.2%

Table 6.2: Pirkwieser/Cordeau instances

time window size of the depot in percent ($\text{avg}(window_i)$) is shown, where $window_i = \frac{l_i - e_i}{l_0 - e_0}$.

The Cordeau as well as the Pirkwieser/Cordeau instances can be grouped into 4-day period problem instances with narrow time windows (instances 1a - 6a), 6-day period problem instances with narrow time windows (instances 7a - 10a), 4-day period problem instances with medium sized time windows (instance 1b - 6b), and 6-day period problem instances with medium sized time windows (instances 7b - 10b). In each group the problem instances consist of different numbers of customers: from 48 customers up to 288 customers. For Pirkwieser/Cordeau instances the subscript in the instance name indicates that this instance was generated by removing customers from the original Cordeau instance.

The time windows for Cordeau instances have been constructed using a method that leads to a normal distribution around the average time window size, and the time window positions (early or late during the day) were assigned randomly.

Table 6.3 shows the Pirkwieser/Solomon instances which all consist of 100 customers. They can also be differentiated into instances with a planning period of four days indicated by the instance name prefix "p4" and those with a planning period of six days with prefix "p6". A further characteristic of the Pirkwieser/Solomon instances is the distribution of customer vertexes. In instances where the name contains an "r" the customers' locations are randomly set, in those with a "c" the customers are clustered, and in those with an "rc" there is a mixture of randomly placed customers and customer clustering. Based on period and customer placement six groups can be distinguished, each containing five problem instances.

For the Pirkwieser/Solomon instances it should be noticed that the average time window sizes vary severely. But even in instances with large average time window sizes there are customers with very narrow time windows. This is because in Solomon instances the time window sizes are not normally distributed around the average. Furthermore, there are always two classes of customers: some with very large time windows (ca. 90% of the depot's time window size) and some with very narrow time windows (ca. 10% or less of the depot's time window size); customers with medium sized time windows do not exist. Therefore, larger average time window sizes of an instance result from the fact that the instance contains more customers with large time windows and vice versa.

The first instance of each instance group (p4r101, p4c101, p4rc101, p6r101, p6c101, p6rc101) consists only of customers with narrow time windows. The subsequent instances of the instance groups include more and more customers with large time windows, except for the last instance per group (...05): here all customers have narrow time windows again, but the time window has about twice the size of the ...01 instances. An exception to this schema are the "rc" instance groups with a mixture of random and clustered customer placement: here the instances p4rc105 and p6rc105 consist additionally to the customers with narrow time windows of customers with medium to larger window sizes (50% of the day time). Generally, narrow time window customers within the "rc" instance group have time windows twice the size of the other instance groups.

The time window positions (early or late during the day) of the Pirkwieser/Solomon instances are randomly distributed over the day.

The visit combinations follow a strict construction schema for all instances. The 4-day

---

position of the two vertexes given by the coordinates x and y: $c_{i,j} = z_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

Instance	n	m	t	avg(f_i)	Q	avg(q_i)	D	avg(d_i)	avg(z_{i0})	avg($window_i$)
p4r101	100	14	4	2.4	200	14.6	∞	10.0	24.9	4.3%
p4r102	100	13	4	2.4	200	14.6	∞	10.0	24.9	25.0%
p4r103	100	10	4	2.4	200	14.6	∞	10.0	24.9	44.8%
p4r104	100	7	4	2.4	200	14.6	∞	10.0	24.9	64.5%
p4r105	100	11	4	2.4	200	14.6	∞	10.0	24.9	13.0%
p4c101	100	10	4	2.4	200	18.1	∞	90.0	28.9	4.9%
p4c102	100	8	4	2.4	200	18.1	∞	90.0	28.9	26.4%
p4c103	100	7	4	2.4	200	18.1	∞	90.0	28.9	47.6%
p4c104	100	7	4	2.4	200	18.1	∞	90.0	28.9	69.0%
p4c105	100	8	4	2.4	200	18.1	∞	90.0	28.9	9.8%
p4rc101	100	10	4	2.4	200	17.2	∞	10.0	33.1	12.5%
p4rc102	100	10	4	2.4	200	17.2	∞	10.0	33.1	29.8%
p4rc103	100	8	4	2.4	200	17.2	∞	10.0	33.1	46.9%
p4rc104	100	7	4	2.4	200	17.2	∞	10.0	33.1	64.4%
p4rc105	100	11	4	2.4	200	17.2	∞	10.0	33.1	22.6%
p6r101	100	14	6	3.0	200	14.6	∞	10.0	24.9	4.3%
p6r102	100	12	6	3.0	200	14.6	∞	10.0	24.9	25.0%
p6r103	100	9	6	3.0	200	14.6	∞	10.0	24.9	44.8%
p6r104	100	8	6	3.0	200	14.6	∞	10.0	24.9	64.5%
p6r105	100	9	6	3.0	200	14.6	∞	10.0	24.9	13.0%
p6c101	100	7	6	3.0	200	18.1	∞	90.0	28.9	4.9%
p6c102	100	7	6	3.0	200	18.1	∞	90.0	28.9	26.4%
p6c103	100	6	6	3.0	200	18.1	∞	90.0	28.9	47.6%
p6c104	100	6	6	3.0	200	18.1	∞	90.0	28.9	69.0%
p6c105	100	7	6	3.0	200	18.1	∞	90.0	28.9	9.8%
p6rc101	100	10	6	3.0	200	17.2	∞	10.0	33.1	12.5%
p6rc102	100	9	6	3.0	200	17.2	∞	10.0	33.1	29.8%
p6rc103	100	7	6	3.0	200	17.2	∞	10.0	33.1	46.9%
p6rc104	100	7	6	3.0	200	17.2	∞	10.0	33.1	64.4%
p6rc105	100	9	6	3.0	200	17.2	∞	10.0	33.1	22.6%

Table 6.3: Pirkwieser/Solomon instances

period instances contain customers with one, two, or four visits per period; the 6-day period instances customers with one, two, three, or six visits. The visit combinations for a customer are constructed such that each single day of the planning period is contained in only one single visit combination. E.g. the visit combinations for a customer of a 4-day period instance with two visits per period are: day 1 and day 3 (first visit combination); day 2 and day 4 (second visit combination).

Table 6.4 displays the schema for the construction of the visit combinations. Each row represents a single visit combination: a filled square (■) indicates that this day is part of it; an

4-day period					6-day period						
	visit day					visit day					
frequency	1	2	3	4	frequency	1	2	3	4	5	6
1	■	□	□	□	1	■	□	□	□	□	□
	□	■	□	□		□	■	□	□	□	□
	□	□	■	□		□	□	■	□	□	□
	□	□	□	■		□	□	□	■	□	□
2	■	□	■	□		□	□	□	□	■	□
	□	■	□	■		□	□	□	□	□	■
4	■	■	■	■	2	■	□	□	■	□	□
						□	■	□	□	■	□
						□	□	■	□	□	■
					3	■	□	■	□	■	□
						□	■	□	■	□	■
					6	■	■	■	■	■	■

Table 6.4: Construction schema for the visit combinations of the used PVRPTW test instances

empty square (□) that this day is not part of it.

The Cordeau and Pirkwieser/Cordeau instances use this schema to assign visit combinations to the customers, where in 4-day period instances half of the customers are set with frequency 1, a quarter are set with frequency 2 and the remaining quarter with frequency 4. In 6-day period instances the four different frequencies 1, 2, 3 and 6 are distributed equally among the customers, each to a quarter of the customers.

For the Pirkwieser/Solomon instances the schema was applied in a way that in all instances – regardless if 4-day or 6-day period – the different frequencies were distributed equally among the customers.

6.2 ACO for Pricing Problem

The column generation approach to solve the LP-relaxed PVRPTW was implemented in C++ compiled with GCC 4.3. The RMP was solved using IBM ILOG CPLEX 12.2. The tests were executed on a 2.83 GHz Intel Core2 Quad Q9550 CPU with 8 GB RAM using a Linux server with 2.6.38 kernel.

Table 6.5 shows the results for the Pirkwieser/Cordeau instances comparing the ACO algorithm as an approximate ESPPRC solver with the other implemented metaheuristics. As a baseline column generation was started without any approximate ESPPRC solver using only the exact dynamic programming based ESPPRC solver with a forced early termination. Each configuration was started 30 times using slack variables instead of providing initial solutions. The table contains the problem instance, the lower bound of the LP-relaxed problem (LB), and the average CPU times of the different variants calculating this lower bound. t_{CG} is the average CPU time of the whole column generation algorithm in seconds; t_{Heur} is the average CPU time percentage of the approximate ESPPRC solver as part of the whole column generation process.

		Exact	GRASP		ILS		VNS		ACO		
Instance	LB	t_{CG}	t_{CG}	t_{Heur}	t_{CG}	t_{Heur}	t_{CG}	t_{Heur}	t_{CG}	t_{Heur}	$sd(t_{CG})$
1a	2882.01	23.6	2.4	84%	2.3	84%	2.4	85%	2.8	85%	(0.3)
2a	4993.48	852.3	18.3	45%	19.6	50%	22.6	55%	22.7	54%	(3.5)
3a	6841.44	3129.0	113.5	15%	115.5	17%	143.5	20%	121.6	20%	(11.9)
$4a_{r1}$	6641.67	7183.6	358.9	7%	341.7	9%	390.5	11%	355.9	10%	(29.7)
7a	6641.39	48.7	9.8	72%	9.9	76%	12.1	77%	10.4	74%	(0.8)
$9a_{r1}$	8035.09	170.8	48.3	39%	47.2	43%	57.0	44%	50.3	45%	(13.0)
$9a_{r2}$	8140.15	1461.0	304.2	13%	306.0	14%	328.8	17%	296.6	16%	(3.1)
8a	9153.79	1522.5	195.3	40%	193.0	43%	226.2	48%	208.6	44%	(47.0)
$2b_{r1}$	2682.52	43.6	16.0	13%	15.2	14%	16.5	12%	15.8	11%	(5.3)
1b	2258.85	73.0	30.2	10%	27.8	13%	26.4	15%	27.5	18%	(4.7)
$2b_{r2}$	2733.55	188.0	51.0	9%	48.7	11%	50.1	13%	53.0	15%	(11.5)
$3b_{r1}$	3241.90	1334.1	252.9	2%	236.1	3%	275.8	3%	248.8	3%	(62.2)
$7b_{r1}$	3677.21	1.4	1.1	92%	1.2	93%	0.9	90%	1.0	91%	(0.1)
$8b_{r1}$	3476.43	27.5	4.2	60%	4.0	64%	4.5	60%	4.0	61%	(1.2)
$7b_{r2}$	3599.72	39.7	11.6	39%	11.4	43%	12.6	39%	12.2	46%	(0.4)
$8b_{r2}$	4324.87	2835.6	595.8	1%	597.9	2%	655.3	2%	633.4	2%	(187.1)

Table 6.5: Column generation approach: results for Pirkwieser/Cordeau instances regarding CPU time

		Exact		GRASP		ILS		VNS		ACO	
Instance	LB	$UB_{B\&B}$	$S_{B\&B}$	$UB_{B\&B}$	$S_{B\&B}$	$UB_{B\&B}$	$S_{B\&B}$	$UB_{B\&B}$	$S_{B\&B}$	$UB_{B\&B}$	$S_{B\&B}$
1a	2882.01	3094.83	100%	3018.88	100%	3014.33	100%	3013.70	100%	3010.82	100%
2a	4993.48	5159.97	100%	5125.07	100%	5112.35	100%	5099.61	100%	5118.69	100%
3a	6841.44	7425.35	100%	7378.81	100%	7380.00	100%	7391.17	100%	7390.07	100%
$4a_{r1}$	6641.67	7221.26	100%	7246.70	100%	7230.85	100%	7222.80	100%	7246.81	100%
7a	6641.39	7218.52	100%	7365.18	47%	7349.47	73%	7313.71	80%	7324.91	67%
$9a_{r1}$	9153.79	10507.5	100%	10483.1	100%	10473.5	100%	10471.2	100%	10466.3	100%
$9a_{r2}$	8035.09	9128.64	100%	9059.07	100%	9035.00	100%	9007.51	100%	9007.82	100%
8a	8140.15	9163.46	100%	9133.61	100%	9143.06	100%	9170.15	100%	9153.41	100%
$2b_{r1}$	2258.85	2287.77	100%	2290.39	100%	2289.13	100%	2293.51	100%	2290.17	100%
1b	2682.52		0%	2775.06	60%	2760.85	83%	2765.95	80%	2781.76	73%
$2b_{r2}$	2733.55	2816.36	100%	2815.35	100%	2806.75	100%	2807.64	100%	2811.32	100%
$3b_{r1}$	3241.90	3416.69	100%	3361.23	100%	3356.06	100%	3358.24	100%	3356.54	100%
$7b_{r1}$	3677.21	3934.84	100%	3909.92	100%	3916.42	87%	3915.10	93%	3943.35	90%
$8b_{r1}$	3599.72	3834.09	100%	3840.60	100%	3854.30	100%	3828.17	100%	3848.71	100%
$7b_{r2}$	3476.43		0%		0%		0%		0%		0%
$8b_{r2}$	4324.87		0%		0%		0%		0%		0%

Table 6.6: Column generation approach: results for Pirkwieser/Cordeau instances regarding quality of columns

For the ACO variant we also report the standard deviation of CPU times in seconds ($\text{sd}(t_{CG})$).

In addition to the runtime of the algorithm the quality of the generated columns was analyzed. For this we used the branch and bound component of the IBM ILOG CPLEX optimizer to solve the RMP including integrality constraints. This "price and branch" approach generates a feasible solution of the PVRPTW whereas the value of the cost function represents the quality of the columns injected into the RMP during the column generation process. Because of the set covering formulation of the problem a simple repair step is applied to the branch and bound solution correcting over-covered visit constraints and cover constraints. If the total costs of the PVRPTW solution are low we deduce that the generated columns have to be relevant and therefore of high quality and vice versa. Table 6.6 shows the results of the column's quality analysis for the Pirkwieser/Cordeau instances. The average total costs of the PVRPTW solution generated with branch and bound is displayed in $\text{UB}_{\text{B\&B}}$. We limited the runtime of the branch and bound procedure to 10 minutes; the percentage of runs that produced a solution within this time is shown in $S_{\text{B\&B}}$.

The results for the Pirkwieser/Solomon instances regarding CPU time are shown in table 6.7. Table 6.8 shows the corresponding results of the analysis of the columns' qualities. Notice that for the instances p4c104 and p6c104 no results have been produced since the runtime of the column generation algorithm exceeded several hours preventing us from generating statistically significant results.

In general the column generation approach requires more CPU time the larger the time window sizes of the problem instance. This is because the performance of the dynamic programming approach of the exact ESPPRC solver scales with the solution space of the problem. Wider time windows allow more feasible solutions which slow down the exact ESPPRC solver; narrower time windows reduce the set of feasible solutions allowing the exact ESPPRC solver to increase performance. The CPU time of approximate ESPPRC solvers based on metaheuristics do not typically depend on the time window sizes. This is confirmed by the percentage of the approximate ESPPRC solver as part of the whole column generation process: for instances with narrow time windows t_{Heur} indicates a stronger influence of the approximate ESPPRC solver than for instances with larger time window sizes.

When comparing the column generation approach without an approximate ESPPRC solver with the variants with an approximate ESPPRC solver the later show significantly reduced CPU times. This effect is most pronounced for larger instances or instances with wider time windows. This confirms the results presented by Pirkwieser and Raidl [74] which performed similar experiments with a slightly different column generation algorithm and by providing initial solutions instead of using slack variables.

By comparing the CPU time of the ACO variant with the other metaheuristics used as approximate ESPPRC solver the GRASP and ILS metaheuristics show in general a better runtime behavior than ACO which in turn shows a better performance than our VNS implementation. The higher CPU times for the ACO variant might be explained by the higher complexity of the algorithm which consists of a computationally expensive construction procedure and it has to manage the pheromone structure. Apparently the computationally simpler metaheuristics GRASP and ILS can provide columns more quickly without forcing the column generation algorithm to increase the number of iterations significantly which would negate the positive CPU

		Exact	GRASP		ILS		VNS		ACO		
Instance	LB	t_{CG}	t_{CG}	t_{Heur}	t_{CG}	t_{Heur}	t_{CG}	t_{Heur}	t_{CG}	t_{Heur}	sd(t_{CG})
p4r101	4151.54	4.7	3.4	88%	3.2	87%	5.8	92%	3.6	89%	(0.3)
p4r102	3729.84	15.2	8.6	62%	8.3	64%	11.4	69%	10.0	67%	(1.0)
p4r103	3154.65	67.1	14.8	51%	15.5	54%	19.5	57%	18.0	53%	(2.4)
p4r104	2528.55	2197.7	426.1	2%	465.1	2%	500.8	3%	473.4	3%	(109.4)
p4r105	3605.43	12.7	6.4	69%	6.4	69%	9.1	75%	6.7	70%	(0.5)
p4c101	2910.89	25.3	6.5	85%	6.5	86%	7.1	33%	7.6	88%	(0.8)
p4c102	2874.62	119.3	22.4	51%	23.2	56%	26.8	58%	26.6	61%	(3.2)
p4c103	2682.76	536.3	33.9	37%	32.4	41%	41.9	40%	40.8	40%	(5.7)
p4c104	2406.80										
p4c105	2882.37	92.1	11.3	83%	11.2	82%	16.0	81%	11.9	81%	(1.2)
p4rc101	3920.75	8.9	5.1	83%	5.0	81%	8.8	86%	5.7	84%	(0.7)
p4rc102	3726.64	21.6	9.6	56%	9.8	58%	12.4	63%	10.3	63%	(0.7)
p4rc103	3410.76	154.6	19.8	37%	19.7	39%	23.3	43%	21.6	43%	(2.6)
p4rc104	2952.09	4553.8	4401.4	0%	3208.2	0%	4859.0	0%	3375.7	0%	(996.6)
p4rc105	3894.41	14.1	7.0	64%	7.3	65%	9.2	70%	7.6	68%	(0.5)
p6r101	5341.25	8.5	6.1	93%	6.2	93%	9.2	93%	6.6	93%	(0.6)
p6r102	5234.66	23.7	14.5	59%	14.9	61%	18.5	65%	15.8	65%	(1.6)
p6r103	3809.98	98.5	41.1	36%	41.8	40%	46.5	42%	43.7	42%	(2.8)
p6r104	3250.04	275.6	112.8	18%	111.6	20%	124.2	22%	117.5	21%	(13.4)
p6r105	4163.21	23.7	13.8	79%	14.0	79%	16.6	80%	13.3	79%	(1.1)
p6c101	3809.55	112.4	20.1	82%	22.4	84%	27.0	86%	20.8	87%	(3.2)
p6c102	3777.94	164.8	23.0	70%	26.9	76%	29.5	76%	26.7	75%	(3.3)
p6c103	3442.89	335.6	60.2	39%	63.0	44%	72.4	45%	66.9	44%	(7.5)
p6c104	3093.83										
p6c105	3991.09	98.7	23.6	83%	23.7	82%	29.5	84%	25.4	85%	(4.3)
p6rc101	5607.60	13.1	8.1	81%	8.0	82%	12.4	86%	8.0	83%	(0.7)
p6rc102	5195.34	31.5	16.3	68%	16.5	70%	18.8	66%	17.2	71%	(1.5)
p6rc103	4112.32	123.3	56.6	27%	58.5	29%	62.2	32%	59.3	30%	(6.7)
p6rc104	3923.74	3080.2	3246.7	1%	2193.7	1%	3317.2	1%	2951.0	1%	(776.9)
p6rc105	5081.55	26.5	12.8	69%	13.0	69%	16.9	74%	12.7	71%	(0.8)

Table 6.7: Column generation approach: results for Pirkwieser/Solomon instances regarding CPU time

Instance	LB	Exact UB_B&B	S_B&B	GRASP UB_B&B	S_B&B	ILS UB_B&B	S_B&B	VNS UB_B&B	S_B&B	ACO UB_B&B	S_B&B
p4r101	4151.54	4236.14	100%	4194.97	100%	4194.32	100%	4193.92	100%	4192.83	100%
p4r102	3729.84	3806.70	100%	3773.64	100%	3779.96	100%	3772.02	100%	3770.79	100%
p4r103	3154.65	3274.37	100%	3230.69	100%	3226.15	100%	3222.20	100%	3237.92	100%
p4r104	2528.55	2673.32	100%	2683.56	97%	2692.57	97%	2700.10	90%	2693.53	87%
p4r105	3605.43	3763.25	100%	3756.82	100%	3760.67	100%	3760.10	100%	3755.99	100%
p4c101	2910.89	2930.70	100%	2922.34	100%	2922.36	100%	2926.72	100%	2923.60	100%
p4c102	2874.62	2919.05	100%	2915.35	100%	2914.94	100%	2911.84	100%	2921.53	100%
p4c103	2682.76		0%	2878.88	60%	2855.89	93%	2840.79	70%	2841.73	63%
p4c104	2406.80										
p4c105	2882.37		0%	2941.12	97%	2942.12	100%	2951.15	100%	2932.61	97%
p4rc101	3920.75	4120.78	100%	4069.02	100%	4082.63	100%	4082.77	100%	4079.08	100%
p4rc102	3726.64	3846.47	100%	3821.05	100%	3816.03	100%	3818.42	100%	3813.47	100%
p4rc103	3410.76	3590.75	100%	3632.61	100%	3624.35	93%	3609.02	100%	3620.07	100%
p4rc104	2952.09	3065.24	100%	3105.35	100%	3108.34	100%	3108.82	100%	3130.40	93%
p4rc105	3894.41	4012.46	100%	4006.88	100%	4007.17	100%	4004.48	100%	4006.74	100%
p6r101	5341.25	5534.54	100%	5489.49	100%	5483.33	100%	5500.05	100%	5493.39	100%
p6r102	5234.66	5424.42	100%	5454.09	100%	5457.57	100%	5438.50	100%	5451.63	100%
p6r103	3809.98	4126.51	100%	4135.07	100%	4135.61	100%	4135.17	100%	4153.54	100%
p6r104	3250.04	3508.25	100%	3578.00	100%	3578.65	100%	3574.43	100%	3588.02	100%
p6r105	4163.21	4566.12	100%	4519.91	100%	4533.33	100%	4521.92	100%	4531.89	100%
p6c101	3809.55		0%		0%		0%		0%		0%
p6c102	3777.94		0%	4083.55	13%	4235.81	27%	4125.09	13%	4169.78	23%
p6c103	3442.89		0%		0%		0%		0%		0%
p6c104	3093.83										
p6c105	3991.09		0%		0%		0%	4314.67	3%		0%
p6rc101	5607.60		0%	6009.78	100%	6019.65	100%	6031.97	100%	6018.97	100%
p6rc102	5195.34	5613.13	100%	5620.42	100%	5615.99	100%	5578.86	100%	5614.20	100%
p6rc103	4112.32		0%	4592.24	93%	4583.52	90%	4566.81	97%	4579.08	100%
p6rc104	3923.74	4338.57	100%	4344.25	100%	4332.28	100%	4319.21	100%	4333.74	100%
p6rc105	5081.55	5506.94	100%	5463.80	100%	5477.66	100%	5459.61	100%	5464.46	100%

Table 6.8: Column generation approach: results for Pirkwieser/Solomon instances regarding quality of columns

time effect. Interestingly, our VNS variant cannot outperform the ACO variant regarding CPU time.

The analysis of the quality of the columns reveals a different picture. Although, the column generation variant without an approximate ESPPRC solver produced by far the most columns, the quality is not superior to the metaheuristically enhanced variants which produce only a fraction of these columns. This indicates that the introduction of an approximate ESPPRC solver does not necessarily decrease the quality of the columns; rather it seems to find the relevant columns for the PVRPTW solution more quickly.

Moreover, it can be observed that the branch and bound process to generate a feasible PVRPTW solution for instances with clustered customer locations has difficulties in producing such solutions in the provided time with the routes generated during column generation. The quality of the columns for these instances seems not to be sufficient for branch and bound. Interestingly, the introduction of an approximate ESPPRC solver improves the quality of the columns as can be observed for the Pirkwieser/Solomon instances p4c103, p4c105, p6c102, p6rc101, and p6rc103.

A Wilcoxon rank-sum test (P-value < 0.05 one-tailed) performed between the different variants showed that each of the metaheuristically enhanced ESPPRC solvers produces columns with significantly better quality then the exact ESPPRC solver. By comparing the approximate ESPPRC solvers no significantly preferable variant could be found, except for the VNS variant: the quality of the columns produced by the VNS metaheuristics is significantly higher for 7 instances compared to the ACO variant, for 10 instances compared to the GRASP variant, and for 7 instances compared to the ILS variant.

A possible explanation for this result could be that the main component of the different approximate ESPPRC solvers is the local search. It seems to be more important than the type of the metaheuristic and it takes a dominant role regarding column quality. The VNS naturally emphasizes local search since it searches in different neighborhoods. This assumption is supported by the results obtained from the analysis of the parameters for the ACO variant: Surprisingly, parameter values of $\alpha = 0$ and $\beta = 0$ would also produce acceptable columns which is only possible when the local search becomes the main factor for column generation.

6.3   ACO for whole Problem

The cascaded ACO to solve the whole PVRPTW was implemented in C++ compiled with GCC 4.3. The tests were executed on a 2.83 GHz Intel Core2 Quad Q9550 CPU with 8 GB RAM using a Linux server with 2.6.38 kernel. Due to scheduling conflicts tests were also executed on a 2.53 GHz Intel Xeon Core2 Quad E5540 CPU with 24 GB RAM also using a Linux server with 2.6.38 kernel. Extensive testing showed that the second system increases CPU time for about 10% ($\pm 3\%$) compared to the first system when executing cascaded ACO. Therefore we divided CPU times measured on the second system by 1.1 and reported it as CPU time for the first system.

For all experiments we applied the feasibility rule proposed by Savelsberg [85] for the VRPTW. This rule allows a delay in the start of the vehicle from the depot to the latest possible time by introducing a forward time slack making more routes feasible regarding the duration

constraint. For the comparison part of this section we only present results from previous work which also uses the same feasibility rule[2].

Instance	n	m	t	avg$(c)_{30}$	min$(c)_{30}$	BKS	gap
1a	48	3	4	2926.24	2909.02	2909.02	0.00%
2a	96	6	4	5076.71	5035.59	5026.57	0.18%
3a	144	9	4	7270.52	7179.31	7023.90	2.21%
4a	192	12	4	8127.14	8037.12	7755.77	3.63%
5a	240	15	4	8840.18	8740.60	8311.17	5.17%
6a	288	18	4	11271.12	11180.09	10473.24	6.75%
7a	72	5	6	6825.90	6797.93	6782.68	0.22%
8a	144	10	6	9916.52	9762.91	9574.80	1.96%
9a	216	15	6	13871.96	13676.71	13201.06	3.60%
10a	288	20	6	18322.97	17890.98	16920.96	5.73%
1b	48	3	4	2289.66	2277.44	2277.44	0.00%
2b	96	6	4	4203.95	4155.67	4121.50	0.83%
3b	144	9	4	5727.39	5655.25	5489.33	3.02%
4b	192	12	4	6680.08	6571.51	6347.77	3.52%
5b	240	15	4	7314.60	7178.08	6777.54	5.91%
6b	288	18	4	9273.37	9181.62	8582.72	6.98%
7b	72	4	6	5572.30	5511.49	5481.61	0.55%
8b	144	8	6	7912.07	7785.00	7599.01	2.45%
9b	216	12	6	11216.51	11077.92	10532.51	5.18%
10b	288	16	6	14510.05	14356.55	13406.89	7.08%

Table 6.9: Cascaded ACO for whole problem: results for Cordeau instances

Table 6.9 shows the results of the cascaded ACO for the Cordeau instances. In addition to instance name, number of customers (n), fleet size (m), and number of days of the planning period (t), the table contains the average solutions costs (avg$(c)_{30}$), and the costs of the best solution found (min$(c)_{30}$) for the 30 runs performed on each problem instance. Additionally the best known solution (BKS) based on previous work reviewed in chapter 3 is provided. The last column contains the gap between the best solution found by cascaded ACO for the 30 runs and the best known solution in percent.

A detailed comparison between cascaded ACO and the results from previous work for the Cordeau instances is shown in table 6.10. Here the results of the improved tabu search algorithm (improved TS) by Cordeau et al. [15], the best VNS variant ((R)VNS) by Pirkwieser and Raidl [73], the improved ACO algorithm (IACO) by Yu and Yang [97], the best variant of the parallel hybrid iterated tabu search (ITS) by Cordeau and Maischberger [16], and the hybrid genetic search with adaptive diversity control (HGSADC) by Vidal et al. [95] are presented and

---

[2]Notice that this rule is relevant only for instances with a duration constraint. Therefore, there is no difference if forward time slack is applied or not for the Pirkwieser/Solomon instances since these instances contain no duration constraint ($D = \infty$)

Instance	improved TS min$(c)_{10}$	avg(t)	(R)VNS min$(c)_{30}$	IACO avg$(c)_{10}$	min$(c)_{10}$	avg(t)	ITS min(c)	HGSADC avg$(c)_{10}$	min$(c)_{10}$	avg(t)	cascaded ACO avg$(c)_{30}$	min$(c)_{30}$	avg(t)
1a	2911.03	30	2909.02	3107.04	2959.09	2.9	2909.02	2909.05	2909.02	1.1	2926.24	2909.02	1.7
2a	5055.05	70	5036.27	5658.66	5323.29	5.7	5026.57	5031.50	5026.57	3.3	5076.71	5035.59	7.8
3a	7229.73	109	7138.70	8158.86	7554.50	10.2	7062.00	7091.51	7050.72	8.1	7270.52	7179.31	19.7
4a	7953.08	155	7882.06	9117.42	8364.61	29.0	7807.32	7818.75	7791.93	17.9	8127.14	8037.12	41.2
5a	8593.00	189	8492.45	9591.97	8964.46	30.2	8358.96	8368.98	8341.93	31.0	8840.18	8740.60	74.2
6a	10927.45	245	10713.75	12346.09	11122.60	47.9	10542.10	10595.85	10477.01	65.4	11271.12	11180.09	113.0
7a	6825.07	53	6787.72	8023.27	7100.24	8.8	6782.68	6788.67	6783.23	3.8	6825.90	6797.93	5.3
8a	9748.36	114	9721.25	11305.93	10094.58	14.2	9603.92	9623.72	9593.43	17.0	9916.52	9762.91	30.1
9a	13614.47	206	13463.96	15936.16	14356.90	44.6	13299.80	13285.89	13247.38	45.9	13871.96	13676.71	84.2
10a	17735.59	290	17650.89	19151.86	17733.20	63.4	17261.30	17058.89	16999.88	96.0	18322.97	17890.98	162.6
1b	2294.03	37	2277.44				2277.44	2277.44	2277.44	0.8	2289.66	2277.44	2.2
2b	4257.40	78	4137.45				4124.76	4130.64	4122.03	4.9	4203.95	4155.67	11.1
3b	5648.76	120	5575.27				5489.84	5555.77	5521.71	8.4	5727.39	5655.25	29.3
4b	6594.54	190	6476.67				6383.28	6400.55	6352.28	27.8	6680.08	6571.51	62.0
5b	7054.95	222	6970.33				6800.45	6838.54	6790.44	47.5	7314.60	7178.08	104.4
6b	8928.37	293	8819.32				8659.44	8647.15	8595.10	77.5	9273.37	9181.62	168.8
7b	5505.23	73	5504.67				5481.61	5491.08	5481.61	3.6	5572.30	5511.49	8.0
8b	7875.31	148	7729.32				7656.13	7665.10	7619.95	16.8	7912.07	7785.00	44.5
9b	10889.77	253	10885.93				10579.50	10653.60	10569.68	68.1	11216.51	11077.92	130.7
10b	13980.55	318	13943.61				13490.80	13502.65	13442.57	110.0	14510.05	14356.55	245.6

Table 6.10: Cascaded ACO for whole problem: comparison of results for Cordeau instances

compared to the results from cascaded ACO. The column avg$(c)_x$ presents average solution costs performed on x runs, min$(c)_x$ contains the costs of the best solution found based on x runs performed, and avg(t) contains the average runtime of the algorithm in minutes. Be aware that the runtime is based on different test systems and therefore only gives an indication of the runtime behavior of the algorithms.

The results of the cascaded ACO for the Pirkwieser/Solomon instances are shown in table 6.11. Because previous authors interpreted the instances differently we provide results for both variants: in the first variant travel costs and travel duration of an arc base on the coordinates x and y and are not modified; in the second variant they are truncated to the first digit, that is, $c_{i,j} = z_{i,j} = \frac{\lfloor 10\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \rfloor}{10}$.

Table 6.12 allows a comparison of the cascaded ACO with the only published results of the variant without truncation of travel costs and duration: the hybrid genetic algorithm (HGA) by Nguyen et al. [67]. A detailed comparison for the variant with truncation is shown in table 6.13. The results of cascaded ACO are compared with the best variant of the VNS improved with integer linear programming techniques (best VNS/ILP) by Pirkwieser and Raidl [75], the best variant of the multiple VNS approach optionally improved with integer linear programming techniques (best mVNS/ILP) by Pirkwieser and Raidl [76], the evolutionary algorithm initialized by the routes generated with column generation (CG-EA) by Pirkwieser and Raidl [77], the hybrid genetic algorithm (HGA) by Nguyen et al. [67], and the hybrid genetic search with adaptive diversity control (HGSADC) by Vidal et al. [95]. Notice that the average runtimes of the last algorithm marked with an asterisk (*) represent average runtimes over the whole instance group of five instances since no detailed runtimes are provided.

Cascaded ACO generated feasible solutions in all runs for all tested instances. This indicates that the parameters of the algorithm have been set adequately to enable stable execution.

Instance	n	m	t	without truncation to the first digit				with truncation to the first digit			
				$\text{avg}(c)_{30}$	$\text{min}(c)_{30}$	BKS	gap	$\text{avg}(c)_{30}$	$\text{min}(c)_{30}$	BKS	gap
p4r101	100	14	4	4254.16	4214.25	4142.35	1.74%	4160.29	4114.0	4082.0	0.78%
p4r102	100	13	4	3809.08	3772.58	3739.34	0.89%	3792.74	3763.2	3724.3	1.04%
p4r103	100	10	4	3212.85	3173.71	3165.62	0.26%	3199.04	3159.6	3153.1	0.21%
p4r104	100	7	4	2639.72	2611.02	2582.67	1.10%	2625.43	2602.1	2566.0	1.41%
p4r105	100	11	4	3740.79	3712.50	3664.14	1.32%	3722.31	3691.6	3638.9	1.45%
p4c101	100	10	4	2919.78	2916.59	2913.81	0.10%	2914.41	2910.2	2907.4	0.10%
p4c102	100	8	4	2931.30	2892.03	2888.31	0.13%	2921.01	2882.9	2882.9	0.00%
p4c103	100	7	4	2779.61	2748.53	2742.17	0.23%	2772.77	2741.6	2734.5	0.26%
p4c104	100	7	4	2483.15	2451.74	2446.85	0.20%	2483.75	2453.4	2419.0	1.42%
p4c105	100	8	4	2917.36	2893.99	2893.99	0.00%	2921.17	2884.5	2884.1	0.01%
p4rc101	100	10	4	4058.08	4005.47	3975.39	0.76%	4037.64	3998.3	3955.9	1.07%
p4rc102	100	10	4	3839.92	3803.01	3765.03	1.01%	3844.21	3816.3	3755.7	1.61%
p4rc103	100	8	4	3567.99	3517.67	3472.07	1.31%	3559.31	3523.8	3449.9	2.14%
p4rc104	100	7	4	3081.21	3036.18	3004.59	1.05%	3078.90	3034.9	2991.5	1.45%
p4rc105	100	11	4	4038.71	3997.99	3953.91	1.11%	4029.74	3988.8	3932.6	1.43%
p6r101	100	14	6	5451.70	5420.99	5394.13	0.50%	5435.61	5400.2	5376.1	0.45%
p6r102	100	12	6	5356.49	5336.09	5295.50	0.77%	5265.33	5239.3	5201.6	0.72%
p6r103	100	9	6	4066.84	4016.98	3961.67	1.40%	4046.36	3993.9	3940.5	1.36%
p6r104	100	8	6	3417.14	3378.12	3361.71	0.49%	3400.10	3369.0	3335.8	1.00%
p6r105	100	9	6	4422.53	4350.61	4308.19	0.98%	4415.55	4329.6	4272.9	1.33%
p6c101	100	7	6	4081.72	4002.62	3992.66	0.25%	4068.31	4000.8	3981.2	0.49%
p6c102	100	7	6	3888.62	3855.63	3850.02	0.15%	3884.89	3847.2	3841.7	0.14%
p6c103	100	6	6	3579.20	3547.92	3535.06	0.36%	3576.29	3535.2	3523.6	0.33%
p6c104	100	6	6	3298.96	3260.18	3244.48	0.48%	3290.89	3255.0	3206.3	1.52%
p6c105	100	7	6	4144.34	4087.71	4059.07	0.71%	4133.33	4066.6	4052.1	0.36%
p6rc101	100	10	6	5887.99	5821.50	5799.67	0.38%	5865.51	5817.0	5781.5	0.61%
p6rc102	100	9	6	5504.74	5426.32	5387.76	0.72%	5488.80	5433.4	5333.3	1.88%
p6rc103	100	7	6	4418.70	4355.65	4316.78	0.90%	4407.55	4341.7	4273.1	1.61%
p6rc104	100	7	6	4216.36	4158.93	4109.99	1.19%	4195.85	4126.9	4062.0	1.60%
p6rc105	100	9	6	5382.81	5341.07	5280.32	1.15%	5376.57	5311.8	5227.1	1.62%

Table 6.11: Cascaded ACO for whole problem: results for Pirkwieser/Solomon instances without and with truncation of $c_{i,j}$ and $z_{i,j}$ to the first digit

	HGA			cascaded ACO		
Instance	$\text{avg}(c)_{30}$	$\text{min}(c)_{30}$	$\text{avg}(t)$	$\text{avg}(c)_{30}$	$\text{min}(c)_{30}$	$\text{avg}(t)$
p4r101	4163.43	4142.35	51.5	4254.16	4214.25	4.6
p4r102	3744.30	3739.34	73.6	3809.08	3772.58	6.3
p4r103	3168.57	3165.62	71.8	3212.85	3173.71	7.7
p4r104	2592.07	2582.67	76.0	2639.72	2611.02	10.0
p4r105	3678.06	3664.14	65.7	3740.79	3712.50	5.6
p4c101	2913.83	2913.81	68.5	2919.78	2916.59	6.4
p4c102	2893.86	2888.31	71.3	2931.30	2892.03	8.9
p4c103	2763.43	2742.17	88.6	2779.61	2748.53	10.6
p4c104	2468.79	2446.85	93.9	2483.15	2451.74	11.3
p4c105	2907.47	2893.99	69.3	2917.36	2893.99	7.4
p4rc101	3977.81	3975.39	60.9	4058.08	4005.47	5.5
p4rc102	3777.56	3765.03	66.1	3839.92	3803.01	7.1
p4rc103	3479.30	3472.07	64.9	3567.99	3517.67	8.6
p4rc104	3019.73	3004.59	70.2	3081.21	3036.18	10.0
p4rc105	3959.46	3953.91	65.9	4038.71	3997.99	6.1
p6r101	5398.65	5394.13	77.5	5451.70	5420.99	5.9
p6r102	5302.56	5295.50	75.0	5356.49	5336.09	8.2
p6r103	3980.51	3961.67	89.5	4066.84	4016.98	10.2
p6r104	3375.91	3361.71	95.1	3417.14	3378.12	11.3
p6r105	4321.17	4308.19	77.5	4422.53	4350.61	8.1
p6c101	4015.34	3992.66	75.9	4081.72	4002.62	9.7
p6c102	3858.76	3850.02	88.6	3888.62	3855.63	11.9
p6c103	3575.18	3535.06	104.5	3579.20	3547.92	14.3
p6c104	3259.09	3244.48	105.6	3298.96	3260.18	15.1
p6c105	4076.46	4059.07	85.6	4144.34	4087.71	10.1
p6rc101	5812.68	5799.67	76.0	5887.99	5821.50	7.2
p6rc102	5402.64	5387.76	83.9	5504.74	5426.32	8.9
p6rc103	4339.45	4316.78	84.0	4418.70	4355.65	11.6
p6rc104	4152.33	4109.99	97.1	4216.36	4158.93	12.4
p6rc105	5290.84	5280.32	80.2	5382.81	5341.07	8.5

Table 6.12: Cascaded ACO for whole problem: comparison of results for Pirkwieser/Solomon instances without truncation of $c_{i,j}$ and $z_{i,j}$

Instance	best VNS/ILP		best mVNS/ILP		CG-EA		HGA			HGSADC		cascaded ACO		
	$\text{avg}(c)_{30}$	$\text{avg}(t)$	$\text{avg}(c)_{30}$	$\text{avg}(t)$	$\text{avg}(c)_{30}$	$\text{avg}(t)$	$\text{avg}(c)$	$\text{min}(c)$	$\text{avg}(t)$	$\text{min}(c)$	$\text{avg}(t)$	$\text{avg}(c)_{30}$	$\text{min}(c)_{30}$	$\text{avg}(t)$
p4r101	4095.28	1.0	4090.09	0.5	4162.54	0.5	4085.94	4082.6	51.5	4082.0		4160.29	4114.0	4.7
p4r102	3748.38	0.9	3732.34	0.5	3780.50	0.5	3730.56	3725.2	73.6	3724.3		3792.74	3763.2	6.3
p4r103	3181.41	1.1	3165.72	0.6	3217.31	0.6	3160.81	3153.1	71.8	3153.1	3.0*	3199.04	3159.6	7.7
p4r104	2599.15	1.2	2595.44	0.7	2673.09	0.7	2581.53	2570.8	76.0	2566.0		2625.43	2602.1	10.1
p4r105	3675.09	1.0	3679.66	0.8	3745.00	0.5	3650.45	3638.9	65.7	3638.9		3722.31	3691.6	5.6
p4c101	2910.17	0.6	2909.39	0.4	2921.08	0.6	2907.49	2907.4	68.5	2907.4		2914.41	2910.2	6.3
p4c102	2940.16	0.9	2905.16	0.6	2963.28	0.7	2890.98	2883.3	71.3	2882.9		2921.01	2882.9	8.8
p4c103	2804.11	0.7	2759.78	0.9	2825.01	0.7	2746.23	2735.8	88.6	2734.5	2.7*	2772.77	2741.6	10.5
p4c104	2468.82	1.2	2454.69	0.8	2518.90	0.8	2450.91	2424.3	93.9	2419.0		2483.75	2453.4	11.2
p4c105	2957.54	0.7	2906.69	0.6	2977.45	0.6	2895.33	2884.1	69.3	2884.1		2921.17	2884.5	7.3
p4rc101	3981.48	1.1	3974.09	0.8	4047.87	0.5	3963.02	3955.9	60.9	3956.4		4037.64	3998.3	5.5
p4rc102	3796.19	0.9	3764.99	0.7	3869.21	0.5	3761.92	3755.8	66.1	3755.7		3844.21	3816.3	7.0
p4rc103	3485.47	1.2	3466.99	0.6	3549.13	0.6	3454.60	3450.1	64.9	3449.9	3.9*	3559.31	3523.8	8.5
p4rc104	3045.37	0.7	3031.49	1.0	3114.51	0.7	3008.34	2996.5	70.2	2991.5		3078.90	3034.9	10.1
p4rc105	3985.82	1.0	3970.49	0.8	4040.32	0.5	3954.16	3942.6	65.9	3932.6		4029.74	3988.8	6.1
p6r101	5389.07	1.3	5385.03	0.8	5453.07	0.7	5379.73	5377.5	77.5	5376.1		5435.61	5400.2	5.8
p6r102	5237.75	1.1	5244.59	0.5	5318.87	0.7	5215.61	5206.4	75.0	5201.6		5265.33	5239.3	8.1
p6r103	4001.86	0.9	3991.46	0.5	4120.37	0.8	3968.69	3946.9	89.5	3940.5	4.5*	4046.36	3993.9	10.0
p6r104	3372.30	1.1	3372.81	1.0	3441.55	0.9	3362.09	3352.9	95.1	3335.8		3400.10	3369.0	11.2
p6r105	4334.60	0.8	4337.54	0.5	4457.93	0.7	4302.94	4291.0	77.5	4272.9		4415.55	4329.6	7.9
p6c101	4070.44	0.7	4050.81	1.0	4162.92	0.8	3995.69	3984.3	75.9	3981.2		4068.31	4000.8	9.5
p6c102	3877.56	1.4	3861.86	1.0	3950.54	0.9	3853.83	3841.7	88.6	3841.7		3884.89	3847.2	11.7
p6c103	3594.89	0.9	3576.50	1.1	3719.95	0.9	3555.71	3529.6	104.5	3523.6	4.0*	3576.29	3535.2	14.1
p6c104	3280.58	0.8	3284.07	0.6	3422.22	0.9	3248.35	3236.5	105.6	3206.3		3290.89	3255.0	14.8
p6c105	4158.06	0.7	4104.31	1.0	4181.50	0.9	4060.14	4052.1	85.6	4052.1		4133.33	4066.6	9.9
p6rc101	5818.06	1.3	5821.63	0.9	5909.63	0.7	5801.08	5791.9	76.0	5781.5		5865.51	5817.0	7.0
p6rc102	5467.22	1.4	5446.00	1.0	5553.47	0.7	5373.82	5352.6	83.9	5333.3		5488.80	5433.4	8.8
p6rc103	4344.02	0.8	4351.50	0.6	4476.44	0.8	4298.33	4288.1	84.0	4273.1	4.9*	4407.55	4341.7	11.5
p6rc104	4122.25	1.0	4130.70	0.6	4267.67	0.8	4100.14	4092.5	97.1	4062.0		4195.85	4126.9	12.3
p6rc105	5319.48	0.7	5321.82	0.5	5450.10	0.7	5263.35	5253.0	80.2	5227.1		5376.57	5311.8	8.5

Table 6.13: Cascaded ACO for whole problem: comparison of results for Pirkwieser/Solomon instances with truncation of $c_{i,j}$ and $z_{i,j}$ to the first digit

It can be observed that the solutions of the cascaded ACO for small instances can compete with state-of-the-art algorithms developed recently. For problem instances with less than 100 customers the gap between the best known solutions and the best solutions generated by cascaded ACO is between 0% and 1%. With increasing problem instance size the quality of the solutions decrease compared with algorithms from previous work. For the largest tested instances with 288 customers the gap to the best known solutions reaches about 7%.

We explain the decreasing solution quality for large instances with reference to the characteristics of the upper level ACO. In our current implementation the upper level ACO lacks a heuristic component that can guide the algorithm to better regions of the search space. With an increasing number of customers the size of this search space increases exponentially. Without the heuristic component and by using the current parameter setting the upper level ACO converges to an arbitrary combination of visit combinations too quickly and with too little guidance. So for large instances the algorithm tends to pick a combination of visit combinations randomly and then to explore just the close neighborhood of this combination.

A second factor is the intensification phase of the cascaded ACO. We observed that for small instances intensification does not improve the solution quality at all. For medium sized ($n \approx 100$) instances intensification improves the solution by up to 1%; and for large instances the intensification phase can decrease solution costs by several percent. This indicates that the lower level ACO of the regular construction phase is able to generate semi-optimal VRPTW solutions for small problem instances. With an increasing number of customers the lower level ACO is stopped too early to reach a semi-optimal state – the quality of the VRPTW solutions become merely indicators for the PVRPTW evaluation. Therefore we suspect that the lower level ACO is not well suited for the intensification phase. Either the parameters of the lower level ACO should be revised for the intensification phase or a different algorithm should optimize the VRPTW solutions for intensification.

In general cascaded ACO enriches the portfolio of algorithms solving the PVRPTW. It replaces the previously published ACO based algorithm by Yu and Yang as best ant colony optimization implementation for the PVRPTW, although competing with but not reaching the results of the most recently published algorithms by Pirkwieser and Raidl, Cordeau and Maischberger, Nguyen et al., and Vidal et al.

Conclusion

The PVRPTW is a highly constrained variant of routing problems which belongs to the complexity class of $\mathcal{NP}$-hard problems. On the other hand, ACO is a metaheuristic solution strategy for combinatorial optimization that uses nature inspired techniques to solve such problems approximately. We applied ACO to the PVRPTW in two different ways: first, to solve the pricing subproblem of a column generation approach that provides lower bounds for the LP-relaxed PVRPTW which could be embedded in an exact solution algorithm; second, as a method to solve the whole problem approximately by decomposing it into a optimization problem for visit combinations (upper level) and an ordinary VRPTW (lower level) leading to our cascaded ACO approach.

ACO bases on a fairly large set of parameters to setup and tune the solving power of the metaheuristic. For both applications of ACO we used a method to find good parameter settings which bases on statistical comparison of the performance for selected test instances.

For the column generation approach ACO was used to solve the pricing subproblem that was formulated as ESPPRC. We compared ACO with three other metaheuristics that should generate relevant columns in good quality during the approximate state of the algorithm. Although, ACO showed no superior performance regarding CPU runtime and quality of columns compared to the other metaheuristics, the comparison with the variant, which does not apply any metaheuristic during the solving procedure, showed clearly an advantage for the metaheuristically enhanced algorithm.

The analysis of the quality of the columns showed that there is no metaheuristic that could be preferred for the solution of the ESPPRC. Therefore, it appeared that the type of metaheuristic is not the main characteristic to determine the solution quality. In fact, we deduced that the local search is the component that influences most the quality of the columns for the pricing subproblem solution.

Therefore we can conclude that ACO is suited to generate high quality columns for the pricing problem of the LP-relaxed PVRPTW as long as it contains a strong local search component,

but there is no reason to prefer it over any other approximate solving procedure with the same precondition.

For the application of ACO to solve the whole problem we introduced a new algorithm: the cascaded ACO. The algorithm decomposes the problem into two separate parts that are both solved by ACO variants. The upper level ACO optimizes the visit combinations and the lower level ACO solves the resulting VRPTW, whereas the two ACOs are applied in a cascaded manner.

We combined several techniques published in the literature to create an efficient solving algorithm. These techniques introduced several parameters that had to be setup appropriately.

In an extensive comparison with previous publications which presented results for the solution of the PVRPTW we showed that the cascaded ACO can compete with these algorithms. Cascaded ACO outperforms the sole other ACO algorithm solving the PVRPTW published by Yu and Yang. Although, recently developed algorithms based on hybrids of metaheuristics and/or exact solution techniques show for large problem instance better solution power than cascaded ACO.

Additionally, this thesis pointed out some open issues that merit more detailed investigation and that can potentially increase performance of the cascaded ACO for large problem instances. These issues include the exchange of the lower level ACO with another fast converging VRPTW solving algorithm and the optimization of the lower level algorithm or its parameters for the intensification phase. Also a field of interest is the development of a heuristic construction component for the upper level ACO to guide the selection of visit combinations. And finally the cascaded ACO could be modified to allow infeasible solutions regarding capacity, duration, and/or time window constraints as also the recently published successful algorithms propose.

We conclude that the cascaded ACO is an efficient application of ACO which enriches the portfolio of algorithms solving the PVRPTW bearing the potential of even better performance especially for large problem instances.

Glossary

ACO
> ant colony optimization - metaheuristic.

cascaded ACO
> cascaded ant colony optimization - algorithm.

ESPPRC
> elementary shortest path problem with resource constraints.

GA
> genetic algorithms - metaheuristic.

GLS
> guided local search - metaheuristic.

GRASP
> greedy randomized adaptive search procedure - metaheuristic.

ILP
> integer linear programming.

ILS
> iterated local search - metaheuristic.

IP
> integer linear programming (see also ILP).

LP
> linear programming.

LU decomposition
> a matrix decomposition into a lower and upper triangular matrix.

MILP
> mixed integer linear programming.

MIP
mixed integer linear programming (see also MILP).

NIM
node insertion move.

PVRP
periodic vehicle routing problem.

PVRPTW
periodic vehicle routing problem with time windows.

RCL
restricted candidate list.

RMP
restricted master problem.

SA
simulated annealing - metaheuristic.

SPPRC
shortest path problem with resource constraints.

TS
tabu search - metaheuristic.

TSP
travelling salesman problem.

VNS
variable neighborhood search - metaheuristic.

VRP
vehicle routing problem.

VRPTW
vehicle routing problem with time windows.

Bibliography

[1] Egon Balas, Sebastián Ceria, Gérard Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996.

[2] Cynthia Barnhart, Christopher A. Hane, and Pamela H. Vance. Using Branch-and-Price-and-Cut to Solve Origin-Destination Integer Multicommodity Flow Problems. *Operations Research*, 48(2):318–326, 2000.

[3] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W.P. Savelsbergh, and Pamela H. Vance. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research*, 46(3):316–329, March 1998.

[4] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209–219, 2006.

[5] Edward J. Beltrami and Lawrence D. Bodin. Networks and vehicle routing for municipal waste collection. *Networks*, 4(1):65–94, 1974.

[6] Mauro Birattari, Luis Paquete, Thomas Stützle, and Klaus Varrentrapp. Classification of Metaheuristics and Design of Experiments for the Analysis of Components. Technical Report AIDA-01-05, Darmstadt University of Technology, Darmstadt, Germany, November 2001.

[7] Robert G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2):103–107, May 1977.

[8] Christian Blum and Andrea Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35(3):268–308, September 2003.

[9] Julien Bramel and David Simchi-Levi. On the effectiveness of set covering formulations for the vehicle routing problem with time windows. *Operations Research*, 45(2):295–301, 1997.

[10] G. Clarke and J.W. Wright. Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, 12(4):568–581, 1964.

[11] Alan Cobham. The intrinsic computational difficulty of functions. In Yehoshua Bar-Hillel, editor, *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*, pages 24–30, 1964.

[12] Jean-François Cordeau, Guy Desaulniers, Jacques Desrosiers, Marius M. Solomon, and François Soumis. VRP with Time Windows. In Paolo Toth and Daniele Vigo, editors, *The Vehicle Routing Problem, Volume 9 of Monographs on Discrete Mathematics and Applications*, pages 157–193. SIAM, 2002.

[13] Jean-François Cordeau, Michel Gendreau, and Gilbert Laporte. A Tabu Search Heuristic for Periodic and Multi-Depot Vehicle Routing Problems. *Networks*, 30(2):105–119, 1997.

[14] Jean-François Cordeau, Gilbert Laporte, and Anne Mercier. A unified tabu search heuristic for vehicle routing problems with time windows. *Journal of the Operational Research Society*, 52(8):928–936, August 2001.

[15] Jean-François Cordeau, Gilbert Laporte, and Anne Mercier. Improved tabu search algorithm for the handling of route duration constraints in vehicle routing problems with time windows. *Journal of the Operational Research Society*, 55(5):542–546, May 2004.

[16] Jean-François Cordeau and Mirko Maischberger. A Parallel Iterated Tabu Search Heuristic for Vehicle Routing Problems. *Computers & Operations Research*, 2011.

[17] Harlan Crowder, Ellis L. Johnson, and Manfred Padberg. Solving Large-Scale Zero-One Linear Programming Problems. *Operations Research*, 31(5):803–834, 1983.

[18] Emilie Danna and Claude Le Pape. Branch-and-Price Heuristics: A Case Study on the Vehicle Routing Problem with Time Windows. In Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors, *Column Generation, GÉRAD 25th anniversary series*, pages 99–129. Springer, 2005.

[19] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press-RAND Coorporation, 1963.

[20] George B. Dantzig, Delbert R. Fulkerson, and Selmer M. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, pages 393–410, 1954.

[21] George B. Dantzig and John H. Ramser. The Truck Dispatching Problem. *Management Science*, 6:80–91, 1959.

[22] George B. Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.

[23] Kenneth A. De Jong. Genetic algorithms: A 10 year perspective. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 169–177. Lawrence Erlbaum Associates, July 1985.

[24] Jean-Louis Deneubourg, Serge Aron, Simon Goss, and Jacques M. Pasteels. The self-organizing exploratory pattern of the Argentine ant. *Journal of Insect Behavior*, 3(2):159–168, 1990.

[25] Martin Desrochers, Jacques Desrosiers, and Marius M. Solomon. A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows. *Operations Research*, 40(2):342–354, 1992.

[26] Jacques Desrosiers and Marco E. Lübbecke. A Primer in Column Generation. In Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors, *Column Generation, GÉRAD 25th anniversary series*, pages 1–32. Springer, 2005.

[27] Jacques Desrosiers and Marco E. Lübbecke. *Wiley Encyclopedia of Operations Research and Management Science*, chapter Branch-Price-and-Cut Algorithms. John Wiley & Sons, 2010.

[28] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[29] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant Colony Optimization: Artificial Ants as a Computational Intelligence Technique. Technical Report TR/IRIDIA/2006-023, Université Libre de Bruxelles, September 2006.

[30] Marco Dorigo and Luca M. Gambardella. Ant Colony System: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[31] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The Ant System: An Autocatalytic Optimizing Process. Technical Report 91-016, Politecnico di Milano, 1991.

[32] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Books. MIT Press, 2004.

[33] Karl F. Dörner, Manfred Gronalt, Richard F. Hartl, Marc Reimann, Christine Strauss, and Michael Stummer. SavingsAnts for the Vehicle Routing Problem. In *Proceedings of the Applications of Evolutionary Computing on EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN*, pages 11–20. Springer, 2002.

[34] Karl F. Dörner and Verena Schmid. Survey: Matheuristics for Rich Vehicle Routing Problems. In María Blesa et al., editors, *Proceedings of the 7th International Workshop on Hybrid Metaheuristics (HM 2010)*, volume 6373 of *LNCS*, pages 206–221. Springer, October 2010.

[35] Dominique Feillet, Pierre Dejax, Michel Gendreau, and Cyrille Gueguen. An Exact Algorithm for the Elementary Shortest Path Problem with Resource Constraints: Application to some Vehicle Routing Problems. *Networks*, 44(3):216–229, 2004.

[36] Thomas A. Feo and Mauricio G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67–71, 1989.

[37] John J. Forrest and Donald Goldfarb. Steepest-edge simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, December 1992.

[38] Peter M. Francis, Karen R. Smilowitz, and Michal Tzur. The Period Vehicle Routing Problem and its Extensions. In Bruce Golden et al., editors, *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43 of *Operations Research/Computer Science Interfaces Series*, pages 73–102. Springer, 2008.

[39] Komei Fukuda and Tamás Terlaky. Criss-cross methods: A fresh view on pivot algorithms. *Mathematical Programming*, 79:369–395, 1997.

[40] Günther Füllerer, Karl F. Dörner, Richard F. Hartl, and Manuel Iori. Ant colony optimization for the two-dimensional loading vehicle routing problem. *Computers & Operations Research*, 36:655–673, March 2009.

[41] David Gale, Harold W. Kuhn, and Albert W. Tucker. Linear programming and the theory of games. *Activity Analysis of Production and Allocation*, pages 317–329, 1951.

[42] Luca M. Gambardella, Éric Taillard, and Giovanni Agazzi. MACS-VRPTW: A Multiple Ant Colony System for vehicle routing problems with time windows. In *New Ideas in Optimization*, Advanced Topics in Computer Science, pages 63–76. McGraw-Hill, 1999.

[43] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, May 1986.

[44] Fred Glover. Tabu search: part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[45] Alan J. Goldman and Albert W. Tucker. Theory of linear programming. *Linear Inequalities and Related Systems, Annals of Mathematical Studies*, 38:53–97, 1956.

[46] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.

[47] Simon Goss, Serge Aron, Jean-Louis Deneubourg, and Jacques M. Pasteels. Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, 76(12):579–581, 1989.

[48] Martin Grötschel, Lászlo Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988.

[49] Pierre Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. In *Proceedings of Congress on Numerical Methods in Combinatorial Optimization*, pages 70–145, 1986.

[50] Paula M.J. Harris. Pivot selection methods of the Devex LP code. *Mathematical Programming*, 5(1):1–28, 1973.

[51] John H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

118

[52] Stefan Irnich and Guy Desaulniers. Shortest Path Problems with Resource Constraints. In Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors, *Column Generation, GÉRAD 25th anniversary series*, pages 33–65. Springer, 2005.

[53] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, December 1984.

[54] Richard M. Karp. Reducibility among combinational problems. *Complexity of Computer Computations*, pages 85–104, 1972.

[55] Scott Kirkpatrick, Charles D. Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[56] Victor Klee and George J. Minty. How Good is the Simplex Algorithm? In Shisha Oved, editor, *Inequalities III, Proceedings of the Third Symposium on Inequalities at the University of California*, pages 159–175. Academic Press, 1972.

[57] A.H. Land and A.G. Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, 28(3):497–520, July 1960.

[58] Gilbert Laporte, Yves Norbert, and Danielle Arpin. Optimal solutions to capacitated multi-depot vehicle routing problems. In *Congressus Numerantium*, volume 44, pages 283–292, 1984.

[59] Eugene L. Lawler, Jan K. Lenstra, Alexander H.G. Rinnooy Kan, and David B. Shmoys. *The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley Series in Discrete Mathematics & Optimization. John Wiley & Sons, 1985.

[60] Ivana Ljubić. A Branch-and-Cut-and-Price Algorithm for Vertex-Biconnectivity Augmentation. *Networks*, 56(3):169–182, October 2010.

[61] Manuel López-Ibáñez, Christian Blum, Dhananjay Thiruvady, Andreas Ernst, and Bernd Meyer. Beam-ACO Based on Stochastic Sampling for Makespan Optimization Concerning the TSP with Time Windows. In Carlos Cotta and Peter Cowling, editors, *Evolutionary Computation in Combinatorial Optimization*, volume 5482 of *LNCS*, pages 97–108. Springer, 2009.

[62] Helena R. Lourenço, Olivier Martin, and Thomas Stützle. Iterated local search. In *Handbook of Metaheuristics*, volume 57 of *International Series in Operation Research and Management Science*, pages 321–353. Kluwar Academic Publishers, 2002.

[63] Ana C. Matos and Rui C. Oliveira. An Experimental Study of the Ant Colony System for the Period Vehicle Routing Problem. In Marco Dorigo et al., editors, *Ant Colony Optimization and Swarm Intelligence, ANTS 2004*, volume 3172 of *LNCS*, pages 286–293. Springer, 2004.

[64] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24:1097–1100, 1997.

[65] Magalie Mourgaya Virapatrin and François Vanderbeck. Column generation based heuristic for tactical planning in multi-period vehicle routing. *European Journal of Operational Research*, 183(3):1028–1041, 2007.

[66] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*, volume 18 of *Wiley Interscience Series in Discrete Mathematics & Optimization*. John Wiley & Sons, 1988.

[67] Phuong K. Nguyen, Teodor G. Crainic, and Michel Toulouse. A Hybrid Genetic Algorithm for the Periodic Vehicle Routing Problem with Time Windows. Technical Report 2011-25, CIRRELT, April 2011.

[68] William Orchard-Hays. *Advanced Linear-Programming Computing Techniques*. McGraw-Hill, 1968.

[69] Manfred Padberg and Giovanni Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1):1–7, March 1987.

[70] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[71] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover, 1998.

[72] Sophie N. Parragh, Karl F. Dörner, and Richard F. Hartl. A survey on pickup and delivery problems, Part II: Transportation between pickup and delivery locations. *Journal für Betriebswirtschaft*, 58(2):81–117, 2008.

[73] Sandro Pirkwieser and Günther R. Raidl. A Variable Neighborhood Search for the Periodic Vehicle Routing Problem with Time Windows. In Caroline Prodhon et al., editors, *Proceedings of the 9th EU/MEeting on Metaheuristics for Logistics and Vehicle Routing*, 2008.

[74] Sandro Pirkwieser and Günther R. Raidl. A Column Generation Approach for the Periodic Vehicle Routing Problem with Time Windows. In Maria G. Scutellà et al., editors, *Proceedings of the International Network Optimization Conference 2009*, April 2009.

[75] Sandro Pirkwieser and Günther R. Raidl. Boosting a Variable Neighborhood Search for the Periodic Vehicle Routing Problem with Time Windows by ILP Techniques. In Stefan Voss and Marco Caserta, editors, *Proceedings of the 8th Metaheuristic International Conference (MIC 2009)*, July 2009.

[76] Sandro Pirkwieser and Günther R. Raidl. Multiple Variable Neighborhood Search enriched with ILP Techniques for the Periodic Vehicle Routing Problem with Time Windows. In María J. Blesa et al., editors, *Proceedings of Hybrid Metaheuristics – Sixth International Workshop, HM 2009*, volume 5818 of *LNCS*, pages 45–59. Springer, 2009.

[77] Sandro Pirkwieser and Günther R. Raidl. Matheuristics for the Periodic Vehicle Routing Problem with Time Windows. In *Proceedings of Matheuristics 2010: third international workshop on model-based metaheuristics*, June 2010.

[78] Jakob Puchinger and Günther R. Raidl. Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization: A Survey and Classification. In José Mira and José R. Álvarez, editors, *Proceedings of the First International Work-Conference on the Interplay Between Natural and Artificial Computation*, volume 3562 of *LNCS*, pages 41–53. Springer, June 2005.

[79] Günther R. Raidl. A Unified View on Hybrid Metaheuristics. In Francisco Almeida et al., editors, *Proceedings of the Hybrid Metaheuristics Workshop*, volume 4030 of *LNCS*, pages 1–12. Springer, October 2006.

[80] Ted K. Ralphs, Laszlo Ladányi, and Matthew J. Saltzman. Parallel Branch, Cut, and Price for Large-Scale Discrete Optimization. *Mathematical Programming*, 98:253–280, September 2003.

[81] M.R. Rao. A Note on the Multiple Traveling Salesmen Problem. *Operations Research*, pages 628–632, 1980.

[82] Marc Reimann, Karl F. Dörner, and Richard F. Hartl. D-Ants: Savings Based Ants divide and conquer the vehicle routing problem. *Computers & Operations Research*, 31(4):563–591, April 2004.

[83] Geraldo Ribeiro Filho and Luiz A.N. Lorena. Constructive Genetic Algorithm and Column Generation: an Application to Graph Coloring. In *Proceedings of APORS 2000 - The Fifth Conference of the Association of Asian-pacific Operations Research Societies*, July 2000.

[84] Stefan Ropke and Jean-François Cordeau. Branch-and-Cut-and-Price for the Pickup and Delivery Problem with Time Windows. *Transportation Science*, 43(3):267–286, 2009.

[85] Martin W.P. Savelsbergh. The Vehicle Routing Problem with Time Windows: Minimizing Route Duration. *ORSA Journal on Computing*, 4:146–154, 1992.

[86] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, April 1970.

[87] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer, 2003.

[88] Marius M. Solomon. Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints. *Operations Research*, 35(2):254–265, 1987.

[89] Thomas Stützle and Holger H. Hoos. MAX–MIN Ant System. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[90] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, 2009.

[91] Tamás Terlaky. A finite crisscross method for oriented matroids. *Journal of Combinatorial Theory, Series B*, 42(3):319–327, 1987.

[92] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 2002.

[93] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265, 1937.

[94] Dalessandro S. Vianna, Luiz S. Ochi, and Lúzia M.A. Drummond. A Parallel Hybrid Evolutionary Metaheuristic for the Period Vehicle Routing Problem. In José D.P. Rolim et al., editors, *Proceedings of Parallel and Distributed Processing, 11 IPPS/SPDP'99 Workshops*, volume 1586 of *LNCS*, pages 183–191. Springer, 1999.

[95] Thibaut Vidal, Teodor G. Crainic, Michel Gendreau, and Christian Prins. A Hybrid Genetic Algorithm with Adaptive Diversity Management for a Large Class of Vehicle Routing Problems with Time Windows. Technical Report 2011-61, CIRRELT, October 2011.

[96] Christos Voudouris and Edward Tsang. Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113(2):469–499, 1999.

[97] Bin Yu and Zhong Zhen Yang. An ant colony optimization model: The period vehicle routing problem with time windows. *Transportation Research Part E: Logistics and Transportation Review*, 47(2):166–181, 2011.

[98] Tao Zhang, Shanshan Wang, Wenxin Tian, and Yuejie Zhang. ACO-VRPTWRV: A New Algorithm for the Vehicle Routing Problems with Time Windows and Re-used Vehicles based on Ant Colony Optimization. In *Sixth International Conference on Intelligent Systems Design and Applications, 2006 (ISDA '06)*, volume 1, pages 390–395, October 2006.