FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

The Vehicle Routing Problem with Compartments

Exact and Metaheuristic Approaches

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Philipp Gebhard, Bakk.techn.

Matrikelnummer 0455783

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 14.10.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Philipp Gebhard, Bakk.techn.
Schenkendorfgasse 44/31, 1210 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 danke meinen Betreuern, Prof. Dr. Günther Raidl und Dr. Sandro Pirkwieser, für die ausführlichen Diskussionen und ihre Unterstützung in den Lehrveranstaltungen und bei der Erstellung dieser Arbeit. Des weiteren möchte ich mich bei den Mitarbeitern der Arbeitsgruppe um Prof. Raidl für die Bereitstellung der Infrastruktur und deren tatkräftiger Unterstützung und hilfreichen Tipps bedanken.

Meine größte Dankbarkeit gilt meinen Eltern Erna und Georg Gebhard, sowie meiner Partnerin Lea und meinen Freunden für ihre Unterstützung. Ich möchte mich bei allen Personen bedanken, die mich während des Studiums unterstützt haben.

Abstract

The Vehicle Routing Problem with Compartments (VRPC) deals with solving a generalized vehicle routing problem with a homogeneous fleet of vehicles having multiple compartments as well as additional constraints on the commodities loaded in each compartment: i.e. incompatibilities between different products in one compartment and between products and compartments. Two slightly different problem formulations, which where inspired by the petrol and food delivery industries, are considered in this work. The vehicles delivering different petrol types, are typically divided into several compartments with a fixed size where different petrol types are not allowed to be loaded into the same compartment as they would agitate. The vehicles in the food distribution industry consist of different compartments with flexible core walls where the products must be loaded into a certain, pre-defined compartment. Routing problems are typically hard to solve as the corresponding decision problems often are members of the so called $\mathcal{NP}$-hard problems.

This work presents two heuristic algorithms to solve the VRPC, which are a randomized Clarke and Wright Savings construction algorithm and an improvement algorithm based on swarm intelligence. Further an exact branch and price approach using column generation is presented, which divides the problem into several independent subproblems that can be solved individually. The cascaded binpacking like problem is solved using a heuristic algorithm and a constraint programming model. The performance of the algorithms is evaluated on three different sets of test instances. The exact approach for the VRPC, like other exact approaches for similar routing problems, is able to solve instances to optimality with a very limited size. In contrast the heuristic approaches where able to solve any instance within a reasonably small gap compared to other algorithms.

Kurzfassung

Das Vehicle Routing Problem with Compartments (VRPC) beschreibt ein generalisiertes Routingproblem mit einer homogenen Flotte an Fahrzeugen mit mehreren Abteilungen, die Bestellungen von Kunden kostenminimal ausliefern sollen. Weiteres gibt es unterschiedliche Produkttypen, die entweder nur in gewisse Abteilungen oder nicht mit anderen Produkttypen ins gleiche Abteil geladen werden dürfen. Diese Arbeit behandelt zwei leicht unterschiedliche Problemdefinitionen die auf Anforderungen aus der Öl- und Nahrungsmittelindustrie eingehen. Die Fahrzeuge, die zum Transport unterschiedlicher Kraftstoffe von einer Raffinerie zu den einzelnen Tankstellen eingesetzt werden, haben typischerweise mehrere Abteilungen mit einer fixen Größe. Die verschiedenen Kraftstoffe dürfen in beliebige Abteile geladen, aber nicht miteinander vermischt werden. Fahrzeuge, die für den Transport von Nahrungsmitteln eingesetzt werden, haben oft mehrere Abteilungen, die durch eine verstellbare Trennwand, variabel in ihrer Größe sind. Eine optimale Lösung für Routingprobleme ist meist nur mit großem Aufwand zu finden, da das zugehörige Entschiedungsproblem zu den sogenannten $\mathcal{NP}$-schweren Problemen gehört.

In dieser Arbeit werden zwei heuristische Algorithmen vorgestellt: ein randomisierter Clarke and Wright Savings Algorithmus und eine auf Schwarmintelligenz beruhende Verbesserungsheuristik. Des Weiteren wird ein exakter Branch and Price Ansatz präsentiert, der mittels Spaltengenerierung das Problem in unabhängige Teilprobleme aufteilt und löst. Das verschachtelte Packproblem wird einerseits heuristisch mit Konstruktionsalgorithmen und andererseits mit einem Constraint Programming Modell gelöst. Die Effektivität der Modelle und Algorithmen wurden auf drei verschiedenen Testdatensätzen evaluiert. Das exakte Verfahren kann, wie auch bei verwandten generalisierten Routingproblemen, nur auf relativ kleinen Probleminstanzen angewendet werden, während die heuristischen Ansätze in der Lage sind alle gegebenen Probleminstanzen relativ gut zu lösen.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In nowadays industry efficient transportation and distribution of goods is essential to the success of many companies. In an environment with increasing energy costs, efficient planning of the delivery paths allows to save resources and money. One of the most fundamental and well studied problems in this environment is the traveling salesman problem (TSP). The aim of the TSP is to find the cheapest path through all cities, the salesman wants to visit. Each city has to be visited exactly once, the tour has to start and stop at the same location. The total tour length or costs have to be minimal. From the theoretical point of view the TSP is a combinatorial optimization problem that is very important for a huge area of applications, including logistics, production paths, semiconductor industry and many more.

The Vehicle Routing Problem (VRP) is a generalization of the TSP with more than one salesman and exactly one depot, where each vehicle has to start and end its tour. In the context of the VRP we talk about customers that are visited by the vehicles instead of cities and the salesman. In the classical VRP, each customer has to be visited exactly once. Today a lot of generalizations of the VRP are subject of intense studies: VRP with Pickup and Delivery, Periodic VRP, Capacitated VRP, VRP with Time Windows, VRP with non uniform vehicles and even more generalizations towards industries needs and applications. In contrast, the Vehicle Routing Problem with Compartments (VRPC) is an optimization problem that has not received much attention yet. The aim of this thesis is to analyze the performance of two solution methods, namely a Particle Swarm Optimization (PSO) algorithm and an exact solution approach based on Column Generation (CG) and Branch & Bound (B&B). These algorithms have not been subject of research on this particular problem yet.

The Vehicle Routing Problem with Compartments has a practical relevance in the petrol and food delivery as well as in waste collection industries. Oil companies need to deliver different types of oil from the refinery to the gas stations, using a single vehicle. The fueling vehi-

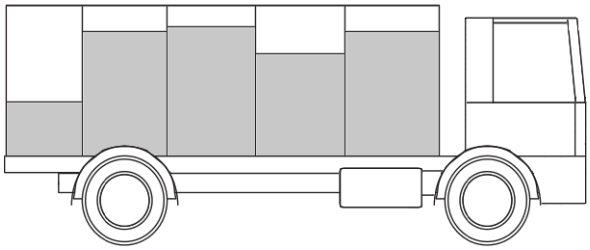

Figure 1.1: A vehicle with fixed compartments for petrol delivery

cles, as the one in Figure 1.1, typically have several compartments which can be filled with different fuel types. It does not matter in which compartment the different types are loaded but it is clearly not allowed to load them in the same compartment as they would agitate. Modern vehicles which deliver food have several compartments with different climatic conditions. This renders a slightly different problem definition, as it is allowed to load different products into the same compartment (see Figure 1.2). Waste collection and recycling are major branches of industry that gain an increasing importance in the logistics supply chain.

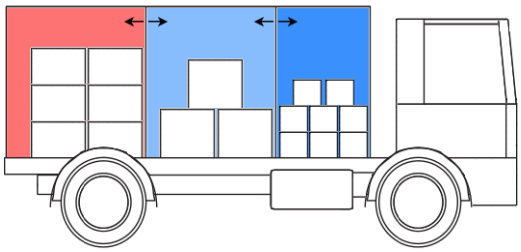

Figure 1.2: A vehicle with flexible compartments for food delivery

This special generalization of the VRP has not received a lot of attention from the scientific part. A short overview of some scientific publications concerning this problem is given in Section 1.3.

1.2 Problem Description and Analysis

The classical VRP deals with solving a multiple TSP, where each city (i.e. customer) is visited by exactly one salesman (i.e. vehicle). All vehicles start from a depot, delivering the goods to the customers and eventually return to the depot. The goal is to minimize the total routing costs, which can either be the total distance or the total time the vehicles drive. There are even more complex cost functions, which take several inputs into account. This problem was first proposed by Dantzig and Ramser in 1959 [12].

The VRPC considers, like the VRP, a set of customers and one central depot from where a certain amount of vehicles deliver the demanded goods to the customers. In addition, con-

straints on the vehicles with loading capacities and incompatibilities between compartments and different product types, are added. Each vehicle has a certain amount of compartments with a maximum loading capacity and a total vehicle capacity that might not be exceeded. The incompatibility constraints define, depending on the problem instance, whether two product types are allowed to be loaded into the same compartment or demands can be loaded into a predefined compartment only. In contrast to the classical VRP, it is allowed to serve a customer by several vehicles.

This thesis deals with two distinct types of properties on the compartments: vehicles with flexible and fixed compartments. The conditions on the problem instances with fixed size compartments are inspired by the petrol industry: each product can be loaded into any compartment, but distinct product types are not allowed to be loaded in the same compartment. For these instances two problems, namely a generalized VRP and a Bin Packing like Problem (BPP) with a limited number of bins, have to be solved simultaneously. It is not possible to solve the problems separately, since the solutions to both highly depend on each other: i.e. for a good routing of the vehicles the demands might not fit into and vice versa.

The problem instances, inspired by the food delivery and waste collection industries, have flexible compartments where the size of each compartment has a continuous domain. This eliminates the need of solving the Bin Packing Problem, since it is enough to check whether the sum of the demands, in all compartments, does not exceed the vehicle capacity.

Linear Program Formulation

The problem formulation as a Mixed Integer Program (MIP) was proposed by Derigs et al. [14]. For sake of completeness it is depicted also here:

$$max. \sum_{v \in V} \sum_{i \in L} \sum_{j \in L} cost_{ij} * b_{ijv} \tag{1.1}$$

$$s.t. \sum_{j \in L_c} b_{0jv} \leq 1 \qquad \forall v \in V \tag{1.2}$$

$$\sum_{i \in L} b_{ikv} = \sum_{j \in L} b_{kjv} \qquad \forall v \in V, \forall k \in L \tag{1.3}$$

$$u_{iv} - u_{jv} + |L| * b_{ijv} \leq |L_c| \qquad \forall v \in V, \forall i \in L, \forall j \in L_c \tag{1.4}$$

$$u_{l_0 v} = 1 \qquad \forall v \in V \tag{1.5}$$

$$\sum_{o \in O} quantity(o) * x_{ovc} \leq compCapa(c) \qquad \forall v \in V, \forall c \in C \tag{1.6}$$

$$\sum_{o \in O} \sum_{c \in C} quantity(o) * x_{ovc} \leq vehCapa \qquad \forall v \in V \tag{1.7}$$

$$\sum_{v \in V} \sum_{c \in C} x_{ovc} = 1 \qquad \forall o \in O \tag{1.8}$$

$$\sum_{o \in ordCust(j)} \sum_{c \in C} x_{ovc} \leq |O| * \sum_{i \in L} b_{ijv} \qquad \forall v \in V, \forall j \in L_c \qquad (1.9)$$

$$\sum_{o \in ordProd(p)} x_{ovc} \leq |O| * y_{pvc} \qquad \forall p \in P, \forall v \in V, \forall c \in C \qquad (1.10)$$

$$y_{pvc} = 0 \qquad \forall (p,c) \in IncProdComp, \forall v \in V \qquad (1.11)$$

$$y_{pvc} + y_{qvc} \leq 1 \qquad \forall (p,q) \in IncProd, \forall v \in V, \forall c \in C \qquad (1.12)$$

$$b_{ijv} \in \{0,1\} \qquad \forall i,j \in L, \forall v \in V \qquad (1.13)$$

$$u_{iv} \in \{1, \ldots |L|\} \qquad \forall i \in L, \forall v \in V \qquad (1.14)$$

$$x_{ovc} \in \{0,1\} \qquad \forall o \in O, \forall v \in V, \forall c \in C \qquad (1.15)$$

$$y_{pvc} \in \{0,1\} \qquad \forall p \in P, \forall v \in V, \forall c \in C \qquad (1.16)$$

The objective function (1.1) minimizes the total travel costs by multiplying the binary variable b_{ijv}, that is 1 if and only if vehicle v drives from location i to j, by $cost_{ij}$, the travel costs from location i to j.

Constraints (1.2) and (1.3) ensure that at each vehicle v starts at most once from the depot, that is location 0, and ends its tour at the depot. Since the sum of all ingoing and outgoing binary variables at each node and for each vehicle v must be equal, the vehicle is forced to end its tour at the depot. The sub-cycle elimination constraints (1.4) and (1.5) enforce that the position variable $u_{iv} < u_{jv}$ if vehicle v drives from location i to j. The depot must be in the first position of the tour. Without these constraints, the result could contain more than one distinct cycle: the tour starting and ending at the depot and other cycles that do not have any customer in common.

Constraints (1.6), (1.7) and (1.8) ensure that neither the compartment nor the vehicle is overloaded and each order is packed in exactly one vehicle and in one compartment. Constraints (1.9) link the tour with the packing variables by enforcing the vehicle v to visit customer j if any demand of customer j is loaded into any compartment of v. Constraints (1.10) link the loading variables x_{ovc} (demand o is loaded in vehicle v and compartment c) with the compartment variables y_{pvc} (product p is assigned to compartment c in vehicle v). They are needed to model the incompatibilities between the demands, other demands and compartments. These incompatibility constraints are modeled with (1.11) and (1.12) and the relations $IncProdComp$ and $IncProd$.

The last constraints (1.13) - (1.16) define the domain of the variables: all variables in the MIP formulation are binary, except the sub-cycle elimination variables u_{iv} that are integers.

Problem Complexity

Complexity theory is a large field of computer science, that is receiving a lot of attention. There exist some unanswered questions with a huge (potential) impact on any other discipline in computer science. One of the most prominent unresolved question in this field is

called $\mathcal{P}$ versus $\mathcal{NP}$: Given a problem, where any solution can be verified efficiently (with a polynomial worst case running time and memory consumption), is there also an efficient method to actually find such a solution? The reader is refereed to [25] and [41] for a very detailed insight in the field of complexity theory in computer science.

The $\mathcal{P}$ versus $\mathcal{NP}$ question is highly important for the area of Operations Research and Combinatorial Optimization Problems (COP) that often arise from this field. A COP deals with finding the best solution among all feasible solutions for an optimization problem. Depending on the objective function the COP is a maximization or minimization problem. For any COP a nonempty subset of all variables has a discrete domain.

Definition 1. *Formally a Combinatorial Optimization Problem P is defined as $\mathcal{P} = (S, f)$*

- *A set of variables with their respective domains $x_1 \in D_1, x_2 \in D_2, \dots x_n \in D_n$*
- *Constraints among the variables (e.g. $x_1 \neq x_2$ or $\sum_{i=0\dots n} x_i \leq C \in D_1 \cap D_2 \cap \dots \cap D_n$)*
- *The fitness or objective function $f : D_1 \times D_2 \times \dots D_n \rightarrow \mathcal{R}$ that evaluates each element in S*
- *A set S of all feasible solutions: $S = \{(x_1 = v_1, x_2 = v_2 \ \dots \ x_n = v_n) \mid \forall i \in \{0, \dots n\}, v_i \in D_i, \ s$ satisfies all constraints $\}$*

The goal is to find an element $s_{opt} \in S : \nexists s' \in S f(s') > f(s_{opt})$ for a maximization problem and $f(s') < f(s_{opt})$ for a minimization problem.

For each COP there exists a corresponding decision problem $\mathcal{D}$ that asks if there exists a solution with an improved objective function value for a given solution. For each COP $\mathcal{P}$ the corresponding decision problem $\mathcal{D}$ determines the complexity of the problem:

Definition 2. *The decision problem $\mathcal{D}$ for a Combinatorial Optimization Problem $\mathcal{P}$ asks if, for a given solution $s \in S$, there exists a solution $s' \in S$, such that $f(s')$ is better than $f(s)$: for a minimization problem this means $f(s') < f(s)$ and for a maximization problem $f(s') > f(s)$.*

Some optimization problems that can be solved in polynomial time, such as shortest paths or minimum spanning trees in a graph, may become a $\mathcal{NP}$-hard problem by adding a single constraint. For example the minimum spanning tree (MST), for any given graph, can be computed in $\mathcal{O}(|V|^2)$ using Kruskals algorithm. But when the total number of vertexes in the resulting tree is constrained, the time complexity of the problem increases exponentially in the size of the input. The resulting k-minimum spanning tree problem [49] is an $\mathcal{NP}$-hard problem.

For most COPs a heuristic algorithm is able to calculate a feasible solution efficiently, but many COPs are hard to solve to optimality, since the size of the set S may grow exponentially

in the size of the input and there is no efficient algorithm known, that is able to explore the whole search space efficiently. Deciding if a solution $s' \in S$ with $f(s') < f(s)$ exists may be $\mathcal{NP}$-complete, depending on the problem:

Definition 3. *A COP is a $\mathcal{NP}$-optimization problem (NPO) if the corresponding decision problem is a $\mathcal{NP}$-complete problem.*

The membership of any problem P_x to the class of the $\mathcal{NP}$-complete problems can be shown by reducing a known $\mathcal{NP}$-complete problem $P_n \in \mathcal{NP}$-complete to P_x in polynomial time. Further it must be shown that verifying whether a solution candidate for P_x is correct can be done in polynomial time. In a formal correct and complete proof this reduction is done in the context of a Touring machine [25, 41]. As the VRP is known to be a $\mathcal{NP}$-complete problem and the VRPC is a generalized variant of it, it can be deduced that the VRPC is $\mathcal{NP}$-complete too.

Due to these definitions the VRPC is a NPO with S, the set of all feasible routes and packing schemes, and $f(s)$, the accumulated routing costs of any vehicle in s.

Solving the instances with fixed size compartments, inspired by the petrol industry, implies solving a Bin Packing like Problem. Although the BPP is known to be $\mathcal{NP}$-hard, the packing problem of the VRPC is easier to solve in practice, as it asks for a feasible packing and not for the minimal number of bins. None the less the decision problem *"Does there exist a solution with a less number of bins?"* is $\mathcal{NP}$-complete and causes the packing problem hard to solve in case the approximation algorithm did not find a feasible packing scheme.

From a practical point of view, the packing problem is easier to solve than the routing problem, since there are efficient approximation algorithms known, which provide a guarantee on the number of bins. First-fit-decreasing is able to solve the BPP in $\mathcal{O}(n * log(n))$ with a guaranteed performance of $^3/_2$ of the optimal solution. Approximation algorithms with a performance guarantee $^x/_y$ provide solutions, where the difference of the worst solution the algorithm calculates ($Alg(I)$) to the optimal solution ($Opt(I)$) is at most $Opt(I) = {}^x/_y * Alg(I)$. Such approximation algorithms are very interesting as they provide guarantees on the bounds of their solutions, which is a very useful property. The interested reader is refereed to [59, 38] for a detailed insight in the field of approximation algorithms and their classification.

1.3 State of the Art and Previous Work

The VRPC has not yet received a lot of attention from the scientific point of view. Although the problem as discussed here was defined only a few years ago some work on very similar problem variants has started already in the 80's [7]. In many publications similar loading constraints were studied. Muyldermans and Pang [39] and Derigs et al. [14] give a short introduction on the effort that has been made.

The problem formulation adopted in this work was defined by Derigs et al. [14] in 2010. They proposed several construction heuristics, a local search and large neighborhood search

algorithms to improve the solutions. They added all to a suite of algorithms where each heuristic has several modification and improvement operators. Further they created a large set of test instances with 10, 25, 50, 100 and 200 customers each ordering more than 5 demands.

El Fallahi et al. [21] studied a problem formulation for the food delivery industry using one compartment for each product and evaluated the performance of a memetic algorithm and a tabu search heuristic. The advantage of using multiple compartments over a single compartments was analyzed by Muyldermans and Pang [39]. They compared the resulting costs by using co-collection, vehicles with several compartments, and a separate distribution by more vehicles. They applied a guided local search and a local search heuristic.

Pirkwieser, Raidl and Gottlieb analyzed Variable Neighborhood Search (VNS) approaches with a large number of neighborhoods for the VRPC [47] and periodic routing problems [46]. They were able to improve most of the best known solutions for the Derigs et al. and obtained competitive results on the modified Eilon and Christofides instances. Pirkwieser analyzed in the context of his PhD thesis [44] several generalized variants of the VRP using different combinations of heuristic and exact algorithms.

The following list gives a short summary of the algorithms that were analyzed by others:

- **Memetic Algorithm**: El Fallahi et al. presented a memetic algorithm, that is the result of combining a genetic algorithm and a local search procedure. Genetic algorithms are inspired by the theory of evolution: when a new individual is created it inherits properties from both parents and is slightly mutated. Over the iteration only the fittest survive, i.e. the individuals with the best objective value.

- **Tabu Search**: The second metaheuristic studied by El Fallahi et al. is Tabu Search, that is basically a local search procedure that is enhanced with memory structures. Such a memory information can be: already performed modifications to the solution are forbidden for a certain number of iterations.

- **Adaptive Variable Neighborhood Search**: AVNS algorithms are based on a local search algorithm that switches between the neighborhoods it searches in, depending on adaptive information, such as the improvements found in each neighborhood. If the algorithm does not find any improvement using a search scheme it switches to the next.

- **Large Neighborhood Search**: Derigs et al. created a large set of neighborhoods, construction and local search algorithms. The majority of the instances used to evaluate the performance of the algorithms, presented later in this work, were proposed by them.

1.4 Aim of this Work

As mentioned before, solving COPs implies the search for the best solution among a possibly huge set of feasible solutions. In general there does not exist a heuristic that performs better on all kind of problems than some other heuristic including random search. This results from the famous *no free lunch* theorem [63]. As a consequence researchers try to acquire and include as much knowledge as possible into sophisticated solvers for the different problem classes. They try to evaluate the performance of different metaheuristic and exact algorithms.

Definition 4. *Metaheuristic algorithms make no assumptions on the problem and (in theory) can be applied on any optimization problem. They define an abstract order of instructions which lead to improved or feasible solutions. In almost any case these instructions must be implemented using problem specific knowledge.*

The FFD heuristic described earlier is a heuristic algorithm as it makes assumptions on the problem: there must be items which can be ordered, whereas the local search procedure is a metaheuristic algorithm and leaves the problem specific implementation to the neighborhood it searches. Section 2.3 provides a detailed description of the local search metaheuristic.

The aim of work is to implement solution methods that have not been subject of publications on the VRPC yet: a Particle Swarm Optimization (PSO) algorithm and an exact solution approach based on Branch & Bound (B&B) and Column Generation (CG). PSOs are not widely used for COPs, since the PSO metaheuristic needs a continuous domain, hence some tricks have to be applied in order to solve a COP with its (partially) discrete domain. But they use three types of information to improve the current solution: the current solution itself, a knowledge and a social component. The combination of these three components might result in improved solutions. Combining a genetic algorithm and a local search procedure is called a memetic algorithm and often results in a good heuristic solver. This idea is applied on the PSO to improve its performance.

The exact solutions are calculated using a Branch & Price (B&P) approach [17], based on a set covering formulation, where each variable represents one feasible tour and every order must be covered by at least one tour. The pricing subproblem searches for variables that might improve the solution or proves that no such improvement is possible under the current conditions. Such an approach for the Periodic Vehicle Routing Problem with Time Windows is described in Pirkwieser et al. [45].

The solution approaches are evaluated in the last chapter using available benchmark instances e.g. those of Derigs et al. [14] and modified Eilon and Christofides instances.

1.5 Methodological Approach

For most $\mathcal{NP}$-hard problems in real world applications, fast heuristics are used, since they usually find good solutions in reasonable time. Especially if the size of the input increases, the total running time increases at most polynomial. It does not matter if the solution is slightly worse than the optimal solution. In addition, most algorithms do not consider noise or side conditions, such as a traffic jam, legal issues or the human factor. In a bad case these side conditions may undo the good results obtained by a sophisticated algorithm that is able to calculate the optimal solution.

For sake of comprehension a very short introduction to the implemented algorithms will be given here. For detailed insight into the algorithms the reader is refereed to the next chapters or the respective publications. Further a short overview of the algorithms, discussed in the literature, that deal with the VRPC is given.

The algorithms discussed in this thesis:

- **Savings algorithm**: The Clarke and Wright [10] savings algorithm starts from individual tours to each customer. Then it iteratively evaluates each pair of tours according their savings (i.e. the reduced costs by merging these tours) and merges the tours with the largest savings.

- **Greedy Randomized Adaptive Search Procedure (GRASP)**: The idea behind a GRASP is to take a greedy algorithm and add a random component. The solution is generated like the greedy algorithm does but instead of taking the best component to extend the solution, the GRASP selects a random component from a set of good candidates. In this work the GRASP build on the savings algorithm, where the set of good candidates consists of pairs of tours.

- **Discrete Particle Swarm Optimization (PSO)**: Particles fly through the solution space and try to reach the best position in the swarm, using local and global information. PSO algorithms are mostly used in the field of continuous optimization. In order to apply it to a combinatorial optimization problem with its discrete domain, the PSO has to be modified slightly: the position of the particles is calculated by drawing a random number.

- **Local Search (LS)**: The algorithm iteratively searches in the so called neighborhood of a solution for an improvement. It accepts either the first, best or a random improved solution. The algorithm converges to a local optimum.

- **Branch & Price** solves the problem to optimality by starting from an reduced model of linear inequations and pricing out new variables that potentially improve the current solution (Column Generation). The main and sub problem are solved until no new variables can be generated and the solution is integral else a new branch is created.

1.6 Structure of this Work

The first chapter gave an introduction to the problem and the problem definition, an overview on its complexity and some general introduction of the used algorithms and methods. Further some general considerations about combinatorial optimization problems in practice are given.

In Chapter 2 the reader gets a detailed description of the implemented algorithms and concepts from a theoretical point of view. Chapter 3 gives a detailed insight into the implemented algorithms for the VRPC. Chapter 4 gives some considerations about the concrete implementation, the libraries that were used and analyzes the performance of the algorithms with a comparison to results from the literature. Chapter 5 gives a critical reflection about the methods and their performance and considerations about potential future work on this implementation and Chapter 6 a short summary.

CHAPTER 2

Solving Combinatorial Optimization Problems

As mentioned in the introduction, heuristic algorithms are essential in solving combinatorial optimization problems and widely used in practice. Among the metaheuristic algorithms two main classes can be distinguished: construction and improvement heuristics. Algorithms from the former class start from an empty solution and successively add solution components until the solution is complete, but might not be feasible. Algorithms from the latter class start from a given solution and improve it by applying their respective methods and finish when a termination criteria (limits on time, iterations, improvement in the fitness values, feasible solution ...) is fulfilled. The *Handbook of Combinatorial Optimization* [19] gives a very detailed insight into different algorithms and problems in this field.

From a theoretical point of view it is interesting to be able to calculate the optimal solution, although a large part of the scientific community assumes that for all NPOs there exists no algorithm which has a guaranteed polynomial worst case execution time complexity. This assumption can be refuted if it can be shown that $\mathcal{P} = \mathcal{NP}$, which can be done by finding an algorithm that solves a $\mathcal{NP}$-optimization problem to optimality with a polynomial time and memory complexity.

Linear Programming (LP) or mathematical programming is a large field of operations research and tries to optimize an objective function under a set of constraints. The objective function and all constraints are linear inequations and equations. Most optimization problems which arise from nowadays industries needs can be written as a linear program. A short introduction to the field of linear programming and the corresponding solving methods is given in Section 2.5. The following sections provide a detailed insight into the implemented metaheuristic algorithms.

2.1 Greedy Randomized Adaptive Search Procedure

Greedy randomized adaptive search procedures (GRASP) are members of the construction algorithms and mostly used to obtain solutions as a basis for other improvement schemes. The big advantage of these algorithms is their fast termination and the random component for diverging solutions. Further the random component is interesting for many improvement heuristics such as swarm intelligence, presented in Section 2.4, and evolutionary algorithms that highly depend on a diversified population to operate on.

A GRASP algorithm starts with an empty solution and iteratively adds solution components in a greedy manner to the partial solution. In every iteration the algorithm evaluates all solution candidates that are possible extensions to the partial solution obtained so far. The evaluated candidates are added to the Candidate List (CL) and a small subset of promising solution components is selected into the Restricted Candidate List (RCL). Most implementations perform a selection of a certain percentage of the best candidates as criteria. In the next step the algorithm selects a random element from the RCL and extends the current partial solution by adding this candidate. This loop is executed until the list of candidates is empty and the solution is complete or an error occurred.

This optimization scheme was introduced by Feo and Resende [24] in 1989. They built upon the book on semi greedy algorithms by Hart and Shogan [29]. Algorithm 2.1 shows the basic version of the GRASP metaheuristic.

Algorithm 2.1: Greedy Randomized Adaptive Search Procedure

input : A problem instance
output: A randomized greedy solution

```
1 sol ← ∅;
2 while sol not complete do
3 |   cl ← getCandidateList(sol);
4 |   rcl ← getRestrictedCandidateList(cl);
5 |   candidate ← selectCandidate(rcl);
6 |   extendSolution(sol, candidate);
7 end
8 return sol;
```

The size of the RCL determines the grade of the random behavior: if only the best element is added to the RCL, the resulting algorithm is a deterministic greedy algorithm that always selects the best element. In the context of the VRPC the GRASP would behave like a Best Fit approximation algorithm. In the contrary case, where CL = RCL, the resulting algorithm conforms to a random search algorithm.

2.2 Clarke and Wright Savings Heuristic

Clarke and Wright presented their Savings Heuristic [10] in 1964. Based on the consequence of the triangular inequality, which states that the longest edge in a triangle is less or equal to the sum of the two shorter edges, a solution for the VRP or a TSP is generated. The algorithm starts with a solution where each customer is visited by exactly one vehicle and successively merges the tours with the largest saving.

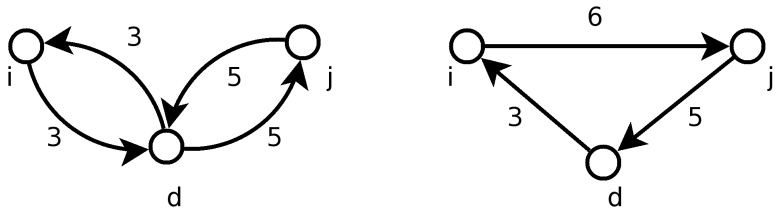

Figure 2.1: The idea behind the Clarke and Wright Savings [10] heuristic: the saving in this case is $S_{ij} = 3 + 5 - 6 = 2$

A tour can be created by appending the second tour at the end of the first tour. If the triangular inequality holds, the costs of the new tour is at most the the accumulated costs of the two separate tours. Figure 2.1 shows the case where the total costs are reduced by 2.

Algorithm 2.2: The Clarke and Wright Savings algorithm

```
1  foreach i ∈ C \ {d} do
2  |   tour_i ← (d, i, d);
3  |   foreach j ∈ C \ {d} do
4  |   |   S ← S ∪ s_ij = c_id + c_dj − c_ij;
5  |   end
6  end
7  sort(S);
8  while ∃ s_ij ∈ S → i ∈ tour_1 ∧ j ∈ tour_2 ∧ tour_1 ≠ tour_2 do
9  |   tour_1 ← tour_1 + tour_2;
10 |   tour_2 ← ∅;
11 end
```

The Savings Algorithm 2.2 presupposes the triangular inequality to be fulfilled. In the beginning the algorithm calculates the savings value for each pair of customers w.l.o.g. i and j: $S_{i,j} = c_{i,d} + c_{d,j} - c_{i,j}$, where $S_{i,j}$ is the saving, $c_{i,j}$ are the traveling costs between customer i and j and d is the depot. In the following loop the algorithm appends the two tours with the largest savings value to one tour. This step is repeated until no two tours can be merged due to side constraints. If there are no constraints that limit the tours, the algorithm calculates a solution with one single vehicle. Therefore the Clarke and Wright Savings algorithm is able to solve TSP instances too.

2.3 Local Search

Local Search (LS) procedures are widely applied on COPs and Constraint Satisfaction Problems (CSP), since they are easy to implement and mostly perform good in matters of execution time and in many cases the solution quality is adequate.

As the name hints, local search algorithms improve the solutions by applying only local changes: i.e. make small changes in the existing solution. In each iteration, the LS algorithm selects the next solution x_{next} from the set of candidate solutions, defined by the so called neighborhood $N(x)$, and sets it as the new solution if its objective value is improved.

The neighborhood depends on the problem and the model of a solution. For example such a neighborhood could be switching the position of two cities, or even a subset of cities, in the tour of the traveling salesman. The size of the neighborhood defines the complexity of the selection step. Switching two cities results in a neighborhood of size $\binom{n}{2} = \frac{n*(n-1)}{2}$. This case is depicted in Figure 2.2. Exploring the whole neighborhood in this example can be done in $\mathcal{O}(n^2)$.

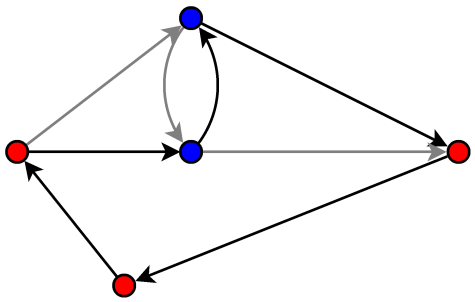

Figure 2.2: Neighborhood for TSP: switch the position of 2 cities

There exist different schemes on how to accept the next solution in the neighborhood: next, random and best improvement. As the names hint, next improvement selects the first, best improvement the best and random improvement a random element from the neighborhood improving upon the current solution. The difference between next and random improvement is the order in which the neighborhood is explored: next improvement explores the neighborhood in a deterministic and random improvement in a random manner. Best improvement means evaluating the complete neighborhood and selecting the solution that optimizes the fitness function the most. Whereas the other selection schemes are able to terminate early and potentially need much less time, this scheme always explores the complete neighborhood and has to be used with care, if the size of the neighborhood grows. From a theoretical point of view they all have the same complexity, which is defined by the size of the neighborhood.

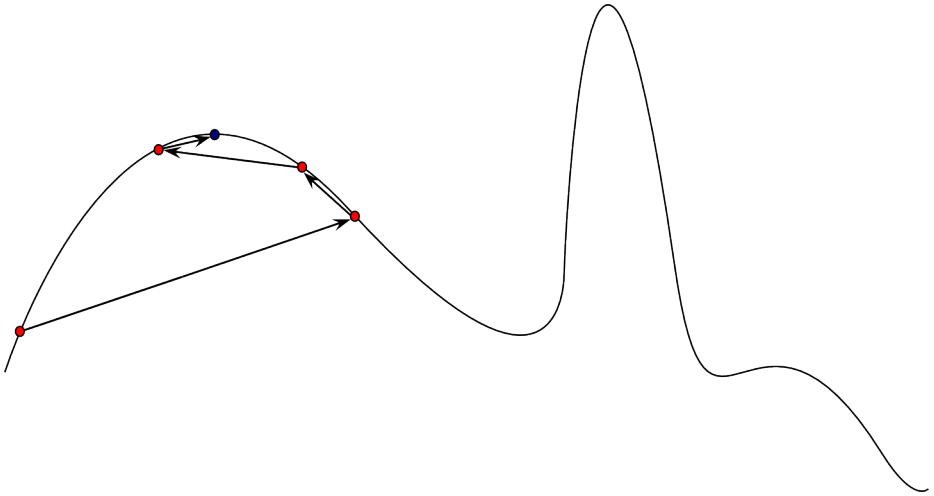

Figure 2.3: A possible series of solutions obtained by the LS algorithm

Algorithm 2.3: Local Search for a minimization problem

```
1 sol ← feasible solution ;
2 repeat
3 |   sol' ∈ Neighborhood (sol);
4 |   if f(sol') ≤ f(sol) then
5 |   |   sol ← sol';
6 |   end
7 until termination criteria holds;
8 return sol;
```

The basic Local Search procedure is shown in Algorithm 2.3. They are often used in combination with other optimization schemes to improve the overall performance, since they converge quickly towards a local optimum. The local optimum may be the global optimum, but in general it is hard to evaluate the quality of the solution obtained by the local search scheme compared to the global, potentially unknown solution.

Definition 5. *A **local optimum** is the best solution within a set of solutions $S' \subset S$. The set S' is defined by a solution and its neighboring solution according to a neighborhood structure. Usually the set S' is a rather small subset of the set of all solutions S.*

Definition 6. *A **global optimum** is the best solution within the set of all solutions S.*

Figure 2.3 shows a possible execution of the local search heuristic. The red dots mark solutions and the arcs the order in which they are generated. The blue dot is the local optimum where the local search procedure terminates. It is obvious that the LS algorithm does not find the global optimum as it converged to a local optimum.

2.4 Particle Swarm Optimization

The particle swarm optimization (PSO) scheme was inspired by the movement of individuals in fish schools and bird flocks that try to reach the best position in the swarm. This optimization scheme was proposed and studied first by Kennedy and Eberhart [34, 20] with the intention to simulate social behavior. PSO algorithms are very robust and can be applied to a large set of optimization problems. Even if the data is noisy, incomplete or changes over time a PSO is able to deal with this inconsistencies.

The PSO metaheuristic starts with a set of P random solutions, called the swarm of particles. Each particle knows its position in the solution space, its velocity and its personal best position $lBest$ during the process of the algorithm. Further all particles know the global best position $gBest$, which is the position of the best particle in history. In each iteration of the algorithm, the velocity of the particles is manipulated in a stochastic manner and the position is updated using the current position and velocity information. The resulting trajectories of the particles depend on the local knowledge of each particle and the global knowledge of the swarm.

The termination criteria of the algorithm can depend on several criteria: the maximum number of iterations, the time limit is reached or the swarm has converged to an optimum, or it oscillates. The PSO, like all other metaheuristic algorithms, converges towards a local optimum that potentially is the optimal value of the objective function. Algorithm 2.4 shows the basic version of the optimization scheme as proposed by Kennedy and Eberhart.

Algorithm 2.4: The PSO metaheuristic

input : A problem instance
output: The particle at the best position

1 *Initialize swarm*;
2 **while** *Termination criteria not met* **do**
3 **for** $i \leftarrow 1$ **to** P **do**
4 $\mathsf{x}_i \leftarrow \mathsf{x}_i + \mathsf{v}_i$;
5 **if** $\texttt{fitness}(\mathsf{x}_i) < \texttt{fitness}(\mathsf{lBest}_i)$ **then**
6 $\mathsf{lBest}_i \leftarrow \mathsf{x}_i$;
7 **end**
8 **if** $\texttt{fitness}(\mathsf{x}_i) < \texttt{fitness}(\mathsf{gBest})$ **then**
9 $\mathsf{gBest} \leftarrow \mathsf{x}_i$;
10 **end**
11 **end**
12 **for** $i \leftarrow 1$ **to** P **do**
13 $\mathsf{v}_i \leftarrow \mathsf{v}_i + \beta * \texttt{rand}() * (\mathsf{x}_i - \mathsf{lBest}_i) + \gamma * \texttt{rand}() * (\mathsf{x}_i - \mathsf{gBest})$;
14 **end**
15 **end**
16 **return** gBest;

The D-dimensional vectors $\mathbf{x}_i = (x_i^1,\ x_i^2,\ \ldots\ x_i^D)$ and $\mathbf{v}_i = (v_i^1,\ v_i^2,\ \ldots\ v_i^D)$ represent the position and the velocity of the i-th particle. The position has to be within the search space, which is defined by the variables domains, and is therefore bounded by $\forall i \in P, d \in D\ x_i^d \in [LB_d, UB_d]$ for each particle i and in each dimension d. UB_d represents the upper bound in the d-th dimension and LB_d the respective lower bound in the continuous domain of the position. The velocity might also be bound in order to avoid particles either missing a promising part of the search space or taking too many iterations to converge towards a local optimum.

The local best position vector $\mathbf{lBest}$ of each particle represents the cognitive component of each particle. The current global best position $gBest$ adds a social component to the current knowledge of the swarm. In contrast to other optimization algorithms, the PSO is influenced by a larger set of different inputs. A genetic algorithm, for instance, usually uses only local knowledge obtained from the genotypes in the crossover and mutation operations.

The velocity of each particle is initialized at random. The position is either obtained from a previously generated solution or is initialized at random, too. In each iteration of the algorithm the position and the velocity are updated using (2.1) and (2.2), respectively:

$$v_i^d(t) \leftarrow v_i^d(t-1) + \beta * r_1 * (lBest^d - x_i^d(t-1)) + \gamma * r_2 * (gBest^d - x_i^d(t-1)) \quad (2.1)$$

$$x_i^d(t) \leftarrow x_i^d(t-1) + v_i^d(t) \quad (2.2)$$

The coefficients β and γ weight the cognitive (local information $\mathbf{lBest}$) and social (global information $gBest$) influence on the velocity of each particle. The random components r_1 and r_2 are introduced to vary the strength of the influence of the cognitive and social components and are generated for each dimension and particle. t represents the current and $t-1$ the previous iteration of the PSO. After each position update the velocity and position have to be checked for feasibility, since it is possible that the particles fly out of the search space or the solutions are invalid.

When the position of all particles in the swarm is updated and bound, the algorithm evaluates the fitness function of each particle and sets the new local and global best positions $\mathbf{lBest}$ and $gBest$. In case of a minimization problem the function to determine the local (2.3) and global (2.4) best particles are the following:

$$lBest_i(t) \leftarrow \begin{cases} x_i(t) & f(x_i(t)) < f(lBest_i(t-1)) \\ lBest_i(t-1) & f(x_i(t)) \geq f(lBest_i(t-1)) \end{cases} \quad (2.3)$$

$$gBest(t) \leftarrow min(lBest_1(t),\ lBest_2(t),\ \ldots\ lBest_d(t)) \quad (2.4)$$

In the original and most common formulation, all particles in the swarm know the same global best position. Newer variants of the PSO define a neighborhood where the global best position is shared among the particles. This results in multiple swarms that may converge to different optimal positions in the solution space.

The PSO and some other heuristic algorithms have a drawback: a potentially early convergence towards a local optimum, where the algorithm is unable to find any improvement in the future. Several extensions to the original PSO were proposed in order to avoid an early convergence. In this work a predator particle was used to simulate hunting the particles that escape if the predator comes too close. This predator particle is able to scare the particles that are stuck in a local optimum out of that part of the solution space. The interested reader is referred to [9] for a more detailed overview.

Discrete PSO algorithms

As mentioned before PSO algorithms operate on a continuous solution space and are not commonly used to solve COPs. Because of the continuous domain of the position and velocity vectors, a mapping from the discrete to the continuous solution space and vice versa has to be defined in order to be able to solve COPs with their discrete domain. Such a mapping can be a simple rounding of the calculated position values in each iteration or more sophisticated functions like recombination operators inspired by genetic algorithms.

The advantage of a PSO over other population based optimization algorithms, like genetic and ant colony optimization algorithms, is the usage of several sources of information: the social component and the knowledge component. Genetic algorithms typically have only a social component and on the other hand ant colony optimization algorithms only the knowledge component, which is the pheromone trail. Further the PSO has not been subject of any research in context of the VRPC.

Kennedy and Eberhart [35] proposed a discrete version of the PSO too: the algorithm uses the same velocity update function (2.1) and the new position update function (2.5) with (2.6) being the sigmoid function.

$$x_i^d(t) \leftarrow f(s_i^d(t) - r) \tag{2.5}$$

$$s_i^d(t) \leftarrow \frac{1}{1 + e^{-v_i^d(t)}} \tag{2.6}$$

The final position of the particle is obtained by applying f, the so called unit step function, on the result of a random number r subtracted from the sigmoid function. The random number r is generated using an uniform distribution. The selection of f directly influences the average velocity needed to set the binary component of the particle. The unit step function f could

be a simple rounding or any other function that transforms a value from the interval $[0 \dots 1]$ into a binary value.

A particle for the VRPC problem for the binary PSO could be the 4 dimensional matrix M, with the number of vehicles in the first, the customer in the second, the demand in the third and the compartment in the forth dimension. If w.l.o.g. the element $e_{v_i,c_j,d_k,comp_l} = 1$ then the i-th vehicle has loaded demand d_k into the compartment l and delivers it to customer j.

Another discrete PSO algorithm is presented in [60]. It operates on domains with more than two distinct values. Similar to the binary PSO above the velocity update function remains the same as in the continuous domain (2.1). The discrete variable values are from the domain $\in \{0, 1, \dots M-1\}$ and the position update function (2.7) is the following, where (2.8) is the sigmoid function:

$$x_i^d(t) \leftarrow round(s_i^d(t) + (M-1) * \sigma * randn(1)) \tag{2.7}$$

$$s_i^d(t) \leftarrow \frac{M}{1 + e^{-v_i^d(t)}} \tag{2.8}$$

The discrete PSO analyzed in this work is the following: the velocity is based on the original PSO formulation with the additional influence of the predator. The velocity vector $\mathbf{v} \in \mathbb{R}^d$ defines the velocity in each dimension and contains the tuple (σ, μ). The position vector $\mathbf{x} \in \mathbb{N}^d$ is obtained by generating a normally distributed random number and applying a simple rounding function. The velocity and position update is done according to (2.9) and (2.10), respectively.

$$\begin{aligned} v_i^d(t) \leftarrow v_i^d(t-1) + \beta * (x_i^d(t-1) - x_{\text{local best}}) \\ + \gamma * (x_i^d(t-1) - x_{\text{global best}}) - \gamma * \Delta \end{aligned} \tag{2.9}$$

$$x_i^d(t) \leftarrow round(randn(v_i^d(t)_\mu, v_i^d(t)_\sigma)) \tag{2.10}$$

The position $\mathbf{x} \in \mathbb{N}^d$ represents a solution and is obtained by choosing a random number using the coefficients from the velocity information. The function Δ returns the distance to the predator.

2.5 Linear Programming

Solving $\mathcal{NP}$-hard optimization problems to optimality can be very time and memory consuming, especially as the size of the problem instances grows. Under the assumption that $\mathcal{P} \neq \mathcal{NP}$ there exists no algorithm that has a guaranteed polynomial time complexity for any problem in $\mathcal{NP}$.

Proven optimal solutions are desirable for all optimization problems, since they provide a minimal lower bound for all other methods and facilitate evaluating the performance of approximate solutions. The classification of problems into complexity classes provides only a worst case analysis of the running time of any algorithm and does not say anything about the average case. Even though solving such problems to optimality is assumed to be very time consuming, practice has shown that there exist several problems where an exact algorithm performs efficiently on a large set of instances.

For example, the TSP has been solved on large instances with more than 10000 cities. In many cases a sophisticated exact algorithm was able to locate the optimal solution to be within a gap that is less than 1 percent from the calculated solution. The three largest instances upon the completion of this work were all solved using CONCORDE [2], the most efficient implementation available.

3. In 2001 the optimal tour through all 15112 communities of Germany was calculated. The resulting total tour has 66000 kilometers length.
2. The optimal tour through all 24978 communities in Sweden with a tour length of approximately 72500 kilometers, solved in 2004.
1. The largest instance of the traveling salesman problem that was solved until now (Sep. 2012) consists of a tour through 85900 locations in a VLSI application that arose in the Bell Laboratories in the late 1980s.

In practice there exist several methods to solve $\mathcal{NP}$-hard problems to optimality. The most prominent member is linear programming (LP), which was studied first by Leonid Kantorovich and George Dantzig, combined with branch & bound. As a member of the military engineering-technical university, Kantorovich was responsible for the socalled road of life to the besieged city of Leningrad. He calculated the optimal distance between cars on ice, depending on the thickness of the ice and the air temperature such that the ice does not crack and the goods that are delivered to Leningrad are maximized. This is known as the first LP model. Independent of Kantorovich, Dantzig formalized linear programming a few years later and published the simplex method in 1947. In the same year John von Neumann developed the theory of duality: for each minimization problem there exists a dual maximization problem.

All LP's are formulated using linear relationships between the variables and can be written in the following basic form:

$$\text{max. } \mathbf{c}^T\mathbf{x} \tag{2.11}$$
$$\text{s.t. } \mathbf{A}\mathbf{x} \leq \mathbf{b} \tag{2.12}$$
$$\mathbf{x} \in \mathbb{R}^n \tag{2.13}$$

The socalled objective function (2.11) defines the formula to be minimized or maximized. The linear constraints (2.12) define the relation between the variables and (2.13) the domain of the variables. The domain of the variables classifies the linear program:

Linear Program(LP): $\mathbf{x} \in \mathbb{R}^n$

Integer Linear Program(ILP): $\mathbf{x} \in \mathbb{Z}^n$

Binary Integer Liner Program(BIP): $\mathbf{x} \in \{0,1\}^n$

Mixed Integer Liner Program(MIP): some $\mathbf{x}_1 \subset \mathbf{x} \in \mathbb{R}^n$, some other $\mathbf{x}_2 \subset \mathbf{x} \in \mathbb{Z}^n$ and a third subset $\mathbf{x}_3 \subset \mathbf{x} \in \{0,1\}^n$

Duality and Relaxation

According to the results of von Neumann, each problem can be converted into a corresponding dual problem. The dual problem of a minimization problem is a maximization problem and vice versa. Converting the primal LP (2.11) - (2.13) into its dual formulation, is the following minimization problem:

$$\text{min. } \mathbf{b}^T\mathbf{y} \tag{2.14}$$
$$\text{s.t. } \mathbf{A}^T\mathbf{y} \leq \mathbf{c} \tag{2.15}$$
$$\mathbf{y} \in \mathbb{R}^n \tag{2.16}$$

The dual objective function (2.14), the dual constraints (2.15) and the domain of the dual variables (2.16) are the basic dual form. When the dual problem is converted into its dual problem, the resulting problem is equivalent to the primal problem.

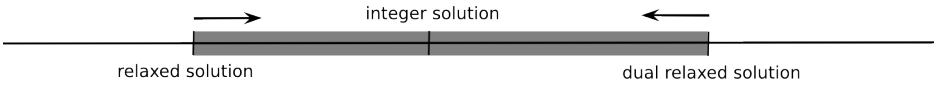

Figure 2.4: The bounds of the ILP

Linear Programs can be solved very efficient by using the Simplex algorithm or the interior point method. The other models are in general $\mathcal{NP}$-hard problems and are solved using advanced methods that use the simplex algorithm in order to solve a relaxed version of the problem. By relaxing the integer condition on the variables, the domain becomes continuous: e.g. a constraint $x \in \mathbb{N}$ becomes $x \in \mathbb{R}$. However the resulting solution might not be integral, i.e. there exists at least one variable having a fractional value.

An important and very useful information that can be obtained from these solutions are the bounds: for a minimization problem the solution of the relaxed problem is a lower bound for the original problem. The solution of the original problem can be at most as good as the solution of the relaxed problem. Already computed feasible solutions for the original problem and the solution of the dual problem provide an upper bound for the original problem. The optimal integer solution is somewhere between the lower and upper bounds that are obtained from solving the relaxed primal and dual problem. See Figure 2.4.

Branch and Bound

Branch and Bound (B&B) is inspired from the *divide and conquer* principle that is widely known for search problems: divide the problem into two or more sub-problems (branches) and solve them independently. Applying this rule to the generated subproblems, too, forms the B&B search tree. Such a subproblem can be generated by adding constraints to the branches of the search tree. A simple constraint for binary problems is shown in Figure 2.5: by fixing w.l.o.g. variable $x_i = 0$ in the first subtree and $x_i = 1$ in the second subtree. The problem in both subtrees is simpler than the whole problem, since the binary variable x becomes a constant in both subtrees.

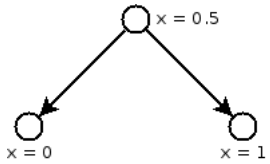

Figure 2.5: Binding the variable x to binary values

If a feasible and improved solution is found, it becomes a global upper bound. If it can be shown that all solutions in a branch are worse than this upper bound, the branch can be pruned from the search tree. B&B basically is a systematic enumeration of all candidate solutions, where a hopefully huge subset of candidate solutions can be discarded.

When solving ILPs, B&B is used to obtain an integer solution from the relaxed solution by creating branches with additional constraints on the fractional variables. For instance and w.l.o.g. if variable x_i in a relaxed ILP has the value 17.326, two subproblems with the constraints $x_i \leq 17$ and $x_i \geq 18$ are created and solved. After solving, each node is checked and possibly pruned.

The optimal solution is found if the primal solution value is equal to the dual solution value and is integral.

A branch in a minimization formulation can be pruned if:

- The Solution is infeasible
- The lower bound is larger than the global upper bound
- Optimal solution is found

The performance of a B&B algorithm highly depends on the branching and variable selection rules. In many cases an empirical analysis shows significant differences in the size of the tree and the total running time for different settings.

Column Generation

The basic idea behind Column Generation (CG) is to focus on an interesting subset of the search space and avoid to explore a larger subset, which is never part of the solution. In addition, a subproblem is used to identify new variables that might improve the current solution. The result is a narrow problem formulation that can be solved in shorter time. When using a CG approach, the original problem is reformulated and split into two problems, the socalled Master Problem (MP) and the Pricing Problem (PP). The unrestricted MP contains all possible variables and equals a complete enumeration, which is very inefficient. Further it would supersede the need to solve the pricing problem. Since this is not practical only a small subset of variables, the Restricted Master Problem (RMP), is initially generated and the PP is used to identify new variables. The Dantzig-Wolfe decomposition provides the formal basics for the column generation approach [19].

The PP highly depends on the formulation of the RMP and the original problem itself. The goal of splitting the original problem is to obtain one or even more problems with a special structure, where sophisticated algorithms are able to solve them efficiently. Even though it is clear that solving the master and pricing problems implies solving at least one problem in $\mathcal{NP}$, there might exist algorithms that are able to solve some of the problems relatively good in practice. E.g. the Knapsack Problem, a weakly $\mathcal{NP}$ problem, is easily tractable using a dynamic programming approach.

As the variables are generated every time after the LP solving process finishes, this approach is also called delayed column generation. The solving process in any column generation approach works as follows: an LP solver, such as the simplex algorithm, solves the RMP with the current available variables and generates the dual variable values for each constraint. When the solver terminates the dual variable values are used in the pricing problem to identify new variables with negative costs. When the PP finds such variables they are added to the RMP and the solving process starts again with the extended master problem formulation. For a MIP formulation the CG process needs to be embedded into a B&B solver

whose decisions also influence the PP. If the pricer does not find any variables the search in the current node in the B&B tree, the column generation process is finished for this particular node and the B&B solver continues.

2.6 Constraint Programming

Constraint Programming (CP) [51] is a programming paradigm that is under growing development since the mid 80's and is an enhancement of logic programming. A CP model is given in form of relations, called the constraints, over binary, integer or set variables. These constraints can have different shapes: linear constraints, n-ary relations, arithmetic constraints and more complex constraints like bin packing, sorting, scheduling constraints, just to name some of them.

The solver tries to assign values to the variables such that all constraints are satisfied. Basically, solving a CP model implies the methods propagation and search using methods like B&B and Depth First Search (DFS). Propagation means removing all values from the domain of a variable that always result in a violated constraint when the variable would be assigned this specific value. The value is removed only if the other variables can assume any value from their domain and there always exists a violated constraint in the model. Each propagation operation is performed by a constraint specific propagator that uses sophisticated methods for the special problem.

For instance the propagator for binary relations like $x_1 < x_2$ deletes all values from the domain of x_1 that are larger or equal than the maximum of x_2 and all values from x_2 that are smaller than the minimum of x_1. A second example is shown in Figure 2.6, showing a combined application of the search and propagation methods on the popular puzzle Sudoku.

The search procedure has to be used if no propagator is able to restrict any domain further and at least one variable has a domain with more than one values. Searching in CP solvers is realized through search trees that restrict the domain of a variable for the left subtree and the exact opposite decision is made for the right subtree. DFS is mainly used for pure assignment problems whereas B&B is normally used for optimization problems.

Constraint programming is used in this work to solve the packing problem, as CP solvers are designed to find assignments efficiently, such as the packing problem for the instances with fixed compartments.

Sudoku, a very popular logic puzzle, is meant to be solved by propagation only even though applying a decision similar to B&B helps to solve the puzzle: if there are two distinct values possible at a node the first one is tried out, if during the further solving process a field having an empty domain is encountered the decision was wrong and the other value is correct.

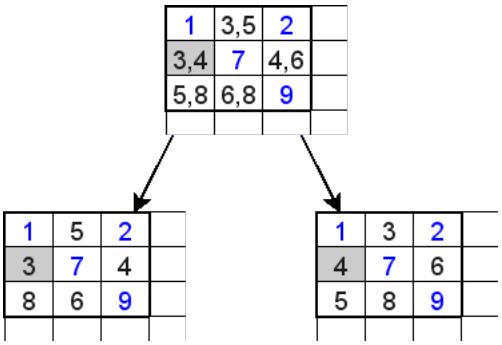

Figure 2.6: Search and Propagation in a part of the Sudoku puzzle. The values in the fields are the respective domains with one or two values. First the search assigns 3 in the left subtree and 4 in the right subtree and then each part of the puzzle can be solved using propagation only.

CHAPTER 3

Algorithms for the Vehicle Routing Problem with Compartments

The two major classes of algorithms implemented and analyzed in this work are heuristic approaches and exact algorithms. The first part of this chapter focuses on the two heuristic approaches, first the GRASP which is used as construction algorithm to obtain the first solutions and second the PSO and LS algorithms that improve these solutions further. The LS scheme optimizes the particles of the PSO after each position update operation to yield a quicker termination and better solution quality.

The second part of this chapter describes the models and algorithms that are used to obtain proven optimal solutions. The column generation algorithm depends on initial variables that are generated by the heuristic algorithms and used to create the RMP. During the solving process, the pricer iteratively generates new variables that are subsequently added to the master problem, which is then resolved.

3.1 Heuristic Approaches

This section provides a detailed description of the two heuristic solution approaches. The first is the GRASP which serves as a provider of an initial solution for other algorithms. The second part describes the heuristic algorithm based on swarm intelligence.

Modeling a Solution

In order to get an efficient program, the organization of the data in the main memory is very important. The goal is to achieve a small memory footprint and a fast way to access the data stored in it. As every algorithm needs special information, the representation of the solutions is slightly different for each algorithm. The basic solution of the heuristic algorithms

is modeled as a two dimensional matrix to represent the tour T and a second one to model the packing P. Figure 3.1 shows both matrices with two vehicles that deliver some goods to customers. The first dimension in both matrices represents the vehicles. The second dimension in the tour matrix T represents the customers and holds the index of the next customer in the tour. The second dimension of the packing matrix P represents the compartments of each vehicle and holds a list of orders, which are loaded into the corresponding vehicle and compartment.

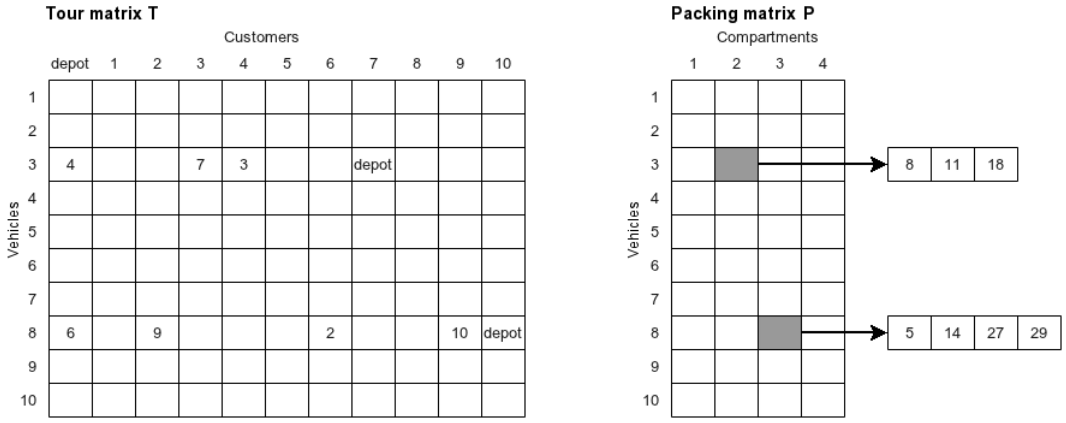

Figure 3.1: A partial solution with vehicle 3 and 8 delivering some orders

As the vehicles are uniform by definition, such a solution approach is possible without any further restrictions on the elements in P. In case the vehicles are not uniform, all compartments that do not exist in the corresponding vehicles have to be constrained to be empty.

In the description of the algorithms set operators are used to modify the packing and tour matrices T and P. This simplifies the pseudo code by abstracting from the data structures and relieves the comprehension. E.g. adding a tour to the tour matrix T is denoted as $T \leftarrow T \cup (l_d, l_1, \ldots, l_d)$.

A GRASP Algorithm for the VRPC

Based on the idea of the GRASP metaheuristic the Clarke and Wright Savings algorithm [10] is modified in order to construct initial solutions with a modifiable grade of diversity. The candidate list holds all possible pairs of tours that can be merged. Further the savings values along the arcs, which connect two different tours, directly provide the metric to order the candidate list.

The GRASP algorithm uses the savings matrix S in order to determine which two tours are connected to a single tour. Each saving w.l.o.g. s_{ij} in the savings matrix S holds the amount of routing costs that can be saved when merging two tours. The first part of the tour starts from the depot and ends at node i and the second begins at node j and ends at the depot.

As the savings do not change over the execution of the algorithm, they are calculated once in the initialization step.

Algorithm 3.1: Initializing the Savings GRASP

```
1  T ← ∅;
2  P ← ∅;
3  foreach l_i ≠ l_j ∈ L \ {d} do
4  |  S_ij = distance(l_i, l_d) + distance(l_d, l_j) − distance(l_i, l_j);
5  end
6  foreach _i ∈ O do
7  |  T ← T ∪ (l_d, l_customer(i), l_d);
8  |  P ← P ∪ pack(i);
9  end
10 return T, P, S;
```

In the first loop Algorithm 3.1 calculates the savings matrix S. In the second loop the tour and packing matrices are initialized by generating a tour for each individual order.

The GRASP metaheuristic needs a list of candidates in order to select a random element from a set of good extensions to the current partial solution. The Savings algorithm is modified in order to fit the needs of the GRASP metaheuristic. Instead of selecting the best saving to merge two tours, the algorithm generates a list of all possible combinations from the current tours.

Algorithm 3.2: Generating the candidate list.

```
   input: The tour, packing and savings matrices T, P and S
1  cl ← ∅;
2  foreach i ≠ j ∈ [1 . . . |T|] do
3  |  if pack(P_i, P_j) ≠ ∅ then
4  |  |  s ← max(S_T_i.b,T_j.b, S_T_i.b,T_j.e, S_T_i.e,T_j.b, S_T_i.e,T_j.e);
5  |  |  cl ← cl ∪ (i, j, s);
6  |  end
7  end
8  return cl;
```

During the main loop of the Savings GRASP Algorithm 3.3 each pair of tours is evaluated due to their savings and checked if the orders in both vehicles fit into a single vehicle. Algorithm 3.2 shows how the CL is constructed. Each pair of tours is checked if the packing

schemes can be merged and the best saving among the four possible connections of both tours is determined. Note: $T_{i.b}$ denotes the index of the first customer in the i-th tour and $S_{T_{i.b},T_{j.e}}$ denotes the saving by appending the end of tour j at the beginning of tour i.

Algorithm 3.3: The Savings GRASP for the VRPC

```
1  T, P, S ← init_savings();
2  cl ← generate_cl(T, P, S);
3  while cl ≠ ∅ do
4  |   sort(cl);
5  |   c ← rand(get_rcl(cl, α ∗ |cl|));
6  |   if c_3 = S_{T_{c_1.e}, T_{c_2.e}} then
7  |   |   T_{c_2} ← reverse(T_{c_2});
8  |   else if c_3 = S_{T_{c_1.b}, T_{c_2.b}} then
9  |   |   T_{c_1} ← reverse(T_{c_1});
10 |   else if c_3 = S_{T_{c_1.b}, T_{c_2.e}} then
11 |   |   T_{c_1} ← reverse(T_{c_1});
12 |   |   T_{c_2} ← reverse(T_{c_2});
13 |   end
14 |   T_{c_1} ← (T_{c_1}, T_{c_2});
15 |   P_{c_1} ← pack(P_{c_1}, P_{c_2});
16 |   T ← T \ {T_{c_2}};
17 |   P ← P \ {P_{c_2}};
18 |   cl ← generate_cl(T, P, S);
19 end
20 if |T| > |V| then
21 |   T, P ← repair(T, P);
22 end
23 return T, P;
```

The packing function is called for every pair of tours and every time the current solution is improved by connecting two individual tours. It takes first the packing scheme P_{c_1} and iteratively calls the online packing function, shown in Algorithm 3.4, for each order in the second packing scheme P_{c_2}. In order to save computation time an additional data structure, which is omitted in Algorithms 3.1, 3.2 and 3.3, holds the information which two tours can be merged or whether the packing was not calculated yet.

After the algorithm selected the candidate to extend the solution it modifies both tours such that the vehicle drives along the direction of the selected saving. This implies that one or both tours may be reversed. Then the second tour is appended at the end of the first tour. The packing is calculated using a first fit heuristic which is depicted in Algorithm 3.4.

After no two tours can be merged, the Savings GRASP is finished. It may happen that the number of generated tours is larger than the number of available vehicles and the solution has to be modified. The repairing function takes the tours with the least number of customers, that cannot be assigned to a vehicle, and iteratively tries to load the orders into other tours regardless if the costs increase. The repairing function always selects the tour that causes the least cost increment.

Algorithm 3.4: The online packing function

input: The demand d that has to be packed into any compartment of vehicle v

```
1  foreach comp ∈ compartments(v) do
2      if flexible compartments then
3          if compatible(d, comp) ∧ load(v) + d.amount ≤ capacity(v)∧
4             load(v, comp) + d.amount ≤ capacity(comp) then
5              L(v, cust) ← L(v, cust) ∪ d;
6              return true;
7          end
8      else
9          foreach d2 ∈ L(v, comp) do
10             if compatible(d, d2)∧
11                load(v, comp) + d.amount ≤ capacity(comp) then
12                 L(v, cust) ← L(v, cust) ∪ d;
13                 return true;
14             end
15         end
16     end
17 end
18 return false;
```

Algorithm 3.4 is a so-called online algorithm, since it does not change any existing partial packing scheme but tries to add orders to it without changing the order in which the orders should be packed.

Definition 7. *An **online algorithm** processes the input immediately without being able to get information about the future input and without changing an already processed output. The online algorithm must process the data as it is passed and does not have the possibility to wait for a later input.*

This packing algorithm is used in all heuristic approaches to load the orders and in the exact algorithm in the first stage when the packing scheme is calculated. This algorithm is referenced several times in the following sections as the *online packing algorithm*.

A discrete PSO for the VRPC

As mentioned in the introduction in Chapter 2.4, the PSO uses two types of information to enhance the solutions: the social component, which is the particle at the global best position, and the cognitive component, the best position each particle has been so far. In the next chapters the discrete PSO will be denoted as PSO for simplicity and to facilitate the understanding and is depicted int Algorithm 3.5.

Algorithm 3.5: The discrete PSO

input: The problem instance and the initial particles

```
1  solution ← ∅;
2  gBest ← ∅;
3  foreach particle do
4  |  particle ← calc_initial();
5  end
6  while Termination criteria is not met do
7  |  foreach particle do
8  |  |  update_position(particle);
9  |  |  repair(particle);
10 |  |  if quarter of time or iteration limit exceeded then
11 |  |  |  optimize(particle);
12 |  |  end
13 |  |  if particle.fitness < gBest.fitness ∨ gBest = ∅ then
14 |  |  |  gBest ← particle;
15 |  |  |  if gBest.fitness < solution.fitness ∨ solution = ∅ then
16 |  |  |  |  solution ← gBest;
17 |  |  |  end
18 |  |  end
19 |  end
20 |  update_predator();
21 |  foreach particle do
22 |  |  update_velocity(particle);
23 |  end
24 end
25 return solution;
```

The algorithm first updates the position of each particle according to the current velocity. Since a position update makes the solution unfeasible with a very high probability, the particles need to be repaired. The optimization function is called only in the last three quarters of the total running time or iterations of the algorithm. If no time or iteration limit is defined the algorithm calls the optimization function in every iteration after a default number of 50

iterations is exceeded. This gives the particles the ability to better explore the search space in the first quarter and forces them towards a local optimum in the last three quarters.

After updating, repairing and optimizing the particles, the algorithm checks if a new global best particle and a best solution can be found in the swarm. The PSO keeps a copy of the best solution that was found during execution of the algorithm. In the second part of the algorithm the predator updates its position depending on the distance to the other particles and their fitness function value. Since the position of the predator depends on the positions of the particles and the velocities of the particles are influenced by the predator's position, the main loop of the original PSO has to be divided into two loops: the particles position update loop and the velocity update loop.

When the termination criteria are met, the best solution is returned. The termination criteria can be a simple time or iteration limit or even more complex such as the number of successive iterations the algorithm found no improvement. In order to be able to compare the results of the algorithm with different datasets, a simple time limit is used as termination criteria. Since the PSO also provides initial solutions for the exact approach, it supports termination after a given iteration limit, too.

In an early development stage the PSO also operated on the packing scheme, but the results were disappointing. The influence of the tour on the loading scheme and vice versa is crucial and prevented the algorithm from finding good solutions. A simple change in the tour has a huge impact on the loading scheme, since the loading values obtained so far become invalid. In the opposite case, when a demand is loaded into a different vehicle, the tour of two vehicles changes.

Due to this strong dependency the PSO has to concentrate either on the tour or the packing. The respective other component is calculated using a simple construction heuristic. When the PSO focuses on the tour the packing is calculated using the online packing Algorithm 3.4. In the opposite case, the PSO assigns the demands to vehicles and compartments and the tour is calculated with a local search procedure using the tour optimization neighborhoods presented later in this Section starting on page 40. An empirical analysis has shown early that the first approach gives better results. As consequence the PSO operating on the packing matrix P is considered rarely in the next sections in this work. Further most of the algorithms are equal or very similar for both PSO variants. In the results Chapter 4 the performance of both algorithms are compared.

The Particles

Each particle is a refinement of the solution the GRASP returns, which is described in Section 3.1. The position of a particle is the current solution with the packing matrix P and the tour matrix T. These matrices are the same as the ones for the GRASP. They are presented in Section 3.1. In addition, the velocity is modeled as a two dimensional matrix V, where each element is a tuple (μ, σ), the coefficients of a normal distributed random number generator. The initial swarm is generated by the GRASP algorithm.

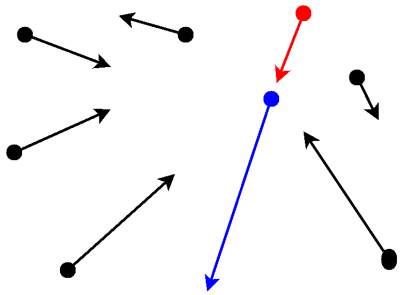

Figure 3.2: The predator particle (red) hunts the current best particle (blue) which escapes from the potentially local optimum. The velocity of the particles near the predator highly depends on the distance between them.

In order to avoid an early convergence a predator particle is used. The predator itself is a particle with a different velocity update and without repair and optimize functions. As the predator only avoids an early convergence, the solution it represents need not be feasible at all. The predator particle aims towards the position of the current best particle. Depending on the distance between the particles and the predator every particle in the swarm reacts differently. Figure 3.2 shows the predator, the red particle, which aims towards the current global best particle, the blue one, and other particles that aim towards the current best position if the distance to the predator is large enough, else the predator scares them away.

Evaluating the Performance of the Particles

The *objective function* returns the total routing costs, which are the total accumulated costs along the route each vehicle drives. Since the solutions represented by the particles contain invalid parts with a very high probability, the particles have to be evaluated using an extended measure: the *fitness function*. It returns a measure depending on the total routing costs given by the *objective function* and additional penalty costs for each invalid part of the solution, which are evaluated by the *error function*.

In order to enhance the exploration of the solution space, especially during the first iterations, the PSO should be able to accept particles as the new global best particle even if they are invalid, but might be better than the current best particle. This can be achieved by adding penalty costs that depend on the severity of the invalid parts of the solution and the progress of the algorithm, that is either the number of the current iteration or the time elapsed. If the

solution, the particle represents, is invalid, weighted penalty costs for each infeasible tour or packing scheme are added to the objective function value. This forces the particles towards feasible solutions. Especially in the last iterations it is crucial that the algorithm produces valid solutions.

Each particle p that was generated by one of the two different PSO algorithms, either operating on the tour or the packing, can be infeasible due to the following reasons:

- **Invalid tours**: Due to the construction of the particles, as described in the position update function on page 37, in most of the cases the tour is feasible, but it may happen that the tour contains cycles. $t(p) = \sum(invalid\ tours\ in\ p)$.
- **Undelivered orders**: All orders that are demanded by the customers, but not delivered, are summed up to: $u(p) = \sum(undelivered\ orders\ in\ p)$.
- **Violated incompatibility constraints**: For each demand that is loaded into an incompatible compartment penalty costs are added. Further if two different demands are loaded into the same compartment, but are not allowed to be in the same compartment, are accumulated to: $i(p) = \sum(incompatibilities\ violated\ in\ p)$.
- **Overloaded compartments and vehicles**: The demands for each compartment and vehicle, which are loaded beyond the respective limit, are summed up to: $o(p) = \sum(overloaded\ vehicles\ in\ p) + \sum(overloaded\ compartments\ in\ p)$.

The first two cases can only happen when the PSO optimizes the tour of the vehicles, while the last two cases are impossible due to the construction process of the packing scheme. In contrast, the first two cases are impossible to happen in case of the PSO operating on the packing, since a feasible tour will be constructed in any case. Further the PSO that optimizes the packing scheme assigns every demand to a vehicle, but may overload and generate incompatible packing schemes, which is impossible for the first PSO.

The resulting *fitness function* (3.1) and *error function* (3.2) for each particle p in the swarm at iteration it are the following:

$$fitness(p, it) \leftarrow objective(p) * (1 + \epsilon * it * error(p)) \tag{3.1}$$

$$error(p) \leftarrow \alpha * t(p) + \beta * u(p) + \gamma * i(p) + \delta * o(p) \tag{3.2}$$

All penalty and routing costs are summed up and weighted with the respective weights $\alpha, \beta, \gamma, \delta$ and ϵ. The increasing *error function* value in later iterations causes the *fitness function* to return larger values and the PSO to accept only feasible solutions.

Updating the Velocity

The velocity update is the most crucial part of a PSO algorithm and difficult to fine tune, since it determines how fast the particles fly towards the promising regions of the search space. In the worst case they fly across the local optimum values and never actually reach them. Contrary, the particles might be too slow and the algorithm takes too long to converge towards a local optimum. In general parameter tuning is always an empirical trial and error approach and the results highly depend on the instances used to tune the parameters, which is not desirable.

An update of the velocity matrix V for every particle is done by updating the velocity of each element. The following equation expresses the velocity update for the element that depicts vehicle i and customer c:

$$\begin{aligned} V(i,c).\mu \leftarrow & V(i,c).\mu + \alpha * (T_{lBest}(i,c).\mu - T(i,c)) \\ & + \beta * (T_{gBest}(i,c).\mu - T(i,c)) \\ & - \gamma * \Delta * (T_{pred}(i,c) - T(i,c)) \end{aligned} \tag{3.3}$$

The component of the velocity update function processes the influence of the local information, the current velocity and the local best position. The second part adds the attraction of the global best position. The third part of the velocity update function is an adaptive component that "scares" the particles depending on the distance Δ between them and the position of the predator particle. The intensity of the influence of the predator on the other particles depends on Δ and is calculated as follows:

$$\Delta \leftarrow \frac{1}{\sum_{i \in T} \sum_{c \in T} (T(i,c) - T_{pred}(i,c))^2} \tag{3.4}$$

Where $\sum_{i \in T}$ depicts all vehicles and $\sum_{c \in T}$ all compartments in the tour matrix T. The value of Δ is almost 0 for all particles that are far away from the predator and increases as the particles get closer to the predator particle. The constants α, β and γ control the respective influence of the local and global information as well as the scaring factor. Especially the value of γ has to be chosen with care, as the algorithm can not produce good results with particles that do not explore a promising region of the search space for a minimal number of iterations, i.e. they are chased away too soon.

Another factor that controls the velocity is the standard deviation σ. It has no direct influence on μ but it highly influences the results of the position update method and indirectly also the velocity. By adapting the value of σ the algorithm reacts on the quality of the current part of the search space the particle explores. If a particle has spent a lot of iterations in a poor region the algorithm increases the difference between two successive positions by

increasing σ. The standard deviation σ is therefore the random and adaptive component of the PSO, as the quality of the current part of the search space also depends on the loading grade of the vehicles. A solution is considered good if the vehicles are almost fully loaded.

$$V(i,c).\sigma \leftarrow \begin{cases} \frac{V(i,c).\sigma}{\varphi} & \text{loading grade}(i) \ \geq \phi \\ V(i,c).\sigma & \phi > \text{loading grade}(i) \ \geq \psi \\ V(i,c).\sigma + \omega & \psi > \text{loading grade}(i) \end{cases} \tag{3.5}$$

The value of σ is updated equally for each vehicle v and is the same for each element in the row of the velocity matrix V that corresponds to vehicle v. The random component σ is modified in a three conditional manner, depending on the loading grade: if the loading grade of v is less than a first threshold ψ, σ is incremented in an additive manner. If the loading grade is superior than a second threshold ϕ it will be decremented in a multiplicative manner. If the loading grade is between the two values σ is not changed. This additive increase / multiplicative decrease approach should allow the algorithm to react quicker on a promising part of the search space and cause the particles to explore it better.

Position update

Updating the position of the particles means generating a random number using the μ and σ values obtained from the velocity matrix V. Each element of the tour matrix is calculated as follows:

$$T(i,c) \leftarrow randn(V(i,c).\mu, V(i,c).\sigma) \tag{3.6}$$

If this position update function is applied on any element in the matrix the resulting tours are invalid in almost any case. The consequence would be a high repairing effort having a high impact on the running time of the PSO. Early tests have shown that it does not make sense to follow this solution approach further. Instead, when updating the position of a particle the algorithm focuses on generating valid tours by applying the position update function to a subset of the elements in T only. All other elements that are not updated preserve their old values. Algorithm 3.6 depicts how the tours are generated.

This algorithm assures that each tour starts and ends at the depot, but it does not prevent sub-tours. In this case the corresponding solution is invalid and the penalty costs of the particle increase.

Algorithm 3.6: Updating the tour of vehicle v

input: The particles velocity and tour matrices V, T and vehicle v

```
next ← randn(V(v, depot).μ, V(v, depot).σ);
T(v, depot) ← next;
current ← next;
while next ≠ depot do
    next ← randn(V(v, current).μ, V(v, current).σ);
    T(v, current) ← next;
    current ← next;
end
return T;
```

The elements in the tour matrix T that were not updated, may form an alternative partial route of vehicle v that potentially leads to an improved fitness function value in future iterations of the PSO. When the algorithm selects a different customer in two successive iterations, at least a subset of the successive tour is different. From a certain point in the tour, the vehicle may also take a completely different path back to the depot. As the number of iterations grows the preserved values of μ and σ along this alternative path potentially lead towards improved solutions.

The resulting complete tour of each vehicle is calculated by following the entries in the tour matrix T. Figure 3.3 illustrates the resulting tour of vehicle 4 and an alternative partial path. The next section discusses how the packing scheme for each vehicle is retrieved.

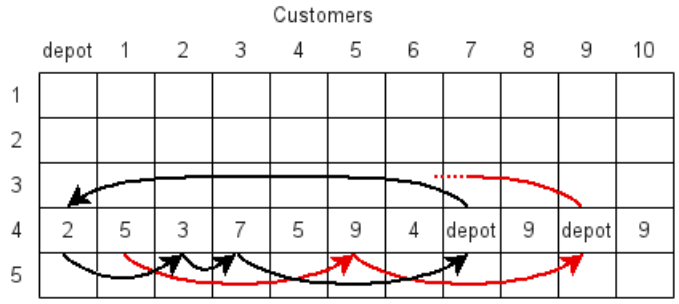

Figure 3.3: The tour matrix T for a vehicle highlighting the actual route of vehicle 4 and an alternative partial route.

Calculating the Packing Matrix

The loading matrix is calculated after the PSO has updated the tour and velocity matrices. The algorithm aims to pack as many orders as possible in the tour of each vehicle in a first fit manner. If no demand of a customer can be packed into a vehicle, this customer is removed from the tour of the corresponding vehicle. As a consequence, if the vehicle is full, the remaining customers in the tour are removed and the vehicle returns to the depot. Algorithm 3.7 shows this procedure.

Algorithm 3.7: Calculating the packing matrix P

input: The particles velocity and tour matrices V and T

```
1  foreach vehicle do
2      curr_cust ← T(vehicle, depot);
3      last ← depot;
4      while curr_cust ≠ depot do
5          skip ← true;
6          foreach o.cust ∉ P do
7              if pack(vehicle, o) then
8                  skip ← false;
9              end
10         end
11         if skip = true then
12             T(vehicle, last) ← T(vehicle, curr_cust);
13         else
14             last ← curr_cust;
15         end
16         curr_cust ← T(vehicle, curr_cust);
17     end
18 end
19 return P;
```

Repairing the Solutions

After calculating the packing matrix P the updated particles may be infeasible due to two reasons: it is either possible that no vehicle reaches some customer or the customer orders cannot be loaded into the vehicles visiting it. The *repair function* searches such orders and tries to load them into vehicles using two different strategies. First the algorithm tries to find the vehicle that can load at least one of these with the lowest additional costs. If no such vehicle can be found, the undelivered demands are loaded into a yet unassigned vehicle that delivers them. Algorithm 3.8 depicts the *repair function*.

Algorithm 3.8: Repairing a particle

input: A particle

```
1  foreach o ∉ P do
2      costs ← ∞;
3      best ← −1;
4      foreach v : P(v) ≠ ∅ do
5          if fits_in(v, o) ∧ costs ≤ additional_costs(v, o.cust) then
6              best = v;
7              costs ← additional_costs(v, o.cust);
8          end
9      end
10     if best ≠ −1 then
11         pack(best, o);
12     else
13         v ← find_empty_vehicle(P);
14         if v ≠ −1 then
15             pack(v, o);
16         end
17     end
18 end
```

In the outer loop the function searches for orders that are not packed yet. In the inner loop the algorithm iterates through all vehicles that have loaded something and checks if the order can be packed into the vehicle. The algorithm selects the vehicle where the insertion generates the least additional costs. If no such vehicle is found the algorithm selects a new vehicle that delivers the order. If no such vehicle can be found the solution the particle represents is invalid and penalty costs increase the fitness function value.

The function `fits_in(v,d)` checks if the order o fits into any compartment of vehicle v and is very similar to the online `pack` function. The only difference is that the order is not added to the packing matrix P. The function `additional_costs(v,o.cust)` calculates the additional costs that result from adding the customer that ordered o to the tour of vehicle v. The function iterates through the tour and searches the cheapest point to insert the customer. The function `find_empty_vehicle` returns an empty vehicle if there exists one, else -1 is returned. As the problem defines an upper bound on the vehicles that are available it is possible that the order cannot be assigned to any vehicle.

Optimizing the Particles

After executing the *repair function* it is possible that vehicles visit customers but deliver nothing to them. This generates additional costs and these customers can safely be removed from the tour. Further, analogously to the idea of a memetic algorithm, an additional,

fast optimization function is executed in each iteration. The local search heuristic tries to improve each particle by searching in appropriately defined neighborhoods for better solutions. These neighborhoods are described in the next sections and are executed in the order as they appear.

Optimizing the tour

The first local search neighborhood performs only changes in the tour matrix T, i.e. it changes the order in which the customers appear in the tour of the vehicles. The algorithm distinguishes between two cases, depending on the number of customers in the tour of each vehicle. If the tour contains less than five customers the algorithm tries all possible moves and selects the cheapest tour. As this conforms to a complete enumeration this can only be applied to a very limited set of customers, since the size of all combinations increases exponentially with the total number of customers in the tour. The enumeration algorithm keeps track of the accumulated tour costs and is able to skip a subset of combinations that exceeds the lower bound, which is the cheapest tour calculated so far. If a partial tour is more expensive than an already computed tour, all possibilities to complete it can be skipped safely. As a consequence the algorithm checks at most 24 distinct cases which can be done in a very short time. The reason for this lies in the third neighborhood, the so called **3-opt**, see Figure 3.6, where the implementation needs at least five customers in the tour to operate correctly.

In the opposite case, with more than five customers in the tour of the vehicle, the algorithm tries to switch the position of customers or a subset of the customers in the tour using a local search method and accepts the first improved solution.

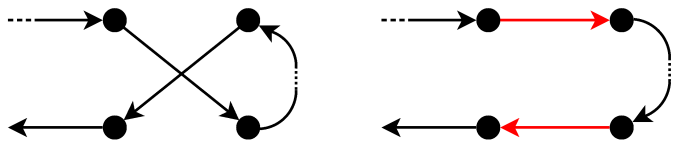

Figure 3.4: A 2-opt move

The neighborhood is known as **2-opt** and switches the position of two customers in the tour. The goal of this neighborhood is to find crossroads in the tour and to reorder the customers in such a way that the tour does not cross over itself. Originally this idea was proposed by Croes in 1958 [11] as a heuristic to improve the tour of the traveling salesman. Figure 3.4 shows the idea of this move. Lin and Kernighan improved this idea further [36] to the `lin-kernighan` TSP improvement heuristic.

The second method that changes the position of customers in the tour is called **2.5-opt** and is depicted in Figure 3.5. For each customer, this neighborhood tries to find another position in the same tour such that the total routing costs of the vehicle decrease. The third and last neighborhood is called **3-opt**, which deletes the link between three customers and reconnects them to a feasible tour in a different order. If the resulting routing costs are less,

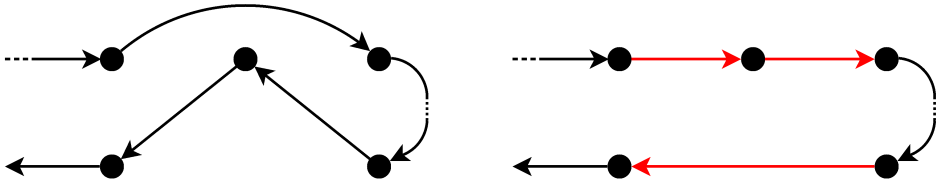

Figure 3.5: A 2.5-opt move

the new route is accepted and the Local Search method continues the search with the next vehicle. The 2-opt and 3-opt neighborhoods are conceptually similar, except that two or three links in the tour are deleted and reconnected differently. Figure 3.6 shows the idea of the 3-opt neighborhood. The tour in this example could also be improved by several 2-opt moves, too, as is a rather simple example.

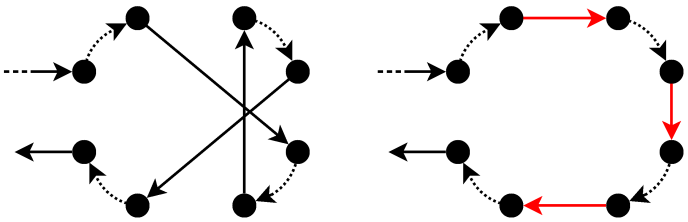

Figure 3.6: A 3-opt move

Optimizing the packing scheme

Until now, the local search method focused solely on the tour of the vehicles, as this directly influences the costs. The last two neighborhoods try to reorganize the packing of the vehicles with the goal to remove a customer from the tour of a vehicle. These neighborhoods might reduce the total routing costs by superseding the need to visit a customer by a certain vehicle.

The first neighborhood performs the search for a vehicle that has loaded a single customer order. The routing costs can be decreased if this demand can be loaded into an other vehicle that already visits the same customer. The local search method updates the packing matrix P and tour matrix T of both vehicles.

The second neighborhood tries to remove all orders of a single customer in the tour of the vehicle and tries to distribute them among the other. In contrast, empty vehicles are not considered here. Both neighborhoods try to shrink the set of vehicles that deliver demands to the same customer, as this might reduce the routing costs.

3.2 A Branch-and-Price Approach

The heuristic solution approaches, presented in the previous section, might actually find the optimal solution, but they cannot proof their optimality. Based on the problem formulation by Derigs et al. [14], presented in Chapter 1, the problem is reformulated in order to be solved by a Branch & Price approach.

The Master Problem

The basic idea of Branch & Price [18] is to utilize an appropriate (re)-formulation and successively add new variables that might improve the current solution. The Master Problem (MP) formulation for the VRPC is obtained by reformulating it as a Set Partitioning Problem (SPP) [32], [23] and [43]. In an SPP the aim is to select a subset of the provided sets with corresponding costs such that each element in the universe is covered by exactly one set and the total costs are minimal.

For the VRPC each set represents a feasible tour and each element in the universe depicts a single order. For each set exists a binary selection variable that defines whether the set is selected or not. The pricing problem then deals with finding new feasible tours with a corresponding valid packing scheme. Such a formulation supersedes the problem of finding feasible packing schemes every time the master problem is solved, as for each variable there already exists a feasible packing scheme. The latter has to be calculated exactly once for each variable, which is done in the pricing subproblem.

Each binary variable in the MP conforms to a feasible tour of a vehicle and determines if the corresponding tour is part of the solution or not. Further each variable covers the demands that are delivered with the corresponding vehicle and has costs associated to it. The costs of each variable are the accumulated costs along the route of the vehicle. As the SPP model deals with finding the sets that partition the universe into distinct subsets with minimal accumulated costs, this corresponds to a set of tours with minimal costs and therefore the optimal solution of the VRPC. The SPP formulation for the VRPC is defined by the linear constraints (3.7) - (3.11).

The variables x_s tell whether the LP solver selected the tour that is represented by this variable, or if this tour is not part of the solution. The objective function (3.7) minimizes the total tour costs. The objective function value is obtained by multiplying the costs of each tour c_s with the corresponding binary selection variable x_s. The partitioning constraints (3.8) are further called the *loading constraints*. They enforce that each demand is delivered by exactly one vehicle. The constants a_{ds} define if vehicle s delivers demand d or not. The third constraints (3.9) are not necessarily needed to calculate a feasible optimal solution, but they provide improved branching possibilities. For each customer c these constraints provide an upper and lower bound for the number of vehicles that visit c and are therefore called the *visit constraints*. Similar to the constants a_{bs} the constants b_{cs} define whether the customer c is visited in the tour s or not. The fourth constraint (3.10) gives an upper limit on the number of vehicles that are allowed to start from the depot and is further called the *fleet*

constraint. The last constraint (3.11) enforces the variables to be binary.

$$\text{min.} \sum_{s \in S} c_s * x_s \tag{3.7}$$

$$\text{s.t.} \sum_{s \in S} a_{os} * x_s = 1 \qquad \forall o \in O \tag{3.8}$$

$$1 \leq \sum_{s \in S} b_{cs} * x_s \leq |V| \qquad \forall c \in L \tag{3.9}$$

$$\sum_{s \in S} x_s \leq |V| \tag{3.10}$$

$$x_s \in \{0, 1\} \tag{3.11}$$

The Set Covering Problem (SCP) is very similar problem formulation, which allows orders to be delivered by more than one vehicle. The BIP formulation of the SCP is almost equal to the one of the SPP except that each demand must be delivered by at least one vehicle, i.e. Constraint (3.8) becomes $\forall o \in O \ \sum_{s \in S} a_{os} * x_s \geq 1$. The big disadvantage of using the set covering formulation, according to [23] and [32] are the weaker dual bounds of the master problem.

Figure 3.7 gives a brief overview on the Branch & Price approach: the initial model is generated using the GRASP algorithm and hence consists of a small subset of all variables that represent individual tours. Running the GRASP several times to generate the initial solutions gives very similar results compared to the PSO but needs only a fraction of the time. By relaxing the domains to be continuous, i.e. $x_s \in [0, 1]$, the model can be solved with the simplex algorithm, provided by the ILP suite. This initial model is called the restricted master problem (RMP). After the RMP is solved the algorithm generates the pricing subproblem and solves it. Each solution to the pricing problem represents a variable for the MP and is added to the model. If the pricer found new variables the RMP is solved again, else the algorithm checks each variable for integrality. This means checking if each binary selection variable has either the value 0 or 1. If there are variables that have a fractional value, the solver makes a branching decision and limits the variable(s) accordingly. After branching the RMP is solved again. The process terminates if all active variables are integral (in this model if $\forall x_s \in S : x_s \in \{0, 1\}$) and if the primal and dual bounds are equal.

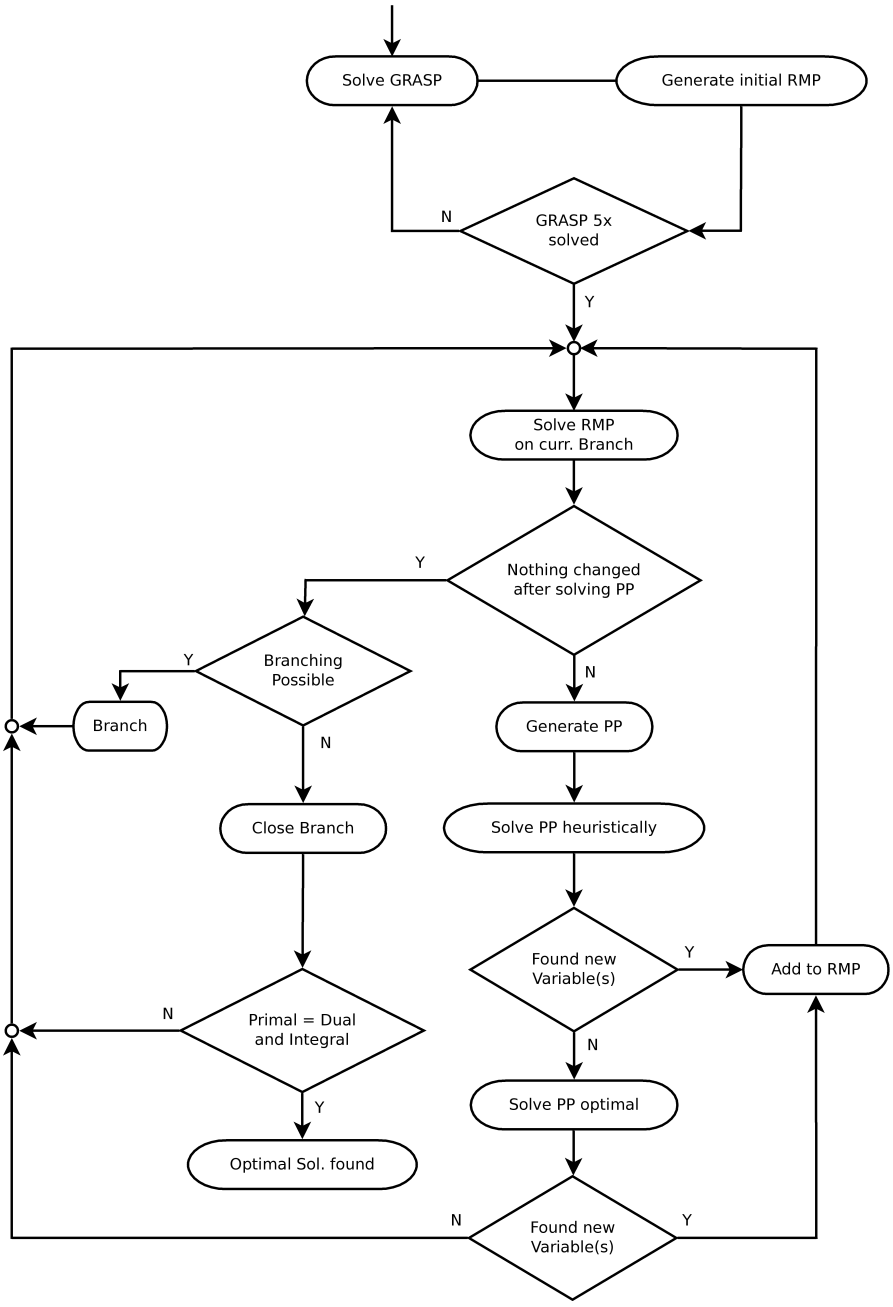

Figure 3.7: A brief overview of the Branch & Price process.

Solving the Pricing Subproblem

The pricing subproblem consists of solving the Elementary Shortest Path Problem with Resource Constraints (ESPPRC) on a directed graph. The goal of the Pricing Problem is to find paths with negative reduced costs, since these are candidates which might improve the objective function value, if selected in the next iteration of solving the RMP.

Definition 8. *In graph theory, a path is called elementary if it contains no cycles.*

In the case of the SCP formulation of the master problem each order can be delivered with multiple vehicles. Further the pricing problem becomes a shortest path problem with resource constraints where the negative cost paths must not be elementary anymore.

The vertexes in the graph $G = (V, A)$ represent the orders and the artificial start v_s and terminal vertexes v_t, which denote the start end the end of each path. Each vertex that represents a demand is adjacent, over outgoing arcs, to all other vertexes, except the start vertex. From the start vertex arcs yield to all vertexes except the start and terminal vertexes. The terminal vertex has no outgoing arcs. The costs on the arcs are calculated depending on the routing costs and the dual variable values of the current LP solution of the MP. To make the long story short, the graph is a complete directed graph with costs associated to each arc, determined by the demands, two additional vertexes, the start vertex and the terminal vertex, and the current LP solution of the RMP. The Pricing Problem consists of finding a shortest path through this graph, which is feasible according to the packing constraints. Figure 3.8 shows a very simple example.

The costs on the arcs are obtained from the dual variable values of the current solution of the RMP and the traveling costs between the depot and the customers. For each arc from a vertex v_i to v_j, where v_i represents either demand i or the start vertex and v_j either the demand j or the terminal vertex, the costs are derived as follows:

$$\hat{c}_{ij} = \begin{cases} c_{ij} - \rho & i \text{ corresponds to start vertex} \\ -\pi_i & customer(i) = customer(j) \\ c_{ij} - \pi_i - \kappa_i & else \end{cases} \tag{3.12}$$

Hence the costs on the arcs are generated differently depending on the demands that the vertexes represent:

1. Each arc that begins at the start vertex: the costs depend on the routing costs c_{ij} from the depot to the customer that ordered demand j, and ρ, the dual variable value of the *fleet constraint* (3.10) in the RMP.

2. All arcs that start and end at vertexes that correspond to demands of the same customer: in this case the routing costs c_{ij} are 0 and only the dual variable value π_i of the *loading constraint* (3.8) of demand i determines the costs.

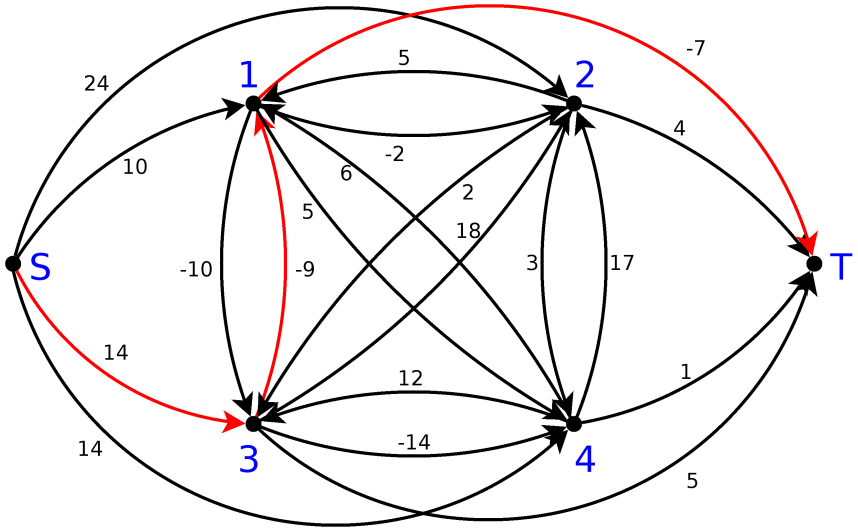

Figure 3.8: A very simple graph with 4 demands and the start and terminal vertexes that represent the depot. The vehicle starts from the depot, delivers demand 3 to the customer that ordered the demand, then it delivers demand 1 to the respective customer and finally returns to the depot. The costs of the resulting tour are -2.

3. For all arcs that are incident to vertexes whose demands come from different customers or the arc ends at the terminal vertex the costs are accumulated from three different values: the routing costs c_{ij} from customer i to customer j or back to the depot, π_i, the dual variable value of the *loading constraint* (3.8) and κ_i, the dual variable value of *visit constraints* (3.9).

The dual variables are part of the dual restricted master problem that provides the lower bound on the objective function value. The dual variables of the *loading* and *visit constraints* are influenced by the branching decisions in the B&B tree.

When solving the ESPPRC the shortest path from the start vertex to the terminal vertex has to be found. In addition, all constraints along the path must be satisfied: i.e. the conditions that the vehicle and compartments must not be overloaded and without violating any incompatibility constraint. For every set of demands along a (partial) path a feasible packing scheme has to be found. If no such packing exists the partial path is infeasible and not extended further.

MIP Formulation

The following MIP model shows the ESPPRC LP formulation for this special pricing problem. The objective (3.13) is to minimize the costs of the path through $G = (V, A)$, which is evaluated by multiplying the binary selection variables x with the costs of the corresponding arcs $\hat{c}$.

$$\min \sum_{i \in V} \sum_{j \in V} \hat{c}_{ij} * x_{ij} \tag{3.13}$$

$$\text{s.t.} \sum_{j \in V \setminus \{s,t\}} x_{sj} = 1 \tag{3.14}$$

$$\sum_{i \in V} x_{ik} - \sum_{j \in V} x_{kj} = 0 \qquad \forall k \in V \setminus \{s,t\} \tag{3.15}$$

$$\sum_{i \in V \setminus \{s,t\}} x_{it} = 1 \tag{3.16}$$

$$u_i - u_j + |V| * x_{ij} \leq |V| - 1 \qquad \forall i,j \in V \tag{3.17}$$

$$u_s = 1 \tag{3.18}$$

$$\sum_{i \in V \setminus \{s,t\}} quantity(i) * y_{ic} \leq compCapa(c) \qquad \forall c \in C \tag{3.19}$$

$$\sum_{i \in V \setminus \{s,t\}} \sum_{c \in C} quantity(i) * y_{ic} \leq vehCapa \qquad \forall v \in V \tag{3.20}$$

$$\sum_{j \in V \setminus \{s,t\}} x_{ij} = \sum_{c \in C} y_{ic} \qquad \forall i \in V \setminus \{s,t\} \tag{3.21}$$

$$y_{ic} = 0 \qquad \begin{array}{l} \forall i \in V, \forall c \in C \\ \forall (i,c) \in IncProdComp \end{array} \tag{3.22}$$

$$y_{ic} + y_{jc} \leq 1 \qquad \begin{array}{l} \forall i,j \in V, \forall c \in C \\ \forall (i,j) \in IncProd \end{array} \tag{3.23}$$

$$u_i \in \{1, \ldots |V'|\} \qquad \forall i \in V \tag{3.24}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in V \tag{3.25}$$

$$y_{ic} \in \{0,1\} \qquad \forall i \in V, \forall c \in C \tag{3.26}$$

Constraints (3.14) - (3.18) define the path through G to be connected and elementary, i.e. every vertex is at most once part of the path and it starts and ends at the start and end vertexes v_s and v_t respectively. The resource constraints limit the path through G further by enforcing the vehicle and compartments not to be overloaded using constraints (3.19) and (3.20) as well as the incompatibility constraints (3.22) and (3.23) to be feasible.

The packing sub problem is modeled using the binary variables y, which describe in which compartment each order, represented by the vertexes along the path, is packed. They are linked with the path selection variables x using (3.21). Constraints (3.24) - (3.26) define the domain of the path selection and packing variables x and y, respectively, to be binary and the sub-cycle elimination constraints to be integer.

Adding a newly generated Variable to the Master Problem

In order to facilitate the comprehension, the procedure of extending the MP is discussed before it will be shown how these variables are actually generated. When one of the pricing algorithms in the next sections finds new paths from the start to the terminal vertex in the ESPPRC graph G, new variables have to be included in the ILP formulation of the RMP. Each new variable x_n is added to the set of variables S and each constraint has to be updated accordingly. For each demand that is delivered with the vehicle represented by x_n, a constant $a_{on} = 1$ is added to the corresponding *loading constraint*. If the demand is not part of the packing $a_{on} = 0$. The same has to be done with the *visit constraints*: if customer c is part of the tour then $b_{cn} = 1$ else $b_{cn} = 0$. The *fleet constraint*, that limits the number of vehicles that start from the depot, is the last constraint that is updated. Algorithm 3.9 shows the function that adds a new variable to the MP.

Algorithm 3.9: The `add_variable` function

input: A new variable x that represents the path p in the ESPPRC graph G

1 objective $\leftarrow$ objective $+$ `costs`$(\mathsf{x}_n) * \mathsf{x}_n$;
2 **foreach** o $\in O$ **do**
3 $\quad a_{\mathsf{on}} \leftarrow \begin{cases} 1 & \mathsf{o} \in \texttt{orders}(\mathsf{p}) \\ 0 & \mathsf{o} \notin \texttt{orders}(\mathsf{p}) \end{cases}$;
4 **end**
5 loading_constraints $\leftarrow$ loading_constraints $+ a_{\mathsf{n}} * \mathsf{x}_n$;
6 **foreach** c $\in L$ **do**
7 $\quad b_{\mathsf{cn}} \leftarrow \begin{cases} 1 & \mathsf{c} \in \texttt{customers}(\mathsf{p}) \\ 0 & \mathsf{c} \notin \texttt{customers}(\mathsf{p}) \end{cases}$;
8 **end**
9 visit_constraints $\leftarrow$ visit_constraints $+ b_{\mathsf{n}} * \mathsf{x}_n$;
10 fleet_constraints $\leftarrow$ fleet_constraints $+ \mathsf{x}_n$;

In order to speed up the generation of new variables, for any variable that is added to the RMP a set of slightly modified variables will be added, too. A dedicated function generates new variables by replacing compatible orders. An order is compatible if the customer who ordered them is the same, the product type is the same, and the amount of the replacing demand is at most as large as the replaced demand.

Heuristic Pricers

Since the ESPPRC is a $\mathcal{NP}$-hard problem and is called every time the master problem has been solved, the overall process can most likely be accelerated by applying a heuristic to find new variables. The basic idea of the heuristics is to divide the path searching process and the packing problem. In the first part different algorithms search negative paths through the ESPPRC graph $G = (V, A)$, whereas the packing scheme is only calculated when a negative path through G has been found. If the packing is unfeasible the generated path is discarded.

For each negative path through G, the CP solver solves the model (3.27) - (3.29) and obtains the packing scheme when facing the $\mathcal{NP}$-hard packing problem, else the online packing algorithm, which is depicted in Algorithm 3.4, is used to calculate the packing scheme otherwise.

Algorithm 3.10: A construction heuristic to solve the ESPPRC

input: The ESPPRC graph G

```
routes ← ∅;
foreach a ∈ A ∧ a.cost < 0 do
    route ← extend_path(a.source, a.target);
    if route.cost < 0 ∧ route.front = v_start ∧ route.back = v_terminal then
        routes ← routes ∪ route;
    end
    route ← extend_path_random(a.source, a.target);
    if route.cost < 0 ∧ route.front = v_start ∧ route.back = v_terminal then
        routes ← routes ∪ route;
    end
end
foreach route ∈ routes do
    if route.costs < 0 then
        tour ← calculate_tour(route);
        packing ← calculate_packing(route);
        var ← generate_variables(tour, tour.costs, packing);
        if var ∉ variables then
            variables ← variables ∪ var;
        end
    end
end
```

The basic idea of the first heuristic is to start from a negative arc and extend the path towards the start and terminal vertexes using two different path extension functions. Algorithm 3.10 describes this process. The first path extension algorithm is called `extend_path` and is

depicted in Algorithm 3.11. This algorithm starts with the partial path (v_i, v_j) that contains only the negative arc a_{ij} and tries to extend this path from the front towards the start vertex by either adding a negative arc or finishing the partial path towards the start vertex if the resulting partial path is negative. The same is done from the end of the partial tour towards the terminal vertex. The path extending loop is terminated if the path is complete, the algorithm is not able to extend the partial path further, or the iteration limit is reached.

Algorithm 3.11: The `extend_path` algorithm

input: A partial path that has to be completed

```
1  improvement ← true;
2  while route.front ≠ v_start ∧ route.back ≠ v_terminal ∧ improvement do
3      improvement ← false;
4      if route.front ≠ v_start then
5          if ∃ a_(v_start, tour.front).cost < 0 then
6              tour ← (v_start, tour);
7              improvement ← true;
8          end
9          if ∃ a_(v_x, tour.front).cost < 0 ∧ v_x ∉ tour then
10             tour ← (v_x, tour);
11             improvement ← true;
12         end
13     end
14     if route.back ≠ v_terminal then
15         if ∃ a_(tour.back, v_terminal).cost < 0 then
16             tour ← (tour, v_terminal);
17             improvement ← true;
18         end
19         if ∃ a_(tour.back, v_x).cost < 0 ∧ v_x ∉ tour then
20             tour ← (tour, v_x);
21             improvement ← true;
22         end
23     end
24 end
```

The `extend_path_random` algorithm is very similar to the `extend_path` algorithm, except that a random arc is accepted to extend the partial tour. This arc must not necessarily have negative costs, instead it is enough that the costs of the resulting partial path are negative. This algorithm terminates if the tour is complete or an iteration limit is reached. It is not depicted in detail here as it does not differ much from the `extend_path` algorithm.

A second heuristic approach is similar to the REUSE heuristic described in [44], which builds upon a similar idea used in combination with a tabu search [15]. The idea is to take the result of the current primal LP solution, slightly modify it and use a local search method to find a path with negative costs. The adapted heuristic takes the current active variables from the current LP solution of the MP and performs random modifications to it without using a successive local search procedure. The modifiers are defined such that the resulting path remains feasible and potentially has negative costs.

Following operations are performed on the paths by the REUSE heuristic. For each variable the algorithm performs up to ten random operations to generate new variables. If the resulting costs are negative and a feasible packing scheme can be found, a new variable is generated. Most functions use a neighborhood of the current customer to select the vertexes for the corresponding operations. Here the neighborhood is defined by the set of all orders of the three nearest customers to the considered customer.

1. **Remove**: If the path contains at least 1.5 times more vertexes than a vehicle has compartments, this function removes one or two random vertexes from the current path. The path is modified such that the vertexes to be removed are skipped without performing any reordering of the remaining vertexes in the path.

2. **Add 1**: This function selects a random vertex and adds one or two random vertexes to the current path. They are selected such that the random vertex and the vertexes that are inserted correspond to orders of the same customer.

3. **Add 2**: This function adds randomly one or two vertexes to the current path at a random position. The vertexes that are inserted are selected randomly from all demands of the three nearest customers of the current vertex.

4. **Exchange 1**: This function selects a random vertex and exchanges it with one or two vertexes belonging to the neighborhood of the removed vertex.

5. **Exchange 2**: This function selects a random vertex and exchanges it with one or two vertexes that correspond to the same customer.

Dynamic Programming Approach

Even though optimality is not needed in each iteration, the algorithm that solves the Pricing Subproblem must find a path with negative costs if one exists. Else the solution obtained by the overall CG approach is not guaranteed to be optimal. Hence applying an exact algorithm to the ESPPRC guarantees that no such path is omitted. The first algorithm is a dynamic programming approach. The idea of dynamic programming goes back to Richard Bellman in 1953 where he had the idea of solving a larger problem by solving smaller subproblems and then combining them to the optimal solution of the larger problem [5]. The results of the subproblems are stored in a table and when needed they are read from it.

More specifically the dynamic programming approach used to calculate the negative paths through the ESPPRC graph G is a label correcting algorithm [32, 50, 23, 43]. Each label represents a path from the start vertex to those where the label is stored and an upper bound on the costs of any path to this vertex. If the algorithm finds a path with less costs ending at the same vertex the old label is discarded and a new, improved upper bound is found. Each label at a specific vertex holds further the accumulated consumed resources along the path it represents. These resources are, as mentioned before, the costs along the arcs of the path, the vertexes that the path visits, a reference to the predecessor label to reconstruct the path, and the amounts of the orders of the vertexes that are part of the path. An implicit bound on the path is due to the total loading capacities of the compartments and the vehicle.

When the label correcting algorithm extends a label that was not extended yet, it generates new labels at any incident vertex that is not already in the path of the label. By applying this extension on all nonextended labels, all possible paths through the graph are enumerated. Yet in most cases only a very limited subset of all paths is generated since new labels that have larger costs than already calculated labels are not extended further. If the accumulated demands cannot feasibly be loaded into the vehicle the label is discarded and not extended further as well.

Algorithm 3.12 shows how this algorithm generates the path from the start vertex to the terminal vertex in the ESPPRC graph G. The algorithm starts with a single label at the start vertex that is pushed into the queue Q. Every time a new label is created it is pushed into Q. The algorithm can pop out each label that is not extended yet and extend it to incident vertexes in G. The function `extend_label` tries to calculate a feasible packing for each new label. If the orders cannot be feasibly packed into a single vehicle, the function returns an empty label to indicate a path that cannot be extended further.

The algorithm uses two main data structures: Q, the priority queue that holds all labels that are not extended yet and a set containing all generated labels L. At the beginning the priority queue holds a single label that marks the root label at the start vertex. The set of labels is divided among the vertexes in G where each one holds a list of labels L_v of partial paths through G ending in v. Further each label $l \in L_v$ holds a reference to its predecessor label. The path ending at any label can be reconstructed by following these references to the root label in the start vertex. Using a priority queue enables the algorithm to extend promising paths with a higher priority, since the labels having the least costs appear first in the queue.

Algorithm 3.12: Label correcting algorithm to find the shortest path in G

input: The graph $G = (V, A)$

```
1  L_start ← {new_label(v_start)};
2  push(Q, L_start);
3  while Q ≠ ∅ do
4      current_label ← pop(Q);
5      foreach (current_label.vertex, v) ∈ A) do
6          if v ∉ current_label.path then
7              new_label ← extend_label(current_label, v);
8              if new_label ≠ ∅ then
9                  if new_label.node = v_terminal then
10                     if L.costs < 0 then
11                         generate_variables(L.tour, L.costs, L.packing);
12                     end
13                 else
14                     not_dominated ← true;
15                     foreach label ∈ L_v do
16                         if new_label dominates label then
17                             recursive_eliminate(label);
18                         end
19                         if label dominates new_label then
20                             not_dominated ← false;
21                         end
22                     end
23                     if not_dominated then
24                         push(Q, new_label);
25                         L_v ← L_v ∪ new_label;
26                     end
27                 end
28             end
29         end
30     end
31 end
```

The **dominance** function avoids the further extension of partial paths in case cheaper paths are already present at the current vertex. Every time the algorithm extends a path to a vertex, the newly generated label is checked whether it is dominated by any of the other labels at this vertex. Further it is checked if the new label dominates any other label. If this is the case all other labels that were extended from this label are superfluous too, since it is possible to generate a cheaper path to all these labels starting from the newly generated label.

Function `recursive_eliminate`, depicted in Algorithm 3.13, finds these labels and eliminates them recursively, since other labels may have already been extended from these dominated labels, too.

Algorithm 3.13: Recursively eliminate all dominated labels

input: A dominated label label

```
1 foreach v ∈ V do
2     foreach l ∈ L_v do
3         if l.pred = label then
4             rekursive_eliminate(l);
5         end
6     end
7 end
8 delete(label);
```

In order to enable a faster termination of the label correcting algorithm two different dominance functions are used: a restrictive, that invalidates more, potentially too many labels, and one that guarantees optimality.

1. Label l_1 dominates label l_2 iff $l_1.cost < l_2.cost$
2. Label l_1 dominates label l_2 iff $l_1.cost < l_2.cost \wedge l_1.demands \subset l_2.demands$

The first dominance function invalidates each label that has larger costs. This causes many labels to be invalidated and results in a faster termination of the algorithm. If the label correcting algorithm with this so-called weak dominance rule does not find any new variable, it is executed again using the second, strong dominance rule.

Breaking the Symmetry

Symmetries in ILP models cause additional and costly computation efforts that make no progress in solving the ILP [37, 19]. In this formulation for the VRPC there occur two different cases of symmetries, one in the MP and one in the ESPPRC formulation.

The symmetry in the MP formulation arises from the permutation in the packing problem: if the tours represented by two different variables contain exactly the same demands, but which are packed in different compartments, they are still symmetric. Note that a permutation in the packing scheme has no influence on the objective function. This kind of symmetry can be eliminated by comparing the packing schemes before adding a new variable to the MP. Each variable holds an array of boolean values that determines which demands are loaded into the vehicle represented by the variable. Lets assume w.l.o.g. that the i-th value

is 1. This means that the i-th demand is loaded into the vehicle. If the costs and these arrays are equal for two variables, the newly generated variable can be omitted safely.

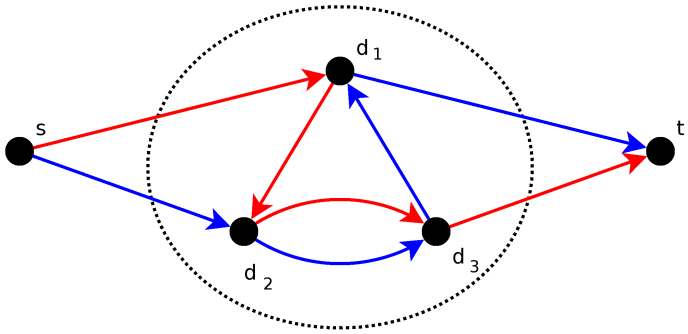

Figure 3.9: A symmetric path in the ESPPRC graph G. All demands are from the same customer, depicted by the dashed line around the demands.

The second symmetry comes from the order of the demands in a negative cost path in the ESPPRC problem. This symmetry causes the label correcting algorithm to extend more labels than actually needed. Figure 3.9 shows a symmetric path in the ESPPRC graph G. The two different paths are $p_1 = (s, d_1, d_2, d_3, t)$, the red path, and $p_2 = (s, d_2, d_3, d_1, t)$, the blue path. This symmetry occurs only when demands of the same customer are permuted in the path. Due to the definition of the costs they are equal for both paths. This kind of symmetry can be avoided by defining an additional constraint for the extension of a label: each label can only be extended to vertexes of demands of the same customer, if these demands have a higher index than the demand of the current label.

Lemma 1. *This symmetry breaking constraint preserves the optimality of the label correcting algorithm.*

Proof. This assertion can be proven by contradiction: Lets assume w.l.o.g. that the only optimal path p violates this constraint and therefore there exists at least one vertex v_i in p where the demand of the successor v_{i+1} has a lesser index than the demand of v_i. For sake of simplicity the order of the vertexes corresponds to the order of the demands, which is a unique natural number. For p this means $\forall j \neq i \in p : v_j < v_{j+1}$ and $v_i > v_{i+1}$.

From the construction of the ESPPRC graph it is obvious that the costs on each arc, where both incident vertexes correspond to the same customer, are equal, namely the dual variable value $-\pi$ (Note: all arcs where the incident vertexes correspond to different customers are not relevant for this constraint and need not be considered in the proof). Let p_c be the path through G where the position of v_i and v_{i+1} is switched. For p_c the following holds: $\forall j \in p_c : v_j < v_{j+1}$ and by assumption $costs(p) < costs(p_c)$.

Since the vertexes v_i and v_{i+1} belong to the same customer and p and p_c have exactly the same customers ($\forall v_j \in p_c \rightarrow \exists v_j \in p$ and $\forall v_j \in p \rightarrow \exists v_j \in p_c$) we have a contradiction:

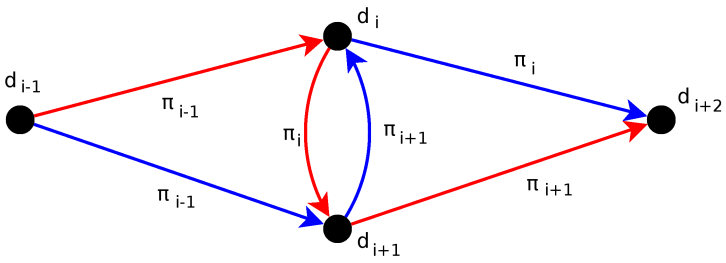

Figure 3.10: The paths p and p_c through G.

it is not possible that $costs(p) < costs(p_c)$, since all outgoing arcs from each vertex are added exactly once and the arcs $(v_{i-1}, v_i) = (v_{i-1}, v_{i+1}) = \pi_{i-1}, (v_i, v_{i+1}) = (v_i, v_{i+2}) = \pi_i, (v_{i+1}, v_i) = (v_{i+1}, v_{i+2}) = \pi_{i+1}$ and both paths must have the same costs. Figure 3.10 shows this case: p is the red and p_c the blue path. □

This constraint invalidates the creation of the blue path in Figure 3.9, since the path cannot be extended from d_3 to d_1.

Branching Rules

For the performance of the B&P approach it is crucial that the B&B tree is balanced. Setting a binary variable to 0 has a very low impact on the LP solutions of this subtree, whereas setting the same variable to 1 has a very high influence on the LP solutions. Especially as the number of variables increase it is important to define sophisticated branching rules that generate a balanced tree and limit the domain of a larger set of variables.

A new branch for a fractional LP solution is opened if solving the ESPPRC did not generate any new variables and the LP solution of the MP did not change since the last time it was solved. The first branching rule counts the number of active variables and checks if the sum is a natural number. This branching rule limits the number of vehicles that start from the depot to be less or larger than a natural number, which is calculated by rounding the summed up, current LP solution values. If the sum is rational the B&B tree is extended by two new subtrees with the corresponding limits on the sum of variables. A short example demonstrates this branching rule: lets assume w.l.o.g. the number of active variables $\sum_{s \in S} x_s = 16.7521$ then two new constraints are added. The first constraint $\sum_{s \in S} x_s \leq 16$ is added to the left subtree and the second constraint $\sum_{s \in S} x_s \geq 17$ is added to the right subtree.

The second branching rule limits the number of vehicles that visit each customer. For this the branching rule determines the number of vehicles that visit each customer and creates new subtrees for the first customer where the number of active variables is fractional. In order to have a link from the variable to the customers, constraints (3.9) and the constants b_{cs} in the MP are used and updated every time a new variable is generated. These two

branching rules provide a balanced B&B tree and the solver does not need to branch on single variables anymore.

Limiting the Number of Variables in the MP

In order to improve the memory usage of the whole program some variables that are added to the MP, but are never part of the basis of the LP, i.e. they are never part of any solution during the solving process, should be removed. The algorithm keeps track of the number of iterations and the number of times the variable is part of the solution. If the number of iterations the variable was part of the LP, but never part of the solution, exceeds some limit, the variable is removed from the MP.

3.3 The Packing Problem

The packing problem is highly relevant for the overall performance, since it has to be solved every time an algorithm extends a partial solution or tries to find a feasible packing for an already calculated tour. Packing demands for the food instances is easy in the sense of computational complexity, since there is only one compartment for each product. Therefore checking if the packing is feasible means checking if the sum of all demands does not exceed the limits of the compartment and the vehicle. The loading scheme for these instances can be calculated with the online algorithm 3.4 that provides optimal solutions in this case. This section describes how the packing for the petrol instances, with the fixed size compartments, is calculated, which is the weakly $\mathcal{NP}$-hard packing subproblem of the VRPC.

Since the heuristic algorithms do not provide optimal routes for the vehicles there is no need to solve the packing problem to optimality. Therefore the heuristic solvers use only the online heuristic algorithm 3.4 to obtain a valid packing scheme. Contrary when the problem is solved to optimality the packing problem has to be solved in an optimal manner, too. Every time the label correcting algorithm adds a vertex to the (partial) path from the start vertex to the terminal vertex, the packing problem has to be solved again.

This special packing problem is simpler to solve than the bin packing problem from a practical point of view: the number of bins is fixed and only a feasible packing has to be found instead of minimizing the total number of used bins. None the less it is important to point out that from a theoretical point of view the packing problem is a weakly $\mathcal{NP}$-hard problem [40]. In the following sections several algorithms are presented that solve this special packing problem.

Approximation Algorithms

Since a feasible packing is known from the previous node, it can be checked if the new demand fits into a bin of the already known packing scheme using the online algorithm 3.4. If this is the case a feasible packing is found and the packing algorithm can terminate, else a new packing scheme is calculated using the greedy First Fit Decreasing (FFD) approxima-

tion algorithm 3.14. The demands are sorted according to their amounts and packed in a first fit manner.

Definition 9. *An algorithm is called **greedy** if it makes a choice at each stage of the algorithm without ever reconsidering any decision made.*

The FFD algorithm starts by inspecting whether the first compartment can accommodate the current demand. If it fits in, the FFD algorithm packs the demand into the first compartment without further consideration. If the current demand does not fit into the current compartment or it is incompatible with any demand in the current compartment it searches in the next compartment. When the FFD algorithm is finally able to pack the current demand it begins the search with the next demand starting at the first compartment again. If the FFD algorithm is not able to pack an item, it finishes without providing a complete solution, and a more sophisticated method has to be used in order to find a feasible packing or proof that none exists for this particular set of demands.

Algorithm 3.14: First Fit Decreasing approximation algorithm

```
1  D ← sort(D);
2  foreach comp ∈ C do
3  |  p_comp ← ∅ ;
4  end
5  foreach dem ∈ D do
6  |  packed ← false ;
7  |  foreach comp ∈ C do
8  |  |  if fits_in(dem, p_comp) then
9  |  |  |  p_comp ← p_comp ∪ {dem};
10 |  |  |  packed ← true ;
11 |  |  |  continue on line 5 with next demand;
12 |  |  end
13 |  end
14 |  if ¬ packed then
15 |  |  return ∅ ;
16 |  end
17 end
18 return p ;
```

Constraint Programming Models

If the approximation algorithms fail, the CP solver either finds a feasible packing scheme or proves the nonexistence of such. In the **first model** the constraints are defined on a two

dimensional binary Matrix M. A value of 1 in the d-th column and the c-th row denotes that demand d is packed into compartment c. The incompatibilities between two demands in the same compartment are modeled by inequality constraints of the corresponding elements in the matrix.

$$\sum_{c \in C} M_{d,c} = 1 \qquad \forall d \in D \tag{3.27}$$

$$(d_1, d_2) \in IncProdProd \rightarrow M_{d_1,c} \neq M_{d_2,c} \qquad \forall d_1 \neq d_2 \in D, \forall c \in C \tag{3.28}$$

$$\sum_{d \in D} amount_d * M_{d,c} \leq CompCapa_c \qquad \forall c \in C \tag{3.29}$$

The set of demands D is the set of demands that has to be loaded into the vehicle and is a subset of all demands. C is the set of compartments. The constraints applied on the matrix are presented in (3.27) - (3.29) and this assignment problem is solved using the depth first search algorithm the CP solver offers.

This model solves only the cascaded bin packing like problem with incompatibilities between two products in the same compartment. Including the incompatibilities between demands and compartments (from the simpler packing problem) in this model can be achieved by adding the following constraint:

$$\forall d \in D, \forall c \in C \qquad (c, d) \in IncProdComp \rightarrow M_{d,c} = 0 \tag{3.30}$$

Many CP solvers offer special constraints to model a specific problem. The CP solver used in this thesis implements a special propagator to solve the Bin Packing Problem. The big advantage of using special constraints is the faster propagation, since algorithms, that use problem specific knowledge, can be implemented. The **second model** uses the bin packing constraint. The CP model becomes an optimization problem with the additional constraint that the number of bins is limited by the number of compartments. The incompatibility constraints remain the same as in the already presented model (3.27) - (3.29). This new model is solved using a Branch & Bound algorithm and is given by constraints (3.31) - (3.33).

$$(d_1, d_2) \in IncProdProd \rightarrow b_{d_1} \neq b_{d_2} \qquad \forall d_1 \neq d_2 \in D \tag{3.31}$$

$$(c, d) \in IncProdComp \rightarrow b_d \neq c \qquad \forall d \in D, \forall c \in C \tag{3.32}$$

$$binpacking(l, b, s) \tag{3.33}$$

The constraints in the second model are defined on a set of integer variables $b = \{b_1, \ldots b_n\}$, the constants $s = \{s_1, \ldots s_n\}$ and $l = \{l_1, \ldots l_m\}$. Each variable $b_i \in b$ represents a

specific demand with the corresponding size information in $s_i \in s$. The value of each variable denotes the compartment into which the demand is loaded. Each constant $l_i \in l$ represents a compartment and holds the size of the corresponding compartment. The first two constraints model the incompatibility constraints and the last one (3.33) is the special binpacking constraint.

The CP solver, used in the implementation of this thesis, allows to implement own specialized branchers, which are classes that are passed to the CP solver in order to make improved branching decisions. These branchers can use problem specific knowledge and sophisticated heuristics to provide a faster pruning of parts of the search tree and therefore speed up the overall process. Heuristics in CP solvers often provide improved bounds on the variables to branch on. Gent and Walsh [26] published the Complete Decreasing Best Fit (CDBF) pruning and propagation rules, which are able to solve the BPP and some related problems to optimality. They used modular arithmetic applied on the constraints to calculate a bound and to prune the search tree. Shaw [56] defined propagation and pruning rules to determine non-packable bins and to decide if items do not fit into bins. They further provided improved lower bounds for partial solutions. Based on these works Schulte implemented the *cdbf* brancher for Gecode which is provided with the examples of Gecode (ver. 3.7.1) [55]. The **third model** is the same as the second one using this *cdbf* branching rule to construct the B&B tree.

The difference between the first and the last two models lies in the way they are solved. The first model is an assignment problem where a feasible assignment of the demands to the compartment has to be found. The second and third models are optimization problems where the CP solver tries to pack the demands in the least number of bins as possible, without an early termination when the demands should already fit into the given compartments. All models have their pros and cons. For a small amount of demands the CP solver finds a feasible packing for the first model faster than for the last two since it terminates quicker, whereas for a larger set of demands the CP solver using the latter two models finds a feasible packing scheme or proves its nonexistence faster compared to using the first model. See Section 4.2 for a discussion about the performance of the different models.

CHAPTER 4

Computational Results

This chapter provides a very brief overview of the algorithms' implementation, the frameworks and libraries applied, the test instances used for the performance evaluation and a detailed analysis of the performance of the algorithms. The first part gives a short overview on the software that was used to implement the solvers.

4.1 Used Frameworks and Libraries

All tests were executed on a scientific cluster, consisting of 16 machines with 14 having two Intel Xeon E5540 processors, running at 2.53 GHz and 24 GB of RAM and two having three such processors. The whole cluster has 136 cores with 3GB of RAM dedicated for each core and is connected with an Infiniband layer. The program was compiled with GCC version 4.6.3. The following libraries and frameworks were used:

- The Standard Template Library (**STL**) provides a large set of basic algorithms and data structures, including container classes (vector, queue, list, set, map, ...), basic algorithms (searching, sorting, transforming, ...) and iterators to iterate over the data in the containers in a common, convenient and efficient manner.
- The **Boost** library is a very large collection of different, smaller libraries. In this work two libraries were used: the boost graph library (BGL) to model the ESPPRC graph and the boost program options library to simplify parsing the arguments passed from the console. The BGL offers the possibility to implement the graphs on a very high level and to add arbitrary objects at the vertexes and edges.
- The Solving Constraint Integer Programs (**SCIP**) [1] B&C&P suite is a fast, non-commercial and highly customizable optimization solving suite, which offers solving linear and nonlinear optimization problems. It supports solving CP models and CG

formulations and allows to use several LP solvers. SCIP provided the lattice for the exact solution approach. Besides the large set of predefined components it provides very simple mechanisms to add custom ones, such as branchers and pricers.

- **Gecode** [55] is one of the fastest and largest CP solvers available under a free license during the implementation of the CG approach for the VRPC. Gecode provides a large set of constraints on binary, integer and set variables that can be solved using DFS, B&B or an algorithm that is able to restart the search with the best solution found so far. Further it provides a simple interface to extend it with custom functions, such as heuristics [6] to provide bounds and improve the overall performance. Gecode is easily extensible with own propagators and branchers, which enable the developer to include problem specific knowledge and algorithms to speed up the overall process.

An extended description of the classes and the interaction between them is beyond the scope of this work, as the main focus lies on the models and algorithms for the VRPC. Note that Chapter 3 already provided a detailed insight on the interesting components of the software from a higher level viewpoint.

4.2 Performance Analysis

For every heuristic and exact algorithm an empirical analysis of the performance is needed in order to state something about the competitiveness to other implementations and models, provided by the scientific community. First a description of the test instances is provided. Then the performance of the algorithms is evaluated on these test sets.

The Test Data

For the performance tests three different classes of test data were used: the test instances provided by Derigs et al. [13], the instances based on Eilon, and newly generated, smaller instances for the CG approach. For each algorithm setting and instance 25 test runs were executed on the scientific cluster with a time limit of 10 minutes for the PSO and a time limit of two hours for the CG approach.

Derigs et al. provided a large benchmark suite of 200 instances for the VRPC to evaluate different algorithms that solve the VRPC. The test instances are available online [13]. The benchmark suite can be divided into two large classes: the food and petrol instances with the corresponding incompatibilities between the product types and the compartments. Further the instances differ by the number of customers (10, 25, 50, 100 and 200), the geographical distribution of the customers (clustered, not clustered), the number of product types (2, 3), and a different maximal amount of the demands (the half of each compartments size, the compartments size).

The GRASP and the PSO are able to handle all instances, whereas the test data provided by Derigs et al. is in almost any case too complex for the CG approach, new test instances

were created in order to identify the main criteria that influences the long execution times of the exact algorithm.

The packing problem in the Derigs et al. benchmark suite only has a minor importance. Although the packing problem for the petrol instances is weakly $\mathcal{NP}$-hard, it can be solved quite easily, since the number of demands that have to be packed into the vehicles is less than 10 and many demands have the same amount: either the half or the full compartment size. All customers for each instance have a very similar or even equal list of demands. In instance `vrpc_p_n25_p3_k1_100`, for example, each customer orders $6 \times 100, 1 \times 83$ for product 1, $2 \times 100, 1 \times 37$ for product 2 and 1×83 for product 3.

The second test set is based on the instances of **Eilon and Christofides**. The original instances are extended versions of the symmetric CVRP instances, which are available online [61]. These instances consider only the simpler case of the VRPC, i.e. the food instances. These instances were generated by doubling each order of the second product type, the vehicle loading capacity, and adding a second, equally sized compartment. The optimal solution for these instances is correlated to the optimal solution for the original instances: the second compartment holds the same amounts as the first one of the other product type. With this scheme the optimal objective value is the same. Two other sets of instances are derived from the Eilon and Christofides instances. In contrast the demands are generated at random, but these instances were not published. The results of them are mentioned in [21, 39].

The third set of **manually derived, smaller instances** is similar to those of Derigs et al. They were generated in order to evaluate the performance of the CG approach and to identify the complicating components of the problem. This set of instances consists of petrol and food type instances, too. The food type instances can be divided further into three subsets where each customer demands 8, 12, or 16 orders. The petrol type instances consist of three subsets, with a different number of compartments, which are 2, 3, or 4 where each compartment has the same size. Each instance can be classified further into 4, 6, 8, or 10 locations, and 2 or 3 different product types. The geographical distribution of the locations, which are the depot and the customers, is determined at random in the interval$[0, 1000]$ without any further constraints. The costs on the arcs between the customers and the depot is the Euclidean distance.

The Performance of the Heuristic Algorithms

First the performance of the GRASP was analyzed with different settings for the size of the restricted candidate list. The settings vary from a strict Clarke and Wright Savings like behavior (RCL = 0) to a random search (RCL = 1), using the configurations described in Figure 4.1. The GRASP is analyzed by solving each instance 25 times for each setting and taking the average costs. Figure 4.1 shows the average performance of the GRASP algorithm executed on four Eilon and Christofides based instances with the different sizes of the restricted candidate list. The red curve represents the GRASP algorithm where the

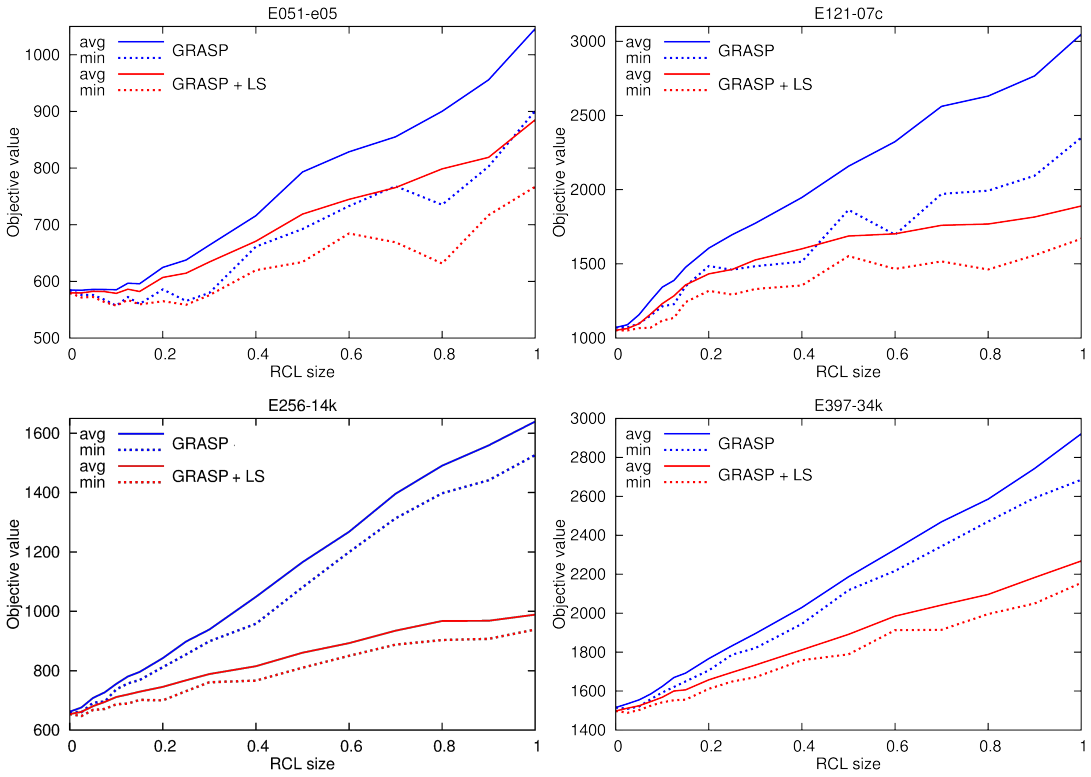

Figure 4.1: Results of the GRASP algorithm with different sizes of the RCL (0.0, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0) and additional local search optimization of the results on four datasets from the Eilon and Christofides dataset.

Local Search algorithm optimizes its results further, the blue one the case without. The dotted lines represent the best solutions among the 25 runs. This Local Search algorithm uses the same neighborhoods as the PSO.

It is observable that the size of the RCL directly influences the performance of the GRASP. The positive influence of the LS procedure on the results is noticeable for any size of the RCL. With an increasing size of the RCL, the random component causes the algorithm to generate randomized solutions having worse quality. It is observable that the additional LS step improves the results in any case and that accepting non optimal local decisions to a certain degree may result in improved solutions. Especially the best results for instance `E051-e05` have significantly improved costs compared to the pure greedy variant (RCL of 0). The first column in Tables 4.1, 4.2, and 4.3 shows the instance, the next few columns the best and average results of the corresponding algorithm and instance and the last column the current BKS.

Table 4.1: Performance of the GRASP algorithms compared to the BKS published in [44]

Instance			GRASP		GRASP + LS		# sign.	BKS
			min	avg	min	avg	better	min
f	10	2	10948.8	11317.2	10808.5	11236.1	1/2	10180.5
		3	14823.8	16046.2	15054.2	16071.7	0/2	14214.0
	25	2	29391.6	30135.8	29264.7	30129.0	0/6	25804.5
		3	24982.1	25872.6	24889.8	25939.4	0/8	22853.6
	50	2	45097.5	46227.7	45123.3	46154.5	1/11	40349.2
		3	42102.5	43332.0	41872.1	43231.0	2/11	38327.1
	100	2	79692.6	81018.9	79512.5	80822.7	4/12	71535.5
		3	76479.6	77942.4	76147.2	77787.8	2/12	69944.7
	200	2	229638.0	232497.0	228455.0	231443.0	4/6	203722.0
		3	156433.0	158794.0	155725.0	157803.0	2/5	145503.0
p	10	2	9893.9	10573.1	10033.5	10530.7	1/4	9510.0
		3	8970.5	9589.5	8888.8	9523.1	1/10	8191.5
	25	2	21476.8	22274.2	21492.5	22249.5	0/15	19811.9
		3	20136.6	20939.3	20328.6	20960.7	2/17	18080.2
	50	2	35831.7	36687.1	35883.1	36711.5	1/18	32849.4
		3	37022.7	38121.3	37038.6	38122.9	3/18	32891.7
	100	2	65630.2	66719.7	65683.0	66715.7	2/16	60615.7
		3	64837.2	65952.5	64665.6	65918.2	3/15	58631.3
	200	2	102749.0	104333.0	102276.0	103835.0	4/6	95263.2
		3	115484.0	118310.0	115380.0	117520.0	4/6	96810.7
avg. costs			59581.1	60834.2	59426.1	60635.3	37/200	53754.5

In the tables showing the results of the heuristic approaches the average solutions, which are significantly better than those of the corresponding other algorithm, are marked bold. The following configurations where compared: the GRASP to the additional LS improvement step and the GRASP without LS, the PSO II to the PSO III, and the PSO III to the PSO IV. The first check determines if the additional LS improvement step is able to significantly improve the GRASP solutions. The second checks if the hunter particle provides any significant improvement and the last one checks if repetitive calling the LS improvement step yields to significant better solutions. The PSO I, which operates on the packing matrix was omitted in the significance checks, since it is not competitive to the other algorithms. The Wilcoxon rank sum test with a significance level of 5% was used to test whether an algorithm variant performs better than another one. The fourth column in Table 4.1 shows how

often GRASP+LS yields a significantly better performance, also stating the total number of instances of this subset. In Table 4.2 the average result of the significance tests was taken in order to determine if a set of solutions is significantly better than the other.

Table 4.1 shows the results of the two GRASP algorithms for different classes of instances taken from the Derigs et al. testbench [13]. It consists of 200 instances that can be divided into different classes, which are: first the problem class (food or petrol industry inspired problem definition), second the number of customers (10, 25, 50, 100 or 200) and third the number of product types (2 or 3). As the number of instances is not equal in each class, the overall averaged results differ from those in Figure 4.1. The GRASP results in Tables 4.1 and 4.2 are generated with a RCL size of 0.25.

As already mentioned, two different PSO algorithms were implemented: one that operates on the tour matrix T and one that operates on the packing matrix P. In order to get a more diverse set of particles, the GRASP uses a RCL size of 25%. The results of the different PSO algorithms are shown in Table 4.2. The columns are the following: the PSO that operates on the packing matrix P (PSO I), which is denoted shortly in Section 3.1, while the next three PSO configurations operate on the tour matrix T, namely the PSO with both, the predator and the LS optimization step (PSO II), the PSO without a predator (PSO III), as well as the PSO without the LS optimization step and without a predator (PSO IV). The instances were divided into the same classes as for the evaluation of the GRASP. The table shows the best and average costs calculated of 25 runs for each data set from the Derigs et al. testbench. The BKS are taken from Pirkwieser [44] who improved most of the results of Derigs et al.

The table gives an overview on the performance of the PSO algorithms with respect to the three different classes of the Derigs et al. instances. As already mentioned, the performance of the PSO that operates on the packing scheme (Packing PSO) is inferior to all other PSO configurations. Although the packing PSO comes close to the performance of the other algorithms for the small classes, its performance is actually worse for each class. Further it can be observed that the predator and the LS optimization step are able to improve the results of the PSO.

Table 4.2: Performance of the PSO algorithms on the Derigs et al. [13] instances

Instance			PSO I		PSO II		PSO III		PSO IV		BKS
			min	avg	min	avg	min	avg	min	avg	
f	10	2	10835.7	11094.8	10742.7	**10808.7**	10747.4	10903.1	10734.5	10888.2	10180.5
		3	14560.8	15061.0	14508.9	**14728.8**	14568.6	14903.0	14514.9	14882.5	14214.0
	25	2	29234.6	29959.3	28542.2	28946.8	28615.0	29078.0	28743.1	29154.4	25804.5
		3	25156.2	25783.2	24214.8	24501.3	24315.8	**24567.2**	24462.2	24847.0	22853.6
	50	2	45961.7	46661.2	44372.1	44804.4	44351.3	44865.3	44598.4	45070.9	40349.2
		3	43246.8	44029.1	41225.5	41585.4	41259.1	**41554.7**	41526.4	42135.6	38327.1
	100	2	82102.7	83058.5	78502.9	78976.3	78329.1	78967.3	78969.8	79790.3	71535.5
		3	80099.3	81113.4	75165.8	75842.6	74935.9	**75608.8**	75951.1	76873.3	69944.7
	200	2	239053.0	241284.0	224444.0	**225770.0**	225715.0	226700.0	229142.0	230718.0	203722.0
		3	167820.0	170023.0	151784.0	152476.0	152101.0	**153276.0**	156708.0	158089.0	145503.0
p	10	2	9875.9	10304.1	9628.8	9659.5	9625.3	**9709.8**	9712.9	9895.1	9510.0
		3	8833.8	9199.4	8479.4	8670.9	8524.7	8674.5	8543.3	8816.3	8191.5
	25	2	21745.5	22350.6	20921.6	21226.9	20978.0	21279.4	21092.0	21447.2	19811.9
		3	20589.7	21160.4	19607.4	19922.9	19617.6	19933.6	19727.5	20129.2	18080.2
	50	2	37420.9	38109.8	35205.7	35637.8	35184.7	35564.6	35594.2	36017.1	32849.4
		3	39142.9	39904.3	36277.7	36812.7	36216.2	**36714.3**	36724.1	37317.1	32891.7
	100	2	70017.7	70945.2	64692.8	**65070.8**	64961.8	**65509.7**	65820.8	66337.2	60615.7
		3	69582.0	70591.6	63724.3	64189.6	64225.2	**64694.7**	64842.6	65449.1	58631.3
	200	2	112246.0	113832.0	100652.0	101357.0	100068.0	**100805.0**	103120.0	104141.0	95263.2
		3	132032.0	134334.0	111708.0	112597.0	111368.0	**112500.0**	116748.0	118420.0	96810.7
avg. costs			62977.9	63939.9	58220.0	58679.3	58285.4	58790.5	59363.8	60020.9	53754.5
avg. Gap to BKS			11.9%	13.7%	6.7%	7.7%	6.8%	7.9%	7.9%	9.2%	

The performance of the heuristic algorithms on the Eilon and Christofides instances is shown in Table 4.3. Looking at the average costs on all instances, it is observable that the PSO, which operates on the tour matrix, has an average 6.1% higher costs than the average best known solutions (BKS). The PSO that operates on the packing matrix has average higher costs of 29.6% than the average BKS. Whereas the influence of the predator with average 6.2% higher costs is relatively low compared to the influence of the LS optimization step with 16.1% higher costs than the average BKS. The best run of the PSO without the predator on instance `E121-07c` almost reached the BKS with only 0.1% higher costs: i.e. 1043.4 compared to 1042.1.

It turns out that the additional hunter particle might be able to result in significant better solutions. But the additional LS optimization step provides even more significant better solutions. Unexpectedly the significance tests show that most of the results of the GRASP with the LS in the Derigs et al. instances are not significantly better in contrast to the Eilon and Christofides instances where all are significantly better.

Table 4.3: Performance of the heuristics on the Eilon and Christofides [39, 21, 61] based instances. BKS are the best known solutions, which are taken from [44].

Instance	GRASP		GRASP + LS		PSO I		PSO II		PSO III		PSO IV		BKS
	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	
E051-05e	573.0	592.9	563.4	**583.0**	619.9	665.7	557.1	571.8	546.9	**559.0**	551.2	567.8	524.6
E072-04f	261.1	280.3	254.1	**263.6**	275.7	289.0	254.1	254.1	244.9	**253.4**	264.8	271.3	237.0
E076-07u	802.5	836.7	779.9	**806.8**	930.2	981.7	769.8	770.0	760.5	**766.6**	787.4	815.0	682.0
E076-08s	795.3	832.8	771.4	**809.2**	928.1	987.0	769.8	769.8	760.5	**764.4**	793.5	818.1	735.0
E076-10e	899.0	926.6	881.2	**910.9**	1065.5	1126.8	846.7	**857.0**	846.7	860.7	893.1	911.7	835.3
E101-08e	907.6	941.9	884.0	**913.6**	1076.6	1121.0	864.2	866.1	850.1	**862.2**	910.4	927.3	826.1
E101-10c	860.3	887.7	827.7	**841.0**	961.6	1068.8	824.8	824.9	823.7	**824.6**	861.7	875.5	819.6
E121-07c	1232.9	1317.5	1075.1	**1226.1**	1373.1	1490.2	1049.9	**1050.4**	1043.4	1051.8	1220.4	1301.4	1042.1
E135-07f	1387.0	1681.8	1393.7	**1543.2**	1523.9	1622.7	1168.2	**1185.9**	1174.6	1205.1	1382.6	1457.3	1162.0
E151-12c	1151.6	1225.6	1144.5	**1179.3**	1457.5	1534.4	1094.4	1104.2	1075.3	**1100.0**	1197.0	1230.0	1028.4
E200-17c	1522.9	1566.9	1417.8	**1486.3**	1917.2	2003.1	1348.5	**1365.0**	1340.4	1373.9	1531.2	1573.9	1291.3
E241-22k	836.4	916.9	796.0	**832.8**	1137.0	1230.7	750.1	757.9	749.8	756.4	870.5	904.0	707.8
E253-27k	990.3	1009.0	968.1	**986.1**	1216.8	1258.4	925.5	936.8	932.8	938.9	994.9	1020.7	859.1
E256-14k	731.6	752.6	675.3	**705.9**	916.2	942.7	633.4	**641.0**	648.0	655.6	753.1	766.6	583.4
E301-28k	1191.9	1249.1	1139.2	**1174.9**	1547.7	1655.0	1054.7	**1065.5**	1069.2	1078.0	1231.1	1262.0	998.7
E321-30k	1258.9	1296.9	1227.6	**1257.4**	1584.7	1652.5	1184.6	**1194.9**	1186.3	1200.3	1295.1	1315.8	1081.3
E324-16k	957.7	978.7	868.0	**904.7**	1156.5	1207.5	809.8	**823.0**	827.6	839.8	982.2	999.5	742.0
E361-33k	1680.8	1746.7	1565.2	**1628.4**	2269.0	2372.9	1445.5	**1459.5**	1473.3	1484.1	1733.4	1767.9	1366.9
E397-34k	1592.4	1627.6	1538.7	**1568.5**	1977.2	2078.6	1469.4	**1481.8**	1476.2	1490.4	1647.9	1660.3	1345.2
E400-18k	1199.5	1228.9	1101.6	**1134.0**	1507.0	1540.7	1021.9	**1029.5**	1042.0	1054.0	1242.3	1273.5	918.5
E421-41k	2246.8	2324.6	2132.6	**2189.0**	2998.5	3129.3	1937.5	**1950.1**	1976.6	1998.5	2343.1	2405.5	1821.2
E481-38k	1970.0	1993.1	1875.6	**1908.3**	2448.5	2535.8	1784.3	**1798.6**	1794.7	1814.1	1989.6	2030.5	1622.7
E484-19k	1512.9	1565.9	1337.5	**1401.3**	2016.5	2110.6	1239.2	**1253.8**	1273.8	1288.8	1628.7	1651.0	1107.2
avg. costs	1154.9	1207.8	1096.4	1141.5	1430.6	1504.6	1034.9	1044.0	1039.9	1053.1	1178.5	1209.0	971.2
avg. Gap to BKS	14.8%	18.6%	10.6%	14.1%	29.6%	33.3%	6.1%	6.9%	6.2%	7.4%	16.1%	18.2%	

Evaluating the CG approach

As the B&P solver generates a lot of variables without making any progress on the Derigs et al. instances, new smaller instances have been created. The results for these food type instances are shown in Table 4.4. This set is divided into the same classes as the Derigs et al. instances and consists of two instances each. One with symmetric and one with asymmetric demands. In the symmetric case each customer demands the same number of orders for each product type. In the asymmetric case each customer demands twice the number of product type 1 than product type 2 and in case of three product types each customer demands twice the number of product type 2 than product type 3. The largest instance in this set has 10 locations, 9 customers and the depot, with each customer demanding 16 orders. The resulting ESPPRC graph has 146 vertexes.

The first column in Tables 4.4, 4.5, and 4.6 shows the instance, the second and third columns the primal and dual bounds obtained by the B&P suite. The fourth column shows the remaining gap between them. The sixth and seventh columns the initial solution, which was obtained by the PSO, and the gap to the primal bound. The next three columns give the number of B&B nodes and the number of initial and priced variables. The last column the average time SCIP needed to calculate the solutions.

Table 4.4: The performance of the B&P approach on the food type self generated instances with each customer demanding 8, 12, and 16 orders.

Instance			Prim. B.	Dual B.	Gap	Initial Solution		B&B Nodes	Init. Var.	Priced Var.	Time
8	4	2	5980.1	5980.1	0.0%	6370.7	6.1%	6.3	12.0	126.3	0.0 s
		3	5532.4	5532.4	0.0%	5532.4	0.0%	43.8	0.0	2199.5	12.1 s
	6	2	12274.2	12274.2	0.0%	13173.8	6.8%	21.4	22.0	575.3	0.6 s
		3	7677.0	7676.9	0.0%	7694.4	0.3%	165.9	6.0	7951.7	227.2 s
	8	2	21685.4	21685.4	0.0%	22916.1	5.4%	176.4	69.3	998.6	10.8 s
		3	9588.9	9539.0	0.5%	9588.9	0.0%	993.9	23.9	42055.2	2582.3 s
	10	2	24031.6	24030.1	0.0%	26259.8	8.5%	2254.9	140.9	3504.6	389.1 s
		3	12926.2	12153.8	7.1%	12928.4	0.0%	506.4	86.3	25140.1	3612.6 s
12	4	2	14079.2	14079.2	0.0%	14760.2	4.6%	10.3	18.0	367.3	0.2 s
		3	11406.8	11406.8	0.0%	11469.0	0.6%	141.8	10.5	6303.4	72.9 s
	6	2	24019.9	24019.9	0.0%	25159.3	4.3%	19.8	89.5	645.0	1.0 s
		3	11143.1	11134.0	0.1%	11239.0	0.8%	514.1	135.5	19964.8	977.4 s
	8	2	31312.7	31306.2	0.0%	32922.8	4.9%	4117.1	234.2	6062.7	1418.8 s
		3	17955.4	17564.8	1.9%	17962.2	0.0%	741.8	346.0	17733.6	5509.0 s
	10	2	41134.8	41133.8	0.0%	43540.0	5.6%	754.2	354.4	5185.4	352.7 s
		3	25023.2	23862.0	4.6%	25141.6	0.5%	350.4	620.8	30297.6	7105.3 s
16	4	2	18779.3	18779.3	0.0%	19186.0	2.1%	14.0	51.7	475.3	0.4 s
		3	8985.1	8985.1	0.0%	8985.8	0.0%	186.0	27.6	7650.9	212.0 s
	6	2	29930.8	29930.8	0.0%	31159.2	3.9%	57.5	212.0	1286.0	10.1 s
		3	19052.5	19004.0	0.3%	19052.5	0.0%	1001.4	213.0	37246.5	3506.9 s
	8	2	44482.2	44417.7	0.1%	46016.5	3.3%	3894.6	395.6	8328.8	3395.8 s
		3	19580.1	19519.7	0.3%	19670.9	0.5%	577.7	561.4	32407.4	4967.5 s
	10	2	51496.8	51496.8	0.0%	53882.6	4.4%	46.7	615.8	3392.6	33.8 s
		3	29663.8	26896.2	9.4%	29840.1	0.6%	217.8	939.0	47596.4	$\geq$ 2 h

The results of the B&P approach on the petrol type instances are shown in Table 4.5. The table shows three subsets of instances, which differ in the number of compartments of the vehicles. The largest instance in this set with 63 orders has 10 locations with nine customers, each demanding 7 orders, and the depot. This instance is relatively small compared to the food type instances with 144 orders.

Table 4.5: The performance of the exact approach on the petrol type self generated instances with each vehicle having 2, 3, or 4 compartments.

Instance			Prim. B.	Dual B.	Gap	Initial Solution		B&B Nodes	Init. Var.	Priced Var.	Time
2	4	2	6982.4	6982.4	0.0%	7042.6	0.6%	4.6	5.5	103.6	0.1 s
		3	4643.8	4643.8	0.0%	4985.1	7.0%	7.2	5.0	147.0	0.2 s
	6	2	12073.4	12073.4	0.0%	12650.2	4.6%	38.4	16.0	556.9	6.1 s
		3	12607.9	12607.9	0.0%	14215.8	11.4%	344.7	15.5	3368.7	130.1 s
	8	2	16393.3	16388.6	0.0%	16831.3	2.3%	2547.4	40.5	2134.4	1202.6 s
		3	14271.8	14271.8	0.0%	15738.2	9.5%	926.0	41.9	8286.6	870.3 s
	10	2	21737.4	21737.4	0.0%	22831.3	4.8%	803.7	93.6	7300.7	1478.4 s
		3	20744.5	20301.3	2.1%	23693.1	12.5%	631.6	103.5	35981.2	$\geq$ 2 h
3	4	2	6405.6	6405.6	0.0%	6405.6	0.0%	11.9	0.0	354.9	1.2 s
		3	5400.9	5400.9	0.0%	5740.1	5.8%	22.3	0.0	701.8	8.9 s
	6	2	8430.9	8421.7	0.1%	8452.4	0.3%	952.2	10.5	7905.3	1503.1 s
		3	8699.9	8531.8	2.0%	9519.9	8.5%	928.4	10.5	33557.9	5742.6 s
	8	2	13310.2	12663.0	4.8%	13497.3	1.3%	651.5	43.9	26723.2	$\geq$ 2 h
		3	8826.0	8219.1	6.5%	9402.2	7.1%	174.5	42.3	16063.6	$\geq$ 2 h
	10	2	16049.6	15494.6	3.2%	16077.3	0.2%	301.5	126.1	22889.4	$\geq$ 2 h
		3	14221.2	12482.9	12.2%	14755.4	3.5%	23.6	104.1	7001.7	$\geq$ 2 h
4	4	2	3896.9	3896.9	0.0%	3896.9	0.0%	13.4	0.0	443.5	10.1 s
		3	4198.3	4198.3	0.0%	4366.0	3.9%	33.4	0.0	2083.1	202.9 s
	6	2	6811.8	6182.4	9.0%	6886.6	1.3%	225.9	5.0	13899.4	$\geq$ 2 h
		3	4584.5	4351.5	6.0%	4784.1	3.5%	116.9	0.0	11205.8	$\geq$ 2 h
	8	2	9264.5	8376.3	8.4%	9304.6	0.4%	64.4	22.4	5870.1	$\geq$ 2 h
		3	9190.7	3483.0	58.0%	9201.1	0.1%	8.0	30.4	4216.5	$\geq$ 2 h
	10	2	13067.0	11894.5	8.0%	13069.6	0.0%	70.8	78.0	4759.5	$\geq$ 2 h
		3	12708.1	—	∞	12731.3	0.2%	1.0	71.8	4960.6	$\geq$ 2 h

A third set of instances was generated in order to show how the complexity increases when the capacity of the compartments and the vehicles are doubled. For this the small instances with each customer demanding 8 orders are taken and the capacity limits were doubled. Table 4.6 shows the results of the B&P model on these instances. Here the paths in the ESPPRC graph now have approx. twice the number of orders resulting in much more combinations.

Table 4.6: The results of the exact approach on the food type instances with each customer demanding 8 orders and each vehicle and compartment having twice the capacity as for the second set.

Instance		Prim. B.	Dual B.	Gap	Initial Solution		B&B Nodes	Init. Var.	Priced Var.	Time
4	2	3185.3	3185.3	0.0%	3185.3	0.0%	37.4	0.0	1706.5	36.7 s
	3	2978.0	2978.0	0.0%	2978.0	0.0%	7.6	0.0	7072.3	227.3 s
6	2	6505.0	6264.8	3.6%	6586.9	1.2%	619.1	10.0	39616.1	5656.1 s
	3	3924.6	1556.6	48.0%	4014.5	3.0%	1.0	0.0	6916.4	5225.2 s
8	2	11449.6	10955.0	4.3%	11458.1	0.1%	865.3	71.3	75231.5	$\geq$ 2 h
	3	5411.8	—	∞	5419.1	0.1%	1.0	0.0	10063.3	$\geq$ 2 h
10	2	12965.0	11917.1	8.0%	13129.9	1.3%	226.5	136.5	25341.8	$\geq$ 2 h
	3	7087.9	—	∞	7214.8	1.8%	1.0	0.0	15241.6	$\geq$ 2 h

The CP Models for the Bin Packing Problem

The performance of the exact solution approach for the VRPC highly depends on the performance of the cascaded packing model as it is solved each time the label correcting algorithm extends a partial path. The models were tested by generating random sets of demands taken from different test instances provided by Derigs et al. [14].

As the demands per customer in the Derigs et al. datasets do not differ too much, one from each of the three main ordering patterns was used for the tests. The number of demands for each customer does not change for different number of customers. The number of compartments is 5 and the size of all compartments is equal. The following is a short overview on the different ordering patterns of the instances with respect to the packing subproblem:

1. At most 15 random demands, 2 product types and 17 orders per customer. All customers order 9 demands of product type 1 and 8 of product type 2. Each demand is at most as large as half of the compartment size.
2. At most 15 random demands, 3 product types where each customer orders 9 demands of product type 1, 3 of product type 2 and 1 of product type 3. Each demand is at most as large as half of the compartment size.
3. At most 10 random demands, 3 product types where each customer orders 4 demands of product type 1, 2 of product type 2 and 1 of product type 3. Each demand is at most as large as the compartment.

To evaluate the three models, the program selects 100000 random sets of demands from each of the three instances. Each of these sets consists between 5 and 15 random demands of random customers.

The performance of the three different search algorithms that Gecode provides, executed with the three models on three different instances, is shown in Table 4.7. It shows the case

Table 4.7: The running times [ms] of the three CP models for the packing problem. Note: $\exists$ P denotes that there exists a feasible packing scheme and $\nexists$ P means it is not possible to pack the given demands into the vehicle.

Model/Algorithm		Instance 1		Instance 2		Instance 3	
		$\exists$ P	$\nexists$ P	$\exists$ P	$\nexists$ P	$\exists$ P	$\nexists$ P
Model 1	DFS	**14.51**	18489	**14.59**	13861	**10.27**	**857**
	B&B	15.69	18371	15.95	**13468**	11.2	**857**
	Restart	16.83	**18079**	17.36	13518	12.62	872
Model 2	DFS	**17.15**	1432	**17.54**	1660	**12.29**	45.4
	B&B	17.75	1379	18.35	1694	12.42	**43.71**
	Restart	18.32	**1326**	19.16	**1626**	13.16	45.49
Model 3	DFS	**19.6**	**12.4**	**19.3**	**19.22**	**12.29**	8.48
	B&B	25.32	12.52	24.99	19.52	18.39	**8.43**
	Restart	28.65	13.23	29.01	22.36	22.57	9.05
Packing found		70149		69071		45674	
No packing found		29851		30929		54326	

where the solver found a packing scheme and the opposite case where no feasible packing exists. It is observable that model 1, the naive assignment formulation, is the fastest if there exists a feasible packing scheme. The reason for this is the formulation of the first and the last two models. While the optimization models try to improve the solution further, the CP solver terminates when it found a feasible assignment for model 1. Due to the definition of the BPP the CP solver tries to determine the minimal number of compartments. This minimization is not needed for the VRPC, if a feasible packing scheme has already be found.

If it is not possible to pack the demands into the vehicle, the naive formulation needs a lot of time to prove it, whereas the other two models outperform the naive model. Using the `binpacking` constraint, the CP solver is able to make better propagation and branching decisions, as it highly uses problem specific knowledge and bounds: it is possible to calculate simple upper and lower bounds for the number of bins needed. Due to the existence of these bounds the resulting search tree is much smaller.

4.3 Comparison to Related Work

Although the PSO was unable to improve any of the current best known solutions it was able to find several solutions which come close to the BKS. A comparison to the BKS of the Derigs et al. instances can be found in Table 4.2. Table 4.3 shows the performance of the heuristic algorithms on the modified Eilon and Christofides [21, 39, 44] instances with the current BKS.

As to the best of our knowledge no exact approaches for the VRPC have been previously reported in the literature, it is not possible to compare the performance of this B&P approach. None the less the performance can be compared to similar problem definitions of generalized VRPs in terms of the size of the instance that are solved. When doing this one encounters the problem of the different computing power of the machines where the results are obtained and the problem of finding a feasible metric to compare the results. One possibility could be to compare the CPUs using the results of benchmarks like [42]. This gives very limited possibilities to compare as the influence of different compilers and operating systems and their respective versions are not taken into account. Yet this only allows a rather coarse comparison as the time limits given in this section are fuzzy and approximate.

Pirkwieser and Raidl [45] provided a CG approach for the Periodic VRP with Time Windows (PVRPTW) which performs relatively good on all instances. They solved instances with up to 160 customers within a small gap of less than 6.5% in approx. 900 seconds on a single core of an AMD Opteron 2214 running at 2.2 GHz with 4 GB of RAM. They used both, a heuristic and optimal pricing algorithm. Compared to the CPU used to evaluate the VRPC these running times should be multiplied with approx. 0.35 resulting in a time limit of 300 seconds.

Ceselli et al. [8] provided a B&P approach for a general VRP with multiple depots and customers that were arranged in groups according to the corresponding geographical region. Further the customers have time windows and a maximal number of allowed visits per day and rest periods for the driver were kept strictly. Incompatibilities between items in the same vehicle and items and customers were also considered. They evaluated datasets with up to 100 orders, consisting of up to 461 items, and up to 47 locations. The performance of their model was measured in three phases. The first and the second phase were not solved to optimality as only a heuristic pricer was used. In the last phase the optimal pricer provided valid lower bounds. As a side effect the computation time increased remarkably. With a time limit of 4 hours on a Pentium IV at 1.6 GHz, this model was able to solve only a small set of instances to optimality. For the larger instances their approach has similar problems: the huge number of variables already generated in the root node of the B&B tree causes the solver to not provide any dual bound. The CPUs, Pentium IV 1.6GHz and Xeon E5540 2.53 GHz, have a huge performance difference. In order to be able to compare the results the time limit for the VRPC model should be set between 20 and 30 minutes.

Desrochers et al. [16] presented an optimal solution approach for the VRPTW using a B&P model. Their approach was able to solve instances with up to 100 customers with an average gap of 1.5% on a SPARC 1 workstation which gives a relative time limit of a few minutes. In these test instances the customers are not fully connected to each other, which simplifies the instances. In the conclusions they observed that the ability of the exact approaches to solve generalized VRPs decreases drastically with the size and complexity of the additional constraints. Further they proposed that the density of the LP and degeneracy become a problem for set partitioning models as the number of customers in the same route increases. This behavior is observable in the results of the B&P approach for the VRPC too, especially

as the number of demands that can be loaded into the same vehicle increases. Table 4.6 shows the results of the smallest set of food instances with compartments which have a doubled capacity where a huge difference to the results in Table 4.4 with the original vehicle and compartment capacity is observable.

Goel [27] describes a B&P approach with only a heuristic pricer for a generalized VRP. This approach is able to find improved solutions but not to prove their optimality. Heuristically solving the pricing problem decreases the running time of the models by a huge factor. The exact approach for the VRPC spends more than 98% of the total running time solving the pricing problem. By running only the heuristic pricers, the total running time could be decreased substantially as the heuristic algorithms terminate quick and omit variables too. They evaluated their model on a AMD Athlon 400 MHz which results in a relative time limit of less than 10 minutes.

Table 4.8 shows a summary of the different approaches for a general VRP. The first column names the authors, the second the largest instance they considered, which is either the number of locations or orders, the third the average gap over all instances and the last column shows the average relative running time for all instances in seconds. As Ceselli et al. used different stages only the results of the first stage with a maximum of 100 orders is shown in the table. As some approaches use heuristics to solve the model it is not possible to provide any gap. The corresponding average gap values are marked with $---$.

Table 4.8: An overview on the results of different B&P approaches

Authors	Size	avg. Gap	rel. avg. Time
Gebhard	144	3.1%	2674
Pirkwieser and Raidl	160	3.3%	137
Ceselli et al.	100	—	625
Desrochers et al.	100	1.5%	30
Goel	1000	—	10

CHAPTER 5

Critical Reflection and Outlook

This chapter provides a critical reflection on the implemented algorithms and approaches to solve the VRPC. The second part describes some considerations to improve the branch and price model further.

5.1 Considerations about the Approaches

The discrete PSO for the VRPC is able to improve the solutions provided by the GRASP algorithm and is able to solve most of the given test instances within a gap of 10% compared to the best known solutions. Especially on the Eilon and Christofides based instances the PSO was able to solve all, except three instances, within a gap of 10%. The average gap over all Eilon and Christofides based instances is 6.7%. Although the PSO was able to improve the results of the Savings algorithm, it is not really competitive to other algorithms, such as the solvers provided by Derigs et al. [14] and the VNS by Pirkwieser et al. [47] which outperformed the former. A huge problem when configuring the PSO is the large set of parameters. The PSO for the VRPC has 9 parameters, namely the swarm size, the initial velocity, the lower and upper bounds for the influence of the packing on the velocity of the particles, the lower and upper bound for the standard deviation σ and the weights of the own velocity, the velocity of the best particle and the predator in the velocity update function.

The idea to improve the results of an algorithm by iteratively calling a simple and fast LS search heuristic is widely applied on genetic algorithms, which is known as memetic algorithm. The positive effect of the LS optimization step on the results of the PSO is observable on the Derigs et al. instances and further proven by the results of the modified Christofides and Eilon instances.

It seems the performance of exact solution approaches for different generalized VRP models highly depends on the additional constraints to be considered. This approach affirms the

observation of Desrochers et al. [16] that the length of the paths in the ESPPRC graph is crucial to obtain competitive results. On the smallest Derigs et al. instances with 10 locations the algorithm generates more than 50000 variables before creating the first branch. Therefore the MIP solver is able to solve problem instances with a very limited size only.

5.2 Open Issues and Considerations about Future Work

From a personal and educational point of view it was very instructive and interesting to deal with the PSO heuristic and the problems it induced. Since the PSO is originally defined on a continuous domain, it is challenging to define proper conversion rules in order to solve a COP with its discrete domain. Further an algorithm with 9 parameters is generally difficult to fine tune. Machine learning provides several methods to tune parameters while taking care of the problems such as overfitting. With an increasing number of parameters the number of possible combinations increase too and it becomes harder to determine good combinations of values. This is also a reason why the proposed PSO should be seen as a proof of concept and not as an algorithm for a practical use on a daily basis.

The large number of generated variables causes a high memory consumption above 4.0 GB within a few hours. By deleting the variables that are never part of the basis of the LP solution, the memory consumption can be handled. None the less the problem of the large number of variables that are generated by the pricers remains. The performance of the branch and price approach might be improved further by using additional methods that are able to generate improved variables, such as stabilization techniques. Another possibility is to solve the pricing problem by using only heuristic algorithms. Though this approach provides only heuristic solutions, as variables that are part of the optimal solution might not be found during the pricing process.

Including Stabilization Techniques

Column generation models highly depend on the dual variable values as they provide the costs for the pricing problems. Stabilization techniques try to increase the quality of these values, such that less columns need to be generated in total. The goal is to generate only those variables that actually improve the current solution. There exist several approaches that can be divided into two classes: stabilization techniques which modify the RMP and ones that operate only on the dual values. See [22] for an overview on the different stabilization techniques. The goal is to penalize dual variable values that are away from a, so called, stability center.

Stabilization potentially enhances the solving process as it effectively helps to avoid creating variables which do not improve the current solution. Hence this approach prevents the creation of many variables that are deleted later on, when never being part of any solution over a certain number of pricing iterations.

Including a Packing Database

Every algorithm has to calculate a packing scheme for a given set of demands, which is time consuming, especially for the weakly $\mathcal{NP}$-hard bin packing like problem. A potential improvement of the performance of the algorithms could be achieved by using a solution database for the packing subproblem, where the algorithms can look for already computed solutions. One potential solution database is a cache that stores the following information for each set of orders: first a boolean state variable, which tells whether the packing scheme is feasible or none exists, and second the packing scheme for the given set of orders.

Especially on the Derigs et al. instances it is important to use the product type and amount as key in order to prevent symmetry, which could occur if identifiers for each order were used instead. As already mentioned in Section 4.2, many amounts of the different demands of the customer are equal: e.g. for the Derigs et al. instance `vrpc_p_n10_p3_k1_100.vrp` every amount of the demands is one of the values in the set $\{37, 80, 83, 100\}$. The resulting packing cache may cover a major part of all feasible packing schemes with a small memory footprint. The packing schemes are evaluated and stored as they occur during the solving process.

Enhancing CG with CP Models

In recent publications several approaches have been studied where the pricing problem was solved with a CP model. The survey paper of Gualandi and Malucelli [28] gives an overview of the problems and applications with some successful combinations of column generation and constraint programming. They successfully implemented or modified already existing B&P approaches where the pricing subproblem is solved with CP. Some of the problems considered in their work are different routing, timetabling, scheduling and assignment problems. They observed a slower convergence of some of the modified B&P models compared to the original models with an alternative pricing approach. They explain the slow convergence by the generation of too many similar variables due to the nature of the depth first search algorithm, which is widely used in CP solvers, and how it explores the search space. This slow convergence also often occurs when solving B&P models in a classical way and could be handled with stabilization methods.

Since the main goal of the pricing subproblem is to find new negative cost variables and not necessarily the path with the least costs, it is enough when at least one new variable is generated. A CP approach to solve the ESPPRC pricing problem can be formulated by restricting the costs of the path to be less than 0 and terminating if such a path results in a new variable. This transforms the pricing problem from an optimization problem into a decision problem: *does there exist a negative path in* G? Gualandi and Malucelli [28] mentioned the advantages of such a transformation, from an optimization problem to an assignment problem. Further the packing subproblem can be included directly into the problem formulation by constraining the paths in G.

The following binary CP formulation was mainly used for testing, but not considered further as the convergence of the CP model was disappointing. The label correcting algorithm outperformed the CP model in any case. Anyway, an advantage of using the CP model was observed: every time after a new branch was created the CP pricer generated more variables than any other pricing algorithm within a few seconds. Another crucial problem was the huge memory consumption of the CP solver, especially as the number of orders increased. On the small instances with 10 customers and approximately 70 orders Gecode quickly needed more than four GB of main memory. As consequence the operating system started to use the swap partition on the hard disk, which slows the computation by a very large factor.

The CP pricing model (5.1) - (5.17) uses the binary matrices $M = (|D| + 2 \times |D| + 2)$ to describe the path through the ESPPRC graph and $P = (|D| \times |C| + 1)$ for the packing scheme. The constant $|D|$ is the number of demands and $|C|$ the number of compartments of each vehicle in the instance. If w.l.o.g. the value of variable $m_{i,j} \in M$ is 1, the path goes from vertex i to vertex j. The matrix P models the packing of the demands. The last column represents a virtual unconstrained compartment where all demands are packed into, whose corresponding vertexes are not part of the path. Constraint (5.1) causes the CP solver to generate negative paths and acts as the objective function.

$$\sum_{i \in row(M)} \sum_{j \in col(M)} c_{i,j} * m_{i,j} < 0 \tag{5.1}$$

$$\sum_{i \in row(M)} m_{i,j} \leq 1 \qquad \forall j \in col(M) \tag{5.2}$$

$$\sum_{j \in col(M)} m_{i,j} \leq 1 \qquad \forall i \in row(M) \tag{5.3}$$

$$m_{i,i} = 0 \qquad \forall i \in V \tag{5.4}$$

$$\sum_{i \in row(M)} m_{i,j} = \sum_{k \in col(M)} m_{j,k} \qquad \forall j \in V^{-} \tag{5.5}$$

$$\sum_{i \in row(M)} m_{i,v_s} = 0 \tag{5.6}$$

$$\sum_{j \in col(M)} m_{v_s,j} = 1 \tag{5.7}$$

$$\sum_{i \in row(M)} m_{i,v_t} = 1 \tag{5.8}$$

$$\sum_{j \in col(M)} m_{v_t,j} = 0 \tag{5.9}$$

$$u_i * m_{i,j} \leq u_j + |D| * (1 - m_{i,j}) \qquad \forall i \in row(M), \forall j \in col(M) \tag{5.10}$$

$$\sum_{i \in row(M)} m_{i,j} = 0 \rightarrow p_{virt,i} = 1 \qquad \forall j \in V^- \tag{5.11}$$

$$\sum_{i \in row(M)} m_{i,j} = 1 \rightarrow p_{virt,i} = 0 \qquad \forall j \in V^- \tag{5.12}$$

$$\sum_{i \in row(P)} p_{i,j} = 1 \qquad \forall i \in V^- \tag{5.13}$$

$$\sum_{j \in col(P)} w_{d_j} * p_{c,j} \leq CompCapa(c) \qquad \forall c \in C \tag{5.14}$$

$$\sum_{c \in C} \sum_{j \in col(P)} w_{d_j} * p_{c,j} \leq VehCapa \tag{5.15}$$

$$(d_1, d_2) \in IncProdProd \rightarrow p_{c,d_1} \wedge p_{c,d_2} = 0 \qquad \forall d_1 \neq d_2 \in D, \forall c \in C \tag{5.16}$$

$$(c, d) \in IncProdComp \rightarrow p_{c,d} = 0 \qquad \forall d \in D, \forall c \in C \tag{5.17}$$

Constraints (5.2) and (5.3) limit the number of times the path enters and leaves each vertex. Constraints (5.4) eliminate each arc starting and ending in the same vertex. The fourth constraints (5.5) guarantee that each path entering a vertex, except the start and terminal vertexes, will also leave it, with V^- being the set of all vertexes, without the start and terminal vertex. The next four constraints define the start and the end of the path: no ingoing (5.6) and one outgoing (5.7) arc at the start vertex and one ingoing (5.8) and no outgoing (5.9) arc at the terminal vertex. The linking constraints (5.11) and (5.12) define the dependencies between the two matrices. The Miller Tucker Zemlin (5.10) constraints are used to eliminate all sub cycles.

Any vertex that is part of the path must be loaded into a vehicle compartment (5.11). Constraint (5.13) forces every demand to be loaded into a compartment, either a compartment of a vehicle or the virtual compartment. Constraints (5.14) and (5.15) limit the packing to the compartment and vehicle capacities. The last two constraints define the incompatibilities between different demands in the same compartment (5.16) and between the demands and compartments (5.17).

It remains to improve this model by evaluating the cause of the poor performance. For example it might be better to use an integer array to model the path through G. This model would need less variables, but the branching becomes more difficult, since for each variable the size of the domain is at least the number of vertexes in G. Gualandi and Malucelli [28] mention that when using sophisticated constraints and propagators it is possible to improve the performance of the model a lot, which is confirmed by the results of the three different packing models shown in Table 4.7. By extending this model with sophisticated propagators and branching rules the search tree might be shrunk, which would lead to a quicker termination and potentially a competitive pricing model.

CHAPTER 6

Summary

The VRPC is a quite recently proposed routing problem which generalizes the classical VRP by considering several compartments instead of a single one. In this work two heuristics to solve the VRPC are presented. The first is an adaption of the GRASP metaheuristic on the Clarke and Wright Savings algorithm to provide the start solutions for further improvement approaches. The second is a combination of a discrete version of a particle swarm optimization algorithm, which is rarely used to solve combinatorial optimization problems, and a local search heuristic, which proved to be advantageous. None the less the combination of these algorithms provided solutions lying within a small gap, compared to current best known solutions.

The second part of this work describes a Branch and Price approach to provide optimal solutions. This model performs relatively good as long as the number of orders that fit into a vehicle is kept low. This affirms the conclusion of Desrochers et al. [16]. This approach was able to solve instances with up to 10 locations and 16 orders for each customer to optimality or with a small remaining gap.

Further the effect of three different Constraint Programming formulations for the packing problem was analyzed. In the case where a feasible packing scheme exists the naive assignment formulation is faster than the sophisticated propagation and branching rules. In contrast, when the given orders do not fit into a vehicle the first model is far from being competitive.

Bibliography

[1] T. Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[2] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. 2007.

[3] N. Azi, M. Gendreau, and J.-Y. Potvin. An exact algorithm for a vehicle routing problem with time windows and multiple use of vehicles. *European Journal of Operational Research*, 202(3):756–763, 2010.

[4] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research*, 46:316–329, 1996.

[5] R. Bellman. *Dynamic Programming*. Dover Publications, 1957.

[6] T. Berthold. Heuristiken im Branch-and-Cut-Framework SCIP. *OR News*, (32):24–25, 2008.

[7] G. G. Brown and G. W. Graves. Real-Time Dispatch of Petroleum Tank Trucks. *Management Science*, 27(1):19–32, 1981.

[8] A. Ceselli, G. Righini, and M. Salani. A Column Generation Algorithm for a Rich Vehicle-Routing Problem. *Transportation Science*, 43(1):56–69, 2009.

[9] S. Chowdhury, R. Moral, and G. Dulikravich. Predator-prey Evolutionary Multi-Objective Optimization Algorithm: Performance and Improvements. In V. Toropov, editor, *Proceedings of 7th ASMO-UK/ISSMO International Conference on Engineering Design Optimization*, pages 1–10, Bath, UK, July 7-8 2008.

[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] G. A. Croes. A Method for Solving Traveling-Salesman Problems. 6(6):791–812, 1958.

[12] G. B. Dantzig and J. H. Ramser. The Truck Dispatching Problem. *Management Science*, 6:80–91, 1959.

[13] U. Derigs, J. Gottlieb, J. Kalkoff, M. Piesche, F. Rothlauf, and U. Vogel. VRP resources. `http://www.ccdss.org/vrp`. [online accessed 30 September 2012].

[14] U. Derigs, J. Gottlieb, J. Kalkoff, M. Piesche, F. Rothlauf, and U. Vogel. Vehicle routing with compartments: applications, modelling and heuristics. pages 1–30. Springer Berlin / Heidelberg, 2010.

[15] G. Desaulniers, F. Lessard, and A. Hadjar. Tabu Search, Partial Elementarity, and Generalized k-Path Inequalities for the Vehicle Routing Problem with Time Windows. *Transportation Science*, 42(3):387–404, 2008.

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

[17] J. Desrosiers and M. Lübbecke. A primer in column generation. In G. Desaulniers, J. Desrosiers, and M. . Solomon, editors, *Column Generation*, pages 1–32. Springer, Berlin, 2005.

[18] J. Desrosiers and M. E. Lübbecke. A Primer in Column Generation. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors, *Column Generation*, pages 1–32. Springer, 2005.

[19] D. Du and P. Pardalos. *Handbook of Combinatorial Optimization*. Number Bd. 1 in Handbook of Combinatorial Optimization. Kluwer Academic Publishers, 1998.

[20] R. C. Eberhart and Y. Shi. Particle swarm optimization: developments, applications and resources. In *Congress on Evolutionary Computation*, volume 1, pages 81–86, 2001.

[21] A. El Fallahi, C. Prins, and R. Wolfler Calvo. A memetic algorithm and a tabu search for the multi-compartment vehicle routing problem. volume 35, pages 1725–1741, 2008.

[22] D. Feillet. A tutorial on column generation and branch-and-price for vehicle routing problems. *4or*, 8(4):407–424, 2010.

[23] D. Feillet, P. Dejax, M. Gendreau, and C. 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.

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

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.

[26] I. P. Gent and T. Walsh. From approximate to optimal solutions: Constructing pruning and propagation rules. In *In Proceedings of the 15th IJCAI*. press, 1997.

[27] A. Goel. A column generation heuristic for the general vehicle routing problem. In *Proceedings of the 4th international conference on Learning and intelligent optimization*, LION'10, pages 1–9. Springer-Verlag, 2010.

[28] S. Gualandi and F. Malucelli. Constraint programming-based column generation. *4OR*, 7(2):113–137, 2009.

[29] J. Hart and A. Shogan. *Semi-greedy Heuristics: An Empirical Study*. Management science working paper. Center for Research in Management, University of California, Berkeley Business School, 1986.

[30] K. Helsgaun. An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[31] Z. hui Zhan and J. Zhang. Discrete Particle Swarm Optimization for Multiple Destination Routing Problems. In *EvoWorkshops*, pages 117–122, 2009.

[32] S. Irnich and G. Desaulniers. *Shortest Path Problems with Resource Constraints*, chapter 2, pages 33–65. GERAD 25th Anniversary Series. Springer, 2005.

[33] B. Kallehauge, J. Larsen, O. B. Madsen, and M. M. Solomon. Vehicle Routing Problem with Time Windows. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors, *Column Generation*, pages 67–98. Springer US, 2005.

[34] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

[35] J. Kennedy and R. C. Eberhart. A discrete binary version of the particle swarm algorithm. In *Systems, Man, and Cybernetics, 1997. 'Computational Cybernetics and Simulation'., 1997 IEEE International Conference on*, volume 5, pages 4104–4108, 1997.

[36] S. Lin and B. W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21(2):498–516, 1973.

[37] F. Margot. Symmetry in Integer Linear Programming. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 647–686. Springer Berlin Heidelberg, 2010.

[38] R. Motwani. Lecture notes on approximation algorithms: Volume I. Technical report, Stanford, CA, USA, 1993.

[39] L. Muyldermans and G. Pang. On the benefits of co-collection: Experiments with a multi-compartment vehicle routing algorithm. volume 206, pages 93–103, 2010.

[40] T. E. O'neil. Sub-Exponential Algorithms for 0/1 Knapsack and Bin Packing. In *Proceedings of the 2011 Intl. Conf. on Foundations of Computer Science*. CSREA Press, 2011.

[41] C. M. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[42] PassMark Software. CPU benchmark. `http://www.cpubenchmark.net`. [online accessed 30 September 2012].

[43] B. Petersen. Unified Label-Setting Algorithm for variants of the Shortest Path Problem with Resource Constraints, 2006.

[44] S. Pirkwieser. *Hybrid Metaheuristics and Matheuristics for Problems in Bioinformatics and Transportation*. PhD thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, 2012.

[45] S. Pirkwieser and G. R. Raidl. A column generation approach for the periodic vehicle routing problem with time windows. In *Proceedings of the International Network Optimization Conference 2009, Pisa, Italy, 26-29 April 2009*, 2009.

[46] S. Pirkwieser and G. R. Raidl. Multilevel variable neighborhood search for periodic routing problems. In *Proceedings of the 10th European conference on Evolutionary Computation in Combinatorial Optimization*, EvoCOP'10, pages 226–238. Springer-Verlag, 2010.

[47] S. Pirkwieser, G. R. Raidl, and J. Gottlieb. Tackling the Loading Aspect of the Vehicle Routing Problem with Compartments. In L. Di Gaspero et al., editors, *Proceedings of the 9th Metaheuristic International Conference (MIC 2011)*, Udine, Italy, 25–28 July 2011.

[48] J. Pugh and A. Martinoli. Discrete Multi-Valued Particle Swarm Optimization. In *Proceedings of IEEE Swarm Intelligence Symposium*, pages 103–110, 2006.

[49] R. Ravi, R. Sundaram, M. V. Marathe, D. J. Rosenkrantz, and S. S. Ravi. Spanning Trees Short or Small. In *SIAM Journal on Discrete Mathematics*, pages 546–555, 1994.

[50] G. Righini and M. Salani. New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Netw.*, 51(3):155–170, 2008.

[51] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming* . Elsevier, 2006.

[52] L.-M. Rousseau, M. Gendreau, G. Pesant, and F. Focacci. Solving VRPTWs with Constraint Programming Based Column Generation. *Annals of Operations Research*, 130:199–216, 2004.

[53] F. A. Santos, G. R. Mateus, and A. S. Da Cunha. A novel column generation algorithm for the vehicle routing problem with cross-docking. In *Proceedings of the 5th international conference on Network optimization*, INOC'11, pages 412–425. Springer-Verlag, 2011.

[54] A. Schrijver. *Combinatorial Optimization : Polyhedra and Efficiency (Algorithms and Combinatorics)*. Springer, 2004.

[55] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and Programming with Gecode, 2010.

[56] P. Shaw. A Constraint for Bin Packing. In M. Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 648–662. Springer, 2004.

[57] A. Silva, A. Neves, and E. Costa. An Empirical Comparison of Particle Swarm and Predator Prey Optimisation. In *Proceedings of the 13th Irish International Conference on Artificial Intelligence and Cognitive Science*, AICS '02, pages 103–110. Springer-Verlag, 2002.

[58] M. F. Tasgetiren, P. N. Suganthan, and Q.-Q. Pan. A discrete particle swarm optimization algorithm for the generalized traveling salesman problem. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 158–167. ACM, 2007.

[59] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[60] K. Veeramachaneni, L. Osadciw, and G. Kamath. Probabilistically Driven Particle Swarms for Optimization of Multi Valued Discrete Problems : Design and Analysis. In *Swarm Intelligence Symposium*, pages 141–149, 2007.

[61] D. Vigo. VRPLIB: A Vehicle Routing Problem LIBrary. `http://www.or.deis.unibo.it/research_pages/ORinstances/VRPLIB/VRPLIB.html`. [online accessed 30 September 2012].

[62] D. Villeneuve, J. Desrosiers, M. E. Lübbecke, and F. Soumis. On Compact Formulations for Integer Programs Solved by Column Generation. *Annals OR*, 139(1):375–388, 2005.

[63] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.