Urban Logistics Routing Problems: Analysis of Heuristic Methods

Masterarbeit

zur Erlangung des akademischen Grades “Master of Science (M. Sc.)“
im Studiengang Wirtschaftswissenschaft der
Wirtschaftswissenschaftlichen Fakultät der Leibniz Universität Hannover

vorgelegt von

Name:	Igler	Vorname:	Jonathan Christopher
Geb. am:	10.10.1993	in:	München, Deutschland
Prüfer:	Prof. Dr. Michael H. Breitner		

Hannover, den 30. September 2019

Abstract

This paper gives an overview of the recent problems that arose in the field of urban logistics focusing on the methods proposed to solve these problems. For this purpose, we introduce several exact methods, (meta-)heuristics, learnheuristics and hybridizations in a unified fashion. We especially stress the importance of recently proposed state-of-the-art machine learning methods incorporating crucial methods of Artificial Intelligence. Further, we propose a new classification scheme for the categorization of heuristics. This scheme is more extensive and provides more diversification dimensions than common in the current literature.

In both a large-scale simulation study and an application on real word data, we implemented major representative methods of each class in order to be able to make a comparative analysis. Building upon our results, we point out the inefficiency of exact methods as well as the power of a new hybrid, the Gnowee algorithm.

Moreover, we showe that learnheuristics generalize well by incorporating training of metaheuristics for a specified problem. Because of their huge potential, they are fast developing and promise steep progress for future research.

To provide an outlook on future research, we eventually identify prospective challenges facing urban logistics such as an increasing environmental awareness, industry 4.0 and Internet of Things. We hereby stress the social drift to a data-driven society and the opportunities green industry provides urban entrepreneurs.

Key Words: Logistic Problems, Urban Logistics, Combinatorial Optimization, Graph Theory, Heuristics, Metaheuristics, Hybrid Heuristics, Learnheuristics, Machine Learning, Neural Networks, Industry 4.0

Acknowledgments

This dissertation has been a journey and I thank everyone who supported me during the whole time. Special thanks go to Marc Sonneberg for guiding me in the creation process. I would also like to thank Professor Breitner for the possibility to write this dissertation and the inspiration of this research.
I extend my sincerest acknowledgment to all my lecturers at Leibniz University Hanover for their previous assistance, scholarly knowledge and enthusiasm.
Last but not least, I would like to express my indebtedness to colleagues and family who never stopped believing in me and helped me by proof-reading this extensive research piece. Special thanks go to my girlfriend for all her love and support.

Contents

List of Tables **I**

List of Figures **II**

Glossary **III**

1 Introduction and Motivation **1**
1.1 A Machine Learning View on Logistic Solutions 1
1.2 Aim of this Paper 2
1.3 Thesis Structure 3

2 Research Design **4**

3 Combinatorial Optimization Problems **6**

4 Introduction to Logistics Problems **8**
4.1 Relation to Graph Theory 11
4.2 Traveling Salesman Problem 12
4.3 Vehicle Routing Problem 14
4.4 Facility Location Problem 16
4.5 Location Routing Problem 17
4.6 Single Shortest Path Problem 18

5 Exact Methods **20**
5.1 Branch-and-Bound Methods 20
5.2 Branch-and-Cut Methods 22

6 Heuristics **26**
6.1 Constructive Heuristics 28
6.2 Improvement Heuristics 30

7 Classification Scheme **37**

8 Metaheuristics **40**
8.1 Introduction and Basic Notations 41
8.2 Individual-based Metaheuristics 43
8.3 Population-based Metaheuristics 54

9 Hybrids **68**
9.1 Collaborative Hybrids 68
9.2 Integrative Hybrids 70
9.3 Other Hybrids 71
9.4 Gnowee 71
9.5 Concluding Remarks 72

10 Statistical Learning Methods **74**
10.1 Machine Learning 74
10.2 Reinforcement Learning 75
10.3 Ant-Q 75

11 Learnheuristics **78**

11.1 Hyperheuristics . . . 79
11.2 Specifically-located Hybridizations . . . 80
11.3 Neural Networks for Learnheuristics . . . 81

12 Experimental Setup **84**
12.1 Problem Definition . . . 84
12.2 Dataset . . . 84
12.3 Methods . . . 87
12.4 Evaluation Criteria . . . 87
12.5 Further Implementation Details . . . 88

13 Results **89**
13.1 Simulated Data . . . 89
13.2 Real World Dataset . . . 93

14 Discussion **97**

15 Future Challenges and Opportunities in Urban Logistics **99**

16 Conclusion **101**
16.1 Results . . . 101
16.2 Outlook . . . 102

References **104**

A Appendix **118**

B Code **122**
B.1 `simulate_graphs.py` . . . 122
B.2 `extract_API.py` . . . 124
B.3 `bnc.py` . . . 128
B.4 `OR_tools.py` . . . 130
B.5 `show_OR_tools_results.py` . . . 135
B.6 `aco.py` . . . 138
B.7 `run_gnowee.py` . . . 140
B.8 `learnheuristic.py` . . . 144
B.9 `requirements.txt` . . . 145

Ehrenwörtliche Erklärung **149**

List of Tables

2 Methodology as proposed by Peffers et al. (2007) for our own framework . 5
3 Examples for Collaborative Hybrids 70
4 Examples for Integrative Hybrids 71
5 Key differences between global and specifical hybrids of ML and (meta-)heuristics 81
6 Average computational time in seconds for solving the TSP on the simulated data 90
7 Average total distance in kilometers for the TSP on the simulated data . . 90
8 Ranking on the simulated dataset 93
9 Longest traveling time of a single path in seconds on the real world dataset 95
10 Ranking on the real world dataset 96
11 Notations 118
12 Literature categorization 119
13 Comparison between exploitation and exploration of the solution space between Heuristics 120
14 Comparison between exploitation and exploration of the solution space between Metaheuristics 121

List of Figures

1 The development of urbanization in Germany and the European Union . . 9
2 Graph theory exposed 12
3 Methodology on one branch of the BnB 22
4 Methodology of Nearest Neighborhood Search 30
5 Methodology of the 2-opt Lin-Kerningham algorithm 35
6 Classification of (meta-)heuristics proposed by us 38
7 Levels of different solvers 40
8 Methodology of individual-based approaches 43
9 Methodology of Local Search algorithms 44
10 Guided Local Search escaping local minima 46
11 Methodology of Tabu Search algorithms 48
12 Generating a partial solution 50
13 Methodology behind Evolutionary Computation 56
14 Methodology of PSO 58
15 Ant Movement 61
16 Methodology of ACO 63
17 Methodology of ABC 66
18 Methodology of Multi-stage Collaborative Hybrids 69
19 Methodology of Sequential Collaborative Hybrids 69
20 Methodology of Parallel Collaborative Hybrids 69
21 Methodology of Full Manipulation Integrative Hybrids 70
22 Methodology of Partial Manipulation Integrative Hybrids 71
23 Methodology of Hyperheuristics 80
24 Location of Hanover universities in a map extracted from Google Maps . . 86
25 Optimal route computed by the Tabu Search algorithm for an exemplary simulated graph with ten nodes 89
26 Relationships between longest distance taken by one vehicle on the simulated dataset and number of nodes of the graph for most important methods 92
27 Relationships between computational time in elapsed seconds for training on the simulated dataset and number of nodes of the graph 93
28 Time needed for the longest tour on the real world dataset 94
29 Computational time of algorithms on the real world dataset 95

Glossary

ABC	Artificial Bee Colony.
ACO	Ant Colony Optimization.
BnB	Branch-and-Bound.
BnC	Branch-and-Cut.
GLS	Guided Local Search.
LSTM	Long Short Term Memory.
ML	Machine Learning.
NN	Neural Network.
PSO	Particle Swarm Optimization.
RNN	Recurrent Neural Network.
SA	Simulated Annealing.
TSP	Traveling Salesman Problem.
VRP	Vehicle Routing Problem.

1 Introduction and Motivation

With the rise of online trading and the resulting merchandise respectively personal movements, urban logistics problems become ubiquitous optimization problems nearly all modern enterprises have to deal with. The approach to logistics determines crucially the effectiveness of economic action. No matter whether a food delivery provider wants to maximize the occupancy rate of his drivers, courier express parcel service providers aim to minimize costs or the growing urban population necessitates more efficient waste collection, logistic challenges require optimization. Urban logistics, thereby, deal especially with the transport of goods in the business-to-consumer relationship. The urban last mile delivery counts towards the most expensive piece in the supply chain because of high personnel costs (Gevaers et al., 2009).

Especially in times of big data and of a social shift to a more data-driven culture, such optimizations cannot be solved exactly in a reasonable amount of time. The need for fast approximate procedures arises. This becomes even more imminent as a result to the increasingly complex environment these companies operate in, the mounting economic pressure and the growing customer demands. Consequently, couriers, food deliverers and waste disposal companies as well as all other businesses operating in urban areas introduce automate logistical concepts for distance minimization (Van Audenhove et al., 2015).

This development makes the following research question necessary:

How do approximate methods compare for solving urban logistics problems?

This question can be further subdivided into the following issues.

- What are approximate methods companies could use to solve their logistic related questions?
- How does the performance of the methods compare?
- Is there a best logistic optimization concept?
- What are future challenges logistic concepts will face?

1.1 A Machine Learning View on Logistic Solutions

One possible way to solve such problems is by the use of machine learning, to be more specific the sub-area of reinforcement learning. Reinforcement learning methods solve optimization problems by letting an agent learn the effects his actions have on the objective function (Birattari, 2004).

Heuristics were already proposed in the bygone century to tackle these kinds of problems. However, they provide us with merely good initial guesses for not more than specific tasks. The surge of literature on metaheuristics, which can be tailored to solve any optimization problem, can be explained by the demand for more general approximate procedures (Osman and Kelly, 1996).

Metaheuristics explore the large solution search space of optimization problems by restricting the effective size of the search space and sifting through it efficiently, often mimicking natural phenomena. In the case of logistics problems, this means that some possible routes are just neglected and other (more likely optimal) routes are improved iteratively. As a result, these methods are not only flexible and robust, but also scalable and fast (Blum and Roli, 2003).

Nevertheless, most of the introduced optimization algorithms have been proposed decades ago. Since then new developments such as a fourth industrial revolution have changed the environment in which urban routing problems are supposed to be solved sustainably. Industry 4.0 and the Internet of Things impose new challenges, but also opportunities in the field of logistics. For example, the latter provides us with real time information on consumer demand and workflows (Van Audenhove et al., 2015).

In order to train such more complicated models, learnheuristics have been introduced. While traditional metaheuristics only build upon reinforcement learning, learnheuristics also incorporate elements of supervised machine learning. The idea is to learn heuristics to make them even more general and less susceptible to local optima.

1.2 Aim of this Paper

Addressing the lack of literature summarizing the recent developments in the field of logistics problems and comparing the performance of diverse optimization algorithms, this paper aims to provide a thorough review of the available methodology to tackle modern logistics. In particular, we will highlight the surge of metaheuristics for solving routing problems and elaborate on a contrasting juxtaposition of these algorithms, namely exact methods and heuristics. It is important to note that a vast amount of relevant methods is proposed every year. While we exemplary present some recently proposed state-of-the-art machine learning models, we cannot cover all algorithms, but carefully choose the most important, diverse and representative ones to be presented in the remaining part of this paper.

We aim to provide a unified view on the algorithms by presenting their implementation using either pseudo code or visualizations based on the presentation of business processes in order to increase the transparency and the comparability across methods. Further, we

introduce a new framework for classification of optimization methods for solving especially urban logistics problems.

In both a large-scale simulation study and an empirical evaluation, we contrast the particularities of the most renown optimization techniques. Our code is made available in an online GitHub repository.

Last but not least, we address future challenges and opportunities a change in consumer behavior, environmental awareness, Internet of Things and Industry 4.0 provide us with. All in all, we aim to address all the research questions listed above.

Among the main contributions of this thesis are especially

- the introduction of a new framework of classification of heuristics,
- the review of the most important such algorithms and
- the implementation of a large scale simulation study for a thorough performance analysis and comparison across methods in Python.

We base our results mainly on papers published in well-known journals and topic-specific books, but also on internet articles. We included websites as *jstor*, *elsevier* or *Springer Link* for the research of relevant literature. Journals such as *Operations Research*, *Journal of Optimization Theory and Applications* and *Advanced Science and Technology Letters* among others were then taken into consideration. We especially focused to present recently proposed methods, such as *Gnowee* which was only published in 2019, in order to capture the most state-of-the-arts algorithms from artificial intelligence.

1.3 Thesis Structure

The remainder of the paper is structured as follows. We start by summarizing related work in Section 2. We then introduce the common approach to solving logistics problem by referring to the design science research concepts used in this paper in Section 2. In Section 3, we provide the reader with the preliminaries needed for the understanding of Section 4, the urban logistics optimization problems. In the following sections, we introduce several solution concepts and classify them. We also present recently proposed learnheuristics which build upon neural networks, a popular method in artificial intelligence. After elaborating on our experimental setup in Section 12, the subsequent part discusses our results. The last section concludes and proposes future directions for research.

2 Research Design

Design Science provides research with guidelines for evaluation. The design cycle is an iterative process that uses several loops and revises developed design artifacts (Van Aken, 2005).

The research of this thesis is based on design science techniques developed by Hevner and Chatterjee (2010) and Van Aken (2005). Through the publications of Hevner and Chatterjee (2010) and Peffers et al. (2007) describing the framework of design science research, this methodology gained on international acceptance (Gregor and Hevner, 2013). The corresponding concepts are then used in order to attain the best results possible during both the building and evaluation of the methods or implementations (Frauchiger, 2017). In our work, this will be consisting in the usage of different exact methods and (meta-)heuristics.

In the following, Hevner and Chatterjee (2010) propose a framework to classify the underlying methods and three intertwined cycles to be attended in order to attain the best results possible. Those need to be clearly identifiable and will thus be presented in the following, taking our own work into consideration:

- **The Relevance cycle** bridges the contextual environment of the research project with the design science activities.
- **The Design cycle** iterates between the core activities of building and evaluating the design artifacts and processes of the research.
- **The Rigor cycle** connects the design science activities with the knowledge base of scientific foundations, experience and expertise that informs the research project.

Since Hevner and Chatterjee (2010) do not propose a methodology itself but solely a framework, we will further examine the methodology provided by Peffers et al. (2007) as shown in Table 2 on the following page.

Furthermore, our literature overview is based on the methodology proposed by Webster and Watson (2002). Accordingly, our review of relevant work was obtained in three stages, elucidated in the following.

The first step required using key words from the definitions of 'Urban Logistics' and 'Heuristics'. We queried databases of all kinds in order to obtain titles and additional key words, setting no period restriction.

In the following we implemented the work cited in the found journals and articles in order to redeem more information and knowledge. For this we made use of books and already

Step	Section	Description
Problem Identification	Sections 1 and 4	Problem description concerning congestions of urban areas as well as the need in solving such
Defining Objectives of a Solution	Section 1.2	Description of both, solutions needed and quality indicators, in order to increase transparency and comparability
Design and Development	Sections 6, 8, 9 and 11	Exposition of the past, present and future in solvers for the problems defined earlier
Demonstration	Section 12	Setting up our empirical study, describing the dataset and the methodology
Evaluation	Section 14	Discussion of the results obtained in the empirical study, comparing the results obtained by the respective methods
Communication	Whole dissertation, mainly Section 15	Structuring the results and taking our conclusions, including an outlook for further research

Table 2: Methodology as proposed by Peffers et al. (2007) for our own framework

existent literature reviews, such as Toth and Vigo (2002). The last step consisted in finding the actuality of the problem by scavenging newspaper articles and websites.

From this systematic and comprehensive search we could redeem a total of 201 articles all listed in the Section References. Please refer to the Tables 12 on page 119 in the appendix for further insights.

For illustration purposes, we followed the presentation of business processes and visualized more complex methodology as a flowchart in a sequence of commands. Thereby, actions are put in squircles, while events are presented in hexahedron. Simpler methods were then explained via pseudo code.

3 Combinatorial Optimization Problems

Basically, combinatorial optimization problems consist in problems with the goal to find a best solution out of a large but finite amount of possible solutions Consoli and Darby-Dowman (2006).

Further, we can distinguish between two groups: problems consisting of continuous and problems consisting of discrete variables. For this dissertation, we shall only consider the latter one. The main goal is then to minimize or to maximize a given objective function under certain restrictions also referred to as constraints (Neumann and Witt, 2010). For the following, we advise the reader to refer to Table 11 on page 118 for a summarized view on the notations used in this work.

Formally, we define a triple (S, f, Ω) as a combinatorial optimization problem where S denotes a given search space, f an objective function to be maximized or minimized and the set of constraints Ω (Gendreau and Potvin, 2005).

Any optimal solution $s \in S$ is a solution that fulfills the above mentioned constraints Ω. To access the quality of this solution we compare the values obtained with the underlying objective function f. Dependent on the problem a high value (for maximization problems) or a low value (for minimization problems) stands for better quality. Once we find no more higher or lower values, this solution is called a globally optimal solution s_{max}. The finding of this one is main element of combinatorial optimization problems (Papadimitriou, 2003).

Furthermore, the computational complexity theory founded by Cook (1971) was developed to categorize the computational requirements of algorithms in order to be able to classify the problem. We differentiate "easy" and "hard" computational problems (Ausiello et al., 2012). This classification is done in order to find the best suitable algorithm of the respective problem. The complexity of a method is then found by the upper bound of its asymptotic behavior.

The most widely used measure for this classification is the computational time, which measures the time needed for an algorithm to come up with a solution. The computational time is often assumed to be proportional to the amount of elementary operations the respective algorithm executes. The latter is typically expressed with respect to the problem's input size, hereby denoted by n (Papadimitriou, 2003).

We formalize time-complexity as follows. Let $f, g : \mathbb{N} \to \mathbb{N}$. Then $f(n) \in \mathcal{O}(g(n))$ if positive and natural numbers c and n_0 exist such that for all $n \geq n_0$ we have $f(n) \leq c \cdot g(n)$. In words, we say that $f(n) \in \mathcal{O}(g(n))$ if it can be asymptotically bounded by $c \cdot g(n)$. This definition of the limiting behavior of a function is also referred to as big *O-notation* (Floudas and Pardalos, 2016).

An algorithm is then considered to be polynomial or efficient if its time complexity is in $\mathcal{O}(p(n))$ where $p(n)$ is polynomial. If it is not possible to express $p(n)$ as a polynomial function, the algorithm is considered exponential or also inefficient (Floudas and Pardalos, 2016).

We can further differentiate between two types of complexity classes. A problem is said to be in **P** iff it is solvable in polynomial time. In contrast, **NP** (nondeterministic polynomial time) denotes the set of all decision problems to which the validity of a solution can be assessed in polynomial time. It is, therefore, not necessary that the solution is computable in polynomial time (Neumann and Witt, 2010).

According to Neumann and Witt (2010), we define a problem to be NP-hard "if it is at least as difficult as any problem in **NP**". The problem does neither have to be in NP nor does it have to be a decision problem at all. What is more, we say a problem A reduces to a problem B if problem A can be transformed into problem B in polynomial time. Consequently, a problem is NP-hard if every problem in **NP** can be reduced to it.

If one NP-hard problem can be shown to be efficiently solvable, this implies that every single problem in **NP** is solvable. The $P = NP$ problem is one of the seven Millennium Prize Problems, each of which is rewarded with a $1,000,000 prize for its first correct solution. The question is whether all **NP** problems are also in **P** and vice versa (Carlson et al., 2006).

The problems encountered in this dissertation belong all to the class of NP-hard problems (Dethloff, 2001). Furthermore, we only analyze decision problems. These are problems to which a solution can either be correct or incorrect. The traveling salesmen problem, "What is the shortest route between all cities?", is for instance solvable and thus a decision problem (Floudas and Pardalos, 2016).

4 Introduction to Logistics Problems

The European Commission (2013) defines urban logistics as '...the movement of goods, equipment and waste into, out, from, within or through an urban area'. To be more specific, the term 'urban logistics' can be split into the terms *logistics* and *urban*.

Literature provides us with many different definitions of the term *logistics*. In our paper, we follow the 'seven rights of logistics' proposed by Plowman. According to this definition, logistics should ensure the availability of the

... the right product

... in the right quantity

... in the right condition

... in the right location

... at the right time

... for the right customer

... at the right costs (Plowman, 1962).

The production of the goods is neglected in this definition. This means that logistics only deals with the transportation of the inputs into the factory and the transportation of the resulting outputs to the customer. Coherent with the worldwide phenomena of a growing population and the concentration of the latter in urban areas, the so called *urban logistics problems* arise. Those are commonly defined through the growing of cities and importance of travel within these areas (Mulligan, 2006).

Urbanization is defined as the bilateral growth of cities with respect to their area and population size (Gebski et al., 2018). Recent extrapolation resulted in the following numbers: Nowadays, around 52% of the worldwide population inhabits urban areas, a number which will rise to 64% until the 2050s. The same development is to be expected for the travel within such urban areas which occupy around 64% of the total amount of movement kilometers. The expected rise in this case is expected to be at a level of around 300% until 2050 (Van Audenhove et al., 2015).

A possible explanation could lie in the general assumption of both better living and working conditions in such urban areas. Further benefits lie in the improved public supply of electricity, gas and water (Mulligan, 2006).

But not only the moving towards greater urban areas is therefore a problem on the rise for operation research: Further, the demand for goods increases, also related to the growing

trend of e-commerce, the number one driver of urban good delivery (Inninger, 2019). A recent study from the "Bundesverband Paket und Expresslogistik" expects a continuous rise in the amount of parcels of up to 4 billion within the following 4 years (Paket und Expresslogistik e. V., 2019). This increase happens on both sides, B2C and B2B whereas the former takes up both, in relative and absolute terms, the greater amount in parcel distribution.

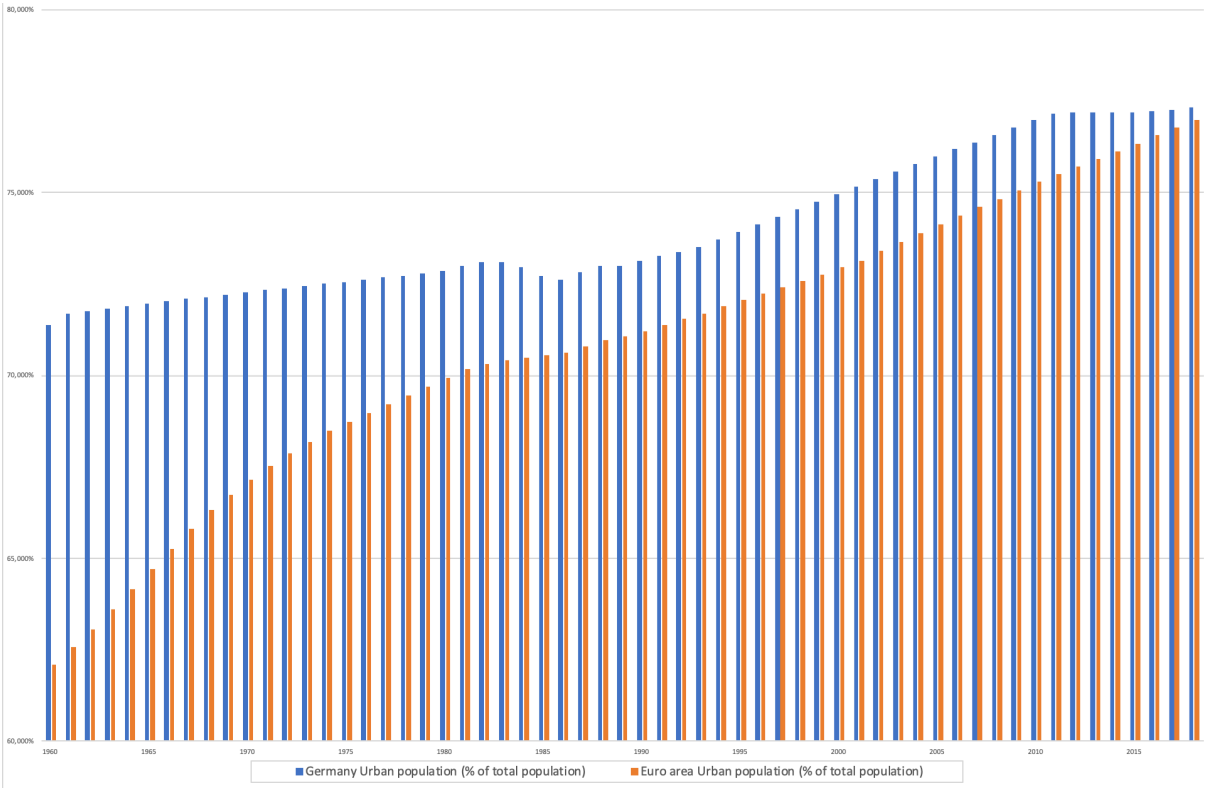

Figure 1: The development of urbanization in Germany and the European Union

The inflating urban population increases the complexity of logistical problems in the most diverse economic sectors, as seen in Figure 1[1]. For example, municipal waste management systems had to be re-developed and adjusted to the greater, more densely living population (Bányai et al., 2019). Furthermore, more and more supermarkets offer delivery services such that not only meals, but also groceries can be conveniently delivered to the house doors of the customers (Gebski et al., 2018).

All in all, this implies that more vehicles traverse already congested urban areas, increasing not only the traffic flow itself, but worsen traffic safety, air pollution, noise and therefore quality of life (Kalnay and Cai (2003), Grimm et al. (2008)).

Recent research has balanced pollution caused by airplanes for the first time on the top

[1]Data retrieved from Worldbank (2019).

ten list of environmental damaging causes in Germany in 2018 (Bann, 2019). Apart from passenger transportation, aeronautics is being used to transport goods, on the rise due to the reasons cited before.

For this, public authorities, transportation firms and retailers, the main stakeholders of urban logistics, see their objectives in the reduction of congestion, amount of cars and trucks in the city, pollution (including CO^2), noise, as well as conservation of energy and development of local retail and housing policies, caused mainly by the increasing urban population (Meixell and Luoma, 2015). The pressure upon those stakeholders increases upon international agreements and protocols, such as the Kyoto protocol or the most recently established Paris agreement (UNFCCC, 2018).

Nonetheless, these stakeholders stand in conflict to each other, resulting from the different goals each one of them tries to achieve (Taniguchi and Van Der Heijden, 2000). Public authorities seize to maximize public utility a goal which is not coherent to the remaining stakeholders who aim to maximize their respective economic wealth by looking only at the relevant costs. Furthermore, the limited possibilities of control make a possible solution finding difficult. While public authorities may only establish regulations, as the recent bans of Diesel-powered vehicles in various German cities, only transportation firms and retailers are in fact able to control the vehicle flow itself (Clarissa, 2019).

As pointed out by Reiche (2019), German transportation firms switch usual trucks for vans, in order to elude tolls and the mandatory breaks, imposed by law for truck drivers. By this, not only safety issues arise but also increases in vehicles in the already congested and polluted urban areas.

Transportation firms and retailers see it as mandatory to satisfy the increasing demands of all kinds. For this it is utterly important to be able to link the different kinds of processes and activities which result in products and services delivered to a final customer (Sachan and Datta, 2005). This is also known as supply chain.

Inside the field of supply chains the problems of the last mile logistics have our main focus. This section is a problem for both, demand and supply side. Within this problem setting, we look at the last part of the delivery of goods up to the front door of the respective client. In other words, the main focus lies on the planning of parcel services which need to satisfy the increased customer demands, the changed behavior of the latter and the promises to be kept made by e-commerce such as “same day delivery“ (Punakivi et al., 2001).

A possible solution would be in tying the separate and individual demands, in order to plan for efficient delivery routes and obtain the lowest transportation costs possible. Additionally there exists the risk of the customer not to be present such that the delivery

service has to return the order postponing the delivery to another point in time causing additional costs. Possible solutions lie in the construction of parcel shop networks which allow the delivery to certain hot spots allowing the customer to be responsible for the successful conclusion of the last mile Song et al. (2009). Another approach would be an improved communication between the customer and the parcel service via apps.

Being the base for the field of operations research, we saw that logistics problems lie within packing and routing problems as well as production scheduling and planning, or decisions over the location of production sites. One of the first problems to appear was the so called Fermat-Torricelli problem in 1636 (Hazewinkel, 2001). In this we take three points as given and shall define a fourth one which has the minimum distance to the three ex-ante defined ones. Even though this problem seems to be out of date, its validation still holds since we could transport this problem to the real world in a situation where we have three demand locations which need to be supplied by a depot to be built.

After a short introduction to *graph theory* we will be introducing the most prominent examples for logistics problems, which are still topics in recent research.

4.1 Relation to Graph Theory

Logistic problems can be simplified to graph theoretical problems: We can interpret the geographical location under consideration as a graph G with its nodes $v_1, ..., v_n$ for some finite $n \in \mathbb{N}$ representing the different stops and (v_i, v_j) denoting the edge between stops v_i and v_j where $i \neq j$ and $i, j \in 1, ...n$. The problems tackled in this work will most often be represented as graphs.

Graphs are uniquely defined by their set of nodes $V = \{v_1, ..., v_n\}$ and their set of edges E (West et al., 1996). We let m denote the number of edges $|E|$. If there are no further attributes, we can thus write $G = (V, E)$. The graph can also be written as the adjacency matrix A with its i^{th} row and j^{th} column equal to 1 if (v_i, v_j) is in E and zero otherwise.

Please refer to Figure 2 on the following page for an exemplary model of a routing problem as a graph. Let us further assume the letters $[A, B, C, D, E, F]$ denote cities and the exemplary distances between those cities is given in meters. One exemplary tour, e.g. $[A-B-F-E-D-C-A]$ with a total distance of $90+120+180+100+200+44 = 836m$. This is only one possibility out of many. The goal is now to minimize the cost and make this route setting more efficient.

In the case of logistics problems it is also possible to assign weight parameters to the edges.

The weight $w_{i,j}$ between nodes v_i and v_j could denote the distance between the corresponding stops. If we let W denote the matrix with edge weight $w_{i,j}$ in its i^{th} row and j^{th} column if v_i and v_j are linked and zero otherwise. In this case, we write $G = (D, E, W)$.

Another extension to allow for node features. Each vertex $v_i \in V$ can be assigned a (possibly Euclidean) vector or single number.

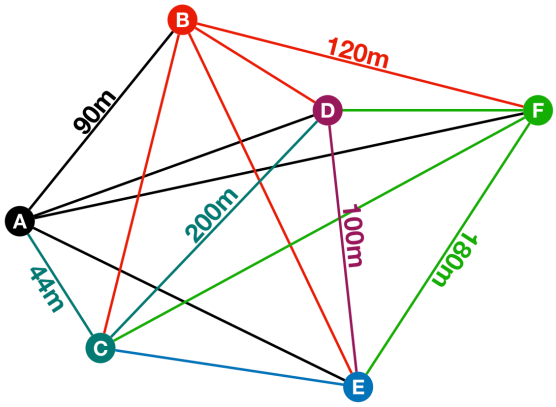

Figure 2: Graph theory exposed

This number can among others stand for a service time that is the time we stay at a stop before continuing.

We further note that this framework allows to present directed as well as un-directed graphs: in un-directed graphs the edge (v_i, v_j) is only in E if (v_j, v_i) is in E. In this case, the adjacency matrix is symmetric. In directed graphs, we also allow for asymmetric adjacency matrices. While un-directed graphs could represent foot paths between stalls on a market, directed graphs can additionally represent the street map of a city with one-way streets (Dethloff, 2001).

4.2 Traveling Salesman Problem

This problem has its origin in both the Seven Bridges of Königsberg proposed by Euler (Euler, 1741) and the Icosian Game by Hamilton (Pegg Jr, 2009).

The former has its origin in a small German city which is divided into five areas which are then connected by seven bridges. The underlying question is now if it was possible to walk through the city crossing all bridges only once and to then return to the starting point. Euler himself affirmed this problem to be unsolvable in this special case (Euler, 1741).

In graph theoretical terms, the bridges represent edges and the five areas are nodes. This problem is also known as the search for a *Eulerian trail*. This describes a closed trail which visits each edge of the given graph exactly once. It can be shown that such a Eulerian trail exists only for Eulerian graphs. Those are described as graphs where all nodes have an even degree (Euler, 1953). The degree of a node is the number of its neighbors. Since the number of edges in the problem of the Seven Bridges of Königsberg is uneven, there is apparently no solution to this question.

Another variation of the Traveling Salesman Problem (further denoted as *TSP*) is the

so called 'Icosian game'. This is a board game which is made up by a wooden board picked with holes. These are connected via segments. The game is about placing the pieces numbered from one to twenty in such a way that two consecutive pieces as well as the initial and end piece are connected by an edge (Pegg Jr, 2009). In graph theoretical terms, we refer to the solution as the so-called *Hamiltonian cycle* that specifies a path where each vertex is visited exactly once.

These two examples describe different views on the same problem setting: the TSP. As we have seen, it can be regarded from an arc-oriented view as in the Seven Bridges of Königsberg but also from a node-oriented view as in the Icosian game (Johnson and McGeoch, 1997).

The TSP is a non-polynomial problem with easy formulation, but difficult and vast solving process Kruskal (1956). Let a set $I = \{1, ..., n\}$ indexing $n \in \mathbb{N}$ locations such as cities be given. The salesman has to visit all of those cities starting from a fixed starting point to which he has to return in the end. Such a path is called *tour* or in terms of graph theory *cycle*.

Traveling from city i to city j, i.e. traversing the edge connecting nodes i and j, the traveling salesman incurs a cost of $c_{i,j}$, $\forall i, j \in I$. The goal is to minimize these travel costs along his route, a.k.a. the distance traveled. Let further $\mathbb{I}_{i,j}$ be the indicator function, being equal to 1 if the salesman travels from city i directly to city j and 0 otherwise. Let $d_i > 0$ for $i \in I$ further denote the positive demand of city i for the goods the traveling salesman sells.

The problem can then be summarized by the following formulas

$$\text{Objective function: } Min \sum_{i \in I} \sum_{j \in I} c_{i,j} \times \mathbb{I}_{i,j} \tag{4.1}$$

$$\sum_{j \in I} \mathbb{I}_{i,j} = 1, \forall i \in I \tag{4.2}$$

$$\sum_{i \in I} \mathbb{I}_{i,j} = 1, \forall j \in I \tag{4.3}$$

$$\sum_{i \in S} \sum_{j \in S} \mathbb{I}_{i,j} \leq Card(J) - 1, \forall S \subset I, 1 \leq Card(J) \leq |\frac{n}{2}| \tag{4.4}$$

Equations 4.2 and 4.3 denote that each city has to be visited once, by departing from another city. Let *Card* further denote the *cardinality* of the equation which is defined as a measure of the elements im the set. The last equation thus allows for the exclusion of sub-cycles such that for example $x_{1,2} + x_{2,1} \leq 1$ eliminates the possibility of the cycle $1 - 2 - 1$, making it thus non-polynomial. Johnson and Garey (1979) have then proved the **NP**-hardness of the problem.

4.3 Vehicle Routing Problem

The Vehicle Routing Problem (further denoted as *VRP*) is one of the most renown combinatorial optimization problems. In general words, the VRP deals is an extension of the traveling salesmen where we have more than one salesman. It was first introduced by Dantzig and Ramser (1959) as a real gasoline delivery problem. Ever since it has gained popularity in operations research (Dethloff, 2001).

We will now specify the problem in more detail. We assume to have several salesmen K which all visit a subset of the same set I of $n := |I|$ customers. Each of these delivery locations is visited exactly once by exactly one salesman. This is thus also called the multiple traveling salesmen problem. The number of salesmen is not necessarily fixed such that it can range from 1 to n. In the case of n salesmen, each salesman would visit only exactly one delivery location.

In the following, we add a depot from where all routes depart from and return to and which has no demand. In this paper, the first city $i = 1$ will be considered as the depot. The goal is to keep the costs and traveling distance at a minimum. In addition, other criteria as minimizing the longest route or only having even routes can be set. Another extension is to constrain the traveling distance of the respective salesmen or vehicles (Dethloff, 2001).

The solution of the VRP is finding a so-called *optimal route*. For this, we find several definitions, ranging from routes with the least total distance to finding a route such that the traveling costs are minimized. In the case of finding no further constraints, this optimal solution would lie in assigning simply one vehicle and finding the shortest route for such. This resembles the traveling salesmen problem (Christofides, 1976).

In order to tackle this, we might modify the definition of the optimal solution: In this, we say that an optimal route is the route, which minimizes the length of the longest single route, concerning all existent vehicles. This definition is needed if the goal is to complete all deliveries in the shortest time possible.

In most problem data we find no location coordinates such that it is sufficient to take the distance matrix into consideration. The latter is available since it is pre-computed. Therefore, the location coordinates are needed just in case one wants to identify the locations in the solution found, denoted by indices i ranging from $[0, 1, ..., n]$, being relevant for visualization only.

Alternatively, assuming now that the salesmen $k \in K$ use vehicles with a maximum capacity of C_k to supply the demand d_i for $i \in I$, we have the capacity-constrained VRP. Parting from the constraints of the TSP we add two more indicator functions, namely $\mathbb{I}_{i,j,k}$ and $\mathbb{I}_{i,k}$. The former takes the value 1 if the vehicle k serves the city j whilst departing

from i and 0 otherwise. The latter takes the value 1 if the vehicle k serves city i and 0 otherwise.

Taking the mathematical concepts presented in the previous section, we increment the following features:

$$\text{Objective function: } Min \sum_{i \in I} \sum_{j \in I} (c_{i,j} \times \sum_{k \in K} \mathbb{I}_{i,j,k}) \tag{4.5}$$

$$\sum_{k \in K} \mathbb{I}_{i,k} = n, \text{if } i = 1, \text{ otherwise } 1, \forall i \in I \tag{4.6}$$

$$\sum_{j \in I} \mathbb{I}_{i,j,k} = \mathbb{I}_{i,k}, \forall j \in I, \forall k \in K \tag{4.7}$$

$$\sum_{i \in I} \mathbb{I}_{i,j,k} = \mathbb{I}_{i,k}, \forall j \in I, \forall k \in K \tag{4.8}$$

$$\sum_{i \in I} d_i \mathbb{I}_{i,k} \leq C_k, \forall k \in K \tag{4.9}$$

$$\sum_{i \in S} \sum_{j \in S} \mathbb{I}_{i,j,k} \leq Card(J) - 1, \forall S \subset I, 1 \leq Card(J) \leq |\frac{n}{2}|, \forall k \in K \tag{4.10}$$

In this setting, we want to minimize the total distance covered by all the vehicles (equation 4.6). Equation 4.7 sets that every city has to be covered by one vehicle, while the depot is served by all vehicles whereas 4.8 and 4.9 guarantee that a vehicle serving one city has to depart from another and will do the same after serving the city it is currently located in at the moment. As in the traveling salesmen problem, the last constraint, namely Equation 4.10, is to eliminate any possible sub-cycle making it thus non-polynomial Lenstra and Kan (1981).

Variations exist such that a capacity constraint is met. In this case, we add a capacity C such that $C : \sum_{i=1}^{m} d_i \leq C$ holds. In this we affirm nothing but that the demands d_i along the route do not to exceed the capacity of the vehicle.

Further if one aims to minimize the length of the longest route, the objective function would be $Min(max_k(\sum_{i \in I} \sum_{j \in I} (c_{i,j} \times \sum_{k \in K} \mathbb{I}_{i,j,k})))$.

The real world often requires to add some additional constraints. For this purpose, extensions of the basic problem have been proposed. For example, heterogeneous VRP (Roberto et al., 2007) allow to differentiate between several types of salesmen. Solomon (1987) introduces time windows which specify that customers must be served in a given period. Delivery plans can also be optimized for several days at once, for instance, in the periodic VRP (Russell and Igo, 1979).

4.4 Facility Location Problem

The main goal of this problem is to find the best possible placement of one or more facilities, such as the aforementioned depots. Further, this should be the initiating of an effective supply chain which is defined to aid in the minimization of combined costs, both of transportation and building (Shmoys and Aardal, 1997).

This depot is designed to serve the demand of a given set of n customers. Building one location is further related to a given set of costs of o_i where i stands for the facility to be build, as we do not impose the building of the same facility. In the following we assign each customer j to one of the depots.

By this, we impose distribution costs $tr_{i,j}$ resulting from the transportation of a good from facility i to the customer j. The goal is now to find a set-up such that we find the optimal locations for all facilities, defining which client is to be served by which facility minimizing both the costs of building and distribution (Shmoys and Aardal, 1997).

Furthermore, the main components of this problem can be divided into two subgroups, explained in the following. This first case denotes the group of static and incapacitated facility location problems:

$$\text{Objective function: } Min \sum_{i \in I} o_i Y_i + \sum_{i \in I} \sum_{j \in J} tr_{i,j} \mathbb{I}_{i,j} \tag{4.11}$$

$$\mathbb{I}_{i,j} \begin{cases} 1 & \text{if facility i serves customer j} \forall (i,j) \in I \times J \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

$$\mathbb{I}_i \begin{cases} 1 & \text{if facility i is open } \forall i \in I \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

$$\sum_{i,j} \mathbb{I}_{i,j} = 1, \forall j \in J \tag{4.14}$$

$$\sum_{j \in J} \mathbb{I}_{i,j} \leq Card(J) \times \mathbb{I}_i, \forall i \in I \tag{4.15}$$

Further, this model can be extended by setting an individual maximum capacity of C_i for each facility i whereas the demand per customer d_j needs to be supplied. To implement this in our initial problem, we need to exchange 4.15 by $\sum_{j \in J} d_j \mathbb{I}_{i,j} \leq C_i \mathbb{I}_i, \forall i \in I$.

In another case we need to assure the amount k of facilities to be built by adding $\sum_{i \in I} \mathbb{I}_i = k.$ to the equations.

Now, we will set dynamic models which bring in periods of time throughout which the supply chain is able to evolve. In this model, this evolution is set in such a way that

facilities might open or close over time, as shown in the next equations.

$$\text{Objective function: } Min \sum_{t\in T}\sum_{i\ inI} o_i \mathbb{I}_i + \sum_{t\in T}\sum_{i\in I}\sum_{j\in J} tr_{i,j} X_{i,j,t} \tag{4.16}$$

$$\mathbb{I}_{i,j,t} \begin{cases} 1 & \text{if facility i serves customer j in period t } \forall(i,j,t) \in I \times I \times J \times J \\ 0 & \text{otherwise} \end{cases} \tag{4.17}$$

$$\mathbb{I}_{i,t} \begin{cases} 1 & \text{if facility i is open } \forall(i,t) \in I \times T \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

$$\sum_{i,j} \mathbb{I}_{i,j,t} = 1, \forall(j,t) \in J \times T \tag{4.19}$$

$$\sum_{j\in J} \mathbb{I}_{i,j,t} \leq Card(J) \times \mathbb{I}_{i,t}, \forall(i,t) \in I \times T \tag{4.20}$$

$$\mathbb{I}_{i,t} \geq \mathbb{I}_{i,t+1}, \forall i \in I_c, \forall t \in T \backslash \{NT\} \tag{4.21}$$

$$\mathbb{I}_{i,t} \leq \mathbb{I}_{i,t+1}, \forall i \in I_o, \forall t \in T \backslash \{NT\} \tag{4.22}$$

In this setting we differentiate between the closed facilities I_c denoting the group of facilities that may be closed and I_o the group of facilities to be opened. Let further t denote the period of time. Equations 4.17 and 4.18 are merely increased by the time component. Equation 4.21 is augmented by $\mathbb{I}_{i,t}$ in order to guarantee a decreasing sequence for already built facilities, only able to shut down after time t. Equation 4.22 guarantees that the previously sequentially ordered facilities $\mathbb{I}_{i,t}$ form an increasing sequence for other potential facilities to be built over the time horizon.

4.5 Location Routing Problem

If we consider now all three aforementioned problems together, we get to the Location Routing Problem. In its essence, we seek the optimal amount, capacity and location of facilities. In order to be able to supply the overall demand, we further need to determine the optimal amount of vehicles and routes from those facilities (Floudas and Pardalos, 2016).

Moreover, we look at the difference to the simpler version, the Location-Allocation problem. In this the client is visited from the vehicle in direct form and once visited, the vehicle returns to the facility without serving any other client, while in the current problem we assume tours (Cooper, 1963).

Usually, we assume the facility assigned to a customer to directly serve the latter on his or her own route. As this is rather inefficient, we consolidate the customers to such groups that the total distance traveled or the amount of vehicles needed is reduced (Floudas and

Pardalos, 2016).

This implies more decisions to be taken from the model, as well as deciding on

... the amount and the location of the respective facilities.

... the assignment of clients to certain depots and routes.

... the order in which the clients shall be served.

The whole system is designed in such a way that candidates of those facilities are located on candidate sites. Then, delivery routes are fixed for a given set of clients in such a way that the overall costs are minimized. In this setting, our aim is to minimize distribution costs, since we assign vehicles to consumer groups apart from the main objective of designing those routes (Laporte, 1987).

All in all, all problems presented in this section belong to the group of non-polynomial problems. In real world conditions, all of the aforementioned problems face more restrictions such as size limitations and restricted locations.

4.6 Single Shortest Path Problem

The simplest logistics problem one can imagine is possibly the single-pair shortest path problem (Neumann and Witt, 2010). It is simply the question of the shortest path between two locations. Formulated as a graph-theoretical problem, we search for the path connecting two vertices with the smallest sum of weights of its constituent edges.

A parallel problem is the so-called single-source shortest path problem where we have to find the shortest path between a source location v_0 respectively source vertex v_0 to all other locations respectively vertices on the map respectively in the graph. The single-destination shortest path problem, in contrast, asks for the shortest paths from all vertices to a single destination vertex. The solution only differs from the solution of the single-source shortest path problem if we regard directed graphs. In the all-pairs shortest path problem, we search for the shortest paths between every pair of vertices in the graph (Fu et al., 2006).

The single-pair shortest path problem, the single-source shortest path problem and the single-destination shortest path problem can be solved using the algorithm proposed by Dijkstra (1959). Let v_0 denote the source node. We label all vertices with ∞ and the empty path $\{\}$ and say they are unvisited. Then, we iterate over the neighbors of each order to update the first labels to the minimal distance from v_0 and the second label to the shortest path connecting the node to v_0.

The minimal distance of v_0 to v_0 is zero which is smaller than ∞. The first label of v_0 is thus updated to 0, while the second label is set to v_0. We say v_0 is now visited. For each neighbor v_x of v_0 the minimal distance is the edge weight connecting the nodes which is smaller than ∞ and the second label is thus set to $v_0 \rightarrow v_x$. In a next step, we iterate over all neighbors of v_0. Let v_x denote one such neighbor. If the path from v_0 over v_x to the neighbors of v_x is smaller than their current labels, these are updated. The second label constitutes hereby the smallest path from v_0 to v_x continued by the corresponding neighbors of v_x. We can now say v_x is visited. The algorithm continues until all vertices have been visited.

This method requires to compute the distance of n nodes to all of their neighbors. Since a node cannot have more than $n-1$ neighbors, its computational requirement is in $\mathcal{O}(n^2)$. As a result, the single-pair, the single-source and the single-destination shortest path problems are thus in **P**. Following the algorithm proposed by Fredman and Tarjan (1987), the time complexity of the corresponding problems can be even reduced to $O(m+n\log n)$ where m denotes the number of edges in the graph (Dijkstra, 1959).

The all-pairs shortest path problem can be solved as n distinct single-source destination problems. As the function n is polynomial, we can say that the all-pairs shortest path problem reduces to the single-source destination problems. As a result, its computational complexity is also polynomial ($\mathcal{O}(n^3)$). This problem is thus also in **P** (Pearl, 1984).

For this increasing complexity, linear modeling might not be sufficient such that we will use both exact methods and (meta-)heuristics, as introduced in the following sections.

5 Exact Methods

Even though logistics problems are typically NP-hard, there still exist some exact solution methods. Those calculate every possible solution and its respective outcome followed by a comparison of the results from which the best one is chosen. Even though this procedure can be accelerated by dynamic programming, the problems remain NP-hard. It might seem logical, why this kind of methods lack in practical usage: Since the required amount of calculation, complexity and computational time needed is immense, it is rather inefficient (Floudas and Pardalos, 2016).

Eilon et al. (1974) were the first to use dynamic programming in the context of VRPs. The used Branch-and-Bound as performed by Christofides and Eilon (1969). This approach will be elucidated in the following.

5.1 Branch-and-Bound Methods

Among popular exact methods are the so called branch-and-bound (further denoted as *BnB*) methods. Those methods have the advantage of being flexible for many types of problems. The initial idea is to mark all possible solutions in an enumeration-tree. This is problematic, since the amount of solution increases exponentially with bigger problems Christofides and Eilon (1969).

For this we divide the solution space S into smaller parts thus creating so-called *sub-problems*. The latter are then optimized individually. By the estimating upper and lower bounds of the quantity being optimized, we are able to discard of certain sub-problems is achieved (Lawler and Wood, 1966).

The following step, the so-called *bound phase*, denotes the relaxation of the problem, e.g. admit solutions which do not belong to the feasible set of solutions initially set in S. This can be done by not taking into consideration the constraints. When solving this relaxation, we obtain a so-called *lower bound* on the value of the optimal solution (Christofides and Eilon, 1969). By these bounds we differentiate between lower and higher bounds of solutions which can be denoted as limiting factors to the solution in terms of accuracy (Floudas and Pardalos, 2016).

In the following analysis, we look if the latter s^* is part of S or if its costs are equal to the costs of some $s \in S$. If this is the case, we can assume our new solution or s^* to be optimal. In the opposite case, we create n subsets of S such that $\bigcup_{i=1}^{n} S_i = S$ where each S_i is denoted as sub-problem. The latter are then added to a group of possible solutions or solutions yet to be examined, a process called *branching*. In the following, we choose one of those candidates in order to process it (Lawler and Wood, 1966).

During this procedure, we will be confronted with the following four cases:

- Finding a better solution than s^*. The quality of a solution is implied by the objective function f. Iff $f(s^*) \geq f(s)$ where s^* is the current solution, also denoted as the *incubent* or *current solution*. In that case s gets replaced by the solution of the sub-problem s^*.
- For the case that the sub-problem has no solution, we prune it.
- Otherwise, we compare the lower bound of the sub-problem to the global upper bound. The latter is obtained by the value of the best feasible solution found yet. In the case that the lower bound being greater or equal to the global upper bound, we discard the sub-problem, since this means we have no room for improvement left.
- If the sub-problem cannot get discarded, we branch and add the sub-problem to the list of active candidates. We proceed with this, until the amount of candidate sub-problems is zero. The current best solution is then said to be optimal.

By this, we work ourselves through the enumeration tree, without the need to take every single possible solution into consideration. Often it is possible to discard around 90% of unnecessary solutions by this procedure (Laporte, 1992).

For the most prominent case of the TSP, the authors continue in the following way. They needed to construct a solution tree, based on a cost matrix. In this, the costs of traveling from node i to i are set to zero. In the following, the lower bound of the root node, which is nothing else but a tour without constraints is calculated. For this the row-wise sum of the two smallest costs is taken.

From this, two sub-problems are generated with the constraint that a specific tour must or must not be contained, e.g. the tour $[1; 2]$. For each sub-problem the costs are then calculated, e.g. a tour with and one without $[1; 2]$. For the lower bound of the former, we sum the $c_{1;2}$ and the row-wise lowest costs. The procedure is similar to the calculation of the root node's lower bound, as done beforehand.

Then the lower bounds of both sub-problems are compared and the branch with the lower value is chosen, proceeding analogously with other tours. If both happen to have the same lower bound, we need to continue with both sub-problems, branching them to the solution tree.

Noticeable is that even though we do not compute the whole solution tree, the computational effort this method undertakes is immense. Please refer to Figure 3 on the next page for a graphical insight.

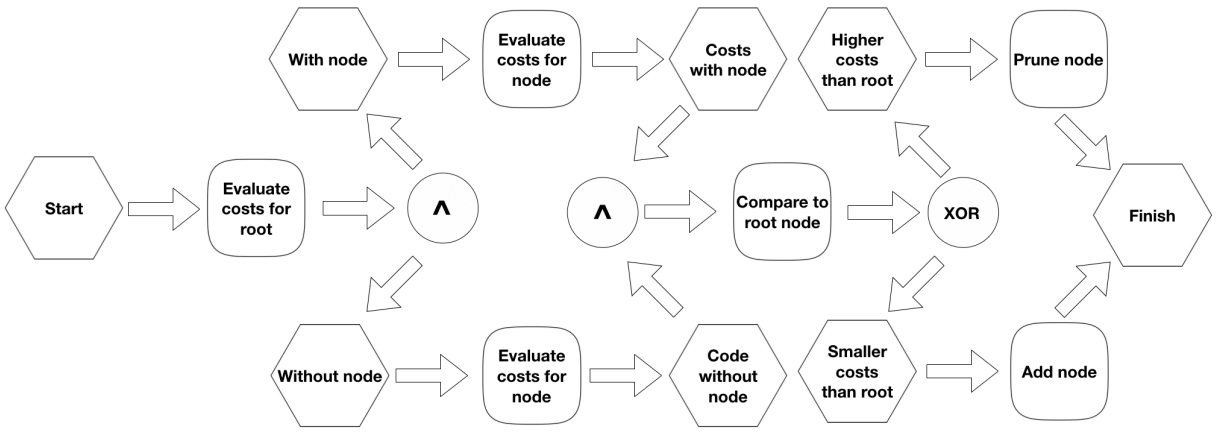

Figure 3: Methodology on one branch of the BnB

Even though for smaller problems BnB is successful with growing problem instances, solution finding gets inefficient. In order to tackle this, the branch-and-cut method was introduced, as elucidated in the following.

5.2 Branch-and-Cut Methods

Combining the methods of cutting-planes and the aforementioned BnB methodology yields in the branch-and-cut (further denoted as *BnC*). Due to the initial cutting phase, it is much more powerful than its elder brother, the BnB (Padberg and Rinaldi, 1991).

The idea of cutting planes was introduced by Dantzig et al. (1954). With this, Padberg and Rinaldi (1991) solved the TSP by iteratively adding cuts. These cuts shrink the feasible solution space, without the deletion of possible solution tours.

This procedure of solving the linear programming relaxation and search for cuts is done as long as it takes to obtain a feasible solution for the original problem. The cutting plane is used until no valid inequalities can be found in the solution space anymore (Rothlauf, 2011). Further we store the best solution for the original and relaxed problem respectively. After this, we start with the already explained BnB-methodology.

The BnC was originally designed to solve the TSP by passing each customer and depot only once. When augmenting it, it is able to solve the VRP by simply adding all individual vehicle tours into one. By doing so we can assume that the number of visits to the depot is as high as the amount of existent tours. Through this we can now reformulate the VRP as a TSP by adding artificial depots all located at the same location and excluding the actual depots (Christofides and Eilon, 1969).

The amount N of those artificial depots is equal to the amount of vehicles employed. In the following, we prohibit traveling from one depot to another. This is done by setting

the distance and the resulting costs between them to ∞. From a cost matrix coherent to the capacity constraints of the respective vehicles a solution is then computed (Puchinger and Raidl, 2005).

This number has a lower bound, which is determined by the vehicle capacity C. If we denote d_i as the respective customers demand, the lower bound results from $N \geq \sum^n \frac{d_i}{C}$. The authors solve the problem now for several N and choose the best suitable solution. Christofides and Eilon (1969) argue that there is no need to implement more than four values for N, when starting with the lowest value possible and increasing it by one per trial.

One could argue against the feasibility of the lowest or even some higher values of N, but as we increase N successively, feasible solution can result in the process, from which conclusions are then taken (Christofides and Eilon, 1969).

Before we proceed the branching of another node, we must make sure that the following constraints have been met: The maximum capacity of the vehicle in question is not to be surpassed by the load proposed by the solution as well as guaranteeing that the total accumulated distance is within the limits of the pre-defined boundaries. Furthermore, considering the k^{th} tour such that $N - k$ tours remain to be planned one must guarantee that the demand of the remaining n customers does not exceed the capacity constraints of the $N - k$ vehicles, namely $\sum^n \frac{q_i}{C > N - k}$.
Christofides (1976) argue if the answer of only one of this constraints is affirmative, we can skip the further exploration of this branch and set off to another.

While we can only consider one of either capacity or distance constraints, we need to take further action to determine bounds for the decision tree. By this we can reduce the search space in order to gain efficiency.

Christofides and Eilon (1969) calculate a so-called *minimal spanning tree*. They proceeded in the following logic: As the route is now a closed cycle with n points such that each point has two links to other points. The authors then define a spanning tree, which is a '...configuration of $n - 1$ straight lines passing through the n points...' and a minimal spanning tree being the shortest sum of those links.

If we take the loop of the tour and remove one link such that $n - 1$ connections remain, we obtain the spanning tree. The result is then a lower bound solution for the spanning tree. Following from this, the minimal spanning tree is also lower bound. As we have to obtain n links, we add the missing one, by linking the points of shortest distance to each other.

The authors then add one point at the same location as an already existing point with the difference that the distance between them is set to ∞. Resulting from this, we acquire

$n + 1$ points. The new minimal spanning tree resulting from this is then a lower bound for the original problem consisting of n points Christofides and Eilon (1969).

Using the method proposed by (Kruskal, 1956), we are now able to come up with the minimal spanning tree of this problem. We repeat the selection of the shortest edge, without forming any loops to edges already chosen, until no edge is left, similar to the neighborhood heuristic explained in Section 6.1.2 on page 29.

We won‘t go any further in detail, but just notify that this is the approach of the BnC procedure, originally proposed by Little et al. (1963) and improved by the authors mentioned before with the help of the methodology by Kruskal (1956). As in the BnB we see that the computational effort is immense, but still used in practice and when it is not needed to come up with feasible solutions quickly (Floudas and Pardalos, 2016). Yet, it is not problem-specific and can thus be used in a variety of problems, making it somehow superior to normal heuristics.

However, even with the most efficient solution method, the best exact algorithm can only solve some instances with less than 200 vehicles via progressive improvement algorithms. Nonetheless, Applegate et al. (2006) computed a solution with a total of 85.900 cities with a BnB and a problem-specific BnC method. Even though their algorithm proved to be successful, it took them 136 CPU-years in order to obtain results.

As the complexity of these problems increases, difficulties in finding feasible solutions in reasonable computational time arise. In a world where time efficiency is of great value it urges to find good solutions in reasonable time, guaranteeing feasibility or optimality (Toth and Vigo, 2002). Nonetheless, exact methods are known to be very effective in solving constraint satisfaction problems. Those consist in tackling hard constraints (Puchinger and Raidl, 2005).

In order to attack problems such as the urban logistics problem explained in Section 4, it is required to obtain a method as a planning instrument which allows us to calculate the desired output needed to achieve the best solution possible (Domschke et al., 2006).

The main components of planning are listed in the following: Accessing and identifying the problem itself and the starting point. From this the following plans of action shall be derived. The latter are sorted in such a manner that each one of them leads to different outcomes, submissive to a given preference or objective function (Streim, 1975).

These plans of action are then considered to be decision variables itself and the existent connections between the latter and the main goal do not need to be simple and can appear to be contending. Whilst simpler connections might be solved analytically, more complex or stochastic systems require simulations in order to rank outcomes, facilitating further

decision making (Thorngate, 1980).

Furthermore, forecasts might be used (Bányai et al., 2019). In practical usage combinatorial problems build a great part such that discrete alternatives of action are of main interest, consisting of selection, assignment and order decision problems (Pearl, 1984).

In such a model, it is utterly necessary to use optimization tools which provide help or even pick the best plan of action (depending on what outcome is desired). This could be achieved by the usage of so called optimized heuristics which appear to have taken over exact programming, such as the computational demanding BnB methods (Lawler and Wood, 1966).

Throughout this thesis, the main components of heuristics and their way of functioning shall be elucidated. We will be defining pure heuristics and pure metaheuristics separately giving the subgroup of learnheuristics in the field of metaheuristics special attention. The importance to introduce pure heuristics arises from the fast that these algorithms are typically used as first solution strategies for metaheuristics.

6 Heuristics

Derived from the Greek word 'heuristikein' which translates to 'find' or 'a guidance to reach a methodical way to knowledge' (Sörensen et al., 2018). In other words, the underlying idea of heuristics lies in solving a problem by experience (Bingham and Eisenhardt, 2011). Often denoted to be 'a rule of thumb', it certainly does not deliver the ideal solution, but nevertheless solves the problem in a more or less efficient way (Pearl, 1984). Further, the implementation of heuristics is rather easy and usually not limited for bigger problem sizes (Streim, 1975).

The acquired knowledge by the implementation of heuristics is often problem specific. Developed and plausible plans of action are currently being formalized and made public via software (Domschke et al., 2006). Example for those are CPLEX, GAMS and Googles OR Tools, among others. This happens through the relinquishing of beforehand mentioned optimization criteria which is mainly defined in such a way that a small computation time and simple implementation can be achieved. This concept will be elucidated throughout the section.

The main problem lies in the identification of easy heuristic plans of action which deliver the best outcomes based on previously defined criteria or constraints (Pearl, 1984). Commonly, heuristics are mostly based on so called *greedy strategies*: Within those, solutions are being constructed piece-wise, following a Markov-process: Each decision node depends on the previous ones (Kennedy, 2001). At every point we opt for a decision that is at least a local optima hoping that it leads to a globally optimal solution. All in all we can say that this kind of algorithms represent a group of iterative heuristics that typically make the most optimal choice in each step as they attempt to find the globally optimal solution (Laporte and Semet, 2002).

To move from this moment on, it is necessary to choose the optimal decision for the underlying node individually. This is also known as *Single Pass Heuristic* (Fisher and Rinnooy Kan, 1988). Please refer to Section 6.1.1 on page 29 for a more detailed explanation.

Further, in the field of heuristic algorithms the justification is based on arguments of plausibility, rather than mathematical proof (Toth and Vigo, 2002). Different from exact solution achieved via mathematical methods, heuristics can neither determine nor guarantee the optimality of both the underlying method and the solution obtained through it. For this particular reason, they are often denoted as 'sub-optimal' procedures (Toth and Vigo, 2002).

In order to access the quality of the utilized heuristics and for the sake of comparison we rely on the following four definitions, accuracy, speed, simplicity and flexibility (Cordeau

et al., 2002).

- **Accuracy** captures the degree of departure of a heuristic solution value from the optimal value. Difficulties arise on capturing the latter specially in cases as the traveling salesman or VRP. Additionally, this measure is often confronted with consistency: It is better to obtain only good results most of the time with some heuristic than opting for heuristics that performs better most of the time, but very poorly in other occasions (Pearl, 1984).

- **Speed** is, as already pointed out in the previous sections, crucial. The importance of this measure is not fixed but rather depends on the problem itself. For example an application for a police redeployment needs much faster calculations than a depot localization problem (Gendreau et al., 2001). We measure speed via clock time or GPU-time needed to come up with a solution.

- **Simplicity:** Apart from numerical measures, programmers aim for simple and easy to implement code. As we argued beforehand, a code should be both easy to understand as to be implemented. Let us assume the Clarke and Wright algorithm from Section 6.2.2 on page 32 (Clarke and Wright, 1964). One of the reasons for its popularity lies in its simplicity.

  But not always 'the simpler the better' holds, as we have to deal with the trade-off raging between complexity and good results. In the heuristics presented in this section we see that over time the amount of parameters increases steadily such that the methods sacrifice simplicity. In order to tackle this problem, we could either set certain parameters to a fixed number or by giving them the possibility to self-adjust.

- **Flexibility** aims to build heuristics in such a way that they are able to adapt to new constraints added for real-life-application reasons. This results normally in performance deterioration by adding new side-constraints. Clarke and Wright (1964) propose to deal with this problem with two steps. A function $F(x)$ computing the routing costs of the solution s. A second function $F'(x)$ summing up $F(x)$ and adding a weighted penalty term associated with violations of each additional side constraint, thus penalizing overfitting.
  In order to exemplify the last point, let us assume a vehicle with capacity C_i and a route duration of $D(x)$ with solution s. In this case, $F'(x) = F(x) + \alpha Q(x) + \beta D(x)$ whereas α and β denote positive self-adjusting penalty parameters. We start by giving them the value 1 and periodically increasing or decreasing them. This happens according to previous solution's feasibility. Through this, we make sure that the algorithm is neither overfitted nor stops evolving over time such that it

gets trapped to local minima. By this we could thus show in a simple way that flexibility and simplicity go hand in hand. By local minima we denote solutions in the solution space, better than all their neighbors, but not necessarily representing the best solution possible, which would then be the global optimum.

Decisive for the success in practical usage are the searching elements which need to be suitable in the processes of *exploitation/intensification* and *exploration/diversification*, following the definition of Osman and Kelly (1996). In other words, these methods describe the way we move through the solution space S. Please refer to the Tables 13 on page 120 and 14 on page 121 in order to get a summarized comparison between the respective techniques used.

By the former we denote focusing the search process in a specific and therefore most promising solution area. This means that we exploit knowledge which has already been acquired beforehand. The latter one denotes the diversification of the searching process in the solution space itself, also in yet unknown areas. Several solution areas shall then be compared which happens in a rather randomized way (Sevaux et al., 2018).

With regard to the use of these two complementary concepts, one can roughly differentiate between an explicit sequence of intensification and diversification or the entanglement elements in the application of the corresponding search operators (Arora and Gigras, 2013).

In the following, we will be introducing some of the most important heuristics. The importance of the respective methods, are based upon the amount of citations and research found on GoogleScholar. Additionally we will be dividing this Section into sub-chapters denoted by the respective classification.

The classification of heuristics is presented in a much simpler way, than for its counterpart the (meta-)heuristics, as will be seen in Section 8 on page 40. Thus, we are able to distinguish two main groups: Constructive and improvement heuristics (Laporte, 1992).

6.1 Constructive Heuristics

These kind of heuristics work on gradually building feasible solutions. Further, they do not omit the fact that the cost constraint has to be met. As we solely construct a route, we do not have an improvement phase per se. All in all, we implement basically two functions only, namely merging existing routes and assigning nodes to the respective routes (Toth and Vigo, 2002).

6.1.1 Bentley's Greedy Algorithms

Bentley's algorithm (Bentley, 1980) is possibly one of the most greedy algorithms for the TSP. It starts by sorting all available edge weights in ascending order. Those edge weights are distributed according to the distance between two nodes. This distance is implicitly given by a so-called *distance function* (Bentley, 1980).

In order to create our tour, we then repeat to link those nodes connected by the smallest edge weight that are not linked so far. Additionally, we have to be cautious to not create sub-cycles within our tour (Kennedy, 2001). This way, the algorithm progressively builds the tour fragments. Please refer to Algorithm 1 for the pseudo code.

Algorithm 1 Bentley's greedy algorithm

Input: TSP
Output: `tour`

1: Sort edge weights in ascending order and save the corresponding edges in the respective order in `asc_edges`
2: Set `tour` to `Empty`
3: **for** $i \in \{1, ..., m\}$ **do**
4: **if** `asc_edges`[i] is not in `tour` **and** adding `asc_edges`[i] to `tour` does not introduce sub-cycle in `tour` **then**
5: Add `asc_edges`[i] to `tour`
6: **return** `tour`

This benefits of this approach lie in the low amount of calculation needed and thus being able to perform pretty good on several computational engines boosting the already renown and shortly explained classical greedy-approach (Laporte and Semet, 2002).

The drawbacks of this algorithm lie in the general drawback of all greedy strategies. The algorithm never looks back in order to evaluate the decision taken (Kennedy, 2001).

6.1.2 Nearest Neighborhood Methods

This methods build on the principles of greedy-algorithms. It is initiated with a random location (Rosenkrantz et al., 1977). From this, we part to the closest (also denoted as 'cheapest' node). This process is iterated as often as there are nodes left to not be visited. We end the tour by connecting the last node with the starting node, i.e. $(i_n; i_1)$. Recall Figure 4 on the next page to see the methodology behind the general nearest neighborhood methods.

While the application is rather easy and solves the problem in low computational time, it bears the risk of the salesman or vehicle to travel long distances when the amount of remaining nodes is low (Mladenović and Hansen, 1997). Further, before the implementa-

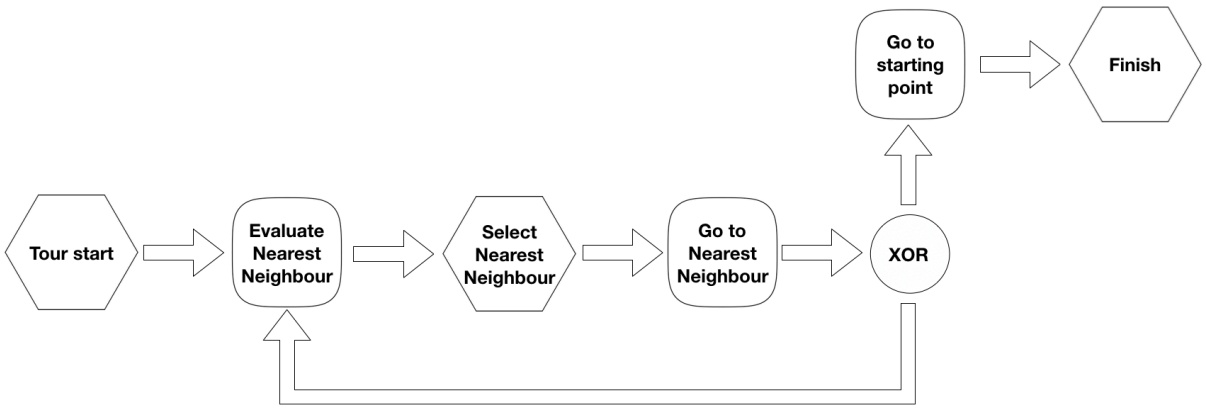

Figure 4: Methodology of Nearest Neighborhood Search

tion, one must take into account that one has to follow the right approach, often hard to be achieved within the methods of greedy heuristics.

We can thus differentiate two cases, one where the algorithm chooses the following destination based on the shortest path to be traveled and the other where the algorithm considers the underlying transportation costs (Lin et al., 1998).

It might have come to the reader's attention that the basic approach is similar to the aforementioned greedy algorithm proposed by Bentley (1980), yet in this setting, we do not order the edges, but take them as they are.

With this technique we were able to see that finding local optima and combining them does not lead to optimal or solutions or does even guarantee even good tours. Since the quality of the solution might be effected, when only a small amount of nodes are left (Bentley, 1980).

Being implemented in a rather easy way and requiring practically no previous gained knowledge, the solutions provided are most often not optimal and these algorithms bear the risk of not even finding a feasible tour, even though there might be one (Lin et al., 1998).

6.2 Improvement Heuristics

In this we start with an already complete tour or solution of the problem, obtained through benchmark processes or beforehand used heuristics. This is then taken as the incubent, current solution s^* and improved in the following. Lin (1965) can be seen as the base for all further improvement heuristics

6.2.1 Multi-Phase-Methods

Within those, we set to decompose the problem into two natural-components: node clustering and route construction. Further, it is possible to imply feedback algorithms between the two groups. More is more, the order in which we impose both procedures, is not fix. For procedures which cluster first to routing, we organize the nodes to feasible clusters. The route is than built upon them. In the opposite case, the tour is built initially and the clusters segment it into feasible vehicle routes (Hendrickson and Leland, 1995).

Motivated by their work on graph partitioning which deals with the reduction of a graph in sub-graphs by partitioning its nodes into several mutually exclusive groups, Walshaw (2002) propose a multi-level approach to the TSP. The fundamental idea is to summarize pairs of nodes to one node based on different criteria such that the graph size is reduced. This procedure is applied several times on the reduced graphs until the total number of nodes falls below a pre-defined threshold.

For each merging step, we get another layer of the original graph. We then start by solving the problem, in this example for the TSP, on the last respectively coarsest level of the graph and propagate the solution as initial starting value to solve the respective problem on the graph of the preceding level. The implicit assumption is that the fundamental graph features are pertained in the coarser level of the graph. Accordingly, the search space is reduced (Walshaw, 2002).

In more intuitive terms, imagine you want to determine the shortest path connecting Berlin and Munich. We let the 16 states of Germany constitute the last level of the graph. We further assume that the cities make the second level of the graph. The first level is the original graph itself with all the streets in Germany. You could start by first determining the states you would have to cross to get from Berlin to Bavaria where Munich lies.

The minimization problem reduces to a graph with 16 nodes and 27 edges. The edge weights are then determined by the shortest distance between the respective states. We thus have that the shortest path starts in Berlin, goes over Brandenburg and passes Saxony before it arrives in Bavaria. In a next step, we would solve the minimization problem of which cities we would have to pass if we assume that we follow the sequence of states listed above.

This route is not fixed and is refined in a next step by optimizing over its local neighbors. It might even happen that the route does not pass the states mentioned above, even though it is more probable that it does. Last but not least, we could determine the shortest route within all these cities connecting their preceding and subsequent neighbor city.

Multi-level approaches are versatile and can be used in the most diverse settings: graph partitioning (Hendrickson and Leland, 1995), graph drawing (Walshaw, 2000) and even graph embedding (Chen et al., 2018). Implementing a multilevel algorithm requires specifying a refinement strategy, a coarsening algorithm, an initialization algorithm and an extension algorithm. The different components will be explained in the following.

In the first step, the matching process aims to fix the those edges that are most likely to appear in a high quality tour. Walshaw (2002) proposes to match each node with its nearest neighbor according to the so-known minimum-weight perfect matching problem, which is thus the coarsening algorithm. In this way, we coarsen the graph iteratively until we achieve the desired graph size.

Walshaw (2002) assumes that the coarsening ceases when further coarsening would make the graph degenerate. This means that the graph is reduced to a size of only two nodes. Initialization is then trivial.

In a third step, the refinement algorithm is used to improve the tour created throughout the initial two steps, by guaranteeing that every node that already has been visited is not revisited and thus being responsible for not including any sub-cycles.

Crucial for this methodology to work lies in the three algorithms cited above as well as finding reasonable ways in clustering. Those are not always as easy to implement as in our example.

6.2.2 Clarke and Wright's Savings Algorithm

One of the conceptually slightly less simple heuristics for solving instances as the VRP and most prominent in the field of heuristics is the Savings Algorithm introduced by Clarke and Wright (1964). Similarly to the heuristics presented before, this is again a greedy algorithm.

Please refer to Algorithm 2 on the following page for the pseudo code of this algorithm. Note that we formulated the algorithm in such a way that it can be both used for undirected as well as directed networks (West et al., 1996).

Put differently, the algorithm proceeds as follows: We start by assigning each delivery location to another salesmen. We thus create several tours which contain both, the depot and one other node. The paths of those delivery locations which give the highest savings when merged are merged as long as no more saving is possible (Clarke and Wright, 1964).

If we say that one route would be traveling from $(0, 1, ..., i, 0)$ and the other $(0, 1, ..., j, 0)$. When we merge those two routes, we get to save costs of $cs_{i,j} = c_{i,0} + c_{0,j} - c_{i,j}$ once generating one tour, $(0, 1, ..., i, j, ..., 0)$. We then proceed by calculating the respective savings $cs_{i,j}$, $\forall i, j \in \mathbb{N}$, without $i = j$. In the following, we create n routes namely

Algorithm 2 Saving Algorithm according to Clarke and Wright (1964)

Input: n delivery locations
Output:

1: Set `routes` to `Empty`
2: **for** $i \in I$ **do** Add $c_0 \to c_i \to c_0$ to `routes`
3: Initialize $n \times n$ matrix `savings`
4: **for** $i \in I$ **do**
5: **for** $j \in I$ **do**
6: Set $\texttt{savings}_{i,j}$ equal to saving for merging delivery locations c_i and c_j given by $\texttt{savings}_{i,j} = d_{0i} + d_{j0} - d_{ij}$
7: Sort all entries of `savings` in descending order and save in `desc_savings`
8: **for** iter= 0 to $n \cdot n$ **do**
9: **if** `desc_savings`[iter]> 0 **then**
10: Set i and j equal to the first and second delivery locations that were merged to get the saving `desc\_savings`[iter]
11: **if** the route containing i in `routes` ends on i before returning to the depot and the route containing j in `routes` begins on j after departing from the depot **then**
12: Delete the corresponding routes from `routes` and add the route merging the deleted ones
13: **return** `routes`

$(0, i, 0)$ for all i.

While the initial step of calculating the savings is always the same, we differentiate between the following two procedures:

- **Parallel Saving:** After ordering the savings, we start at the top with the following condition: Given $s_{i,j}$, the algorithm looks for two routes containing or edge $(0, j)$ and the respective other with $(i, 0)$ in order to be merged. If this is the case, we might exclude the single tours, unifying them to (i, j). This method is also referred to as the best feasible merge (Laporte, 1992).

- **Sequential Saving:** Our starting point is now the route $0, i, ..., j, 0$. We now look for the first saving either $s_{k,i}$ or $s_{l,j}$. In order to feasibly merge some other route containing either the edge $(k, 0)$ or $(0, l)$. If this is the case, the route is merged and we proceed with the next route analogously. This is solely stopped once no further merge is considered feasible. Further, we denote this method as a route extension (Laporte, 1992).

The drawback of the presented algorithm lies in the variability of the reported results retrieved from the saving algorithm. One possible explanation lies in the lack of specification in research of which of the respective methods, parallel or sequential savings has

been used. Generally authors consider the parallel method to be dominant, mainly on computational time needed Laporte (1992).

Furthermore, the complex structure of the algorithm rises computational effort, as we need to calculate all possible savings. In the following, those savings need to be sorted, being the most consuming operation. Such a savings lists, in the case of n cities results in a matrix with k^2 elements such that we obtain a complexity of $O(k^2 \log k)$. In some exceptional but worst cases, it can be necessary to conduct an experimental setup for every saving such that the right vehicle was able to merge the routes (Laporte and Semet, 2002).

Further, the algorithm does neither incorporate fixed costs caused by the vehicle nor the fleet size, although the former is easily implemented by adding a constant to every cost $c_{i,j}, \forall j \in [2; n]$. The latter can then be achieved through two different approaches. In the first, we repeat the merging procedure as often as routes are required, even if the respective savings become negative (Laporte, 1992). In another setting, we extend the algorithm for a fixed amount of $|K|$ salesmen or vehicles. If the algorithm gives a result where we need less than $|K|$, we can just leave the remaining ones unoccupied. In a case where the algorithm results in more routes than we have salesmen or vehicles, some of the latter depart again from the depot after returning to it (Laporte and Semet, 2002).

This original procedure by Clarke and Wright (1964) can be executed in $O(n^2 \log n)$ time. Through extensions, research was able to reduce this complexity through different data structures, as in Pichpibul and Kawtummachai (2012) (Laporte, 1992).

6.2.3 Lin-Kernighan Algorithm

Lin and Kernighan (1973) propose this algorithm which constructs and tests complex local search methods composed by simpler ones. Basically, this algorithm improves a given tour by modifying it in order to reduce costs. The condition to be met is $c_{i,i+1} + c_{j,j+1} > c_{i,j} + c_{i+1,j+1}$ where c denotes the respective costs of the vertices.

This is done by repeating a replacing procedure where sets of tour edges get replaced by cheaper alternatives if possible. The case of a $k = 2$ is shown in Figure 5 on the next page.

This Lin-Kerningham heuristic has its origin in the k-opt heuristic which originates from the 2-opt technique (Croes, 1958), as shown in the Figure 5 on the following page. In the setting of the latter, the algorithm iteratively removes all possible pairs of non-adjacent edges observed in a single tour. In the following, the resulting two unconnected sub-tours are then re-connected, resulting in a minimal tour length. If the distances are now said to be Euclidean (on a two-dimensional plane), we obtain a non-crossing tour.

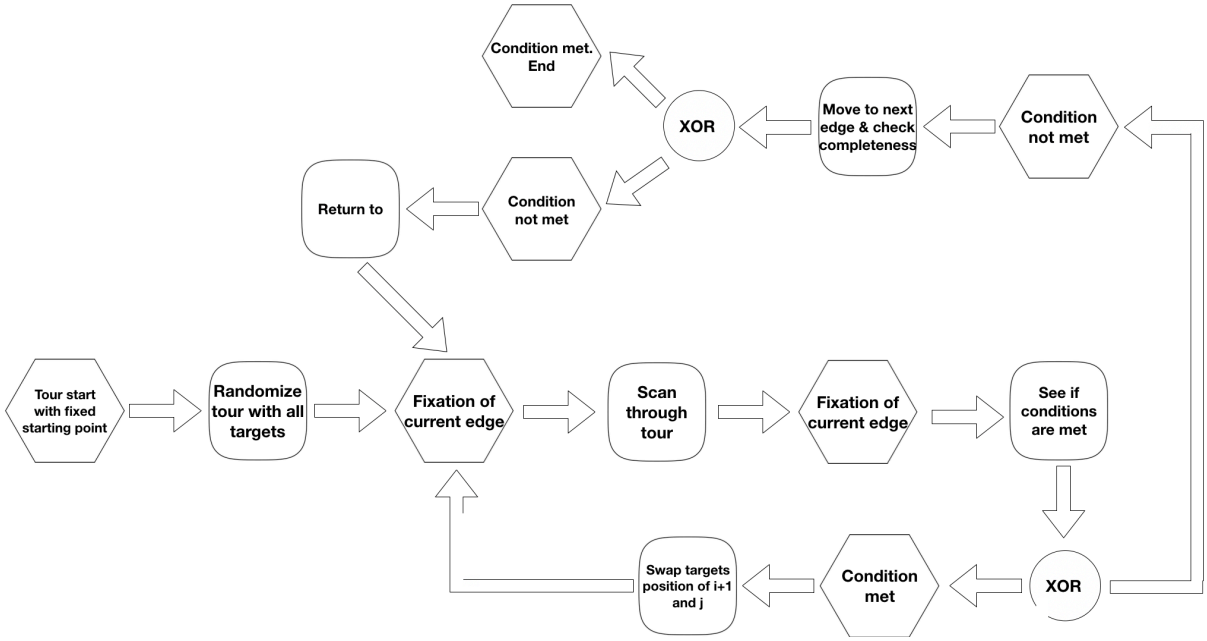

Figure 5: Methodology of the 2-opt Lin-Kerningham algorithm

Lin (1965) generalized the 2-opt heuristic into a λ-opt heuristic with $\binom{n}{\lambda}$ k-subsets of existent edges. Instead of analyzing pairs, we analyse all λ-subsets of edges in the tour.

We take each subset of S into account, testing if our tour can be optimized by replacing edges in S with λ new ones. This procedure is denoted as a so-called λ-switch. If the latter does not bring any benefits, we have a λ-optimum (Lin, 1965).

The authors iteratively perform such switches until obtaining the optimum. Further research has shown that λ optimally lies within the range of $[1; 3]$. Within this range the application of the algorithm is still practical, as the amount of iterations needed is polynomial. In the 2-opt case for Euclidean problems we have $O(n^{10}log_n)$ runs, for the λ-opt case this amount increases to $O(\frac{1}{4}n\frac{1}{2k})$ (Chandra et al., 1994).

Returning to the Lin-Kerningham algorithm, (Lin and Kernighan, 1973) extended Lin's research from the λ-opt and relax their assumptions. In more detail, they allow λ to vary along the search and the algorithm not to immediately implement all encountered improvements.

The algorithm starts with a random tour where it removes a path from tour. From the latter, we take the end and connect it to a random internal node, which in consequence creates a tailed circle. If this operation of possible interchanges has been exhausted in finding improvements, this tour replaces the current tour. Afterwards, the procedure is repeated (Lin and Kernighan, 1973).

From this, we once more remove an edge, resulting in a new path. We repeat this as long as both the sum of 'gain-sum' is positive and un-added or removed edges exist. The

former is thus defined as the sum of removed, except for the last one, and the sum of added edges. Further, the authors compute the costs by joining the endpoints. In the case of being lower than on the original tour, we say the current tour to be the solution.

Johnson and McGeoch (1997) even define this algorithm as the world champion heuristic for the traveling TSP from 1973 to 1989, as it is effective on real world problems. Walshaw (2002) uses the Lin-Kernigham Algorithm as a local search method, even though other (meta-)heuristics could be used as well. For an explanation of the local search method, please refer to Section 9 on page 44.

Even though being state-of-the-art, the Lin-Kerningham obtains its good results with much computational effort. The latter can thus grow exponentially depending on the problem instance (Lin and Kernighan, 1973).

As a concluding remark on heuristicsl we can say that throughout a long time they have been the state-of-the-art methods in solving prominent problems as the salesman or VRP. Nonetheless, we see its limitations in being too problem specific, and lacking in the aforementioned quality criteria of flexibility. For this, a sub-group, the meta-heuristics arise, which will be elucidated in Section 8 on page 40. For the classification of the latter being challenging, we start by introducing our methodology in classification in the following Section 7 on the next page.

7 Classification Scheme

In the following we will be elucidating the reason we chose to classify the metaheuristic employed in this work between 'Individual-based' and 'Population-based-methods', since there several authors use different approaches in separating or grouping those algorithms (Puchinger and Raidl, 2005).

One intuitive way would lie in the separation of 'nature-based and 'non-nature-based' procedures. This classification might have been able to capture metaheuristics in their early days, but after the enforced discovery of hybrids, further described in Section 9 on page 68 a classification by this means does not bring any enlightment. Those, as will be shown can't be put in either of this classes as most often components of both groups are combined Blum et al. (2011).

Another popular classification scheme would be imposing 'dynamic' against 'static objective functions'. The core of this classification is to escape the trap of only achieving local optima, by allowing for a change in the search landscape itself (Arora and Gigras, 2013). Although it might seem reasonable to classify metaheuristics by this way, we argue against it, since many of the presented 'population-based' methods even though being fixed on their objective functions, rely on a learning component such that they mimic the functionality of 'dynamic objective functions' although being considered 'static'.

Furthermore, we could also differentiate between 'one' or 'various neighborhood structures'. Within the latter, we denote a changing fitness landscape topology throughout the algorithm. This is used in order to diversify the search by swapping between those landscapes. We chose not to follow this classification, as only a small amount of metaheuristics, as the Variable Neighborhood Search established initially by Mladenović and Hansen (1997) and not further explained in this work make use of those 'neighborhood structures'. Thus a classification in these groups is not useful.

In other works, the authors based their classification on whether the algorithm use a 'memory' component or not. In the latter case, the information used to plan the next actions is based on the current information state and follows thus a Markov-process. We further could differentiate between long and short-term memory-usage. While the latter relies only on recently performed steps, the former takes note of an accumulation of synthetic parameters of search (Arora and Gigras, 2013). We will use this classification to enhance our classification, yet the algorithms are not clearly to be categorized into one of those groups.

One alternate differentiation of metaheuristics is based on whether they are 'individual-' or 'population-based'. By this we denote the amount of possible solutions taken into consideration simultaneously on the path of finding the best one. While the first, also

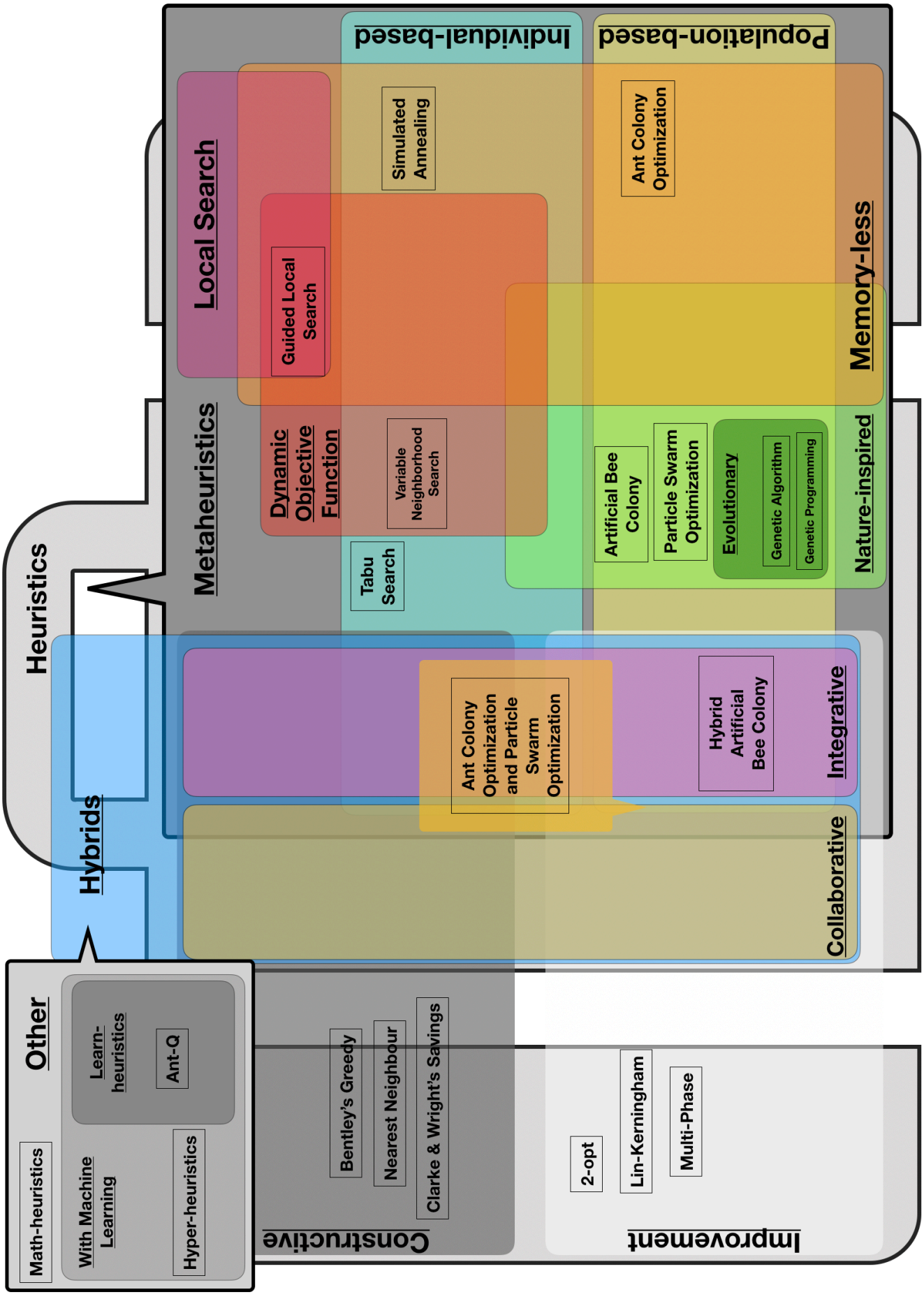

Figure 6: Classification of (meta-)heuristics proposed by us

denoted as 'trajectory-method' describe, as the name states, a trajectory when searching the solution space, the latter perform the search process in such a way that the evolution of a set of point in the solution space is described (Gendreau and Potvin, 2005).

Even though the memory-usage argument is strong, we rely on the last presented methods of classification, since in our view, this is the only way to really obtain a clear distinction between the respective algorithms. This becomes even more evident once we introduce the concept of hybrids in Section 9 on page 68.

Please refer to Figure 6 on the preceding page for a visualizations of the applied classification scheme.

8 Metaheuristics

Metaheuristics build a subgroup of heuristics. These methods are, however, more powerful than standard heuristics since they incorporate a more effective mechanism to reduce the risk of being trapped in local minima as well as not being restricted to certain problems only (Blum and Roli, 2003).

Moreover, a metaheuristic is able to employ heuristic methods by guiding them over the search space in order to exploit its best capabilities to achieve better solutions. In other words, metaheuristics represent algorithms which enhance the efficiency of heuristic procedures (Sörensen et al., 2018). Further, they are designed in such a way that they solve a high number of the aforementioned computational optimization problems, but lack in the need to heavily adapt to each problem (Sörensen et al., 2018).

”Metaheuristics” or ”modern heuristic” methods are used to develop and control heuristic procedures. It is important to note that the prefix ”meta” is not justified by computer scientists and is rather used to denote that the search for the optimal solution is done on an ”upper level” (Gendreau and Potvin, 2005). Please refer to Figure 7 for an illustrative explanation.

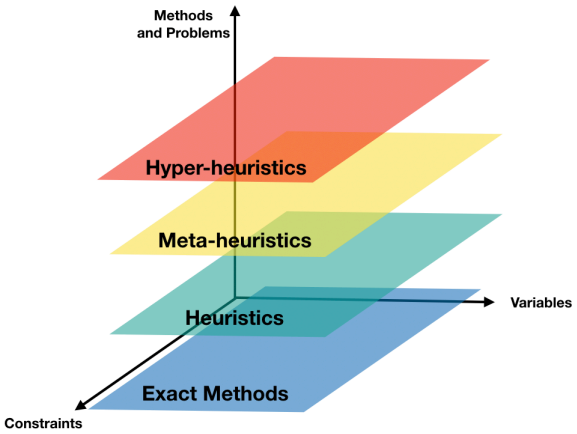

Figure 7: Levels of different solvers

Thereby we denote especially those procedures which aim to find solutions by the means of a run time efficient search on the successive improvement of already processed solutions. This happens via an application of iterative searching operators, building upon local search and/or upon evolutional/recombinatorial principles (Puchinger and Raidl, 2005) which will be explained in the throughout his work.

What unifies both principles is the matter of fact that both function on the principles of a mold which means that both techniques can be used independently from the problem itself. Most often the foundations of these metaheuristics can be found in nature. In contrast, the application of heuristics itself is based upon the adaptation or completion of problem specific properties given by the problem itself (Sörensen et al., 2018).

For metaheuristics being black box processes, scientific research lacks in theoretical knowledge to assume performance of these validations. By black box we denote systems where we see the input, yet we don‘t really know what is done with this data in order to get the output (Gendreau and Potvin, 2005). A majority of these algorithms have a stochastic behavior and mimic biological or physical processes, as we will see in the following

sections.

As one of this papers goals is to provide an overview of classification, we will provide further information on metaheuristic classes, which were already implemented into a graphical representation, as in Figure 6 on page 38. We will thus start by introducing basic notions, important throughout the remaining work.

8.1 Introduction and Basic Notations

We denote a function $N : S \to 2^S$, which matches each solution $s \in S$ with a subset $N(s) \subset S$ (Lin et al., 1998). A solution $s^* \in N(S)$ is called neighbor of a solution s. For this to be true, we need to make sure that the so-called phenomena *locality* exists.

Locality describes the effect on a solution once we move in its representation (San Nah Sze, 2007). This movement is referred to as perturbation. When we perform small modifications on this representation, the solution must present small changes, too. If this is the case, we can denote the neighborhood to have *strong locality*, which means that the search in the solution space is more powerful. The opposite case, *weak locality*, denotes when a large effect in the solution while only small modifications have been done to the representation. This effect can come to an extreme such that the search converges to a random search in the search space S (Gendreau and Potvin, 2005).

The principal component of this method, lies in generating an initial solution to depart from. There are numerous ways in modeling the latter, but we will be focusing only on two of them (Arora and Gigras, 2013).

The easier method consists in obtaining the initial solution randomly. This is done rather quick, but one has to take into consideration that in further solution finding, the solution might take more iterations and computational time to converge. Yet, the more constrained a problem is, the more difficult it might be to find feasible random solutions. For this and other cases, we apply the already explained greedy approach (Arora and Gigras, 2013).

By this we are able to consistently reduce polynomial-time complexity and thus create local optima of better quality. Local optima are thus defined as follows.

$$f(s^*) \leq f(s), \forall s \in N(s^*) \tag{8.1}$$

As described in Equation 8.1, a local optima of one neighborhood does not imply being a local optima for another neighborhood, since the optimality is guaranteed only for the neighborhood $N(s^*)$.

Further, there is a set of local optima $S_N = S_N^- \cup S_N^+$ in the means of the local minima

and maxima, defined in equations 8.2 and 8.3 respectively:

$$S_N^- = \{x \in S, \forall y \in N(x), H(x) \leq H(y)\} \tag{8.2}$$

$$S_N^+ = \{x \in S, \forall y \in N(x), H(x) \geq H(y)\} \tag{8.3}$$

Two flaws appear on this: Inapplicability and ineffectivity.

Inapplicability arises since the amount of elements in S might be so large that memorizing the neighbors of each solution in order to enable comparisons throughout the all neighborhoods, might become unhandy. Ineffectivity appears since this definition does not incorporate any notion of the proximity between two neighboring solutions. The resulting problem is now that close solutions might share features corresponding to similar costs. To tackle this, one might opt to add minimal changes to the characteristics of a solution, leaving the remaining unaltered (Gendreau and Potvin, 2005).

What remains to be said is that the choice of a neighborhood system is crucial in order to implement metaheuristics. It is important to structure these neighborhood systems in such a way that they are both effective and relatively well adapted to the particular problem (Arora and Gigras, 2013).

San Nah Sze (2007) describe an exponentially growing size of the neighborhood depending on the problem, resulting in the negative impacts explained above. In other words, the goal is to either find efficient procedures exploiting the neighborhood in such a way that large neighborhoods might be chosen. Another possibility would imply designing heuristics in such a way that the identification of improving neighbors is identified such a way that the whole neighborhood has not to be taken into consideration (Gendreau and Potvin, 2005).

In order to boost performance, one may also use hybrid strategies. In this setting, we start with a pool of solutions, composed of both, random and greedy solutions (San Nah Sze, 2007). In some exceptional cases, mainly for specific problems of the real world, our initial solution may even be pre-modeled by experts (Arora and Gigras, 2013). The way our neighborhood is modeled depends on the optimization problem itself. The underlying risk in designing a neighborhood structure lies in the size, quality itself and the computational time required in order to explore it (Talbi, 2009).

The size of the neighborhood might be dependent on the problem itself, yet mostly small neighborhoods are used. Those are either created by polynomial functions of the input sizes, mostly linear or quadratic functions. Although, high-order polynomial functions might come into consideration as well as exponential functions (Arora and Gigras, 2013).

While with increasing size, we note improved quality of solutions, we must take more neighbors into consideration for each iteration, resulting in large computational time. We thus denote the size by the amount of neighbors of a given solution s (Arora and Gigras,

2013).

After having introduced the reader to the basic notions, we will now proceed by presenting the most important metaheuristics based on the already elaborated classification scheme, starting with individual-based metaheuristics.

8.2 Individual-based Metaheuristics

Also referred to as 'trajectory methods' or 'S-Metaheuristics', these metaheuristics aim to reduce the search space by relying on a neighborhood system which makes it possible to build a set of neighbor solutions $s^* \in N(s)$ starting from a current solution s. Put simpler, this approach is to improve a given solution (Gendreau and Potvin, 2005).

The trajectory is then performed by iterative procedures, moving from solution to solution. The efficiency of these methods lies in tackling various optimization problems in different domains (Blum and Roli, 2003). Please refer to Figure 8 to get a visualization of the basic procedures of such methods.

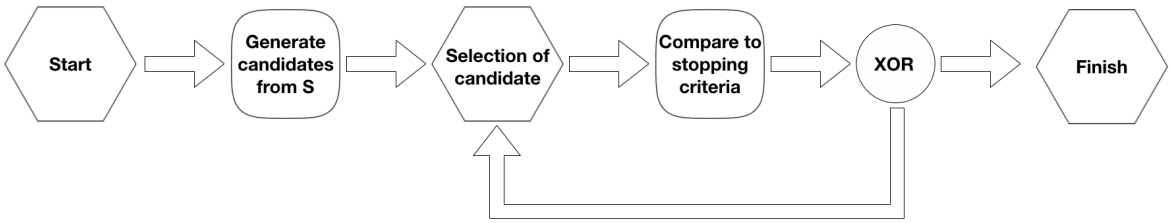

Figure 8: Methodology of individual-based approaches

Individual-based solutions can also contain a memory-component. In this case, both generation and replacement phases are not only based on the current solution employed, but rather stored and then applied when selecting new solutions. Examples of both such methods will be provided in the following (Consoli and Darby-Dowman, 2006).

The common search concepts for all S-metaheuristics are the definition of the neighborhood structure and the determination of the initial solution, crucial for obtaining solutions for the problem. We will be presenting the most iconic metaheuristics in the following.

8.2.1 Local search methods

Once referring to the classification provided by Osman and Kelly (1996), this class does not belong exclusively to the group of metaheuristics. Nonetheless, just like other algorithms, local search methods build upon improving already existing initial solutions, from which it's other name, *iterative improvement method* arises. In the following we assign them

to the class of metaheuristics, even though many heuristics apply the same techniques (Rochat and Taillard, 1995).

Local search methods thus follow the idea of successive improvement: By changing the solution marginally in every step coherent to our goal of lowering the costs continuously. The fundamental idea of such a procedure lies in the assumption of a solution space S which unifies all possible alternatives of action. Further, this method is most often referred to as *Hill Climbing* (Henderson et al., 2003).

We start by either by computing or choosing randomly a feasible solution s^* from the problem in the problem's solution space S. If we find a solution for the problem at hand with a lower cost than the initial solution imposed by us or experts, we update our starting point. We stop once there is no other solution with lower costs than the incubent (Rochat and Taillard, 1995).

More formally, by iterative repetition of modifications of the incubent solution s^* we move to a neighbor solution, defining the latter as the new candidate if it is superior to s^*. Those neighbors have been pre-defined by setting up relationships between all possible solutions in the solution space (Rochat and Taillard, 1995).

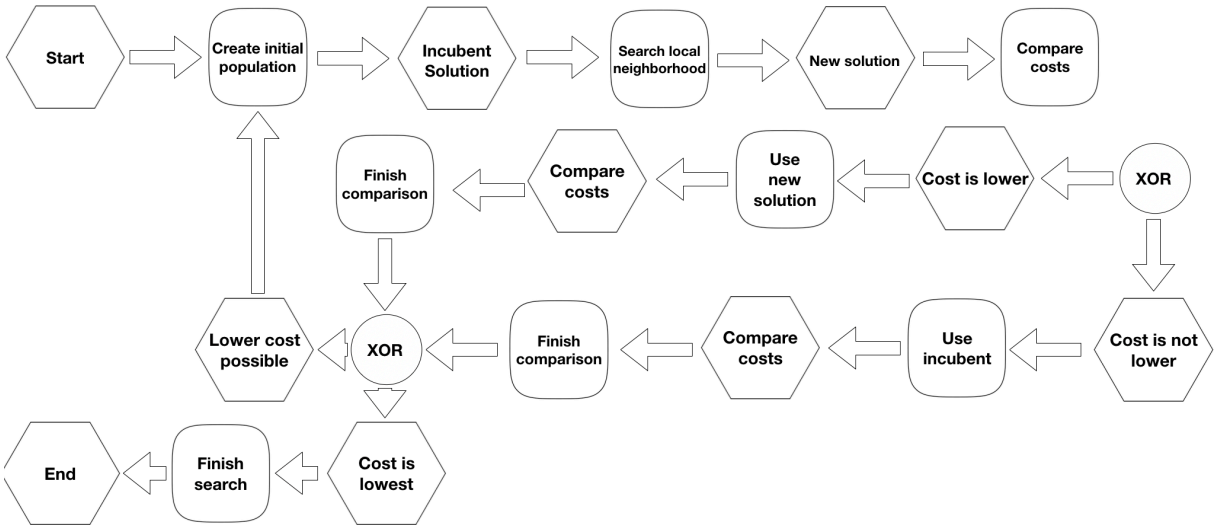

Figure 9: Methodology of Local Search algorithms

The benefits of the local search solutions lie in a computational efficient way of finding good solutions. Yet, one of its major drawback lies in the risk of the algorithm to be trapped in local optima (Lourenço et al., 2003). This risk results from the algorithm to have insufficient knowledge on the solution space S. In order to escape this, a local search method would have to look at all permutations of the locations to be visited, making it thus inefficient (Rochat and Taillard, 1995).

Further, the global optimal solutions do not necessarily need to be in the same neighbor-

hood $N(s^*)$ of the current solution candidate s^*. If no solution with a better condition on the improvement value is found during the local search process, the candidate is a local optimum. This is a global optimum only in exceptions, as greedy algorithms, which build the base of local search methods, can only use information that is available from the current solution (Johnson and McGeoch, 1997).

Bearing this in mind, several improvements have been introduced, namely Simulated Annealing (Section 8.2.3 on page 52), Tabu Search (Section 8.2.2 on page 47) and the Guided Local Search, which will all be analyzed in the respective following sections.

Guided Local Search

This algorithms acts as a master upon a normal Local Search algorithm. In order to start, one defines features needed for a solution to be listed among candidate solutions. Once our Local Search gets trapped in some local optima, we select certain features in order to be penalized (Voudouris and Tsang, 1999). Further we will be referring to this method as *GLS*.

We will call those features *solution features*. Those describe certain kinds of properties or characteristics needed in order to differentiate between solutions. Take the arcs between the cities concerning the TSP as an example for such a feature (Voudouris and Tsang, 1999).

In the following, Local Search scatters the solution space using the initially set objective function plus the accumulated penalties. The most interesting thing in the GLS lies in the way we select the solution features which will cause a penalty: GLS effectively distributes the effort in search, favoring promising areas and finding utilities in penalizing respective features. Possible solutions found by the Local Search are then regularized by such penalty modification such that they stand in accordance to the gathered information of both prior or during the search operations (Rochat and Taillard, 1995).

Please refer to Figure 10 on the next page. A normal Local search algorithm would get trapped in s'. Through the penalties, we worsen the result such that it can proceed to s^*, another solution being even a global optimum. Put differently. in the GLS procedure the objective function is changed in a dynamic way. Let further $\mathbb{I}_i(s)$ be a dummy which takes the value 1 if the feature i is present in solution s, otherwise it will be 0. Further, let us define a new objective function $f'(s)$. This depends on the original objective function $f(s)$ summed to a term, which depends in the m pre-defined features, as seen in Equation 8.4.

$$f'(s) = f(s) + \lambda \sum_{i=1}^{m} p_i \times \mathbb{I}_i(s) \tag{8.4}$$

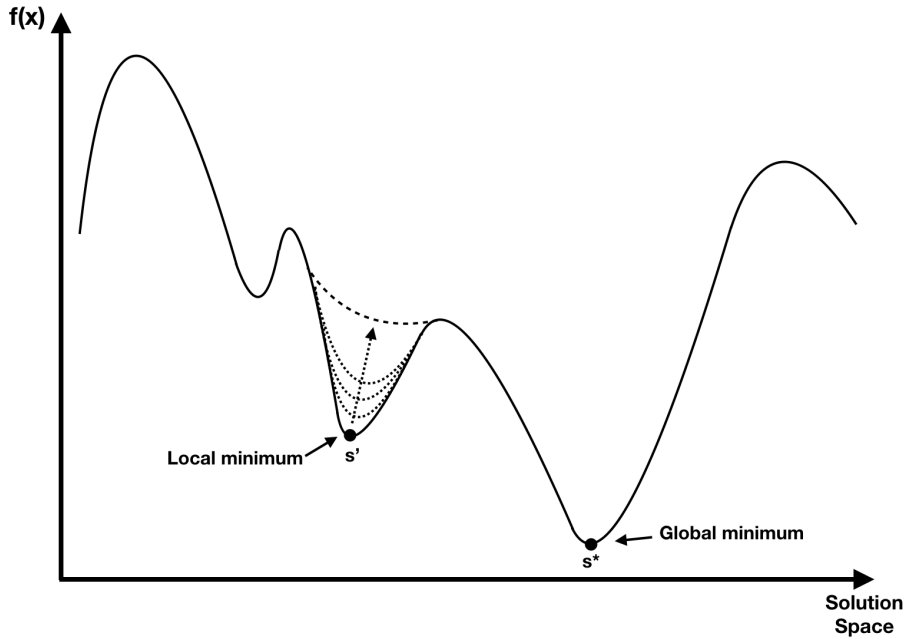

Figure 10: Guided Local Search escaping local minima

Let p_i denote the penalty parameter. The higher p_i, the more importance we delegate to feature i. Further, let λ denote a regularization parameter which balances the importance of all features I in relation to the original set in $f(s)$. Research found that good values for λ are achieved by simply dividing the value of the objective function of a local minimum by the number of features present in it (Voudouris and Tsang, 1999).

In other words, we augment the objective function by adding penalties. It is utterly important to specify the aforementioned features. Let us assume the VRP. One feature could be whether in such a solution candidate tour a vehicle travels immediately from a pre-defined city A to B (Voudouris and Tsang, 1999).

This utility is further defined as $U(s,i) = \mathbb{I}_i(s)\dfrac{c_i}{1+p_i}$ iff a feature i is present in a solution s, otherwise it takes the value 0. Let c_i denote the costs of a feature i. This value is obtained by a heuristic evaluation of the importance relative to all features set (Johnson and McGeoch, 1997).

The GLS would now associate costs and penalties for each of these features. The former is often taken out of the objective function. In our example of the VRP costs could simply be the distance between i and j. In other words, the higher c_i the more utility we relate to this feature. Put differently, the higher the costs, the more utility we have in penalizing it. If a feature is not even met in the solution, we have no utility in penalizing it. As we scale the utility by p_i, we prevent the algorithm to be looking solely at the costs such that it gets more sensitive to the previous results from the search history. If the penalties are frequent, denoted by a high p_i, the lower the utility in penalizing it again (Lourenço

et al., 2003).

In a local optimum, the feature which brings the highest utility once getting penalized will be penalized. In this case, we increase p_i by the value of 1 thus scaled by λ. In certain variants the penalties take place with a lower frequency. Within this, the penalties are then increased by a factor $\alpha \in [0; 1]$ (Mills and Tsang, 2000).

By this, the GLS is able to restrict its search space, reducing the search to more promising areas in the solution space, more specifically only areas with good features, i.e. feature with lower costs c_i. For this to function, penalties are utterly important in order to prevent the search to put all its effort in one particular region of the solution space. The algorithm stops, once a solution $s*$ is best within the original objective function f (Mills and Tsang, 2000).

Extensions exist, as setting a threshold value for the penalties. In the case that a penalty level exceed a pre-defined threshold, we decrease all penalties uniformly. By this, we boost the performance such that even large problem instances can be solved (Voudouris and Tsang, 1999).

8.2.2 Tabu Search

Belonging to the group of local search operators as explained in Section 8.2.1 on page 43, Tabu Search was proposed by Glover (1986) in its modern form. Its main idea can be deduced from its name. The word *tabu* (or *taboo*) namely originates from the aborigines of Tonga Island as a term for things that cannot be touched because they are considered sacred (Edelkamp and Schroedl, 2011).

In Tabu Search, the "states that cannot be touched" are saved in a so-called *tabu list*. In this way, the feasible neighborhood that is explored in the local search is restricted. This helps that the method is not trapped in a local optimum due to adaptive memory (Edelkamp and Schroedl, 2011).

In general, Tabu Search is a simple local search, which explores the solution space by making modifications as it moves from one solution to another. This is done in a rather intelligent way as it incorporates non-static memory and exploration with feedback. Further, past moves are added to a tabu list according to a pre-defined criterion. By this we prevent the algorithm from returning to both recently visited solutions or bad solutions. Put differently, we prevent the algorithm to end up in a cycles (Gendreau et al., 1994).

After each iteration, we define the best solution s^* as the new incubent. This is then added to the tabu list and one element of this very list is then removed in a FIFO order. In other words, Tabu Search methods are also referred to as *dynamic neighborhood searches* (Hansen and Mladenović, 2003)

For instance, we could also update the tabu list such that it includes the past $k \in \mathbb{N}$ moves or the k most frequent past moves. It is important to note that the tabu list does not contain any solution, but only search moves. Prospective moves have to be chosen from outside this tabu list. As a result, we avoid sequences of similar solutions such that the solution space is more extensively explored (Xu and Kelly, 1996).

If all neighbors are in this tabu list, a move increasing the cost of the solution is accepted. This is the decisive step differentiating Tabu Search from other local search methods which would be trapped in this local optimum in such a case. The aspiration criterion allows to ignore the tabu constraint if there is a move that improves all previous solutions (Gendreau et al., 1994).

The decision of how to modify the incubent solution s^* is guided by the information gathered during the search process. This responsive exploration uses the principles of intelligent search, i.e. exploiting good solutions while exploring promising new region in the solution space (Toth and Vigo, 2003). Let us now assume an objective function $f(s)$

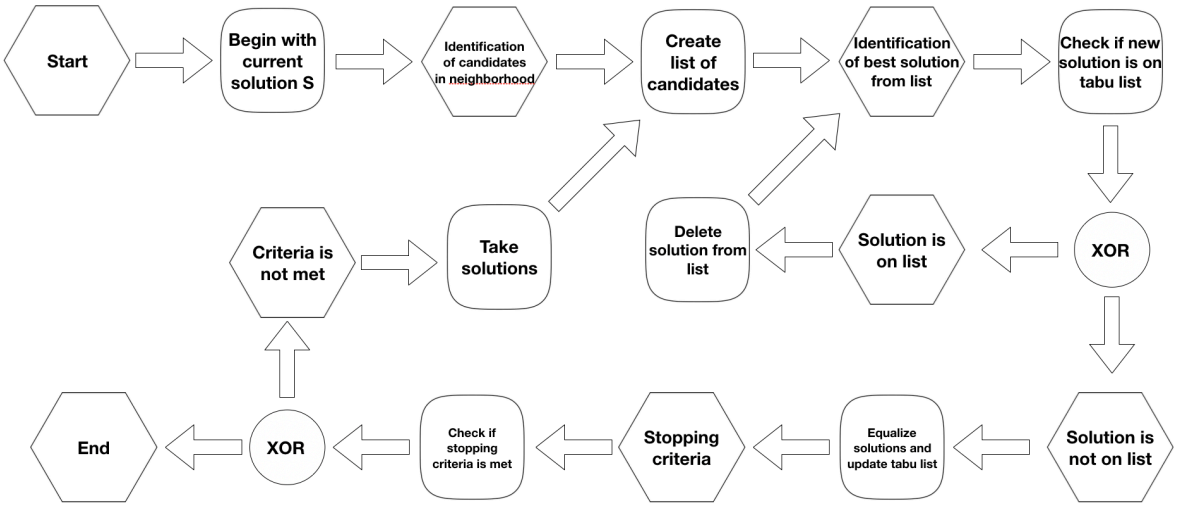

Figure 11: Methodology of Tabu Search algorithms

with $s \in S$ denoting a solution in the solution space. The initial procedure resamples local search, as seen in the previous section, by iteratively moving from one solution to another until a solution s^* fulfills all constraints that are found. Each solution has a neighborhood $N(s) \subset S$. Each other solution is then reached by moving from one solution to another (Toth and Vigo, 2003).

We consider a solution an improvement if $f(s^*) < f(s)$ for all $s \in N(s)$. We further denote a *move value* v that is the gains on the new solution by $v = f(s^*) - f(s)$. If $v < 0$ we call it an improvement, otherwise it is considered a non-improving move. Further, we might use the methodology of steepest descent. In this, the algorithm proceeds by only moving in improvement cases until no more improving solutions can be found in the

neighborhood of the incubent s^* (Toth and Vigo, 2003).

A different approach would lie in the best-improving strategy. In this setting, Tabu Search chooses the best neighbor of the incubent, regardless of $v \geq 0$. The drawback of this strategy lie in the massive computational expenses, as every element of $N(s)$ has then to be evaluated (Gendreau et al., 1994).

In order to reduce computational effort, we can thus incorporate solution or candidate lists whereas we narrow the examination areas. Popular on this behalf is the so-called first-improving strategy. In this setting the algorithm proceeds to the first s^* for which $f(s^*) < f(s)$ holds. If no more such s^* exist, we move to the best non-improving solution that is the solution for which the difference is least (Gendreau et al., 1994).

The main key of Tabu Search lies in finding the right balance between intensification and diversification, e.g. balancing the modification of choice rules in order to encourage move combinations and solution evaluated as 'good' and the encouragement of searching unvisited areas (Toth and Vigo, 2003). The algorithm stops, once a stopping criteria is reached. Please refer to Figure 11 on the preceding page to see a visualization of the general Tabu Search procedure.

Further, extensions are possible. Randomized Tabu Search, for example, combines the idea of the tabu list with the selection mechanism applied in BnC, mentioned in the following. This method can be shown to be asymptotically convergent (Bozkaya et al., 2003).

We further differentiate between the following Tabu-Searching methods:

- **Granular Tabu** methods, introduced by (Toth and Vigo, 2003), view long trails as less likely to be contained in optimal solutions. With this knowledge, all edges with a greater length than a granulating threshold are not considered, reducing thus the amount of elements being examined.

  Toth and Vigo (2003) defined $\nu = \beta\bar{c}$ where β denotes a sparsification parameter ranging from $[1.0; 2.0]$. In contrast, $\bar{c}$ denotes the average edge length, which we get from an a priori fast heuristic. If $\beta \in [1.0; 2.0]$, the authors impose a remaining rate of edges in the graph of about 10%-20%.

  Apart from the theoretic usage, we let β adjust dynamically whenever the algorithm notes no improvement after a fixed amount of iterations additionally to its periodically decreasing characteristics.

  We can thus generate neighbor solutions, once we limit the amount of edge exchanges within the same (or maximally two) routes. For this particular reason, Toth and Vigo (2003) propose a method in order to be able to examine all potential exchanges in $O(|E(\nu)|)$ time. Let $E(\nu) = \{(i,j) \in E : c_{ij} \leq \nu\} \bigcup I$ hold. Further,

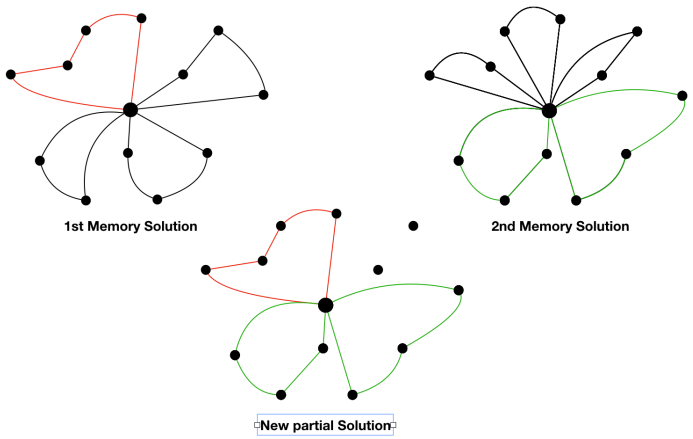

Figure 12: Generating a partial solution

let I denote a set of important edges. Those can be routes to the facility or those belonging to an already computed solution of high quality.

- **Adaptive Memory Procedure** introduced by Rochat and Taillard (1995), requires a special note: Even though it is mostly used in the field of Tabu Search, it is not limited to this particular field of operations.
  A so-called *adaptive memory* is a pool of good solutions which instead of being static, gets to be updated dynamically throughout the solution finding. Similar to Evolutionary Algorithms introduced in 8.3.1 on page 55, we periodically extract solutions from this pool in order to combine those in a different way. By this, we hope to generate new and improved solutions (Rochat and Taillard, 1995).
  For the selection of these routes it is utterly important to check whether a customer is included twice in a solution. For this, the Rochat and Taillard (1995) advise that there is a possibility for the selection to be ended by partial solutions. Those will have to be completed. This can be achieved by some kind of construction heuristic, as shown before.
  We included some example in Figure 12. By extracting the red and green tours. we obtain a partial solution, as two locations remain unvisited. Up to today, this technique remains a state of the art method when solving a wide range of optimization problems (Bozkaya et al., 2003). Further, we can differentiate between short and long memory:

  – **Short Memory** techniques allows the search to continue beyond local optima. This is possible due to the permissions of non-improving moves as well as the modification of the neighborhood structure of following solutions. The latter, denoted by $N'(s)$ is the result of maintaining only partially the history of the

states encountered throughout the search (Bozkaya et al., 2003).
With the tabu element, selected elements of the original neighborhood $N(s)$ are excluded from $N'(s)$. In other words, tabu can be considered a dynamic neighborhood search. A solution s might be visited more than once in this setting, but the considered neighborhood is not constant (Gendreau et al., 1994).

- **Long Memory** enhances the short-memory approach by adding frequency. Frequency-based memory delivers information that complements the information provided by short memory, enhancing the principles by which the selection of preferred moves is done (Gendreau et al., 2001).
  Frequency is then weighted into sub-classes taking into consideration the quality of the underlying solution. Therefore we define penalties and incentive values in order to modify the evaluation of the respective move determining the moves to be selected (Toth and Vigo, 2003).
  Let $v' = v \times (1 + pq)$ denote the modified value function where v is the original move value, p the penalty and q the frequency ratio. The crucial factor here is p, since a $p > 0$ makes the move less attractive. In other words, in this setting the likelihood of duplicating a previous neighborhood, once a solution is revisited or the probability of making choices which repeatedly visit only a limited amount of neighborhoods is equal to zero.

- **Xu and Kelly (1996)** introduce a model where the vertices of two routes are swapped. This yields in a global repositioning of certain vertices in different routes, resulting in local improved routes and solving network flow models, necessary to be able to do the reallocation efficiently.
  Via approximation we compute the costs of ejection and insertion, whilst the underlying capacity constraints are met.
  The route itself is optimized by 3-opt exchanges in the Tabu Search procedure. As in the adaptive memory process, we allow the parameters to dynamically adjust throughout the search process. In the following, a pool of best solutions is created and memorized. Periodically, we use those to re-initiate the search process with new values on the respective parameters (Xu and Kelly, 1996).
  As well as its younger brother, the adaptive memory procedure, this method is a state of the art methodology on certain implementations, since computational effort and the parameter tuning tend to be tedious.

Tabu Search ends upon meeting a pre-defined stopping criteria or when the opposite of the candidate list, the *allowed set* is empty such that all solutions of the neighborhood

$N(s)$ are now on the tabu list.

8.2.3 Simulated Annealing

This class of metaheuristics does not take its inspiration, apart from most metaheuristics, from nature but from industrial processes, more specifically from metallurgical processes. Thus it is one of the oldest algorithms belonging to this group of metaheuristics (Aarts and Korst, 1988).

This procedure was independently discovered by Kirkpatrick et al. (1983) and Černỳ (1985) (Aarts and Korst, 1988). From its name we can already derive its origin, which is from the physical process of annealing from the metallurgical industry. In this procedure, the algorithm mimics thermodynamics, important in the process additional to properties of the processed metals and the heating respective cooling processes.

The former plays an essential role in order to join the materials while the cooling process is important in the process of solidification, since the temperature is to be reduced in such a velocity that the material achieves a firm structure by a minimum input of energy (Aarts and Korst, 1988).

Although it is considered to be one of the oldest metaheuristics, the inventors knew about the problem of (meta-)heuristics getting trapped in local minima. To overcome this, they allow the algorithm to take the opposite approach from the afore-mentioned hill-climbing, the so-called *uphill-move*. In this setting, we allow moves which result in solutions worse than the incubent. To omit the solution to worsen with each move, the authors propose to set the probability in allowing for such solutions to decrease (Kirkpatrick et al., 1983).

In the beginning of a Simulated Annealing (further denoted as *SA*) algorithm the authors depart from an initial solution s, generated either randomly or by pre-computation and the pre-set temperature T. In the following, at each iteration a new solution $s^* \in N(s)$ is sampled at random and thus accepted as the new current solution under consideration of $f(s)$, $f(s^*)$ and the temperature T. Put differently if $\Delta = f(s^*) - f(s) < 0$. If the latter statement is not met such that $\Delta \geq 0$, s^* is being chosen as the new solution with a probability taken from the Boltzman distribution $exp(-\frac{\Delta}{T})$ (Duhr and Braun, 2006).

From the equation we can further note that the probability depends on Δ, on T and its respective decreasing nature. We will now focus on the latter component. Ass mentioned above, the decreasing temperature is defined in a *cooling schedule*. This, defines for every iteration k the respective value of T. We can thus say that $T_{k+1} = Q(T_k, k$ where $Q(T_k, k$ is a pre-defined function of the temperature for each iteration (Aarts and Korst, 1988).

Research on non-homogeneous Markov chains affirm that for certain condition we obtain a global minimum s_{max} for $f \to \infty$. Further, we assume Γ to exist in $\mathbb{R}$ such that

$lim_{k\to\infty} p(s_{max}, k) = 1$ iff $\sum_{k=1}^{\infty} exp(\frac{\Gamma}{T_k}) = \infty$.
In other words, a cooling schedule which is able to produce a global optimum needs to fulfill the above-mentioned equations. In other words,

$$T_{k+1} = \frac{\Gamma}{\log(k + k_0)}$$

with k_0 being a constant. Unfortunately, research has found that such cooling schedules are thus infeasible for practical usage, as the computational effort would break all boundaries (Henderson et al., 2003).

However, one may apply faster cooling schedules, as one of the most iconic ones following geometric law: $T_{k+1} = \alpha T_k$ whereas $\alpha \in [0; 1]$. In other words, we let the temperature decay exponentially. Further, as we said above, the cooling rule has not to be constant throughout the search. This is imposed, since we want to achieve a good balance between exploration and exploitation (Henderson et al., 2003).

Let us assume that we are currently at the beginning of the algorithm. For now our goal is to sample the search space such that we either leave T constant or let it decrease in a slow and linear way. Once our sampling is done, we leave T to follow the geometric rule, as mentioned above, as our goal is now to converge to a local minimum (Aarts and Korst, 1988). Lundy and Mees (1986) opted for a non-monotonic cooling schedule. In this, the cooling alternated in phases, since we allow the algorithm to 're-heat' in order to extend the exploration and thus reduce the risk of being trapped in local minima.

We could see that for the algorithm to work properly, it is of great importance to come up with both, a feasible cooling schedule and a feasible initial temperature, which are prone to the problem at hand. Aarts and Korst (1988) refer to the different search landscapes which result in different costs in order to escape local minima.

The initial temperature T_0 is then determined empirically by initially sampling the search space by the means of a random walk in order to obtain values for the expected value and variance of the objective function throughout the search space (Ingber, 2000).

As the original designers, Kirkpatrick et al. (1983) and Černỳ (1985) described SA to share characteristics with a Markov-chain, e.g. only the previous state matters in order to evaluate the next one, we assume it to be memory-less. Further, research has adapted models in which a memory component is adapted, as in Chardaire and Sutter (1995).

We can thus say that the algorithm proceed its search by the means of (i) a random walk and (ii) an iterative improvement. As in the initial phase of the search is more prone to be a random walk, it is less biased towards improving a given solution, but more prone to explore the solution space. Due to the inconsistent change in temperature, the search

converges then to a local minimum (Henderson et al., 2003).

The algorithm stops under one of the following conditions (Henderson et al., 2003) for pre-defined iterations k_i.

- The value $f(s^*)$ has not decreased by at least $\pi_1\%$ for at least k_1 consecutive cycle of T iterations
- The number of accepted moves has been less than $\pi_2\%$ of T for k_2 consecutive cycles of T iterations;
- k_3 of T iterations have been executed.

Let us assume the TSP. In this, our goal is to find a global minimum which is the shortest traveling route. In the setting of the SA we see the problem itself as the material which is being melted. For this methodology tries to mimic the behavior of atoms, moving at random in the melted metal, we assume these atomal movements to consist in routes. Through the modification implied on our behalf, as described above, one might change the course of these atoms. On colder temperatures, we observe the so-called *crystallization*, which stands for the finding of shorter routes. In other words, the material decreases temperature such that the atom's movement is restricted. Being an atom able to escape the crystallization is denoted with the above given probability and stands thus for increasing the route.

Even though important for solving optimization problems, the results of SA are not competitive to simple Tabu Seach algorithms. This can be explained by the long computing times which arise to the great amount of parameters and the complicated implementation (Gendreau and Potvin, 2005).

This was only a glimpse to the vast category of 'Individual-based' metaheuristics where we focused on the most prominent examples. Further methods may be found, as in Consoli and Darby-Dowman (2006). The main benefit of these approaches lie in the way the solution space or an inbound promising region is explored. We will now proceed to its counterpart, the *Population-based* approach.

8.3 Population-based Metaheuristics

The underlying goal of Swarm Optimization is to design multi-agent systems in a usually nature-inspired way. Those consist most often by simple entities, lacking global control which shall iteratively be improved (Eberhart and Kennedy, 1995).

Swarm-based algorithms use simple particles, also referred to as agents, which interact locally with a certain objective in their respective environment. This implies normally reaching a *food source*. Each agent is then assigned to one or more rules without any centralized structure in order to limit the agents behavior. Through this we induce local and random interactions of the agents through communication. This results in a so called *intelligent global behavior* (Sörensen et al., 2018).

As explained before, through exploitation we are able to expand the search space in such a way that we find an optimum solution close to a good (near-) optimal solution. As our knowledge about the search space is rather limited during the initialization, we need to make sure our swarm continues with the exploitation process to reduce the risk of only obtaining local optima. With each step we exploit more relatively to exploration, since we aim to tune the algorithm and thus the solutions (Sörensen et al., 2018).

This is achieved by initializing with a random population which does not have to be complete since in some examples we allow the particles to set an offspring. In the following, this population is integrated into the solution space in order to exploit it. This population can either have memory in order to recall the initial population or remain memory-less. This procedure is repeated until a pre-defined condition (also known as stopping criterion) is satisfied (Aydin, 2012).

As mentioned before, the first step is to define an initial population. In this we see the exploitation nature of the method because of the vast amount of initial agents in a population. An optimally chosen metaheuristic is diversified in such a manner that the risk of premature convergence to local optima is reduced (Gendreau and Potvin, 2005).

The exploration and exploitation process are being done in three steps, mostly inspired in nature as named in the following: Self-adaptation, Cooperation and Competition. During the first step each agent enhances its own performance before cooperating by transferring the obtained information. During the last step, agents compete in order to survive and remain in the system, directing it to find the best solution possible whereas m solutions are generated per iteration (Gendreau and Potvin, 2005).

In the following we will be introducing the most meaningful metaheuristics belonging to the group of population-based metaheuristics. As for Section 6 on page 26, we will not be including every single one of them, just those which brought a new concept or great innovation.

8.3.1 Evolutionary Computation

Evolutionary computation belong to the first group of population-based metaheuristics and make use of principles inspired by the evolution of species (Darwin, 1859). In other

words, the algorithms in this group try to make the population evolve, so that the individuals continue optimizing a given objective function. The latter can be seen as a measure to indicate the level of adaptation to the solution space (Maier et al., 2018).

Within these, we differentiate between the following classes: Genetic Algorithms, Differential Evolution, Evolutional Strategy and Genetic Programming (Datta et al., 2019). What unifies all of these methods lies in a so-called selection method, which decides about the *strongest individual* to survive and be the base for the next generation (Settles and Soule, 2005).

This chance of survival is given by a probabilistic function whereas so-called higher fitness levels lead to an increased probability of survival. Those are usually measured along the solution's value in the objective function. As observed in nature, parts of the survivors are then joined in order to create an offspring through the crossover operator. In order to guarantee diversity in solutions, we introduce mutation operators which modify the new individual randomly (Maier et al., 2018).

Different from other methodology presented in this work, one of the main advantages of this group of methods lie in the need of little knowledge about the system or the problem itself. Yet if we consider knowledge, it is easily implemented as constraints such that we boost performance and gain on efficiency. Furthermore, this methods appears to have the widest range of implementation within the respective problems (Datta et al., 2019). Please refer to nFigure 13 for a graphical explanation of the algorithm. We will now

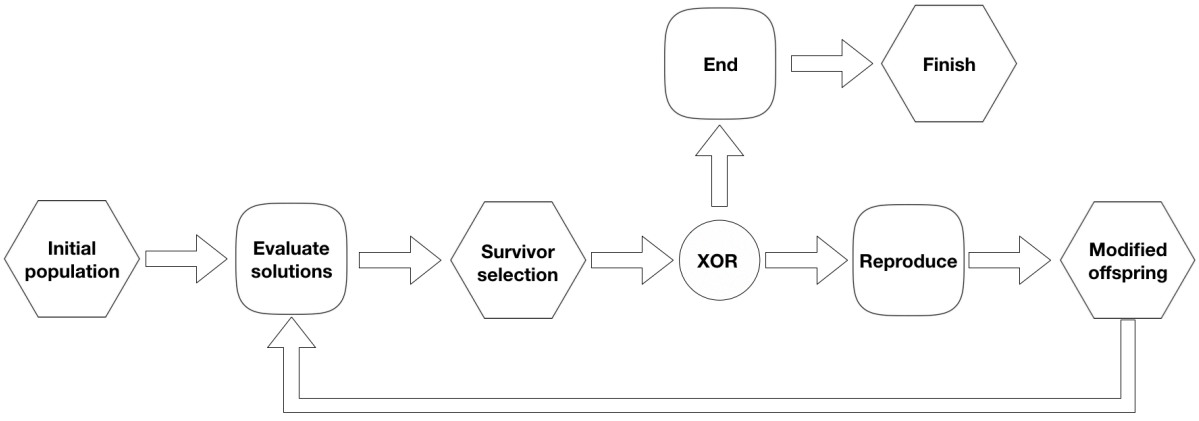

Figure 13: Methodology behind Evolutionary Computation

introduce the two most widely used algorithms belonging to the class of Evolutionary algorithms in the following.

- **Genetic Algorithms**, as introduced by Holland et al. (1992), are inspired by the evolution theory of Charles Darwin according to which the fittest individuals are selected naturally for reproduction and produce the offspring of the next generation

which inherits the characteristics of the parents. This notion can be applied to search problems (Darwin, 1859).

By mimicking the respective events of natural evolution through operators, we randomize the global search process. The Genetic Algorithm then selects the best current solutions, deleting the remainders. From the former we create an offspring, aka new solutions. This happens throughout each iterations such that at the end the authors hope for a perfect offspring, e.g. global optimal solutions (Holland et al., 1992).

In its pure form the authors used some minimal information input, retrieved from some heuristic, about the underlying problem domain. The basic components lie in using a random population of bitstrings $X^1 = \{x_1^1, x_2^1, ..., x_N^1\}$ where each one x_i^1 is a solution to the problem. By applying the aforementioned operators at each iteration $i = 1, ..., T$ we achieve evolution by applying k-times the following steps whereas $k \leq \frac{N}{2}$ is the amount of selections per algorithm and T the amount of generations.

We then select two x_i^1 for reproduction, usually the best solutions, through a crossover operator two offspring shall be generated. With only a small probability, we apply a random mutation on the offspring.

When this has been done for the maximum of $k \leq \frac{N}{2}$ amount of times, we create a new population X^{t+1} whereas $2k$ of the worst solutions from X^t and adding $2k$ of the new offspring. The best solution after T generations is then the solution of the algorithm.

As for the VRP, each particle represents one individual and is this represented as a chain of integers. Each integer, represents either a client or a vehicle.

We thus imply a fitness function $f_{eval}(s^*) = f_{max} f(s^*)$. In this we set $f(s) =$ totaldistance(s^*)+λ overcapacity(s^*)+μ overtime(s^*). The last two factors, namely the overtime and overcapacity give us the capacity and tour-time on a maximal allowed value

In a case where both constraints are violated, our objective function f returns the total distance traveled so far and are penalized in their fitness value for the violation. Otherwise, we use the weighting parameters λ and μ. The best solutions s will have a value $f(s)$ close to f_{max}.

Even though being generally easy to be implemented, drawbacks have to be taken into account: As in nature, there is no guarantee for optimality. This is explained by the difficulty in developing a fitness function, suitable to the problem. Although extensions have been founded, the nature given risk is not to be eliminated (Dréo et al., 2006).

- **Genetic Programming** is the most recent method in the field of Evolutionary Computing. The key difference to other techniques in the field of evolutionary computing lies in the way of presentation: While Genetic algorithms is presented as a list of actions, genetic programs are represented in tree structure. Therefore they are not fixed in length and are thus more flexible in use.
  Resulting from this, they easily grow on complexity but invalid states are only produced in exceptional cases. The explicit structure is thus important in order to entirely avoid operator precedence.
  It is thus a popular method, but due to its computational complexity, we will mention it only for completeness.

8.3.2 Particle Swarm Optimization

Introduced by Eberhart and Kennedy (1995), its name originates from the fact that it models the behavior of a flock of birds or a school of fish. It was originally proposed to solve continuous optimization problems but in the following modified in order to be able to solve combinatorial optimization problems (Clerc, 2004).

Similar to Genetic Algorithms (Section 8.3.1 on page 55) the solution is initialized with a sample of random solutions. What differentiates both is that we assign each potential solution a random velocity and then let them move through a so called hyperspace. We will be referring to these potential solutions as *particles*. Further, Particle Swarm Optimization (further denoted as *PSO*) has no evolution operators such as mutation, while genetic algorithms, as seen in Section 8.3.1 on page 55, have (Gendreau and Potvin, 2005).

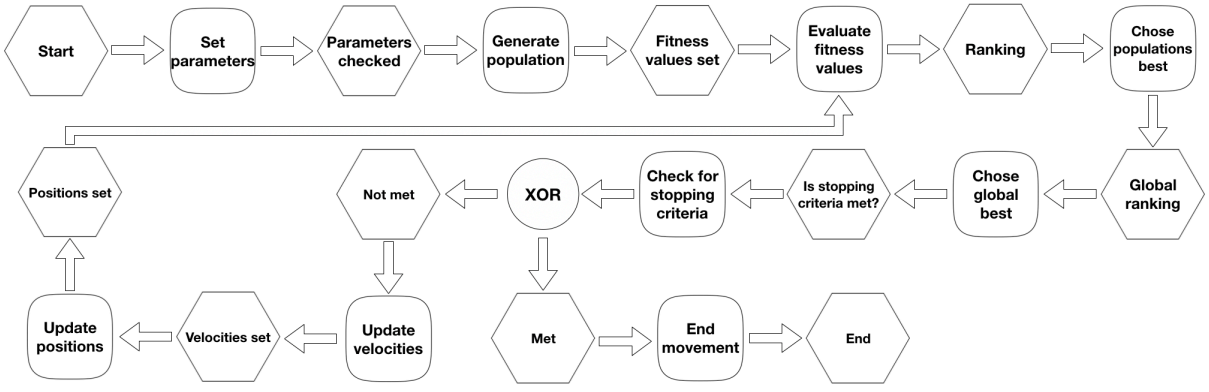

Figure 14: Methodology of PSO

Each particle keeps track of the coordinates of the best solution it has achieved so far. The optimizer also memorizes the coordinates of the best solution achieved in general by all particles and saves it. The global best is then tracked by the global version of the PSO (Gendreau and Potvin, 2005).

We will now further elaborate on the mechanics of the algorithm. The initialization process is defined by distributing the positions randomly to the respective agents. Then, they start moving in the search space by modifying their velocities and position in the following (Eberhart and Kennedy, 1995).

We start by assuming a reference problem, which delivers us $S = [s_1^{min}, s_1^{max}] \times [s_2^{min}, s_2^{max}] \times ... \times [s_d^{min}, s_d^{max}] \subset \mathbb{R}^d$. In case of a missing reference problem, we define specific definitions (Parsopoulos and Vrahatis, 2002).

We now introduce the particles of the swarm $A_i \in A$ with $i \in I = [1; N]$, which iteratively change their position in the search space S. In the agents' memory the information regarding the best solution is kept. The agents move, while their movement is stochastically biased towards the gained experience from previous moves and promising areas in S found by members of its swarm. For the swarm to exchange information, the properties of the individuals converge to one among the whole group (Eberhart and Kennedy, 1995).

One Agent A_i is henceforth characterized by the following four criteria.

- **Current Position** x_i^t where t stands for the respective iteration.
- **Velocity** v_i^t, responsible for the movement of the particle.
- **Particle's best position** p_i^t, which is self-explanatory. In the beginning, this is set according to the reference problem. The best position is a key factor, since it effects the velocity v_i^t.
  One problem arising from this, is that a particle only looking at its own best position, would limit the search capability in such a way that the particle would travel in circles to its best positions visited.
- **Information Exchange:** To overcome the difficulties pointed out in the last point, a subset of the swarm is able to communicate, e.g. $NB_i^t \subset I$.

In a neighborhood, the best position of a single particle can thus be defined as $p_{g_i}^t = argmin_{p_j^t, j \in NB_i^t} f(p_j^t)$, being the second effect on the factor velocity (Shelokar et al., 2007). After the communication phase, the velocity and positions of the particle change as follows:

$$v_{i,j}^t = \chi v_{i,j}^t + C_1(p_{i,j}^t - x_{i,j}^t) + C_2(p_{g_i j}^t - x_{i,j}^t)$$
$$x_{i,j}^t = x_{i,j}^t + v_{i,j}^{t+1}$$

where χ is denoted the constriction coefficient, responsible for keeping the velocity to a reasonable level, further C_1 and C_2 are uniformly distributed terms, being acceleration

constants. They control the attraction of a particle's position towards the neighborhood's best position (Parsopoulos and Vrahatis, 2002).

The first term, related to C_1 is called the cognitive term, since it involves only the information set by the particle itself, the term related to C_2 is the social term, e.g. denotes the communication between the particles (Parsopoulos and Vrahatis, 2002). The position updates happen via

$$p_i^{t+1} = \begin{cases} x_i^{t+1}, \text{ if } f(x_i^{t+1}) \leq f(p_i^t) \\ p_i^t, \text{ otherwise.} \end{cases} \tag{8.5}$$

The algorithm stops, when a certain stopping criteria is met. This can be a time limit or a limit set for the amounts of iterations $t \in T$. A global optima $gbest_{i,j}$ is thus defined, as the minimum of the respective best positions of the particles. With the help of a local search strategy the solutions are then improved. The respective best positions of a particle per iteration and the global best are stored in a memory component (Parsopoulos and Vrahatis, 2002).

Eberhart and Kennedy (1995) propose in their original framework the *global model* to give every particle a friend particle. This model has been improved by so called local methods, which had a slower convergence rate and would therefore not be trapped in a local minima (Parsopoulos and Vrahatis, 2002).

This method benefits from the few hyperparameters it employs. The method is so versatile that it can be used in many settings with only minor adjustments and is thus so popular that many extensions have been proposed Eberhart and Kennedy (1995). Those mainly aim to solve the VRP. We leave it to the reader to explore those as in Golden et al. (2008), Toth and Vigo (2002) or Settles and Soule (2005).

The PSOs we note several advantages. Through is foundation lies in swarm intelligent behavior, its use is not limited. Further, due to the missing overlapping and mutation operations, the amount of calculation is reduced. In order to find feasible solution, we can rely on the respective velocities of the particles. As only the best particles survive and are thus able to transfer information, solution finding is rather fast. Additionally, the few calculations needed to be done in this setting are simple compared to similar developing methods. Nonetheless, we find some drawbacks. One of the main criticism lies in the *partial optimism*. Further, the field where we can apply this metaheuristic is rather restricted (Ting et al., 2005).

8.3.3 Ant Colony Optimization

The Ant Colony Optimization (further denoted as *ACO*) was developed by Dorigo and Di Caro (1999) and was inspired by the experiments on real ants conducted by Goss et al. (1989). It tries to mimic the behavior of ants on their way on search for food to find the shortest path.

What makes this setting so intriguing is the fact that these animals are blind and their movement resembles a random walk. While walking, ants deposit pheromones leaving a chemical trace on the route they chose. Those appear to be very important, since they play an important role in the field of communication of the ants whenever food sources have been found. In other words if an ant comes along a way, following the pheromone trail, it can expect to be getting food (Dorigo and Di Caro, 1999).

It is important to mention that ants do not follow each and every pheromone track they find, since the strength in pheromone odor varies such that the ant might end up following a path with a lower level of pheromones or even opting for a new path at all. However, it remains to be said that the probability of following a path increases with the odor, e.g. the amount of pheromones deposited on it (Goss et al., 1989).

One major problem is that less reinforced by the insects, the pheromone trails are not meant to stay forever, since they tend to gradually evaporate. In other words, trails which no longer lead to food will be used less frequently, leading the ants to follow either the routes maintained with a high level of pheromone or to find new paths and eventually food sources. This is also one of the main advantages of this method, since it is keen to adapt to dynamic environments and is thus optimal for the usage in the field of urban logistics (Dorigo and Di Caro, 1999).

Another problem is that each ant, due to its simple moving and behavior is limited to find solutions of reasonable quality only. This is tackled by its colony, as through the global cooperation and communication within the ants better solutions can be found. To improve the understanding of this method, we will be examining a simple example, given in Figure 15.

In this figure, we can see that the ants choose path *A*, even though it seems to be the less efficient path. But a few ants decide to opt against the more frequented path and decide for the other trail. This one is advantageous, as the ants retrieve the food via path *B* faster, meaning that the pheromone trail is updated more frequently and is thus less prone to lose the pheromone

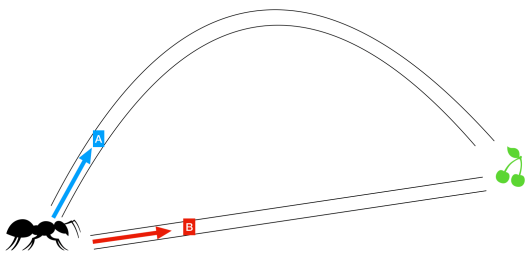

Figure 15: Ant Movement

trails due to evaporation which is the awaited destiny of path A such that only path B will be kept in use (Goss et al., 1989).

The probability of using a path is then given by

$$p_{i,j} = \begin{cases} \frac{(\tau i, j^{\alpha})(\zeta_{i,j}^{\beta})}{\sum_{k \in \Omega}(\tau i, j^{\alpha})(\zeta_{i,j}^{\beta})} \text{ if } v_j \in \Omega \\ 0 \text{ otherwise.} \end{cases}$$

Let $\Omega = \{v_j \in V \text{ where } v_j \text{ is yet to be visited } \} \bigcup \{v_0\}$ and v_j denoting each city. Further, τ denotes the amount of pheromones placed on the edge of i and j, ζ the desirability of the edge i and j (normally we assume it to be scaled with the distances between the edges, e.g. $\frac{1}{d_{i,j}}$. Last but not least, let α denote the attractivity of the path ahead, and β the independence of a single ant (Dorigo and Di Caro, 1999).

As presented we let each ant construct a solution by initiating from an empty solution $s[]$. At each step i we add a component k_i of the set of viable solutions of the neighborhood $N(s)$. Using the above mentioned probabilities a solution is now chosen. Further, for each solution pair $(i; j)$ there will be one separate probability, which can thus be explained by the varying levels of pheromone $\tau_{i,j}$ deposited on the track (Blum, 2005).

As in the Tabu Search, we can boost efficiency by thus creating candidate lists. These are computed ex-ante. One of the most prominent candidate list procedures lies in listing all available nearest neighbors (Blum, 2005).

As noted before, the most crucial factor lies in the pheromone trails, e.g their evaporation or renewal. The latter happens, whenever an ant chooses that path such that $\tau_{i,j}$ gets updated to $\tau_{i,j} = (1 - \rho)\tau_{i,j} + \Delta\tau_{i,j}$ whereas $\rho \in [0; 1]$ is a pre-set evaporation rate and $\Delta\tau_{i,j}$ the deposited pheromone. The latter is usually $\frac{1}{L_k}$, with k denoting an ant and L_k being the costs of travel. Usually we set the costs to be equal to the distance. The first part of the equation $(1 - \rho)\tau_{i,j}$ is further denoted as the evaporation rate (Colorni et al., 1992). Please refer to Figure 16 on the following page for a visualization of the methodology behind the ACO.

One drawback of this model lies in the direct unapplicability for combinatorial optimization problems once applied in its original form, since the associated pheromone levels could link directly to solutions. In order to overcome this, one must have full knowledge of the solution space. As our goal is to find an unknown optimal solution, this is rather impossible . For this, we consider the pheromone levels only to be only components of the solution. Those are then assembled in order to come up with a viable solution (Blum, 2005).

Further, we will explain how to solve the VRP using the aforementioned techniques.

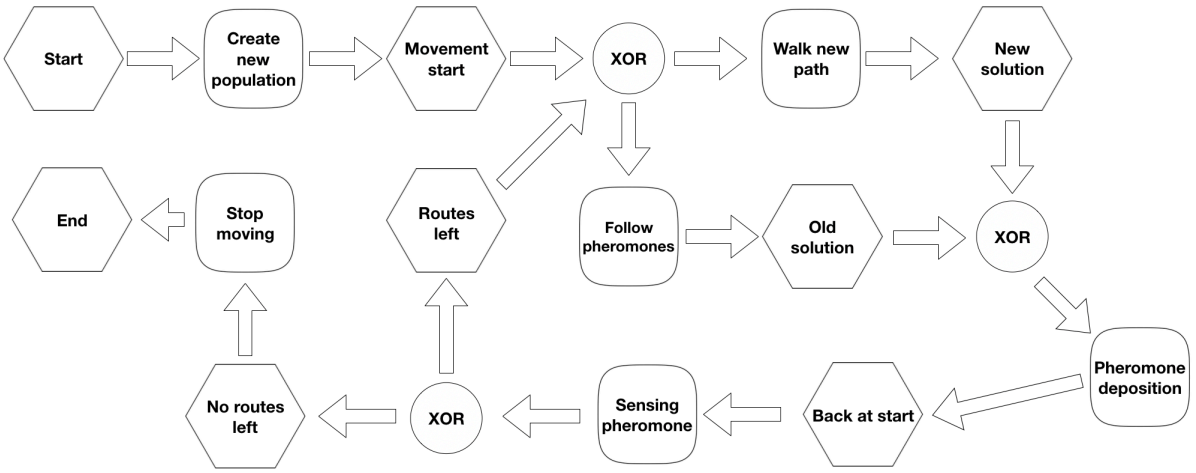

Figure 16: Methodology of ACO

Initially, we set the amount m of ants equal to the amount of customers, supposing they are located at different locations each. In the following, we set a maximal amount of iterations. During those, the ants generate new solutions and thus create candidate lists, by sorting the solutions according to the distances. For the respective routes, the ants choose the cities they want to visit, until every city has been visited. On every node, the decision is then to not choose a solution that would lead to unfeasible solutions (such as exceeding the capacity or maximal route length constraints). If no other possibility is left, the ant returns to the depot and starts anew (Dorigo and Di Caro, 1999).

Concerning the selection of a city that has yet not been visited, the ant on the one hand looks at the pheromone trail in order to get a glimpse of the routes quality. On the other hand, through the $\zeta_{i,j}$ parameter, as explained above. In the following we use the 2-opt heuristic to improve the routes. We then let the ants move again, updating their pheromone trails (Dorigo and Di Caro, 1999).

Due to its success, many extensions have been proposed, which will be elucidated only shortly, since the aim of this work lies in giving a general overview of existent heuristics.

- **Ant System:** Being the first, the pheromonical update rule is denoted as $\tau_{ij}^{new} = \rho\tau_{ij}^{old} + \sum_{\mu=1}^{\sigma 1} \Delta\tau_{ij}^{\mu} + \sigma\Delta\tau_{ij}^{*}$ (Dorigo and Di Caro, 1999). Let $\rho \in [0;1]$ denote the trail's persistence. From this, we deduce $(1-\rho)$ to be the evaporation rate.
  Only in the case that (v_i, v_j) has been used by the μ-th best ant, it gets a pheromone increase of namely $\Delta\tau_{ij}^{\mu}$. This is then equal to $(\sigma\mu)/L_{\mu}$. In the case that the μ-th best ant has opted for another arc, it is 0.
  Once considering all arc, which are in the set of arcs of the best solutions are then emphasized. In the model, we assume an amount of σ elitist ants to have used these arcs such that the pheromone intensity is increased by $\Delta\tau_{ij}^{*}$. This is then equal to $1/L^{*}$ if the arc is part of the set of the so far best solution Dorigo and Di Caro

(1999).

- **Ant Colony System:** As the first major improvement, we stark making use of a so called 'pseudorandom proportional rule'. This describes that each ant's movement now depends on a random and uniformly distributed variable $q \in [0; 1]$ and an additional parameter q_0 (Montemanni et al., 2005).
  If $q \leq q_0$, we choose the component which maximizes $\tau_{i,j} \times \zeta_{i,j}^{\beta}$, otherwise the equations from the traditional model hold. Through this, the exploitation of pheromone information is favored.
  On the other hand, we use a local pheromone update to limit exploitation. This can be described as a pheromone update performed by each ant after each step such that it gets applied only on the last edge. We describe this as $\tau_{i,j} = (1-\phi)\tau_{i,j} + \phi\tau_0$. The new parameters $\phi \in [0, 1]$ describe the pheromone deposition and τ_0 the initial pheromone level set.
  Further, our update functions is now $\tau_{i,j} = (1-\rho)\tau_{i,j} + \rho\Delta\tau_{i,j}^{best}$. The last part, describes the pheromone deposition of the best ant (Dorigo and Gambardella, 1997).

- **MAX-MIN Ant System:** As the name suggest, we set limits to the amount of pheromone deposition $\tau_{i,j}^{min} < t_{i,j} < \tau_{i,j}^{max}$. As the pheromone trails get updated, one would converge fast to only local optima. In order to tackle this problem, we set the update in such a way that in the case of $\tau_{i,j} < \tau_{i,j}^{min}$, $\tau_{i,j}$ is set to the new minimum. Analogously in the opposite case (Stützle and Hoos, 2000).

Even though it belongs to one of the most used and modified methods, according to the amount of Google Scholar citations, it has shown to be present drawbacks. As we have seen, theoretical analysis is rather difficult, compared to other (meta-)heuristics presented in this work. Further, our probability distribution changes throughout the iterations such that we can say the research to be rather experimental than from theoretical nature. Nonetheless, the algorithm converges to solutions, yet it is unknown how long this takes as the algorithm is usually stopped before final convergence is achieved (Blum, 2005).

On the other hand, due to the feedback system, results are obtained swiftly once positive feedback is obtained. For simpler problems, as the TSP, it shows to be an efficient solver. Last but not least, due to its dynamic nature, dynamic applications become solvable (Blum, 2005).

8.3.4 Artificial Bee Colony

In this setting we imagine a colony of bees consisting of three types of the animal, which mimic the foraging behavior observed in nature and according to their respective food

selection process: The employed bees, onlocker bees and scout bees. Their goal is to maximize the amount of nectar transported to their hive (Karaboga, 2005). This method will further be denotes as *ABC*.

We associate the first group with all information belonging to the specific food sources and with the work on the collection and transportation of the food to the hive. Such that the onlocker bees watch the initial group in order to deduce if a certain food source is not worthy anymore, e.g. the food source gets exhausted. These two are so called *local searches* which means that their search for food is based on a probabilistic and deterministic selection what holds mainly for the group of onlocker bees. The last group, the scouts, look for food in a randomized way without any previously acquired knowledge (Karaboga and Basturk, 2007).

The decision of which food source to be chosen is based on the amount of nectar found. In order to be able to do so, the scouts memorize the previous amount found and comparing them. If the newer source shows a higher amount of nectar, the bees update their memory, as the new solution is "better". If the source appears to have less nectar, the bee counts the amount of searches around that source in their memory. In the beginning we set 50% of the bees being employed, the same amount of unemployed bees and one scout only Karaboga (2005).

In this setting, each food source is seen as one solution candidate in the solution space, while the amount of nectar denotes the quality of this food source. We set the amount of food source equal to the amount of bees. The quality of the former is then set depending on the value of the objective function f on that position. We will further denote this by its *fitness value* (Kong et al., 2012).

We further differentiate between three phases (Karaboga, 2005):

- **Employed Bees Phase:** In this, scouts search for new solutions s^* aka food sources whilst food sources with more nectar from a solution's neighborhood $N(s*)$ are preferred. The location of a solution s^* is then memorized. When finding a new food source, their respective fitness levels are then compared by applying a greedy technique.

  The scouts become employed bees and start 'extracting food', returning to their hive in order to share the information with the onlockers in the hive. In the following, the employed bees can either return to their food source or settle.

- **Onlocker Bees Phase:** By the means of probabilistic choice, the onlockers choose the food source. This is biased by the information provided by the scouts. The selection is then based on the respective fitness levels.

  Once a selection has been made, the onlocker distinguish a neighborhood source.

The respective fitness value is then calculated and compared.

- **Scout Bee Phase:** In this, we differentiate between onlocker bees, whose solution can not be improved inside a limited amount of trials or iterations become scouts. The solutions found up to then, are then abandoned. A new random search begins.

This three steps are then repeated for a limited set of iterations. Refer to Figure 17 for a visual insight.

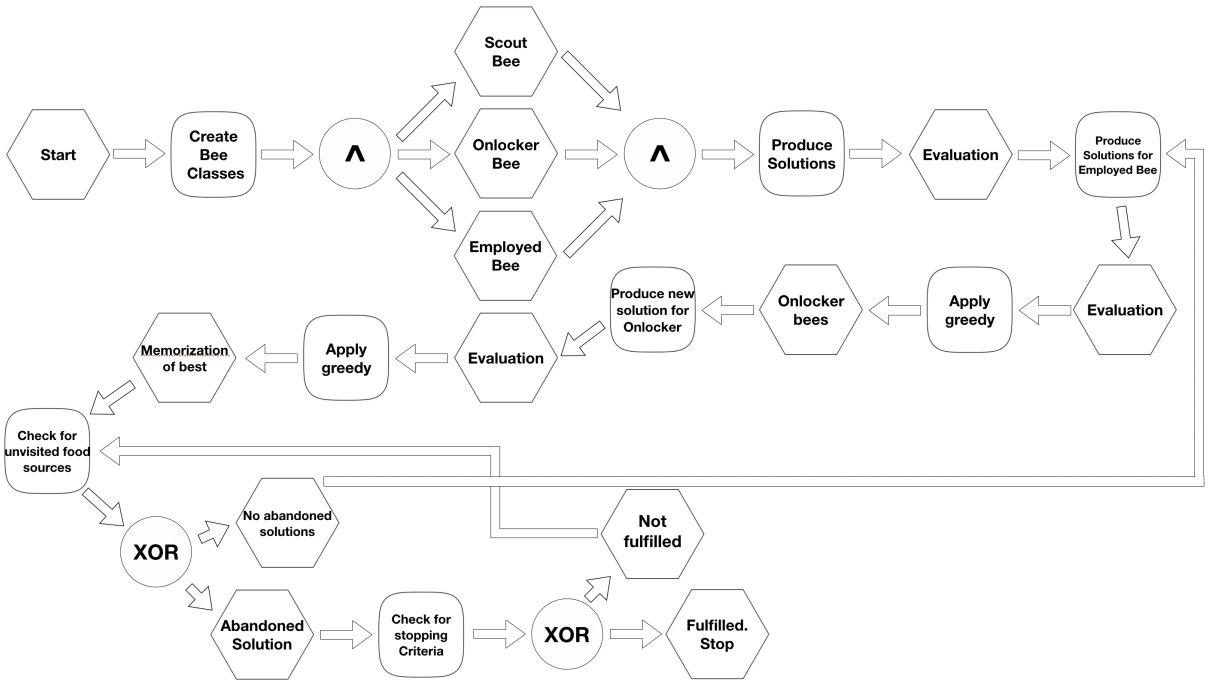

Figure 17: Methodology of ABC

By this, the food selection is based o the group‘s experience and their respective nest mates, repositioning the latter since communication between the bees is possible (Karaboga, 2005).

In the beginning we create a food source population by

$$s_{i,j} = x_j^{min} + rand(0;1)(s_j^{max} - s_j^{min})$$

where $rand(0;1)$ is a function to which a random number in the interval $[0;1]$ is assigned to. By

$$s_{i,j}^* = s_{i,j} + \phi_{i,j}(s_{i,j} - x_{s,j})$$

the employed bee conducts its search around the food source from their memory. If $s_{i,j}^*$ is better than $s_{i,j}$, we update the bee's memory. The parameter $k \in [1, FS]$ is a random index, and FS denoted the amount of food sources. And $j \in [1, D]$ represents a random

index, too. Further D stands for the dimension of the problem at hand (Karaboga and Basturk, 2007).

The communication between the employed and unemployed bees is done by assigning each solution found a normalized probability based on their fitness (incorporated as a quality measure), as shown in the next equation:

$$p_i = \frac{fitness_i}{\sum_{i=1}^{CS} fitness_i} \tag{8.6}$$

We could thereby show that the ABC balances both operations, exploration and exploitation, incorporating local and global search methods to obtain the best results possible (Karaboga, 2005).

Being one of the most recent metaheuristics, it impresses with its simplicity, flexibility and robustness. Due to the swarm behavior, it is thus able to explore local solutions handling the objective costs simultaneously. All in all, we can affirm it to be, once the parameters have been set, easy to be implemented for a huge amount of problem instances. Yet, every new parameter need to undergo fitness tests as we require a new and more evaluation of the objective function, making it thus slow in sequential processing

Different from the trajectory methods, we saw that the main forces behind population-based metaheuristics lie in the possibility of recombination of current solutions in order to obtain new ones. Even though powerful, we saw that the amount of parameter to be tuned increased, what makes the construction of the algorithm a quite complex task. On the other side, they prove to have their strength in slower convergence such that the risk of being trapped in local minima gets reduced.

For the Population-based metaheuristics we see due to the random explorations (caused mainly by the crossover coefficient) it is able to penetrate areas in the solution space normally not found by local search. Yet, the great amount of parameters necessary to be tuned in order to make the algorithm work is great. For the random exploration, performance replication is difficult, too.

In order to tackle the problems related to both groups of metaheuristics, we will try to take the respective benefits of the groups merging them into hybrid (meta-)heuristics, as explained in the following section.

9 Hybrids

Further it is possible to combine the aforementioned metaheuristics (and others not named in this work) to form more powerful and efficient algorithms. One must further note that in the group of *population-based* metaheuristics, techniques as in the ACO and ABC the hybridization is implicitly given, as often a Local Search operator is already implemented inside the code (Blum et al., 2011).

Now, we try to join the benefits, exposed in the respective ends of Sections 8.2 on page 43 and 8.3 on page 54 to form new and more powerful (meta-)heuristics. Research concentrated on the essence of merging existing (meta-)heuristics is vast. Yet, we differentiate between and will respectively explain the two groups, which combine (meta-)heuristics solely, as explained briefly in the following.

9.1 Collaborative Hybrids

In this setting, we combine one or more algorithms in order to obtain better results either in parallel or sequential manner. We can further weight the parts being used. In the normal case we distribute the weights equally among the (meta-)heuristics applied (Hogg and Williams, 1993).

As already mentioned, we are able to differentiate three distinct mechanisms, explained in the following.

a) **Multi-Stage Methods:** Refer to Figure 18 on the following page. We initiate the operation with *Algorithm I*, which is used as a global optimizer. After shrinking the search space, *Algorithm II* performs a local search on the restricted solution space. In other words, our walk in the search space gets to be more structured than in the separated manner (Ting et al., 2005).
   Commonly a population-based algorithms is used as the first algorithm, as those methods, as already shown, obtain feasible solutions by recombining solutions found by parts of the population finding promising areas in the search space. In other words, the exploitation done by the swarm is able to reduce the solution space such that further processing is eased. Individual-based algorithms do this recombination via one or more recombination operators, being more efficient in finding solutions, once the search space has already been reduced (Ting et al., 2005).

b) **Sequential Methods:** In this setting we let both algorithms run simultaneously, until we achieve convergence. For simplicity reasons, both algorithms are thus supposed to run the same amount of iterations Ciornei and Kyriakides (2011).

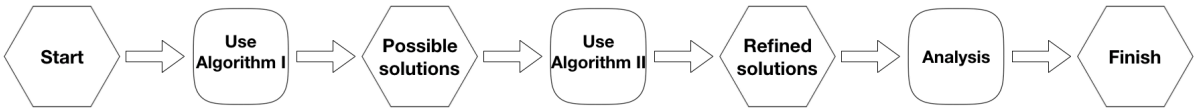

Figure 18: Methodology of Multi-stage Collaborative Hybrids

In the case of using two population-based metaheuristics, we achieve a better exploitation of the solution space in a more diverse manner than simply increasing the amount of particles of the respective swarm (Shelokar et al., 2007).
Please refer to Figure 19 for a graphical representation of the sequential hybrid running for two (meta-)heuristics.

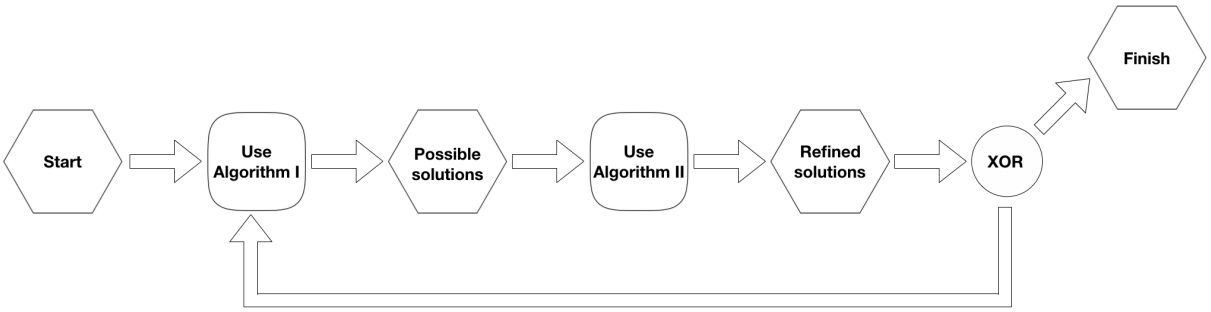

Figure 19: Methodology of Sequential Collaborative Hybrids

c) **Parallel Collaborative Methods:** In this setting, both algorithms run simultaneously whilst modifying the same pre-set hybrid population (refer to Figure 20). We thus allow one of the algorithms to be executed on an ex-ante defined time on the algorithm in order to pre-process it (Crainic and Toulouse, 2002).
Please refer to Figure 20 for a graphical representation of the sequential hybrid running for two (meta-)heuristics.

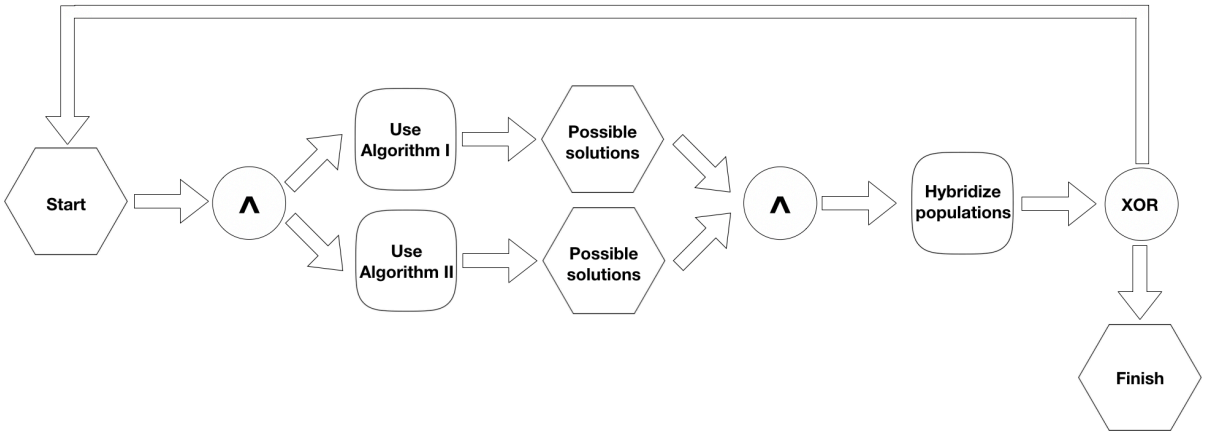

Figure 20: Methodology of Parallel Collaborative Hybrids

As most of these methods have never been used in the field of logistics, we name one example per sub-group such that the reader is able to deepen his knowledge on his own. Please refer to the Table 3 for examples.

Area	Example	As seen in...
Multi-Stage Collaborative Methods	Merge General algorithm and Particle Swarm Optimizer	Ting et al. (2006)
Sequential Collaborative Methods	Sequential run of Genetic Algortihm and ACO	Ciornei and Kyriakides (2011)
Parallel Collaborative Methods	PSO and ACO	Shelokar et al. (2007)

Table 3: Examples for Collaborative Hybrids

9.2 Integrative Hybrids

For the core idea of this type of hybrids, we divide our algorithms in two: A *master* and a sub-ordinate heuristic, the *slave*. The latter is then embedded in the first. The weight with which we do the embedding is normally chosen to range from 10% to 20% (Lozano and García-Martínez, 2010).

To be able to perform this embedding, there is the urge to have an operator, the manipulatory operator from the sub-ordinate to the master heuristic. This has been done by several authors, as shown in Table 4 on the next page. We now set to differentiate between the distribution of the respective manipulation, as follows.

- **Full Manipulation:** In this setting, at each iteration we take modifications on the population. Usually we apply this, by assigning one algorithm as a sub-routine into the source code (Kong et al., 2012).

Figure 21: Methodology of Full Manipulation Integrative Hybrids

- **Partial Manipulation:** In this setting, we accelerate the population partially with local search methods. Therefore, we just have to guarantee the right portion and thus the candidates to be accelerated by the slave(Tien and Li, 2013).

As most of these methods have never been used in the field of logistics, we again name only one example per sub-group such that the reader can enlarge upon interesting topics on his own. Please refer to the Table 4 on the following page for examples.

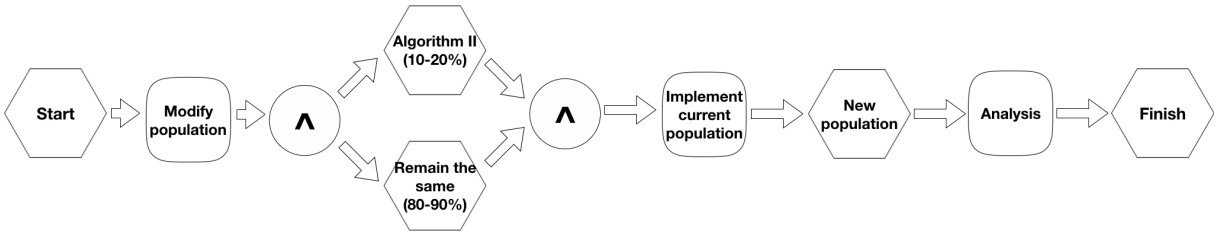

Figure 22: Methodology of Partial Manipulation Integrative Hybrids

Area	Example	As seen in...
Full Manipulation	Hybrid ABC	Kong et al. (2012)
Partial Manipulation	Hybrid Taguchi-Chaos of ABC	Tien and Li (2013)

Table 4: Examples for Integrative Hybrids

9.3 Other Hybrids

For now, we have just analyzed hybridization across (meta-)heuristics. Further, research has already implemented other ways of enhancing (meta-)heuristics, as the combination of exact methods and (meta-)heuristics (Blum et al., 2011).

Even though we have seen exact methods to lack in general performance mainly driven by computational time for routing, they are able to perform well on tasks apart from that topic resulting in more robust results. As we noted earlier, exact methods and their strength in attacking hard constraints problems and (meta-)heuristics in optimizing objective functions could be joined (Raidl, 2006).

In those, we use the prominent constraint programming. The main importance is now to define the areas in which the algorithms operate, one being the master and the other the slave (Blum et al., 2011).

Further, one may also use so-called math-heuristics. In those (meta-)heuristics are combined with mathematical programming techniques. The latter can be described as using mathematical models to enhance overall performance. In recent research, the already defined dynamic programming, relaxation, decomposition techniques or tree searches might be used to enhance (meta-)heuristics. Being out of the scope of this research, we leave it to the reader to investigate those methods by listing exemplary literature in the following (Maniezzo et al. (2009), Congram et al. (2002), Blum (2005), Raidl (2006)).

9.4 Gnowee

We will now present an even more complicated hybrid method, Gnowee, which was proposed for solving the TSP. Being a reference to the aboriginal goddess of the sun, the

recently published Gnowee algorithm proposed by Bevins and Slaybaugh (2019) aims to combine a set of diverse, robust heuristics to solve computational optimization problems. The authors promise it to converge rapidly to nearly global optima. This rapid conversion to a minimum fitness threshold is assumed to be thus an at least nearly global optimum, which is not being evaluated.

In the setting of this hybrid, the authors combine in a new way diverse, coherent and robust (meta-)heuristics. By this, the algorithm is designed to find the right balance of exploration and exploitation. Gnowee is as such a hybrid consisting of Neighborhood Search, Hill Climbing, accepting negative moves, Multi Start, Adaptive Memory Programming, Population-based Search, Intermediate Search, Directional Search and Variable Neighborhood Search.

While in Neighborhood Search the current solution is only changed step-wise, or equivalently one variable, the current solution is changed by only accepting moves to a better solution in Hill Climbing. By also allowing for negative moves, the solution space is more thoroughly explored. Multi-Start lets the search process restart from a different starting location, once it converged to a local optimum. Population-Based Search has been extensively discussed in Section 8.3 on page 54. Intermediate Search focuses on exploring the space between two highly performing solutions. Further, Directional Search methods learn the direction that a current solution can be improved in, typically based on gradients. Next, Variable Neighborhood Search extends basic Neighborhood Search by letting the definition of the neighborhood vary over the iterations for a wider exploration of the solution space. Finally, Search Space Mapping constructs a map or partitions of the solution space by incorporating expert knowledge.

By combining all these approaches, the authors were able to balance the different purposes of the methods, when being used separately, namely: balancing directional search and thus accepting negative moves. While the former is used to achieve rapid convergence, the latter aims to increase the chance to explore the solution space. Conclusively, it is less susceptible to local optima than the methods it incorporates. In order to accelerate the search process, the authors provide a parallelized implementation of their code (Bevins and Slaybaugh, 2019).

9.5 Concluding Remarks

Even though we named the many advantages of hybrids in the beginning of the Section 9 on page 68, there are some drawbacks apart from simple naming conventions, presented in the following.

As already noticed, we could see that the implementation of single metaheuristics might be

simpler than for exact methods, yet there are a few challenges left. When combining two or more metaheuristics, those difficulties tend to rise. The more complicated the structure of the algorithm, the more complicated it gets gaining insights on why the underlying method performs that well, turning it more and more into a black-box-method (Ting et al., 2005).

We can clearly observe this in the structure of the algorithm itself: As we create the need for extra components, the complexity of the algorithm rises as the amount of parameters increase, making it more difficult to tune the latter properly. This increased complexity puts stumbling blocks on the path of acceptance between researchers. Whilst analyzing the fields of citations on 'GoogleScholar', we can see that among hybrid methods, integrative hybrids seem to redeem most of the aforementioned success (Ciornei and Kyriakides, 2011).

Additionally, one of the main benefits of (meta-)heuristics was the easy implementation, which gets lost due to the increasing amount of parameters being more prone to errors. Further, the interpretation and analysis should be taken with much caution (Settles and Soule, 2005).

Furthermore it is noticeable that the convergence rate and computational speed improve on the use of hybrids. Yet, mainly the former is a measure needed to be analyzed separately, as the respective authors graphed it with respect to the number of iterations needed not the convergence speed itself. As we had already shown, hybrids tend to use more iterations, as these are done implicitly (Shelokar et al., 2007).

Further, we can say that research in the field of integrative hybrids is by far more recent than that of its counterpart, the collaborative hybrids. One explanation could lie both in the wider acceptance of the former from research and its simplicity in application.

10 Statistical Learning Methods

We initiate this by introducing one of the basic foundations as will be seen in the following. In this our aim is to build functions capable of predicting by establishing links between the variables in use (Hastie et al., 2005).

In our case, we use these predictions for the costs of an edge for a given state. This prediction is necessary, as the actual costs will only be revealed at the moment of realization (Arnau et al., 2018).

For the range of statistical learning methods is vast, we have to choose the right technique to solve the particular problem. As in the heuristics setting presented in the following, we need to deal with a trade-off between computational time, accuracy and interpretability (Hastie et al., 2005).

For this we will set our focus to machine learning methods which have the attention of recent research. Within these methods, the reinforcement learning appears most suitable to tackling the problems presented in Section 4 on page 8.

10.1 Machine Learning

Machine learning (further denoted as *ML*) denotes the field in data science and artificial intelligence and is one example in the field of statistical learning methods, in which algorithms laid upon datasets lead an engine to learn (Hastie et al., 2005). The result of this lies in the capability of the engine to make predictions based on the underlying dataset used for the learning process. In order to classify the wide range of machine learning algorithms, three basic ML paradigms have been established.

We differentiate between:

- **Supervised Learning.** This is the broadest field in ML. The basic concept of this method is a supervisor which gives direct feedback on the results of the algorithms. This is achieved by specifying an error function in order determine whether the algorithm is approaching or deviating from a pre-defined goal. In other words, we can assume this method to be based on a ”trial and error” function (Kotsiantis et al., 2007). Possible application areas range from simple regression tasks up to classification.

- **Unsupervised Learning.** In unsupervised learning, the dataset is analyzed with respect to unknown patterns which cannot be unambiguously evaluated. As there is no feedback per se, the algorithm learns by identifying commonalities in the data upon which it reacts. Among others, this paradigm covers dimensionality reduction

and clustering. The main goals of this field lie in anomaly detection, dimension reduction, time-series modeling, and latent variable models (Hastie et al., 2005).

- **Semi-supervised Learning.** This is similar to supervised learning, but as we use this technique in fields where large amounts of unsampled data is available, not all examples have an output value.

Further we will be taking a closer look at Reinforcement learning, since this methodology has been widely used in order to either tackle problems as the traveling salesman or vehicle problem as well as helping heuristic methods, as can be seen in Section 11 on page 78.

10.2 Reinforcement Learning

In operations research, we typically make use of reinforcement learning algorithms belonging to the class of supervised-learning (Hastie et al., 2005). In these kinds of methods, we assume that an agent in a given state can decide among a finite set of actions. Dependent on the decision, the state and the set of possible actions changes. The agent can further get an immediate reward, such as a result of the respective action, whilst maximizing the expected reward is the objective of the agent. The reinforcement is often modeled as stochastic Markov decision process (Kaelbling et al., 1996).

In reinforcement learning, we let an agent learn the situation-dependent effects of his actions. To be more specific, we let an agent operate in a pre-specified environment. We let him then choose among a set of environment-state and agent-state-dependent actions. Based on the states and the taken action, the agent receives an immediate reward and the environment and the agent states change accordingly. The agent aims to maximize his cumulative reward (Kaelbling et al., 1996).

Even though ML arose as new innovative field in the recent decades, it cannot be clearly distinguished from traditional methods such as prominent (meta-)heuristics. For example, the ACO can be interpreted as distributed reinforcement learning technique as has be seen in Section 8.3.3 on page 61.

In the following, we present an exemplary method which stresses the incorporation of reinforcement learning even more.

10.3 Ant-Q

The most prominent learnheuristic was introduced by Gambardella and Dorigo (1995), a family of algorithms building upon the well-established reinforcement learning method Q-learning (Watkins and Dayan, 1992), and which were then applied to the solution of symmetric and asymmetric instances of the TSP.

This Q-Learning, being part of the group of reinforcement methods, aims to recognize or learn how to perform the best action for every respective state visited by the system. The underlying methodology is the so-called *trial and error*. The agent chooses one plan of action and receives immediate and direct feedback, based on a reward $r(i, a, j)$ plus the value of the upcoming state for this. The latter is then used to update the agents database. Let further i denote the state the agent is in, a the action taken and j the upcoming state (Kaelbling et al., 1996).

The database consists of a so-called *Q-factor*, which is established for every pair of state and respective action. The procedure is now as follows: Whenever the agent gathers positive feedback on the action, the respective Q-factors value increases. Analogously we find a decreasing value on negative feedback. Further, the value attributed to the Q-factor is also the value attributed to the following state j and is defined by the maximum Q-factor of that state. In the case of having two actions in a state, we assume the maximum of both Q-factors to be the value of that state (Kaelbling et al., 1996).

In this setting, we once more define the problem as a graph, yet the nodes stand now for low-level heuristics related to directional edges. In the initial setting, we disperse our fixed amount of ants on those nodes such that they start their way.

This procedure can be described as follows, each ant decides which other node to visit, applying the heuristic to the problem at hand. As in the classical ACO, each ant deposits pheromones on the track such that other ants are able to follow. For this to be efficient, we further assume the ants to do this kind of exploration in an improved manner, e.g. having a learning mechanism which is then able to guide the search.

Put differently, the ants interact with the solution environment by taking into account previous results such that they try to predict future utilities of exploring certain nodes. During this, we are able to gradually construct sequences of those heuristics.

Once applied to the TSP, we have our problem specification as already explained in Section 4.2 on page 12. Let us further define the positive and real number Ant-Q-Value $AQ_{i,j}$, which is similar to the traditional Q-value form the Q-learning algorithm. This denotes the usefulness of making a move from node i to node j.

Further, we have have the heuristic value of an edge denoted with $HE_{i,j}$. This one is used, when we want to evaluate the quality of the moves. In the example of the TSP we take the inverse of the distance as a measure.

In the following the authors describe an ant k, responsible to create a tour. We assume all ants $k \in K$ to have memory such that in the latter there is a list $J_k(i)$, denoting the nodes which have yet not been visited where i denoted the node. By that we can say this algorithm to have a memory component. This list is needed, since our aim lies in

proposing a tour which consists of visiting all nodes only once.

The movement of an ant k is then guided by the following rule:

$$j = \begin{cases} argmax_{u \in J_k(i)}\{[AQ_{i,u}]^{\delta}[HE_{i,u}]^{\beta}\} \text{ if } q \leq q_0 \\ J \text{ , otherwise.} \end{cases} \tag{10.1}$$

Let $q \in [0;1]$ be a randomly and uniformly distributed value, q_0 a parameter, denoting the probability of making a random choice such that a high value on q_0 imposes a smaller probability in making a random choice. The symbols δ and β are weighting parameters weighting the relative importance of the respective AQ and HE parameters. Further, J denotes a random variable selected according to the probability distribution of the aforementioned parameters AQ and HE.

As stated before, the respective AQ-values get updated by $AQ_{i,j} \leftarrow (1-\alpha)AQ_{i,j} + \alpha(\Delta AQ_{i,j} + \gamma max_{z \in J_k(i) AQ_{s,z}})$. Let further α denote the learning step and γ a discount factor. Usually, $\Delta AQ_{i,j}$ is zero except in cases when all agents have completed their respective tours. This is normally biased towards the tour length, e.g. $\frac{1}{L_k}$. By the list, denoted in $J_k(i)$ we have a memory component.

We now explain this in the setting of the VRP. As proposed initially, we have a set of n cities and the respective distances $d_{i,j}$. Following Equation 10.1, the vehicles located in i are now supposed to decide which city to visit next (Gambardella and Dorigo, 1995).

We then assign an initial AQ-value and place our vehicles on a city i. We further create the list of unvisited cities J_k. In the following, each vehicle moves and the respective AQ-values are thus updated. This will be repeated until all cities have been visited and thus have returned to the depot (Gambardella and Dorigo, 1995).

After the return, the tour length per vehicle is calculated such that only the AQ-values of the globally best tours get updates. In the last step, Gambardella and Dorigo (1995) check for the stopping criteria, which is usually a fixed amount of iterations. In cases where the optimal tour is reached earlier, the algorithm stops before the maximum amount of iterations is reached.

Gambardella and Dorigo (1995) imposed this algorithm for the TSP, yet the implementation for the VRP should be possible, too, yet not employed by researchers for this purpose.

11 Learnheuristics

Whilst we could see that exact methods of those NP-hard combinatorial optimization problems require either a simplification of the problem or addressing prohibitive computing times. Following (meta-)heuristics, it could be shown that the optimality of the provided solution may not be granted, yet are beneficial in terms of computational time. Additionally, those methods, especially in terms of metaheuristics, are applicable with more variety, since they are not problem specific and can therefore be used under more realistic conditions (Sörensen et al., 2018).

One different approach would be to combine machine learning features, as introduced in Section 10 on page 74, with the previously explained idea of meta-heuristics. This is thus an extension of the hybrid models introduced in Section 9 on page 68 (Calvet et al., 2017). Due to its importance in recent research we dedicate the field of 'learnheuristics' a separate chapter.

A special benefit arising from the use of learnheuristics is the possibility in solving combinatorial optimization problems with dynamic inputs. Those are special, principally in practical use, e.g. real world conditions, since the problem inputs, either the constraints or the objective function itself, are not fixed upfront (Calvet et al., 2017).

The latter might surplus vary, possibly not randomly, but for the statistical learning method in a predictable way. This results from the matter of fact that we build the solution partially by the means of an iterative process, based on heuristic procedures (Arnau et al., 2018).

Furthermore, the variability in the constraints is plausible, since for example a consumer's willingness to pay a specific price for a certain product varies with the availability of the latter, as this implies a rise in the price, which is nothing but basic economics (Hazlitt, 2010).

Mainly because of this variation in input factors, the coordination between the learning and metaheuristic mechanisms must be set such that at each iteration, the learning procedure is trained and updates within the input are then implemented in the following by (meta-)heuristics (Arnau et al., 2018).

We differentiate between methods where ML is used to improve (meta-)heuristics and the other way around. Both methods will be elucidated shortly in the following sections (Calvet et al., 2017).

11.1 Hyperheuristics

The term hyperheuristics was initially founded by Cowling et al. (2000). These methods aim for a simplification of the search space, e.g. reducing the search space, boosting efficiency and are thus part of the group where ML is used to enhance (meta-)heuristics.

While in the field of (meta-)heuristics our search lied in the search space of solutions to the problem at hand, hyper-heuristics aim to find the best (meta-)heuristics within the search space of (meta-)heuristics. For this, not only single techniques are searched but also in which sequence those (meta-)heuristics shall be applied. Therefore not the solution to the problem is the goal but rather the methodology in order to do so.

We thus say that these operations work on a higher level than metaheuristics, as seen in Figure 7 on page 40. Technically, hyperheuristics choose which low-level (meta-) heuristic is to be used for any given instance for the type of underlying exploration of the search space. Cowling et al. (2000) introduced this methodology in order to solve scheduling problems.

We can further differentiate between hyperheuristics able to select or even to generate own or even new heuristics. Please refer to Figure 23 on the next page. The algorithm is thus able to learn, via a feedback mechanism. The latter can be achieved in three distinct forms (Epitropakis and Burke, 2018):

- **Online Learning:** These types of hyperheuristics do the learning while the algorithm is solving the problem at hand. In this approach, we have two separate search spaces: the space with the sequences of the respective heuristics and the solution space of the problem itself whereas the former is usually a subset of the latter. Throughout the search procedure, the algorithm is able to learn.
- **Offline Learning:** These types of hyperheuristics gather knowledge in the form of rules or programs, from an a priori set of training instances, capable of generalizing the solving of the problems at hand. The learning in this setting gathers knowledge and tests it on a test set which is composed of unseen instances.
- **No Learning:** As the name suggests, no learning is implemented in this setting.

Further, one can differentiate between the analysis itself. Pertubative methods consider the whole solution space, changing them by modification of one or even several components. Constructive methods on the other hand consider the solution space only partially such that one or more components of the respective solutions are missing. By this, the algorithm needs to extend the missing solutions (Calvet et al., 2017).

The procedure for the high-level heuristics is to efficiently search the solution space, restarting it with different sequences. For this the already introduced hybrids could

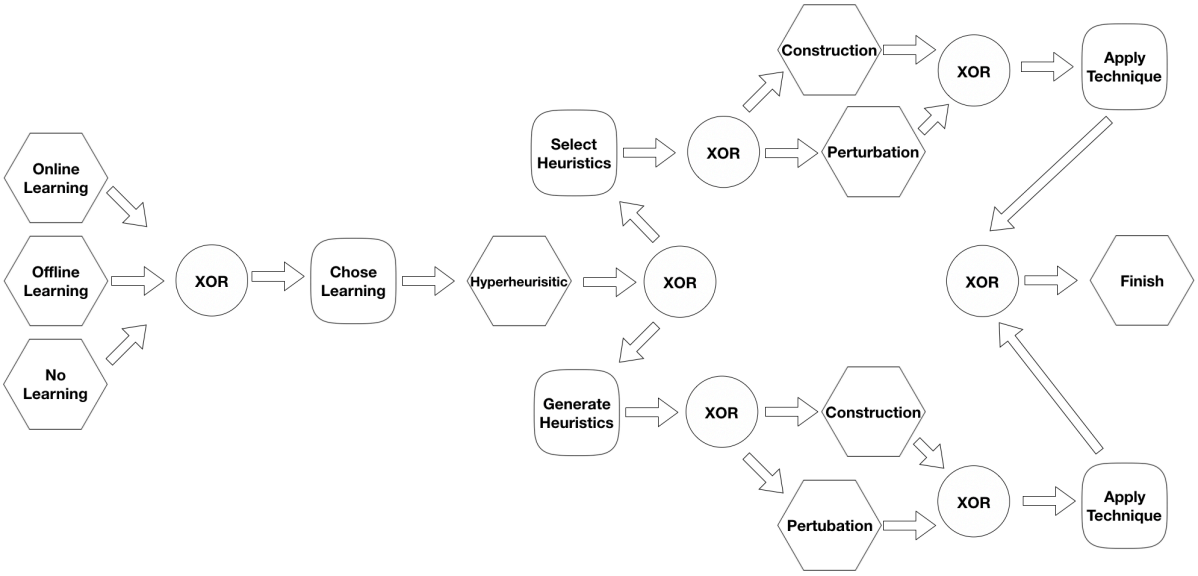

Figure 23: Methodology of Hyperheuristics

come in handy, as those might conduct the search in both solution spaces simultaneously (Epitropakis and Burke, 2018).

Being introduced recently to the field of logistics, Garrido and Castro (2009) were the first to use hyperheuristics in the field of VRP. Based on the already explained hill-climbing methods, they search the heuristic space consisting of sequences of constructive and pertubative pairs of low-level heuristics. In the following, the authors apply those sequences constructing or even improving already obtained partial solutions.

In a follow-up analysis, the authors improved their results, by augmenting the methodology (Garrido and Riff, 2010). Their method was able compete with some renown heuristics proposed in the previous sections.

As the implementation of hyperheuristics is not usual in the field of logistics problems, we name it for completeness and leave it for further research. Not to be forgotten is the fact that through these techniques, we might be able to broaden the level of generality of search methodologies, even of (meta-)heuristics.

11.2 Specifically-located Hybridizations

Crucial for a (meta-)heuristic's performance is the tuning of the parameters involved. Before machine learning techniques were merged with (meta-)heuristics, researchers did not tune their parameter, but used values generated through exhaustive testing or simply copying values used in similar problem settings (Birattari, 2004).

One can achieve this by controlling the parameters directly, implying deterministic or adaptive rules. The latter bases its results on some sort of feedback function, as in Jeong

et al. (2009) or Battiti and Brunato (2010). Another approach would lie in tuning the parameters. By this, the algorithm is said to be robust enough such that it is comparable to values from comparable problems with fixed parameters. Last but not least, we combine the aforementioned techniques and fix the parameters but letting them be specific for each instance. All in all we can say that this kind of techniques is not often used in solving logistic problems, due to their complexity in application. Yet, once the algorithms have been adapted to certain problems, they become powerful solvers of the aforementioned problems (Battiti and Brunato, 2010).

In Table 5, we review the two main groups and compare their methodology. This might be used on further research to access the right method for the right problems.

Global Hybrids	Specifical Hybrids
Reduction of Search Space	Fine Tuning of Metaheuristic parameters
Selection of algorithms	Initialization, Evaluation
Implementation of Hyper-heuristics	Populations Management
Cooperative Strategies	Operators
New Metaheuristics	Local Search

Table 5: Key differences between global and specifical hybrids of ML and (meta-)heuristics

11.3 Neural Networks for Learnheuristics

Neural networks (NN) are one of the most popular state-of-the-art machine learning methods. They are widely used in the most diverse data processing areas: speech recognition, image classification and machine translation. In simple terms, neural networks can be described as successive transformed regressions. This means that over several layers where each layer contains one regression, the input data is firstly multiplied by weight matrices, secondly modified by the addition of a bias, thirdly non-linearly transformed and eventually taken as the input of a following layer if the corresponding layer is not the last layer of the neural network. Please refer to Haykin (1994) for a thorough introduction to the topic.

Special neural networks which take sequential data as input are called *recurrent neural networks* (RNNs). These networks are characterized by a chain-like architecture that is the successive concatenation of simple NNs. Simple RNNs typically only condition on recent information such that each neural network in the chain incorporates knowledge gained by the preceding neural networks. However, RNNs tend to forget over long periods. To encounter this problem, Hochreiter and Schmidhuber (1997) introduce *Long Short Term Memory* (LSTM) networks, as special kind of RNNs. Their signifying property is to remember information over long periods.

The different dimensions of the input to a neural network can have different importance for predicting the output. When a neural network has, for instance, to differentiate between a picture from a cat and a picture of a dog, it should not focus so much on the fact that both have four legs, but more on the different aspects of the pets' heads. As this example shows, it is reasonable to let the neural network put varying focus, respectively *attention*, to the different dimensions of its input. This is made possible by the inclusion of attention mechanisms (Vaswani et al., 2017). In technical terms, attention defines a distribution over the weight parameters such that they become random variables and are not longer fixed parameters.

In order to understand how NNs can be applied to routing problems, we have to consider the aim of routing to be to generate an optimal sequence of locations. This optimal sequence is typically learned based on supoptimal tours, which can also be expressed as sequences. As such, we aim for LSTM networks which transform sequences into sequences. Sequence to sequence models are based on an encoder-decoder architecture. This means that a first LSTM finds a latent representation of the input sequence (it thus encodes the input sequence), while a second LSTM decodes the latent representation. That means it transforms the latent representation back to a sequence.

Pointer networks, introduced by Vinyals et al. (2015), are neural networks which transform a sequence by yielding a succession of pointers to the elements of the input series. In contrast to sequence to sequence models, they thus do not output a sequence. The models perform better when augmented by attention.

Bello et al. (2016) were one of the first to introduce the implementation of neural networks to solve the TSP. They train a LSTM model that predicts a distribution over different location permutations given a list of all locations. A permutation is here one possible tour which includes each location exactly once. The model parameters are learned by reinforcement learning where the negative tour length is given as a reward to the agent. In particular, Bello et al. (2016) use policy gradient learning where the agent learns parameters to characterize the action he takes conditional on his and the environment's state. The conditional action decision is also referred to as *policy*. When the parameters are learned by improving them in the direction of the reward, we speak of *actor-critic* policy gradient methods since the reward serves as a critic to the agent's policy (the *actor*). The neural network is first learned on a training dataset. For instance, its performance is optimized for solving the TSP on a set of several graphs with 20 nodes. When the parameters of the NN are learned, the model can be seen as a fixed metaheuristic. The network can thus simply be applied on another graph with 20 nodes which it has never seen before.

Recently, Kool et al. (2018) extended the model proposed by Bello et al. (2016). The

authors train this model using REINFORCE with an effective greedy rollout baseline. This means that they specify the training procedure of the NNs in more detail.

The benefit of this kind of learnheuristics lies especially in its flexibility to learn strong heuristics for the most diverse settings.

Further, we might also use metaheuristics in order to improve machine-learning. In the field of supervised learning, we can enhance the machine learning algorithms in both its main functions, classification and regression. In the unsupervised field, improvements can be achieved in the fields of clustering and rule mining. As these fields do not belong to the group of logistics problem solving, we won't provide further explanation on the topics. We advise the reader to discover the following literature in order to gain knowledge on this field of learnheuristics: Stein et al. (2005), Candelieri (2011) and Turner and Miller (2014)

12 Experimental Setup

For our own empirical study, we propose as similar simulation setup as presented by Leyerer et al. (2018).

12.1 Problem Definition

In order to cover as wide a range of different settings as possible, we will take a look on the major routing problem concepts introduced in Section 4 on page 8. These are the TSP and the VRP. We regard the VRP without capacity constraints such that we minimize the length of longest route taken by one vehicle instead of the total distance covered. We regard a simple model where goods do not have to be submitted at a given time.

We won't be considering location problems, such as facility location problems, since these are not as common as routing problems in urban logistics. We suggest that further research can evaluate our simulation setup for location problems.

We focus on the most important part in the delivery process, the last-mile. For concepts of today's business needs could be explored as well: For example food deliveries have to operate on a first-comes-first-serves basis such that ready meals are delivered before they get cold. The mathematical intuition behind this problem has already been given, refer to 4.3 on page 14.

12.2 Dataset

For the comparison of theoretical properties of (meta-)heuristics, it is most useful to evaluate their performance on simulated data. Nonetheless, practitioners, one of the main users of the applied methods, are often not interested in these theoretical properties of (meta-)heuristics. Instead, they value those methods more which work best in practice. For this reason, we also measure the performance of the different algorithms on a real world data example with use case. In the following, we will be giving more insight to the datasets used.

12.2.1 Simulated Data

In a first step, we evaluate the performance of the optimization algorithms on simulated data for the TSP. For this purpose, we simulate respectively ten completely connected graphs with 30, 70 and 100 nodes. The x- as well as the y-coordinates of the nodes are drawn independently from a uniform distribution over the interval 0 to 100. The distances between locations are calculated using the Euclidean distance $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$,

which results in minimizing the air route between two coordinates (x_1, y_1) and (x_2, y_2).

However, there is no special reason to use this definition. Other possible distance measures include, for instance, the Manhattan distance $|x_2 - x_1| + |y_2 - y_1|$ which measures the distance between two points along axes at right angles. This is inspired by the gridlike street geography of the New York borough of Manhattan.

The distances are then rounded to integers. The first simulated coordinate in each graph is set equal to the depot. We simply set the cost to be the distance between locations.

In the following we will exemplary present the obtained distances for one graph with ten nodes in a distance matrix.

$$\begin{bmatrix} 0 & 35.54 & 85.02 & 54.23 & 80.30 & 37.67 & 21.09 & 38.39 & 54.55 & 44.68 \\ 35.54 & 0 & 49.88 & 25.89 & 44.78 & 23.66 & 19.36 & 3.53 & 19.40 & 35.27 \\ 85.02 & 49.88 & 0 & 46.87 & 13.10 & 63.67 & 65.70 & 46.75 & 30.52 & 71.50 \\ 54.23 & 25.89 & 46.75 & 0 & 35.90 & 19.89 & 44.03 & 26.42 & 26.94 & 24.94 \\ 80.30 & 44.78 & 13.10 & 35.90 & 0 & 54.3 & 62.7 & 42.2 & 26.9 & 60.9 \\ 37.67 & 23.66 & 63.67 & 19.89 & 54.35 & 0 & 34.46 & 26.53 & 37.07 & 11.65 \\ 21.09 & 19.36 & 65.70 & 44.03 & 62.71 & 34.46 & 0 & 20.95 & 35.89 & 45.21 \\ 38.39 & 3.53 & 46.75 & 26.42 & 42.20 & 26.53 & 20.95 & 0 & 16.22 & 38.04 \\ 54.55 & 19.40 & 30.52 & 26.94 & 26.98 & 37.07 & 35.89 & 16.22 & 0 & 47.20 \\ 44.68 & 35.27 & 71.50 & 24.94 & 60.93 & 11.65 & 45.21 & 38.04 & 47.20 & 0 \end{bmatrix}$$

12.2.2 Real World Data

In order to incorporate an empirical application, we consider the logistic concepts in the German city Hanover, the capital of Lower Saxony. With a total area of 204.01 km^2 and over 500,000 inhabitants, Hanover is the largest city in Lower Saxony.

The problem we want to solve is the following. In order to promote the annual event 'IWI-Grillen' from the institute of business informatics at the Leibniz University Hanover, we would like to distribute flyers in all university campus in the local area. Let us further assume that the amount of flyers in our case is infinite and stored at the IWI-institute itself, being our depot.

We pre-selected 38 locations, which are all part of the campus of the universities in Hanover. As our aim is to have as many guests as possible, we even consider campus seemingly out of range, such as in the north-western of Hanover in 'Garbsen' or the southeastern campus in 'Südstadt'.

In order to reduce the fossil fuel burn and consequently pollution, as being one of the main problems encountered in the fields of logistics, bicycles are our vehicle of choice. The capacity constraint is thus a simple backpack, which will not be specified further,

since we do not impose a demand to be supplied, but simply placing the flyers on the most frequented places on the campus. We further employ students to make the delivery. As those seem not to have good fitness conditions, we aim to reduce the longest tour. In

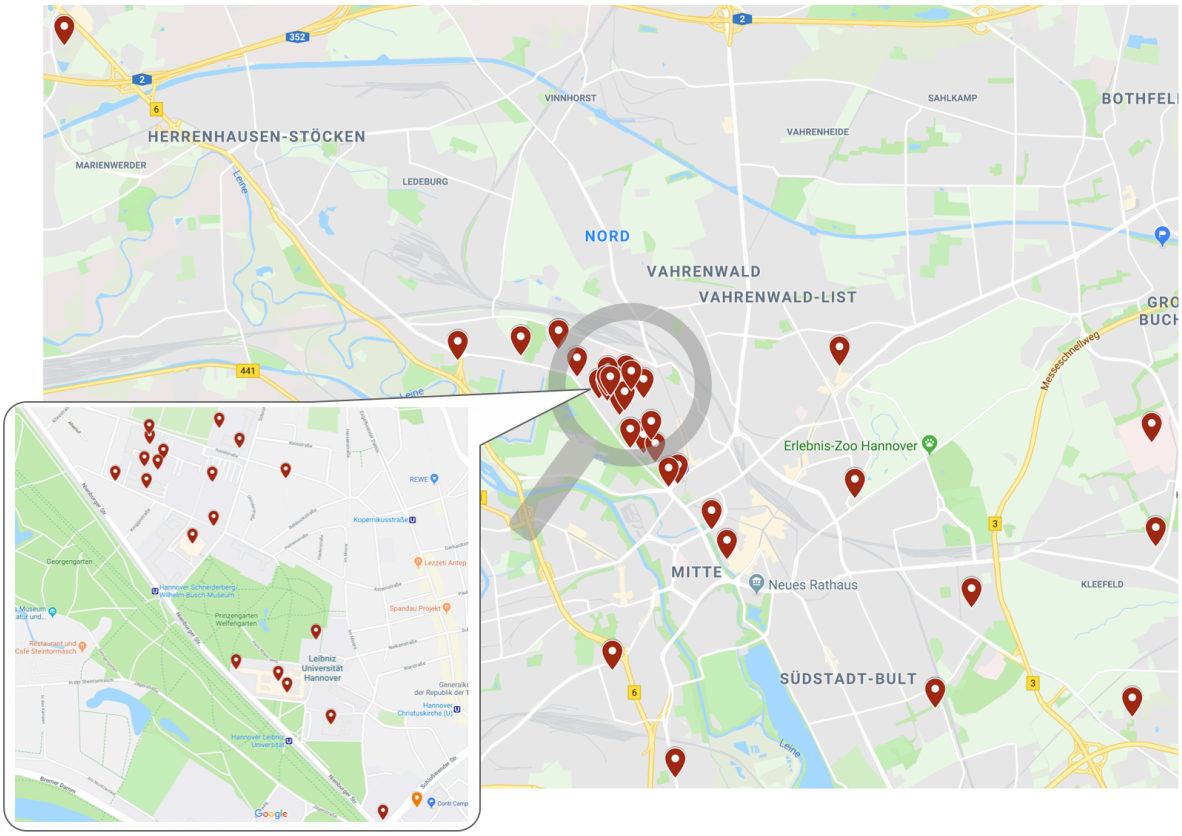

Figure 24: Location of Hanover universities in a map extracted from Google Maps

Figure 24, we illustrated a map of the catchment area. The localizers (marked in red) denote the locations we will be departing to in our empirical study. Those are the locations of 38 university campus and the depot, marked in orange, in Hanover and Garbsen.

In Figure 24 the cluster in the northern part of the city catches our attention ('Nordstadt'). This area has been augmented, as denoted by the magnifying glass. Even then, the clustering is still existent such that those areas will most probably be served in one setting, due to the proximity of the locations.

We extract the traveling time using the Google Distance Matrix API. We use the default parameters given on `https://developers.google.com/maps/documentation/distance-matrix/intro#DistanceMatrixRequests` with the change that we set the `mode` to bicycling. One reason for us to choose the Google API is that Google considers usual traffic conditions and averages those for bicycle drivers.

12.3 Methods

It is not in the scope of this paper to compare all available methods, but to focus on a comparison of the most influential algorithms. We chose at least one algorithm per exact method, heuristic and metaheuristic respectively. Namely, we considered 'Branch-and-Cut', 'Greedy Descent', 'Guided Local Search', 'Simulated Annealing', 'Tabu Search', 'Ant Colony', the recently added 'Gweeno' and the 'Learnheuristics imposed by Kool et al. (2018)'.

Further we let the respective method solve the problem at hand, measuring both, the time needed in order to come up with a feasible solution and the distance of or time needed to finish the longest route obtained by the method.

12.4 Evaluation Criteria

We will be evaluating the quality of the solutions, based on the following criteria proposed by Talbi (2009):

- **Quality of the solution:** This measurement is usually used to get a glimpse on the precision of the algorithm. For this, we might take a previously calculated global optimum based on an exact algorithm as a comparative measure. If the latter is not available, we might take the comparison to the lower respectively upper bound solutions. Those are normally easy to compute and will not be elucidated any further in this work.

  Another method would be the comparison to the best known solutions, available through open libraries or to those solutions applied in practical terms, since those seem to be the most viable. We consider the difference between the distance determined by the heuristics and metaheuristics compared to the exact algorithms as the error rate.

- **Computational Effort:** denotes the actual time the algorithm needed in order to come up with solutions. Mostly used is the measure in wall-clock-time or CPU time. We can further differentiate with or without input/output and pre-processing/post-processing time. Analyzing this key number alone is not practical, since this depends on the hardware components, such as RAM-memory. We measure the time in elapsed wall-clock seconds using CPUs.

- **Ordinal Data Analysis:** In this, we compare all n implemented (meta-)heuristics for a given number of conducted experiments m. We then distribute a set of ordinal values o_k with $k \in [1; m]$. In one experimental setting, let o_k denote the rank of

a (meta-)heuristic to its competitors where $o_k \in [1; n]$. Let there then be a single linear order O, which summarizes all m linear orders o_k, yielding in the final ranking.

12.5 Further Implementation Details

Our code is mainly implemented in Python since this programming language has gained huge popularity in machine learning in the past years (Grus, 2019). For further details, please refer to the code in the Appendix B on page 122 or the GitHub repository available under `https://github.com/JonathanIgler/Urban-Logistics-Routing-Problem`.

We utilize the Google OR Tools (Google, 2019) commonly used in literature (Nazari et al., 2018, e.g.). These tools are used for Greedy Descent, GLS, SA, Tabu Search. PSO is implemented using the pyswarm library. The ACO is implemented from the ACO-Pants package. Further, the BnC algorithm is implemented using the mip package. We set the maximal number of seconds of training to be 3,000. If this limit is reached, BnC does no longer give an exact solution but is also only just a heuristic. For Gnowee, we use the GitHub repository made available by the authors on `https://github.com/SlaybaughLab/Gnowee/`. Since it requires manual hyperparameter tuning, we hypertune `gh.maxGens` over 500, 750 and 1,000, `gh.stallLimit` over 10,000, 15,000 and 20,000 and `gh.optConTol` over 0.001, 0.01 and 0.1. The results are presented for the best parameter constellation. The learnheuristic proposed by Kool et al. (2018) was implemented based on the repository of the authors available on `https://github.com/wouterkool/attention-learn-to-route`. We train and validate the networks based on respectively 10,000 randomly generated instances.

The simulations are solved on a standard computer with an Intel Core i5 processor with 3,1 GHz (3.7 GHz Turbo) and 8 GB LPDDR3 under macOS 10.16.6. We set the random seed of the number generator such that all results of this paper can be replicated.

We note that there are also other solvers freely available such as the Concorde TSP solver (Cook, 2016). This algorithm has been used to solve the TSP for up to 85,900 cities. Since Concorde is written in the ANSI C programming language and not in Python, we opted for Google OR tools.

13 Results

In the following, we will be presenting our results for both study cases. We will thus begin with the simulated data.

13.1 Simulated Data

To begin with, we display an exemplary simulated graph in Figure 25. This is also the graph corresponding to the distance matrix shown in Subsection 12.2.1 on page 84. The depot is denoted as the yellow dot. If we use, for instance, the Tabu Search method

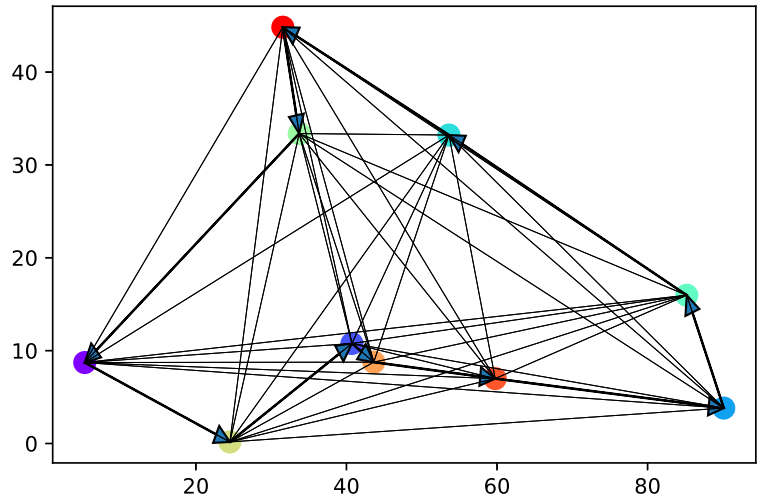

Figure 25: Optimal route computed by the Tabu Search algorithm for an exemplary simulated graph with ten nodes

to solve the corresponding TSP, we obtain the optimal route which is also illustrated in Figure 25. All in all, we get a total distance of 214.13 kilometers to be traveled.

If we run the algorithm again, we obtain another optimal route and total distance. For this reason, we display average results over ten runs in the following such that the influence of outliers in the solution space is mitigated. In order to ensure clear results, in the following plots we only illustrated the most important algorithms as pointed out earlier.

Please refer to the Table 6 on the following page and Table 7 on the next page for an overview on the average computational time and average total distance achieved when solving the TSP on the simulated dataset. For the overall averages we summed up the values in question and divided this sum by the amount of methods, e.g. 8.

While the Greedy approach delivers results above average in no time due to its simplicity, the GLS outperforms it when it comes to the solution itself, e.g. the distance to be

Method/number of nodes	10	30	100
Branch-and-cut	0.33	133.43	3000.00
Greedy Descent	0.01	0.05	0.76
Guided Local Search	30.00	30.00	30.00
Simulated Annealing	238.85	214.20	30.00
Tabu Search	30.00	174.00	30.00
Ant Colony	0.07	0.4427	4.22
Gnowee	3.09	20.10	171.05
Learnheuristic (Kool et al., 2018)	5,738	25,748	-
Overall Average	755.04	3,290.03	466.58

Table 6: Average computational time in seconds for solving the TSP on the simulated data

Method/number of nodes	10	30	100
Branch-and-cut	286.07	467.12	961.66
Greedy Descent	286.07	472.00	816.78
Guided Local Search	286.07	467.12	796.05
Simulated Annealing	286.07	474.33	821.64
Tabu Search	286.07	470.05	810.24
Ant Colony	286.26	539.91	1210.49
Gnowee	286.07	467.12	805.76
Learnheuristic (Kool et al., 2018)	286.07	469.23	820,47
Overall Average	286.10	478.36	880.38

Table 7: Average total distance in kilometers for the TSP on the simulated data

traveled for larger amounts of nodes. Yet, the computational effort was more elaborate than the Greedy approach, which remained equal for all amounts of nodes.

The Tabu Search, with reasonable to good results, has a time peak at 30 nodes with decreasing time for 100 nodes, as seen in Figure 27 on page 93. One possible explanation could be that the algorithm got stuck in a local minimum for the latter case such that it was able to give feasible results above average in less time.

Out of all methods, we see SA was both, the most time inefficient heuristic whilst reporting the second longest tours. As pointed out, this might be due to a not well tuned cooling schedule, which gets noticeable mainly for larger graphs.

Noticeable was the performance for the ACO: Even though it is the overall second fastest algorithm, its tour length was only optimal for the minimal case of ten nodes. From 30 nodes on, the tours given by this method proved to be far beyond the average of 484.68 and 891.04 km respectively. We could explain this by the matter of fact of the artificial ants not to necessarily travel the shortest route existent, but merely include parts of this route in their final solution such that we have segments of an optimal route which altogether show to be of rather worse quality.

Overall, the Gweeno delivered the shortest route for most instances. and was beaten by the GLS only. As seen in Table 7 on the previous page, the route proposed by it was even shorter than the route of the exact method in the case of 100 nodes. This is thus explained by the fact that we programmed the exact method to stop after 3,000 seconds such that we took the best result obtained so far. Thus we can say that BnC has worked similarly to a common heuristic in this case.

However with increasing amount of nodes, the Gweeno continued delivering efficient paths but lacked in computational efficiency. One possible explanation lies in the immense amount of calculations and heuristics to be applied such that computational time increases exponentially.

As for the learnheuristic imposed by Kool et al. (2018), we achieve average results, yet is the computational time considerably high. For the instance of 100 nodes, the algorithm did not achieve any results, since it took too long. The explanation for such is simple. In the learnheuristic setting, the algorithm needs to be trained initially on a training set consisting of different graphs with the same amount of edges. For such big amounts of data one usually opts to train the algorithm on GPU, yet we were able to train it on CPU only. Note that the computational time required does not increase exponentially with the number of nodes. This results from the fact that a major part of the computations is taken by the training of the neural networks.

In the following, we impose a VRP to the simulated data. In Figure 26 on the following page, we depict the relationships on the simulated dataset between the longest distance taken by a vehicle and the number of nodes of the graph for solving the VRP with one to five vehicles. The arrangement is such that it has to be read row-wise, such that one vehicle is added column-wise, e.g. starting with one vehicle in the top left corner and finishing with five vehicles in the bottom left corner.

Trivially, the longest distance of a vehicle decreases with the number of vehicles employed, independently from the method chosen as seen on the shrinking values of y-axis on the respective graphical representations. But with increasing nodes the longest distances of a vehicle respectively rise.

Through all implied methods we notice seemingly no differences, apart from the case where we employed five vehicles. In this, Tabu Search might be distributing the most efficient distances to the remaining four vehicles such that the last vehicle remains with the longest tour, consisting of non-improving solutions only. We have reason to believe so, as in the overall tour lengths the algorithm performed above average.

Further, we note that the distances seem to be equal for up to 3 vehicles among all methods, but diverges from that moment on. Such that for the maximum amount of vehicles,

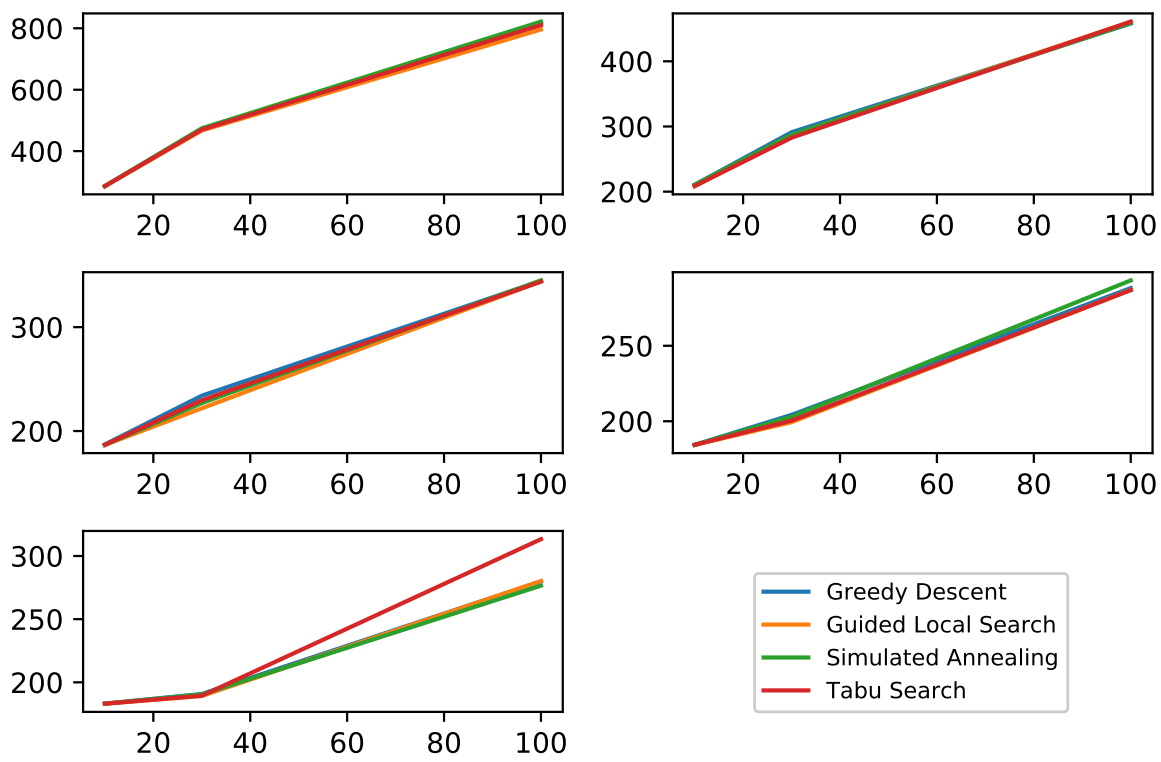

Figure 26: Relationships between longest distance taken by one vehicle on the simulated dataset and number of nodes of the graph for most important methods

We start with a VRP consisting of only one vehicle, e.g. TSP, on the upper left corner. For each row we regard column-wise one additional vehicle such that in the last graph we look at the results of a VRP consisting of five vehicles.

Tabu Search emerges as the worst solution finder. Always within the best solutions, consistently for all categories we find the GLS and SA.

In Figure 27 on the next page, we analogously illustrate the relationships between the computation time of running the heuristics and the number of nodes. One might note that in some cases the computational time needed decreased with an increasing among of notes. This can be explained by the matter of fact that for larger instances the problem aggravates such that the algorithms get stuck in local minima, delivering solutions in less time.

For Table 8 on the following page, we ranked the methodologies separately for each node composition, e.g. 10, 30, 100 nodes and the respective amounts of vahicles, e.g. 1, 2, 3, 4 and 5. In order to validate, we assigned values from $[1; 8]$, the higher the points the better the method in ranking. For the overall rating, we summed up all points and compared the results. As for the separate case, the more points an algorithm had gathered, the higher its ranking.

For the quality measure we only took the distances in kilometers into consideration. For

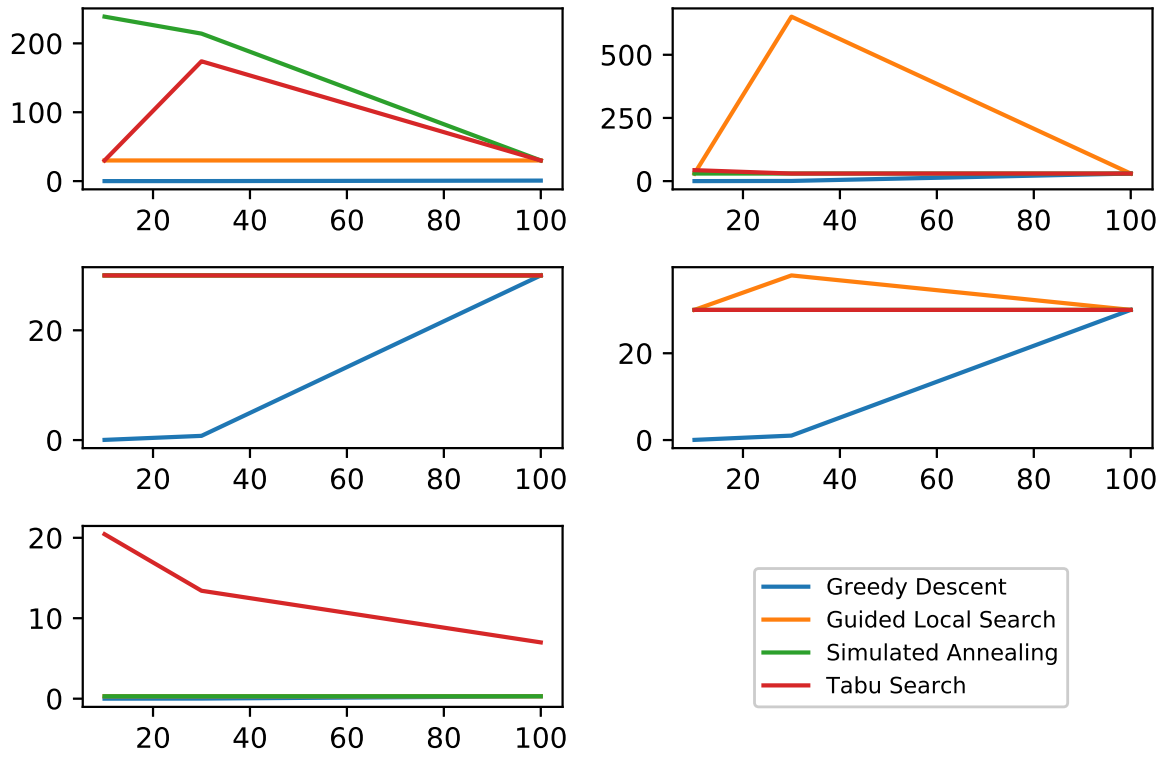

Figure 27: Relationships between computational time in elapsed seconds for training on the simulated dataset and number of nodes of the graph

We start with a VRP consisting of only 1 vehicle, e.g. TSP, on the upper left corner. For each row we add column-wise one vehicle such that in the last graph we look at a VRP consisting of 5 vehicles.

computational effort, the measured wall-clock-time was compared. If it came to a clash between two methods, we opted for the most efficient respective better tour solution.

Between the Branch-and-cut and Tabu Search it came to a clash, when comparing the quality. As both had a score of 18, we looked at their ranking on the computational effort ranking. As Branch-and-cut was more time demanding, it took the worse ranking, e.g. rank 4.

Analysis/Method	BnC	Greedy	GLS	SA	Tabu	ACO	Gweeno	Learn
Quality	3	6	1	7	3	8	2	5
Computational Effort	6	1	3	7	5	2	4	8
Overall Ranking	5	1	2	7	4	6	3	8

Table 8: Ranking on the simulated dataset

13.2 Real World Dataset

For results on the real world dataset, please refer to Figure 28 on the next page for the time needed for the longest tour on the real world dataset conditional on the number of

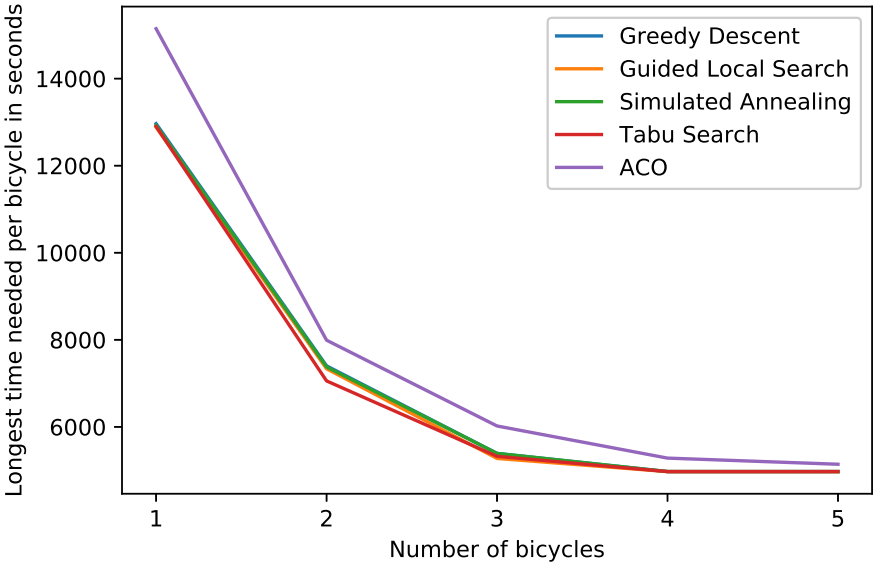

Figure 28: Time needed for the longest tour on the real world dataset

bicycles. In Figure 29 on the following page we depict the computational time of the algorithms also depending on the number of bicycles. The figures only depict results for the standard metaheuristics in order to ensure clarity of the illustrations. An overview of all results on the real world dataset can be found in Table 9 on the next page.

All methods generated solutions of similar quality when it comes to the longest routes employed such that only for one and two vehicles employed slight differences arise such that Tabu Search delivers the best results throughout the process. ACO follows the same trend, yet has computed longer tours. We can conclude that considering the maximum time spent on bicycles has been computed with only slight differences among all (meta-)heuristics.

As we pay our driver based on the minutes driven, we can see that opting for more than four bicycles is implausible, since the marginal product of doing so is minimal or not even visible.

With respect to the efficiency, we are able to say that Greedy Descent outperforms all remaining methods (Figure 29 on the following page). While GLS remains at constant computational time, the remaining methods oscillate in their efficiency. If we imposed more constraints, we might therefore say that both, GLS and SA would take the same computational time into account. For the other algorithms such a statement is not possible, as the remaining methods do not seem to be consistent in the respective computational time neede.

Yet, BnC, Gnowee and the learnheuristic were the most time inefficient algorithms. An

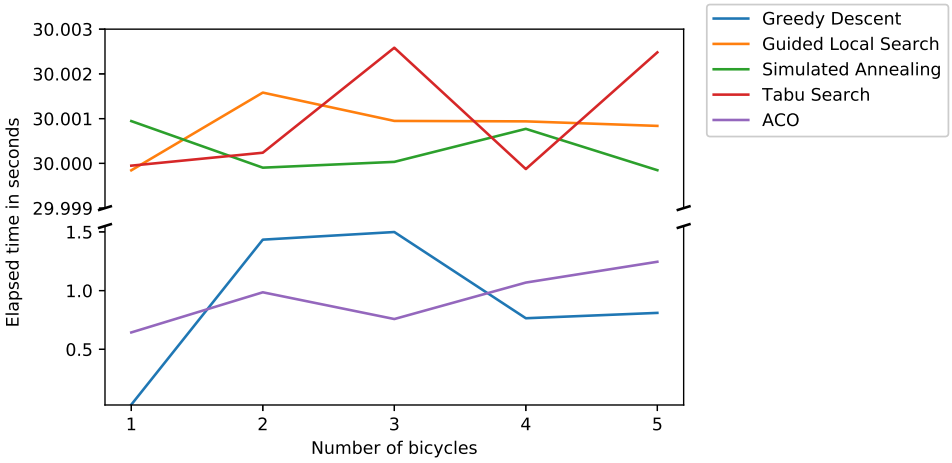

Figure 29: Computational time of algorithms on the real world dataset

Method/Number of vehicles	1	2	3	4	5
Branch-and-cut	11,041	-	-	-	-
Greedy Descent	12,965	7,402	5,391	4,974	4,974
Guided Local Search	12,883	7,335	5,275	4,973	4,973
Simulated Annealing	12,928	7,377	5,391	4,974	4,974
Tabu Search	12,901	7,056	5,324	4,973	4,973
Ant Colony	15,146	7,991	6,021	5,283	5,142
Gnowee	11,040	-	-	-	-
Learnheuristic (Kool et al., 2018)	12,573	-	-	-	-

Table 9: Longest traveling time of a single path in seconds on the real world dataset

explanation seems trivial: For the BnC the limitation rises due to the matter of fact of it being an exact method such that in the case of obtaining results, we can guarantee their optimality yet the computational time that has to be taken is the largest. Gnowee however, makes use of such a vast amount of (meta-)heuristics that is has difficulties for larger instances. Nonetheless, as seen for the simulated data in Section 13.1 on page 89, the results when obtained, belonged to the three best in terms of quality. For the learnheuristic one has to take the training into account, as explained in the previous section.

The BnC algorithm found the optimal solution after 856.83 seconds already. However, it iterates for more than 50 minutes for the remaining vehicle combinations until it terminates because of the imposed time limitations. Further, we opted against the calculation for the subsequent amounts of vehicles, since this would exceed the pre-set time boundaries of 3,000 seconda[2].

[2]For the calculation of 2 vehicles, the algorithm stopped after four hours

We can thus see that the overall traveling time obtained through the exact method is by far the lowest compared to the methods applied in (meta-)heuristics. Such that for one vehicle, its route is 38% lower than the overall traveling time of the route computed by the best performing (meta-)heuristic, namely the GLS. Yet, for the Gnowee we obtained the same result as BnC. However, this method has the same problems as the exact-method: Due to the vast amounts of (meta-)heuristics hybridized in this method, adding further constraints, as increasing the amount of vehicles, results in exponential time growth as already seen for the simulated data as in Section 13.1 on page 89.

The same problems arose for the learnheuristic imposed by Kool et al. (2018). As the computational effort exceeded our limited CPU, we were only able to compute results for the single-vehicle case. This result is further of average quality.

For Table 10, we ranked the methodologies separately for the different amount of vehicles imposed, e.g. 1, 2, 3, 4 or 5 bicycles. In order to validate, we assigned values from $[1; 8]$, the higher the points the better the method in ranking. For the overall rating, we summed up all points and compared the results. As for the separate case, the more points an algorithm had gathered, the higher its ranking.

For the quality measure we only took the distances in kilometer of the longest tour into consideration. For the measure of computational effort, the measured wall-clock-time was compared. If it came to a clash between two methods, we opted for the most efficient respective better tour solution.

Once more we had a clash. In this case between the Greedy Descent and the GLS. As both were strong in terms of their solution's quality, Greedy showed itself to be more time efficient and thus leading the overall ranking.

Analysis/Method	BnC	Greedy	GLS	SA	Tabu	ACO	Gweeno	Learn
Quality	6	4	1	3	2	7	5	8
Computational Effort	7	1	4	3	5	2	6	8
Overall Ranking	7	1	2	3	4	5	6	8

Table 10: Ranking on the real world dataset

As to our technical limitation, the statement of this ranking is also limited. Due to the missing values of BnC, Gweeno and the learnheuristic, we were not able to compare those results in terms of quality, yet we know that all of the three lack in computational efficiency. Even if the results obtained through one of these methods would have been of superb quality, they would nonetheless have turned out to respectively have lower ranks due to time inefficiency.

14 Discussion

From what we saw, (meta-)heuristics are cheap, efficient and scalable methods to solve logistics problems. Even though, mainly metaheuristics which were inspired by natural phenomena, have been targets of criticism among researchers. This is mainly due to the fact that they try to disguise the missing aspect of innovation behind complex metaphors. As for novel swarm optimizers, like the firefly algorithm proposed by (Yang, 2009) or the bat algorithm proposed by Yang (2010) can be considered to merely be copies of the well-established PSO or ACO and differ only in a nearly imperceptible way.

As could already be shown in previous sections, mainly metaheuristics represent highly customizable algorithms and are thus not problem specific. Even though our results hint a direction, one must not generalize the obtained results, as we set several limitations to our setup for the simulation and empirical study, which will be elucidated in the following.

The comparative study suffers from the most diverse and different implementations. While Google OR-Tools have been developed by experienced developers and researches such that these methods will most probably be reviewed and updated frequently, the code proposed by freelancing programmers via GitHub repository often lack on efficient implementations. For this particular reason, the observed performance with main focus on the computational efficiency of these algorithms is less favorable than their theoretical fundamentals would advise to. However, if the lack of efficient implementation of these models is not fixed, practitioners will be forced to spend additional resources in order to improve already existent approaches. Nonetheless, algorithms which have already been implemented will remain favorable.

For this particular reason, it becomes only more reasonable to compare the proposed methods both, in a simulation study and on real world data, as it has been done in this paper.

For the BnC's performance disadvantage we note the vast and depending on the problem itself growing amount of optimizations to be memorized in order to being solved. Mainly on large problem instances, we note high computational effort in both calculation and memorization. This is implied not to the solution finding itself, but more due to the fact that this algorithm proves global optimality. Nonetheless, being the only exact method we imposed, it delivered throughout every possible constellation the best results. Whilst Bentley's Greedy Algorithm works efficiently on the majority of the instances, it does frequently not provide optimal solutions. Nonetheless, all kinds of greedy approaches are simple in both coding and implementation. Before implementation, one must take into account that the right approach has to be followed, often hard to be achieved within the methods of Greedy Heuristics.

Depending on what kind of an initial state we obtain, the Local Search algorithm can easily get stuck in local optima. This can also be caused by the little amount of knowledge we have about the solution space. For this, its extension, the GLS was introduced. With this method we overcome getting stuck in local optima, but one needs to define the features first such that the true power of this algorithm can only be captured after a difficult design.

In the setting of the SA, the crucial factor is laid upon the temperature and thus the cooling schedule. For the design of the latter showing itself to be challenging, the algorithm performs efficiently on rather small instances. On larger problems, it is not comparable to other heuristics, lacking in efficiency and the ability to take notes on the quality of the solution obtained, yet finding an optimal solution is statistically guaranteed.

While Tabu Search has an extra advantage in allowing us to select the maximum number of elements of the tabu list, we face the trade-off between boosting efficiency or gaining on the quality of the results. Furthermore, the more knowledge about the problem the better the obtained results. Since our knowledge, mainly on the simulated data is rather limited, results were poor in quality.

For the advantages of the ACO are vast, including positive feedback within the system (responsible for a swift solution finding) or its efficiency on simpler problem instances (as the TSP) and problems with dynamic applications (due to quick adaptation), we note several drawbacks. ACO suffer mainly from the difficult theoretical analysis, for which the vast amount of research is considered to be more an experimental analysis than a theoretical, resulting from the fact that the artificial ants operate in a more or less 'black-boxerish' way. This being said, the probability distribution changes within the iterations. Surplus, even though the sequences of decisions might be random, yet not independent. All in all, the algorithm guarantees convergence, yet the time to achieve this is uncertain.

For the Gweeno algorithm delivered one of the best results, its drawback lie in the computational effort that has to be invested. For simple instances, we were able to compute results, yet for larger and more complex instances it failed thus resulting in overall low ratings for the real-world dataset.

Learnheuristics have been shown not always to outperform standard metaheuristics. Their strength lies in the their flexibility to learn task-specific heuristics for a wide range of different practical problems for which no good heuristics exist so far. We have seen that learnheuristics are computationally not feasible when trained using CPUs. The high amount of matrix operations requires training using GPUs.

15 Future Challenges and Opportunities in Urban Logistics

With the change in consumer behavior, the rise of new technologies and the persistent globalization, the environment in which urban logistics operate are subject to continuous change. While consumers opt for more and more to e-commerce, their awareness towards environmental issues increase. In the same time, new (especially green) technologies for transportation are developed. Future optimization algorithms should be able to be changed such that they take into account these changes. As an example we could name the rising trend of e-mobility vehicles which need to be charged from time to time, limiting both its range and time of use.

Because of continuous growth in globalization and online commerce, consumer demands have also been increasing. On the one hand, operations research thus has to deal with new challenges such as same day delivery. On the other hand, mobile gadgets facilitate new possibilities such as on-demand transportation currently only available for private purposes with e.g. Uber or Lyft.

Leyerer et al. (2018) differentiate three areas of sustainability where the latter shall be elaborated on more closely in the following:

- Economic sustainability,
- Social sustainability and
- Environmental sustainability.

Recent arising social and political awareness for environmental sustainability and new developments in the area of green technologies necessitate that operations research does not only address the direct costs for the company, but also takes into account noise emissions, reputation damage, carbon tax or other costs resulting from environmental pollution in form of incineration, packaging or exhaust gases. For example, the European Commission has defined the goal of forbidding conventionally powered vehicles by 2050 and achieving "CO^2-free city logistics in major urban centers by 2030" (European Commission. Directorate-General for Mobility and Transport, 2011). These goals will probably require to prohibit certain types of vehicles. So even though the need for general, versatile algorithms has arisen in the past decades on the one hand, we argue that specific objective functions addressing environmental issues should be developed on the other hand. Accordingly, we maintain that further advances in urban logistics should be further researched in the framework of green information systems.

The Frauenhofer Institut is, for example, already researching in this area (Inninger, 2019). They differentiate three options of electrification of urban logistics:

- Replacing Diesel motors with electric motors.
- Establishing an e-concept with cargobikes for instance.
- Creating a new logistics concept such as pull concepts, green logistics or night logistics.

The options increase in risk and effort in the order listed.

New logistic concepts could extend the carsharing approach to the business sector. Carsharing is the offer of use of vehicles for trip-dependent fees to people who have the necessary permits (Kuehne et al., 2017). This concept could, for example, reduce the costs of empty trips. A suitable logistic concept has still to be developed to this case and the optimization algorithms have to be adjusted to this much more complicated problem setting.

What is more, arising industry 4.0 and the Internet of things provide future logistics with new opportunities. Lee et al. (2016) incorporate vehicle location, vehicle condition information and dynamic demand forecasting in their optimization algorithm of the vehicle routing problem. For instance, mobile applications allow a company to track the GPS positions of their 'salesmen' in real time. Further, with live information on the traffic situation, it is reasonable to require from future optimization algorithms to be dynamic such that solutions can be online updated. Online optimization allows to take into account traffic as well as update the customer base to be served such that new locations can be pursued.

16 Conclusion

This thesis introduced the reader to the area of urban logistics. In the scope of this work, we dealt with optimization solution approaches to solve corresponding logistical concepts. We presented a general framework to classify a wide range of optimization algorithms for urban logistics problems. We differentiated between exact methods, heuristics, metaheuristics and thus hybridizations of all the aforementioned instances, while focusing on the (meta-) heuristic for their variability.

We especially stressed the importance of state-of-the-art machine learning methods. Learnheuristics where supervised machine learning methods are employed to (meta-)heuristics for a specified problem, have shown to generalize well. To incorporate crucial methods of Artificial Intelligence remains to be a fast developing, promising and steep progress in the future.

In both a large-scale simulation study and an application to real-world-conditions, we compared the performance of a chosen subset of relevant methods. From this we could quantify both, advantages and disadvantages of the employed methods. All in all, we provide decision support for parcel service providers, city authorities, and other relevant businesses.

16.1 Results

As a result of our analysis we found out that when it comes to the quality of solutions, nothing was able to compete with the exact methods, as theoretically explained. Among the simple (meta-)heuristics, we could not identify a single best solution. Yet, being only introduced recently to the field of combinatorial optimization, the Gnowee algorithm performed best and came to close to the results obtained by exact methods. We recommend to further investigate the field of hybrids then.

Further, it could be shown that learnheuristics do not improve the quality of solutions provided by both, traditional (meta-)heuristics and hybrids. However, they can be applied in a more general manner, being even less problem-specific than metaheuristics, but leave a lot of scope for further research.

Since the exact optimization algorithms are not feasible (for the most simple instance we had a total calculation time of over 30 minutes), we aim to find a balance between the accuracy, the scalability and the computational time solver requires. In most cases, there exists a trade off between the optimality of the solution and the time it takes to find it. As Einstein once said,

> "Everything should be made as simple as possible, but not simpler."

Following this quotation, we recommend the use of metaheuristics instead of heuristics, due to their wider range of application.

Further, the implementation of nature-inspired techniques in the field of computational optimization shall be given less attention. Even though most of the explained and used metaheuristics belong to this particular group, the amount of innovation in this setting is minimal and so is the gained performance.

16.2 Outlook

There are several directions for future work that arise from the results of this paper.

(Meta-)Heuristic methods have been proposed in a time where computational power was rare. In modern times where we have vast access to fast computing machines with sufficient working memory, we can also make use of either more accurate methods. In particular, we highlight the importance of cloud computing as evolving paradigm that describes the 'ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources' (Mell et al., 2011). Even though its use still incurs costs, computational more exhausting methods can be used. If the goal is to reduce these costs, one might opt to use hybrid methods as the Gweeno. For its configuration is tedious, once established it delivers results comparable or even equal to the results delivered by exact methods in a fraction of time.

But not only the medium to solve routing problems has changed, but also the complexity of the problems themselves. In recent time, there has been a surge of papers that analyze dynamic logistics problems (Arnau et al., 2018). It is reasonable to assume that the cost a path incurs depend on a state and cannot be prognosticated as a fixed number beforehand. For example, we can think of the traveling time in a metropolis during rush hour. The same street could take less than a tenth of the time in the night compared to the time it would take to traverse it at 5 pm.

In an ever-changing economic environment, the requirements towards optimization methods of logistics problems are fast evolving. Accordingly, new application areas of routing problems arise. For example, demand-side management in station-based car sharing concepts make adjustments of generally applicable metaheuristics necessary (Broihan et al., 2018). Especially, in urban areas green zones in city centers allow environmentally friendly cars only to traverse them. Consequently, optimization algorithms should differentiate between different types of salesmen. Another simulation study based on our results could address especially the needs of newly evolved routing problems and propose task-specific fitting.

All in all, urban logistics are expected to become even more important with the rise of

Industry 4.0. Optimization algorithms dealing with them should thus be advanced.

References

Aarts, E. and Korst, J. (1988). Simulated annealing and Boltzmann machines.

Applegate, D. L., Bixby, R. E., Chvatal, V., and Cook, W. J. (2006). *The traveling salesman problem: a computational study*. Princeton university press.

Arnau, Q., Juan, A., and Serra, I. (2018). On the use of learnheuristics in vehicle routing optimization problems with dynamic inputs. *Algorithms*, 11(12):208.

Arora, T. and Gigras, Y. (2013). A survey of comparison between various metaheuristic techniques for path planning problem. *International Journal of Computer Engineering & Science*, 3(2):62–66.

Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., and Protasi, M. (2012). *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media.

Aydin, M. E. (2012). Coordinating metaheuristic agents with swarm intelligence. *Journal of Intelligent Manufacturing*, 23(4):991–999.

Baldacci, R., Hadjiconstantinou, E., and Mingozzi, A. (2004). An exact algorithm for the capacitated vehicle routing problem based on a two-commodity network flow formulation. *Operations research*, 52(5):723–738.

Bann, E. (2019). Ryanair joins the club of europe's top 10 carbon polluters. `https://www.transportenvironment.org/press/ryanair-joins-club-europe's-top-10-carbon-polluters`. (Accessed on August, 23^{rd} 2019).

Bányai, T., Tamás, P., Illés, B., Stankevičiūtė, Ž., and Bányai, Á. (2019). Optimization of municipal waste collection routing: Impact of industry 4.0 technologies on environmental awareness and sustainability. *International journal of environmental research and public health*, 16(4):634.

Battiti, R. and Brunato, M. (2010). Reactive search optimization: learning while optimizing. In *Handbook of Metaheuristics*, pages 543–571. Springer.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

Bentley, J. L. (1980). A parallel algorithm for constructing minimum spanning trees. *Journal of algorithms*, 1(1):51–59.

Bevins, J. E. and Slaybaugh, R. (2019). Gnowee: a hybrid metaheuristic optimization algorithm for constrained, black box, combinatorial mixed-integer design. *Nuclear Technology*, 205(4):542–562.

Bingham, C. B. and Eisenhardt, K. M. (2011). Rational heuristics: the 'simple rules' that strategists learn from process experience. *Strategic management journal*, 32(13):1437–1464.

Birattari, M. (2004). The problem of tuning metaheuristics as seen from a machine learning perspective.

Blum, C. (2005). Ant colony optimization: Introduction and recent trends. *Physics of Life reviews*, 2(4):353–373.

Blum, C., Puchinger, J., Raidl, G. R., and Roli, A. (2011). Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151.

Blum, C. and Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308.

Bozkaya, B., Erkut, E., and Laporte, G. (2003). A tabu search heuristic and adaptive memory procedure for political districting. *European Journal of Operational Research*, 144(1):12–26.

Broihan, J., Möller, M., Kühne, K., Sonneberg, M., and Breitner, M. H. (2018). Revenue Management meets Carsharing: Optimizing the Daily Business. In *Operations Research Proceedings 2016*, pages 421–427. Springer.

Calvet, L., de Armas, J., Masip, D., and Juan, A. A. (2017). Learnheuristics: hybridizing metaheuristics with machine learning for optimization with dynamic inputs. *Open Mathematics*, 15(1):261–280.

Candelieri, A. (2011). *A hyper-solution framework for classification problems via metaheuristic approaches*. PhD thesis, Springer.

Carlson, J. A., Jaffe, A., and Wiles, A. (2006). *The millennium prize problems*. American Mathematical Society Providence, RI.

Černỳ, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51.

Chandra, B., Karloff, H. J., and Tovey, C. A. (1994). New Results on the Old k-opt Algorithm for the TSP. In *SODA*, pages 150–159.

Chardaire, P. and Sutter, A. (1995). A decomposition method for quadratic zero-one programming. *Management Science*, 41(4):704–712.

Chen, H., Perozzi, B., Hu, Y., and Skiena, S. (2018). Harp: Hierarchical representation learning for networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Christofides, N. (1976). The vehicle routing problem. *Revue française d'automatique, informatique, recherche opérationnelle. Recherche opérationnelle*, 10(V1):55–70.

Christofides, N. and Eilon, S. (1969). An algorithm for the vehicle-dispatching problem. *Journal of the Operational Research Society*, 20(3):309–318.

Ciornei, I. and Kyriakides, E. (2011). Hybrid ant colony-genetic algorithm (GAAPI) for global continuous optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(1):234–245.

Clarissa (2019). Dieselfahrzeuge: Warum ein Verbot in Deutschland beschlossen wurde. `https://www.bussgeldkatalog.org/dieselfahrzeuge-verbot/`. (Accessed on August, 30^{th} 2019).

Clarke, G. and Wright, J. W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581.

Clerc, M. (2004). Discrete particle swarm optimization, illustrated by the traveling salesman problem. In *New optimization techniques in engineering*, pages 219–239. Springer.

Colorni, A., Dorigo, M., Maniezzo, V., et al. (1992). Distributed optimization by ant colonies. In *Proceedings of the first European conference on artificial life*, volume 142, pages 134–142. Cambridge, MA.

Congram, R. K., Potts, C. N., and van de Velde, S. L. (2002). An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 14(1):52–67.

Consoli, S. and Darby-Dowman, K. (2006). Combinatorial optimization and metaheuristics. Technical report, Brunel University.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM.

Cook, W. (2016). Concorde TSP solver. `http://www.math.uwaterloo.ca/tsp/concorde.html`.

Cooper, L. (1963). Location-allocation problems. *Operations research*, 11(3):331–343.

Cordeau, J.-F., Gendreau, M., Laporte, G., Potvin, J.-Y., and Semet, F. (2002). A guide to vehicle routing heuristics. *Journal of the Operational Research society*, 53(5):512–522.

Cowling, P., Kendall, G., and Soubeiga, E. (2000). A hyperheuristic approach to scheduling a sales summit. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 176–190. Springer.

Crainic, T. G. and Toulouse, M. (2002). Introduction to the special issue on parallel meta-heuristics. *Journal of Heuristics*, 8(3):247–249.

Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812.

Dantzig, G., Fulkerson, R., and Johnson, S. (1954). Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410.

Dantzig, G. B. and Ramser, J. H. (1959). The truck dispatching problem. *Management science*, 6(1):80–91.

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection Or the Preservation of Favoured Races in the Struggle for Life.* H. Milford; Oxford University Press.

Datta, S., Roy, S., and Davim, J. P. (2019). Optimization Techniques: An Overview. In *Optimization in Industry*, pages 1–11. Springer.

Dethloff, J. (2001). Vehicle routing and reverse logistics: the vehicle routing problem with simultaneous delivery and pick-up. *OR-Spektrum*, 23(1):79–96.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

Domschke, W., Scholl, A., et al. (2006). *Heuristische Verfahren.* Univ., Wirtschaftswiss. Fak.

Dorigo, M. and Di Caro, G. (1999). Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477. IEEE.

Dorigo, M. and Gambardella, L. M. (1997). Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66.

Dréo, J., Pétrowski, A., Siarry, P., and Taillard, E. (2006). *Metaheuristics for hard optimization: methods and case studies*. Springer Science & Business Media.

Duhr, S. and Braun, D. (2006). Thermophoretic depletion follows boltzmann distribution. *Physical review letters*, 96(16):168301.

Eberhart, R. and Kennedy, J. (1995). A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43. Ieee.

Edelkamp, S. and Schroedl, S. (2011). *Heuristic search: theory and applications*. Elsevier.

Eilon, S., Watson-Gandy, C. D. T., Christofides, N., and de Neufville, R. (1974). Distribution management-mathematical modelling and practical analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, (6):589–589.

Epitropakis, M. G. and Burke, E. K. (2018). Hyper-heuristics. *Handbook of Heuristics*, pages 489–545.

Euler, L. (1741). Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, pages 128–140.

Euler, L. (1953). Leonhard Euler and the Königsberg bridges. *Scientific American*, 189(1):66–72.

European Commission (2013). A call to action on urban logistics.

European Commission. Directorate-General for Mobility and Transport (2011). *White Paper on Transport: Roadmap to a Single European Transport Area: Towards a Competitive and Resource-efficient Transport System*. Publications Office of the European Union.

Fisher, M. L. and Rinnooy Kan, A. H. (1988). The design, analysis and implementation of heuristics. *Management Science*, 34(3):263–265.

Floudas, C. A. and Pardalos, P. M. (2016). Encyclopedia of optimization.

Frauchiger, D. (2017). Anwendungen von Design Science Research in der Praxis. In *Wirtschaftsinformatik in Theorie und Praxis*, pages 107–118. Springer.

Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615.

Fu, L., Sun, D., and Rilett, L. R. (2006). Heuristic shortest path algorithms for transportation applications: state of the art. *Computers & Operations Research*, 33(11):3324–3343.

Gambardella, L. M. and Dorigo, M. (1995). Ant-Q: A reinforcement learning approach to the traveling salesman problem. In *Machine Learning Proceedings 1995*, pages 252–260. Elsevier.

Garrido, P. and Castro, C. (2009). Stable solving of cvrps using hyperheuristics. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 255–262. ACM.

Garrido, P. and Riff, M. C. (2010). DVRP: a hard dynamic combinatorial optimisation problem tackled by an evolutionary hyper-heuristic. *Journal of Heuristics*, 16(6):795–834.

Gebski, S. A., Czerwinski, P., Leyerer, M., Sonneberg, M.-O., and Breitner, M. H. (2018). Ein Entscheidungsunterstützungssystem zur Tourenplanung am Beispiel eines innovativen Lebensmittel-Lieferkonzeptes.

Gendreau, M., Hertz, A., and Laporte, G. (1994). A tabu search heuristic for the vehicle routing problem. *Management science*, 40(10):1276–1290.

Gendreau, M., Laporte, G., and Semet, F. (2001). A dynamic model and parallel tabu search heuristic for real-time ambulance relocation. *Parallel computing*, 27(12):1641–1653.

Gendreau, M. and Potvin, J.-Y. (2005). Metaheuristics in combinatorial optimization. *Annals of Operations Research*, 140(1):189–213.

Gevaers, R., Van de Voorde, E., Vanelslander, T., et al. (2009). Characteristics of innovations in last mile logistics-using best practices, case studies and making the link with green and sustainable logistics. *Association for European Transport and contributors*.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549.

Golden, B. L., Raghavan, S., and Wasil, E. A. (2008). *The vehicle routing problem: latest advances and new challenges*, volume 43. Springer Science & Business Media.

Google (2019). OR-Tools. `https://opensource.google.com/projects/or-tools`.

Goss, S., Aron, S., Deneubourg, J.-L., and Pasteels, J. M. (1989). Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76(12):579–581.

Gregor, S. and Hevner, A. R. (2013). Positioning and presenting design science research for maximum impact. *MIS quarterly*, pages 337–355.

Grimm, N. B., Foster, D., Groffman, P., Grove, J. M., Hopkinson, C. S., Nadelhoffer, K. J., Pataki, D. E., and Peters, D. P. (2008). The changing landscape: ecosystem responses to urbanization and pollution across climatic and societal gradients. *Frontiers in Ecology and the Environment*, 6(5):264–272.

Grus, J. (2019). *Data science from scratch: first principles with python*. O'Reilly Media.

Hansen, P. and Mladenović, N. (2003). Variable neighborhood search. In *Handbook of metaheuristics*, pages 145–184. Springer.

Hastie, T., Tibshirani, R., Friedman, J., and Franklin, J. (2005). The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85.

Haykin, S. (1994). *Neural networks: a comprehensive foundation*. Prentice Hall PTR.

Hazewinkel, M. (2001). Fermat-Torricelli problem, Encyclopedia of Mathematics.

Hazlitt, H. (2010). *Economics in one lesson: The shortest and surest way to understand basic economics*. Currency.

Henderson, D., Jacobson, S. H., and Johnson, A. W. (2003). The theory and practice of simulated annealing. In *Handbook of metaheuristics*, pages 287–319. Springer.

Hendrickson, B. and Leland, R. W. (1995). A Multi-Level Algorithm For Partitioning Graphs. *SC*, 95(28):1–14.

Hevner, A. and Chatterjee, S. (2010). Design science research in information systems. In *Design research in information systems*, pages 9–22. Springer.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hogg, T. and Williams, C. P. (1993). Solving the really hard problems with cooperative search. In *Proceedings of the national conference on artificial intelligence*, pages 231–231. JOHN WILEY & SONS LTD.

Holland, J. H. et al. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*.

Ingber, L. (2000). Adaptive simulated annealing (ASA): Lessons learned. *arXiv preprint cs/0001018*.

Inninger, W. (2019). Urbane Logistik in der Zukunft. `https://www.it-logistik-bayern.de/uploads/media/Vortrag_Wolfgang_Inninger_Fraunhofer_IML.pdf`.

Jeong, S.-J., Kim, K.-S., and Lee, Y.-H. (2009). The efficient search method of simulated annealing using fuzzy logic controller. *Expert Systems with Applications*, 36(3):7099–7103.

Johnson, D. S. and Garey, M. R. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. WH Freeman.

Johnson, D. S. and McGeoch, L. A. (1997). The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1(1):215–310.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285.

Kalnay, E. and Cai, M. (2003). Impact of urbanization and land-use change on climate. *Nature*, 423(6939):528.

Karaboga, D. (2005). An idea based on honey bee swarm for numerical optimization. Technical report, Technical report-tr06, Erciyes university, engineering faculty, computer ….

Karaboga, D. and Basturk, B. (2007). A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. *Journal of global optimization*, 39(3):459–471.

Kennedy, K. (2001). Fast greedy weighted fusion. *International Journal of Parallel Programming*, 29(5):463–491.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, 220(4598):671–680.

Kong, X., Liu, S., Wang, Z., and Yong, L. (2012). Hybrid artificial bee colony algorithm for global numerical optimization. *Journal of Computational Information Systems*, 8(6):2367–2374.

Kool, W., van Hoof, H., and Welling, M. (2018). Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*.

Kotsiantis, S. B., Zaharakis, I., and Pintelas, P. (2007). Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24.

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

Kuehne, K., Sonneberg, M.-O., and Breitner, M. (2017). Ecological & profitable carsharing business: Emission limits & heterogeneous fleets.

Laporte, G. (1987). Location-routing problems.

Laporte, G. (1992). The vehicle routing problem: An overview of exact and approximate algorithms. *European journal of operational research*, 59(3):345–358.

Laporte, G. and Semet, F. (2002). Classical heuristics for the capacitated vrp. In *The vehicle routing problem*, pages 109–128. SIAM.

Lawler, E. L. and Wood, D. E. (1966). Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719.

Lee, K., Kim, D.-h., Choi, H. R., Park, B. K., Cho, M. J., Kang, D. Y., Park, S. H., and Sung, S. H. (2016). A study on vehicle routing problems considering iot based real time information. *Advanced Science and Technology Letters*, 140.

Lenstra, J. K. and Kan, A. R. (1981). Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227.

Leyerer, M., Sonneberg, M.-O., and Breitner, M. H. (2018). Decision Support for Urban E-Grocery Operations.

Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269.

Lin, S. and Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516.

Lin, T. Y. et al. (1998). Granular computing on binary relations i: Data mining and neighborhood systems. *Rough sets in knowledge discovery*, 1:107–121.

Little, J. D., Murty, K. G., Sweeney, D. W., and Karel, C. (1963). An algorithm for the traveling salesman problem. *Operations research*, 11(6):972–989.

Lourenço, H. R., Martin, O. C., and Stützle, T. (2003). Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer.

Lozano, M. and García-Martínez, C. (2010). Hybrid metaheuristics with evolutionary algorithms specializing in intensification and diversification: Overview and progress report. *Computers & Operations Research*, 37(3):481–497.

Lundy, M. and Mees, A. (1986). Convergence of an annealing algorithm. *Mathematical programming*, 34(1):111–124.

Maier, H. R., Razavi, S., Kapelan, Z., Matott, L. S., Kasprzyk, J., and Tolson, B. A. (2018). Introductory overview: Optimization using evolutionary algorithms and other metaheuristics. *Environmental modelling & software*.

Maniezzo, V., Stützle, T., and Voß, S. (2009). Hybridizing metaheuristics and mathematical programming. series: Annals of information systems.

Meixell, M. J. and Luoma, P. (2015). Stakeholder pressure in sustainable supply chain management: a systematic review. *International Journal of Physical Distribution & Logistics Management*, 45(1/2):69–89.

Mell, P., Grance, T., et al. (2011). The NIST definition of cloud computing.

Mills, P. and Tsang, E. (2000). Guided local search for solving sat and weighted MAX-SAT problems. *Journal of Automated Reasoning*, 24(1-2):205–223.

Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100.

Montemanni, R., Gambardella, L. M., Rizzoli, A. E., and Donati, A. V. (2005). Ant colony system for a dynamic vehicle routing problem. *Journal of Combinatorial Optimization*, 10(4):327–343.

Mulligan, G. F. (2006). Logistic population growth in the world's largest cities. *Geographical analysis*, 38(4):344–370.

Nazari, M., Oroojlooy, A., Snyder, L., and Takác, M. (2018). Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9839–9849.

Neumann, F. and Witt, C. (2010). Combinatorial optimization and computational complexity. In *Bioinspired Computation in Combinatorial Optimization*, pages 9–19. Springer.

Osman, I. H. and Kelly, J. P. (1996). Meta-heuristics: an overview. In *Meta-heuristics*, pages 1–21. Springer.

Padberg, M. and Rinaldi, G. (1991). A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100.

Paket und Expresslogistik e. V., B. (2019). Kep-studie 2019. `https://www.biek.de/publikationen/studien.html`. (Accessed on August, 22^{nd} 2019).

Papadimitriou, C. H. (2003). *Computational complexity*. John Wiley and Sons Ltd.

Parsopoulos, K. E. and Vrahatis, M. N. (2002). Particle swarm optimization method in multiobjective problems. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 603–607. ACM.

Pearl, J. (1984). Heuristics: intelligent search strategies for computer problem solving.

Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77.

Pegg Jr, E. (2009). The icosian game, revisited. *The Mathematica Journal*, 11(3):310–314.

Pichpibul, T. and Kawtummachai, R. (2012). An improved clarke and wright savings algorithm for the capacitated vehicle routing problem. *ScienceAsia*, 38(3):307–318.

Plowman, E. G. (1962). *Elements of business logistics*. Graduate School of Business, Stanford University.

Puchinger, J. and Raidl, G. R. (2005). Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In *International Work-Conference on the Interplay Between Natural and Artificial Computation*, pages 41–53. Springer.

Punakivi, M., Yrjölä, H., and Holmström, J. (2001). Solving the last mile issue: reception box or delivery box? *International Journal of Physical Distribution & Logistics Management*, 31(6):427–439.

Raidl, G. R. (2006). A unified view on hybrid metaheuristics. In *International Workshop on Hybrid Metaheuristics*, pages 1–12. Springer.

Reiche, M. (2019). Mit dem "Polensprinter" an der Maut vorbei. `https://www.tagesschau.de/inland/speditionen-umgehen-maut-101.html`.

Roberto, B., Maria, B., and Daniele, V. (2007). Routing a heterogeneous fleet of vehicles. *DEIS, University Bologna via Venezia*, 52:47023.

Rochat, Y. and Taillard, É. D. (1995). Probabilistic diversification and intensification in local search for vehicle routing. *Journal of heuristics*, 1(1):147–167.

Rosenkrantz, D. J., Stearns, R. E., and Lewis, II, P. M. (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581.

Rothlauf, F. (2011). *Design of modern heuristics: principles and application*. Springer Science & Business Media.

Russell, R. and Igo, W. (1979). An assignment routing problem. *Networks*, 9(1):1–17.

Sachan, A. and Datta, S. (2005). Review of supply chain management and logistics research. *International Journal of Physical Distribution & Logistics Management*, 35(9):664–705.

San Nah Sze, W. K. T. (2007). A comparison between heuristic and meta-heuristic methods for solving the multiple traveling salesman problem.

Settles, M. and Soule, T. (2005). Breeding swarms: a GA/PSO hybrid. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 161–168. ACM.

Sevaux, M., Sörensen, K., and Pillay, N. (2018). Adaptive and multilevel metaheuristics. *Handbook of Heuristics*, pages 1–19.

Shelokar, P., Siarry, P., Jayaraman, V. K., and Kulkarni, B. D. (2007). Particle swarm and ant colony algorithms hybridized for improved continuous optimization. *Applied mathematics and computation*, 188(1):129–142.

Shmoys, D. B. and Aardal, K. (1997). *Approximation algorithms for facility location problems*, volume 1997. Utrecht University: Information and Computing Sciences.

Solomon, M. M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265.

Song, L., Cherrett, T., McLeod, F., and Guan, W. (2009). Addressing the last mile problem: transport impacts of collection and delivery points. *Transportation Research Record*, 2097(1):9–18.

Sörensen, K., Sevaux, M., and Glover, F. (2018). A history of metaheuristics. *Handbook of heuristics*, pages 1–18.

Stein, G., Chen, B., Wu, A. S., and Hua, K. A. (2005). Decision tree classifier for network intrusion detection with ga-based feature selection. In *Proceedings of the 43rd annual Southeast regional conference-Volume 2*, pages 136–141. ACM.

Streim, H. (1975). Heuristische Lösungsverfahren Versuch einer Begriffsklärung. *Zeitschrift für Operations Research*, 19(5):143–162.

Stützle, T. and Hoos, H. H. (2000). MAX–MIN ant system. *Future generation computer systems*, 16(8):889–914.

Talbi, E.-G. (2009). *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons.

Taniguchi, E. and Van Der Heijden, R. E. (2000). An evaluation methodology for city logistics. *Transport Reviews*, 20(1):65–90.

Thorngate, W. (1980). Efficient decision heuristics. *Behavioral Science*, 25(3):219–225.

Tien, J.-P. and Li, T. (2013). Hybrid taguchi-chaos of artificial bee colony algorithm for global numerical optimization. *Int. J. Innovative Comput. Inf. Control*, 9(6):2665–2688.

Ting, T., Wong, K., and Chung, C. (2005). Investigation of hybrid genetic algorithm/-particle swarm optimization approach for the power flow problem. In *Proceedings of 2005 International Conference on Machine Learning and Cybernetics*, volume 1, pages 436–440.

Ting, T., Wong, K., and Chung, C. (2006). A hybrid genetic algorithm/particle swarm approach for evaluation of power flow in electric network. In *Advances in Machine Learning and Cybernetics*, pages 908–917. Springer.

Toth, P. and Vigo, D. (2002). *The vehicle routing problem*. SIAM.

Toth, P. and Vigo, D. (2003). The granular tabu search and its application to the vehicle-routing problem. *Informs Journal on computing*, 15(4):333–346.

Turner, A. J. and Miller, J. F. (2014). Neuroevolution: evolving heterogeneous artificial neural networks. *Evolutionary Intelligence*, 7(3):135–154.

UNFCCC (2018). The Paris Agreement. `https://unfccc.int/process-and-meetings/the-paris-agreement/the-paris-agreement`. (Accessed on August, 30^{th} 2019).

Van Aken, J. E. (2005). Management research as a design science: Articulating the research products of mode 2 knowledge production in management. *British journal of management*, 16(1):19–36.

Van Audenhove, F.-J., De Jongh, S., and Durance, M. (2015). Urban logistics. `https://www.adlittle.com/en/UrbanLogistics`. (Accessed on August, 22^{nd} 2019).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700.

Voudouris, C. and Tsang, E. (1999). Guided local search and its application to the traveling salesman problem. *European journal of operational research*, 113(2):469–499.

Walshaw, C. (2000). A multilevel algorithm for force-directed graph drawing. In *International Symposium on Graph Drawing*, pages 171–182. Springer.

Walshaw, C. (2002). A multilevel approach to the travelling salesman problem. *Operations research*, 50(5):862–877.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

Webster, J. and Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a literature review. *MIS quarterly*, pages xiii–xxiii.

West, D. B. et al. (1996). *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ.

Wolpert, D. H., Macready, W. G., et al. (1997). No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82.

Worldbank (2019). Database. `https://data.worldbank.org/indicator`. (Accessed on August, 10^{th} 2019).

Xu, J. and Kelly, J. P. (1996). A network flow-based tabu search heuristic for the vehicle routing problem. *Transportation Science*, 30(4):379–393.

Yang, X.-S. (2009). Firefly algorithms for multimodal optimization. In *International symposium on stochastic algorithms*, pages 169–178. Springer.

Yang, X.-S. (2010). A new metaheuristic bat-inspired algorithm. In *Nature inspired cooperative strategies for optimization (NICSO 2010)*, pages 65–74. Springer.

A Appendix

Symbol	Definition
$\mathbb{I}$	Indicator function, equal to 1 for a predefined condition and 0 otherwise
c_x	Transportation Costs
C_x	Capacity of x
d_x	Demand of x
$f(x)$	Objective function
I_x	Facility information (o - to be opened; c - to be closed)
i and j	Nodes i and j
k	Amount of agents/particles/salesmen
L_k	Length of tour k
n	Input size
o_x	Costs for facility building
Ω	Set of constraints
S	Solution
$s \in S$	Single solution from the Solution space
s_{max}	Global optimum
s^*	Incubent solution
$s[]$	Empty solution
$tr_{x,y}$	Distribution costs
v_x	Edges

Table 11: Notations

Methodology	Literature
Research Design	Webster and Watson (2002), Van Aken (2005), Peffers et al. (2007) Hevner and Chatterjee (2010), , Gregor and Hevner (2013), Frauchiger (2017)
Combinatorial Optimization Problem	Cook (1971), Dethloff (2001), Papadimitriou (2003), (Gendreau and Potvin, 2005), Carlson et al. (2006), Consoli and Darby-Dowman (2006), Neumann and Witt (2010), Ausiello et al. (2012), (Floudas and Pardalos, 2016)
Introduction to Logistic Problem	European Commission (2013), Mulligan (2006), Gebski et al. (2018), Van Audenhove et al. (2015), Paket und Expresslogistik e. V. (2019), Bányai et al. (2019), Gebski et al. (2018), Kalnay and Cai (2003), Grimm et al. (2008), Bann (2019), Meixell and Luoma (2015), Worldbank (2019), Taniguchi and Van Der Heijden (2000), Clarissa (2019), Sachan and Datta (2005), Punakivi et al. (2001), Song et al. (2009), Hazewinkel (2001), Euler (1741), Pegg Jr (2009), Euler (1953), Pegg Jr (2009), Kruskal (1956), Johnson and Garey (1979), Dantzig and Ramser (1959), Lenstra and Kan (1981), Roberto et al. (2007), Solomon (1987), Russell and Igo (1979), Shmoys and Aardal (1997), Neumann and Witt (2010), Dijkstra (1959), Fredman and Tarjan (1987), Floudas and Pardalos (2016), Cooper (1963). Laporte (1987), Fu et al. (2006), Dethloff (2001), Christofides (1976), Johnson and McGeoch (1997)
Exact Methods	Floudas and Pardalos (2016), Christofides and Eilon (1969), Padberg and Rinaldi (1991), Rothlauf (2011), Baldacci et al. (2004), Dantzig et al. (1954), Christofides and Eilon (1969), Little et al. (1963), Kruskal (1956), Applegate et al. (2006), Toth and Vigo (2002), Domschke et al. (2006), Streim (1975), Thorngate (1980), Pearl (1984), Lawler and Wood (1966), Puchinger and Raidl (2005)
Heuristics	Bingham and Eisenhardt (2011), Pearl (1984), Domschke et al. (2006), Pearl (1984), Kennedy (2001), Fisher and Rinnooy Kan (1988), Toth and Vigo (2002), Cordeau et al. (2002), Gendreau et al. (2001), Clarke and Wright (1964), Bentley (1980), Rosenkrantz et al. (1977), Walshaw (2002), Hendrickson and Leland (1995), Walshaw (2000), Chen et al. (2018), Clarke and Wright (1964), Lin and Kernighan (1973), Lin (1965), Chandra et al. (1994), Johnson and McGeoch (1997), Streim (1975), Kennedy (2001), Lin et al. (1998), Laporte and Semet (2002), Croes (1958), Sörensen et al. (2018), Pichpibul and Kawtummachai (2012)
Classification Scheme	Mladenović and Hansen (1997), Gendreau and Potvin (2005), Arora and Gigras (2013)
Metaheuristics	Talbi (2009), Gendreau and Potvin (2005), Puchinger and Raidl (2005), Wolpert et al. (1997), Osman and Kelly (1996), Lin et al. (1998), Osman and Kelly (1996), Glover (1986), Edelkamp and Schroedl (2011), Toth and Vigo (2003), Rochat and Taillard (1995), Xu and Kelly (1996), Kirkpatrick et al. (1983), Černỳ (1985), Aarts and Korst (1988), Eberhart and Kennedy (1995), Clerc (2004), Dorigo and Di Caro (1999), Goss et al. (1989), Karaboga (2005), Karaboga and Basturk (2007), Datta et al. (2019), Holland et al. (1992), Blum and Roli (2003), Sörensen et al. (2018), Gendreau and Potvin (2005), Sevaux et al. (2018), San Nah Sze (2007), Consoli and Darby-Dowman (2006), Voudouris and Tsang (1999), Lourenço et al. (2003), Settles and Soule (2005), Maier et al. (2018), Arora and Gigras (2013), Henderson et al. (2003), Johnson and McGeoch (1997), Toth and Vigo (2003), Bozkaya et al. (2003), Gendreau et al. (1994), Gendreau et al. (2001), Aarts and Korst (1988), Henderson et al. (2003), Sörensen et al. (2018), Aydin (2012), Golden et al. (2008), Toth and Vigo (2002), Settles and Soule (2005), Blum (2005), Colorni et al. (1992), Stützle and Hoos (2000), Dorigo and Gambardella (1997), Dréo et al. (2006), Parsopoulos and Vrahatis (2002), Montemanni et al. (2005), Lundy and Mees (1986), Ingber (2000), Chardaire and Sutter (1995), Hansen and Mladenović (2003)
Hybrids	Blum et al. (2011), Arora and Gigras (2013), Hogg and Williams (1993), Ting et al. (2005), Crainic and Toulouse (2002), Blum et al. (2011), Ting et al. (2005), Shelokar et al. (2007), Lozano and García-Martínez (2010), Raidl (2006), Bevins and Slaybaugh (2019)
Statistical Learning Methods	Hastie et al. (2005), Kotsiantis et al. (2007), (Arnau et al., 2018), Kaelbling et al. (1996), Maniezzo et al. (2009), Congram et al. (2002), Blum (2005)
Learnheuristics	Calvet et al. (2017), Cowling et al. (2000), Garrido and Castro (2009), Garrido and Riff (2010), Gambardella and Dorigo (1995), Watkins and Dayan (1992), Birattari (2004), Jeong et al. (2009). Battiti and Brunato (2010), Stein et al. (2005), Candelieri (2011), Turner and Miller (2014), Epitropakis and Burke (2018), Haykin (1994), Hochreiter and Schmidhuber (1997), Vaswani et al. (2017), Vinyals et al. (2015), Bello et al. (2016), Kool et al. (2018)

Table 12: Literature categorization

	Methods	Exploration	Exploitation	Literature
Constructive Heuristics	Bentley's Greedy Algorithms	Exploration is limited, as only neighbors are compared. Thus we can say, it to be exploration-free	The exploitation is simple, as the lowest edge weight is implemented into the solution	Bentley (1980)
	Nearest Neighbour Methods	Limited, since we stay in the neighborhood only	The exploitation is simple, as the lowest edge weight is implemented into the solution	Rosenkrantz et al. (1977), Mladenović and Hansen (1997), Lin et al. (1998)
Improvement Heuristics	Multi-Level-Approach	Identification of the graphs needed in order to obtain a solution	Reducing graphs into subgraphs	Walshaw (2002), Hendrickson and Leland (1995), Walshaw (2000), Chen et al. (2018)
	Clarke and Wright's Savings Algorithm	All nodes from the solution space are taken into account and therefore combined into tours	Analyzing which the greatest savings achieved by merging separate tours	Clarke and Wright (1964), Laporte and Semet (2002), Pichpibul and Kawtummachai (2012)
	Lin Kerningham	Divide solution space into subsets	Replacing edges of the subset under examination	Lin and Kernighan (1973), Lin (1965), Chandra et al. (1994), Johnson and McGeoch (1997), Walshaw (2002)

Table 13: Comparison between exploitation and exploration of the solution space between Heuristics

	Methods	Exploration	Exploitation	Literature
Individual-based	Local Search Methods	Random selection of initial solutions	Iteratively modifying the incubent solutions s^*	Osman and Kelly (1996), , Lourenço et al. (2003), Rochat and Taillard (1995), Lourenço et al. (2003), Johnson and McGeoch (1997), Gendreau et al. (1994), Toth and Vigo (2003)
	Guided Local Search	Penalizing solution features, such that we escape local minima and obtain feasible incubents	As in the Local Search	Voudouris and Tsang (1999) Mills and Tsang (2000)
	Tabu Search	Through adaptive memory creating Tabu lists in order to explore promising areas only. List contains past or frequent moves considering taken actions	Responsive, such that search effort is taken only to promising areas thus being dynamic	Glover (1986), Edelkamp and Schroedl (2011), Toth and Vigo (2003), Rochat and Taillard (1995), Xu and Kelly (1996), Bozkaya et al. (2003), Gendreau et al. (1994), Gendreau et al. (2001)
	Simulated Annealing	When temperature is high, the focus lies on exploration. But this happens only shortly, such that the exploration is rather random	Once the temperature has settled and cooled, we focus on exploiting the actual solution to its best	Kirkpatrick et al. (1983), Černỳ (1985), Aarts and Korst (1988), Henderson et al. (2003)
Population-based	Evolutionary Computation	Through crossover and mutation operators	Selecting the fittest to survive	Datta et al. (2019), Holland et al. (1992), Settles and Soule (2005), Maier et al. (2018), Dréo et al. (2006)
	Particle Swarm Optimization	At the beginning through large velocities and random initialization positions	Once the velocities decrease, the best solutions are then exploited by considering the best positions achieved	Eberhart and Kennedy (1995), Clerc (2004), Gendreau and Potvin (2005), Golden et al. (2008), Toth and Vigo (2002), Settles and Soule (2005), Parsopoulos and Vrahatis (2002)
	Ant Colony Optimization	Upon large values of α, the pheromone trail is set to be attractive such that the ant diverts its own way to this route.	Upon large values of β, ants move more liberally and independently in order to find the best path	Dorigo and Di Caro (1999), Goss et al. (1989), Blum (2005), (Colorni et al., 1992), Stützle and Hoos (2000), Dorigo and Gambardella (1997), Montemanni et al. (2005)
	Artificial Bee Colony	Scout bees searching the solution space for food sources	Employed and onlocker bees	Karaboga (2005), Karaboga and Basturk (2007)

Table 14: Comparison between exploitation and exploration of the solution space between Metaheuristics

B Code

The code can be easily implemented by cloning the git repository from `https://github.com/JonathanIgler/Urban-Logistics-Routing-Problem` to the local machine. In the following, we explain how to execute each file and additionally provide the codes in a consecutive shaded box.

B.1 `simulate_graphs.py`

Execute file by parsing

```
    $ python <path>/simulate_graphs.py --loc <path>
```

where <path> denotes the path to the directory where the results should be saved. The directory should contain a folder called ”Data”.

```
"""
    Simulate locations
    parser:
        loc - path to save distance matrices
"""

# import packages and functions
import argparse
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np
from random import random
from random import seed
from scipy.spatial import distance_matrix
import pickle

# arguments given in command line
parser = argparse.ArgumentParser(description='Simulate some distance
    matrices.')
parser.add_argument('--loc', nargs='?', help='Path to save distance
    matrices.')
args = parser.parse_args()
loc = args.loc

# define variables
seed(123)
num_graphs = 10
num_locations = [10,30,100]
location_area = [[0,100], [0,100]]


# randomly generate coordinates
coordinates  = []
```

```
for k in range(len(num_locations)):
    coordinates.append([])
    for i in range(num_graphs):
        coordinates[k].append([])
        for j in range(num_locations[k]):
            coordinates[k][i].append(
                    [random()*(location_area[0][1]-location_area[0][0])+
                     location_area[0][0],
                     random()*(location_area[1][1]-location_area[1][0])+
                     location_area[1][0]])

# plot exemplary graph
colors = cm.rainbow(np.linspace(0, 1, len(coordinates[0][0])))
x = [coordinates[0][0][i][0] for i in range(len(coordinates[0][0]))]
y = [coordinates[0][0][i][1] for i in range(len(coordinates[0][0]))]
plt.scatter(x, y, c=colors, s=100)
for i in range(len(coordinates[0][0])):
    for j in range(len(coordinates[0][0])):
        plt.plot([x[i],x[j]],[y[i],y[j]], 'k-', lw=0.1)

# include optimal route
opt_route = [0, 6, 1, 7, 8, 2, 4, 3, 9, 5, 0]
for i in opt_route[:-1]:
    plt.arrow(x[opt_route[i]],y[opt_route[i]], x[opt_route[i+1]]-
              x[opt_route[i]],y[opt_route[i+1]]-y[opt_route[i]], width =
    0.05,
              length_includes_head = True, head_width = 2, head_length= 2)
plt.savefig(args.loc + '/ex_graph.eps', format='eps')

# generate distance matrices
dist_matr = []
for k in range(len(num_locations)):
    dist_matr.append([])
    for i in range(num_graphs):
        dist_matr[k].append(list(distance_matrix(coordinates[k][i],
                 coordinates[k][i])))


# save data
with open(loc + '/Data/dist_matr.pkl', 'wb') as output:
    pickle.dump(dist_matr, output, pickle.HIGHEST_PROTOCOL)

with open(loc + '/Data/coords.pkl', 'wb') as output:
    pickle.dump(coordinates, output, pickle.HIGHEST_PROTOCOL)

```

B.2 extract_API.py

In order to execute the following code, first request an Google Distance Matrix API key on `https://developers.google.com/maps/documentation/distance-matrix/start`. Then, execute the file by parsing

```
$ python <path>/extract_API.py --loc <path> --API_key <API_key> --get_coor <get_coor>
```

into the command line. Again, <path> denotes the path to the directory where the files are saved. Insert your API key in <API_key>. Eventually, set <get_coor> to True if coordinates should be extracted instead of distances. False otherwise. Default is False.

```
"""
    Extract distance matrix from Google Distance Matrix API
    parser:
        loc - path where distance matrices are saved and results will be saved
        API_key - API key for Google Distance Matrix API
        get_coor - True if coordinates should be extracted instead of distances.
                   False otherwise. Default is False.
"""

# import packages and functions
import argparse
import json
import pickle
import urllib.request
from urllib.request import FancyURLopener

# adjust the User Agent for opening URLs without string checking
class MyOpener(FancyURLopener):
    version = 'User-Agent'

# arguments given in command line
parser = argparse.ArgumentParser(description='Extract real world data.')
parser.add_argument('--loc', nargs='?', help='Path where distance matrices are saved.')
parser.add_argument('--API_key', nargs='?', help='API key for Google Distance Matrix API.')
parser.add_argument('--get_coor', type=bool, help='True if coordinates should be extracted instead of distances. False otherwise. Default is False.', default = False)
loc = parser.parse_args().loc
API_key = parser.parse_args().API_key
get_coor = parser.parse_args().get_coor

# how to send requests
def send_request(origin_addresses, API_key, get_coor = False, dest_addresses = None):
```

```
    """
        Build and send request for the given origin and destination addresses.
        Code changed from https://developers.google.com/optimization/routing/vrp#distance_matrix_api
    """
    def build_address_str(addresses):
        # build a pipe-separated string of addresses
        address_str = ''
        for i in range(len(addresses) - 1):
          address_str += addresses[i] + '|'
        address_str += addresses[-1]
        return address_str
    # build request
    if get_coor == False:
        link = 'https://maps.googleapis.com/maps/api/distancematrix/json?units=imperial'
        origin_address_str = build_address_str(origin_addresses)
        dest_address_str = build_address_str(dest_addresses)
        link = link + '&origins=' + origin_address_str + '&destinations=' + dest_address_str + '&mode=bicycling' + '&key=' + API_key
    else:
        link = 'https://maps.googleapis.com/maps/api/geocode/json?address=' + origin_addresses + '&key=' + API_key

    # send request
    myopener = MyOpener()
    page = myopener.open(link)
    jsonResult = page.read()
    response = json.loads(jsonResult)
    #with urllib.request.urlopen(link) as url:
      #  jsonResult = url.read()
      #  response = json.loads(jsonResult)
    # return
    return response


# define data
data = {}
data["API_key"] = API_key
data['addresses'] = ['Koenigsworther+Pl.+1,+30167+Hannover',
                         # Conti-Campus # depot
                     'Ricklinger+Stadtweg+120,+30459+Hannover',
                         # Hochschule Hannover
                     'Expo+Plaza+2,+30539+Hannover',
                         # Hochschule Hannover Fakultät III
                     'Blumhardtstrasse+2,+30625+Hannover',
                         # Hochschule  Hannover Fakultät V
                     'Welfengarten+1,+30167+Hannover',
                         # Leibniz Universität
                     'Callinstrasse+23,+30167+Hannover',
```

```
78                          # Hauptmensa
79                      'Herrenhaeuser+Str.+2,+30419+Hannover',
80                          # Universitätsbereich Campus Herrenhausen
81                      'Carl-Neuberg-Strasse+1,+30625+Hannover',
82                          # MHH
83                      'Holzmarkt+5,+30159+Hannover',
84                          # Leibnizhaus
85                      'Welfengarten+1A,+30167+Hannover',
86                          # Marstallgebäude
87                      'Welfengarten+1B,+30167+Hannover',
88                          # TIB Naturwissenschaften
89                      'Welfengarten+2C,+30167+Hannover',
90                          # TIB Sozialwissenschaften
91                      'Appelstrasse+4,+30167+Hannover',
92                          # Insschtitut für theoretische Informatik
93                      'Buenteweg+9,+30559+Hannover',
94                          # Tiermedizinische Hochschule
95                      'Neues+Haus+1,+30175+Hannover',
96                          # Hochschule für Musik, Theater und Medien Hannover
97                      'Freundallee+15,+30173+Hannover',
98                          # FHDW
99                      'Lister+Str.+18,+30163+Hannover',
100                         # Fachhochschule des Mittelstands (FHM) - Campus
101                         Hannover
102                     'Schneiderberg+39,+30167+Hannover',
103                         # Laboratorium für Nano- und Quantenengineering
104                     'Klaus-Mueller-Kilian-Weg 2,+30167+Hannover',
105                         # Ehemalige Bürgerschule
106                     'An+der+Universitaet+2,+30823+Garbsen',
107                         # Produktionstechnisches Zentrum Hannover
108                     'Dorotheenstrasse+5,+30419+Hannover',
109                         # Studentenwohnheim
110                     'Herrenhaeuser+Str.+8,+30419+Hannover',
111                         # Dekanat Fakultät für Architektur und Landschaft
112                     'Herrenhaeuser+Str.+2,+30419+Hannover',
113                         # Areal Gartenbau, Landschaftsarchitektur und
114                     Umweltentwicklung
115                     'Hanomagstrasse+8,+30449+Hannover',
116                         # Hanomag-Gebäude
117                     'Nienburger+Strasse+17,+30167+Hannover',
118                         # Parkhaus
119                     'Am+Hohen+Ufer+6,+30159+Hannover',
120                         # Üstra-Gebäude
121                     'Bismarckstrasse+2,+30173+Hannover',
122                         # Pädagogische Hochschule
123                     'Nienburger+Str.+1,+30167+Hannover',
124                         # Klinkerbau Nienburger Straße
125                     'Callinstrasse+3,+30167+Hannover',
126                         # Chemiegebäude
127                     'Schlosswender+Str.+1,+30159+Hannover',
128                         # Areal Schlosswender Straße
```

```
                    'Appelstrasse+9A,+30167+Hannover',
                        # Institut für Baumechanik und Numerische Mechanik
                    'Callinstrasse+38,+30167+Hannover',
                        # Max-Planck-Institut für Gravitationsphysik
                    'Callinstrasse+34a,+30167+Hannover',
                        # Hannover Institute of Technology
                    'Appelstrasse+9,+30167+Hannover',
                        # Institute of Turbomachinery and Fluid Dynamics
                    'Appelstrasse+11,+30167+Hannover',
                        # Institut für Kontinuumsmechanik
                    'Callinstrasse+36,+30167+Hannover',
                        # Institut für Mehrphasenprozesse
                    'Callinstrasse+34,+30167+Hannover',
                        # Fakultät für Bauingenieurwesen und Geodäsie
                    'Appelstrasse+2,+30167+Hannover'
                        # Universität Hannover Learning Lab Lower Saxony
                    ]


# send requests
addresses = data["addresses"]

if get_coor == False:
    ## Code changed from 'create_distance_matrix' from https://developers.google.com/optimization/routing/vrp#distance_matrix_api

    # set parameters
    distance_matrix = []
    max_elements = 100 # only 100 requests allowed per Distance Matrix API request
    num_addresses = len(addresses)
    max_rows = max_elements // num_addresses
    q, r = divmod(num_addresses, max_rows)
    dest_addresses = addresses
    distance_matrix = []

    for i in range(q):
        origin_addresses = addresses[i * max_rows: (i + 1) * max_rows]
        response = send_request(origin_addresses, API_key, dest_addresses = dest_addresses)
        for row in response['rows']:
            row_list = [row['elements'][j]['duration']['value'] for j in range(len(row['elements']))]
            distance_matrix.append(row_list)

    # get the remaining remaining r rows
    if r > 0:
        origin_addresses = addresses[q * max_rows: q * max_rows + r]
        response = send_request(origin_addresses, API_key, dest_addresses = dest_addresses)
        for row in response['rows']:
```

```
            row_list = [row['elements'][j]['duration']['value'] for j in range(len(row['elements']))]
            distance_matrix.append(row_list)

    # save data
    with open(loc + '/Data/Han_distances.pkl', 'wb') as output:
        pickle.dump(distance_matrix, output, pickle.HIGHEST_PROTOCOL)
else:
    coor_matrix = []

    for i in range(len(addresses)):
        response = send_request(addresses[i], API_key, get_coor = True)
        coor_matrix.append([[s['geometry']['location']['lat'] for s in response['results']][0],
                            [s['geometry']['location']['lng'] for s in response['results']][0]])

    # save data
    with open(loc + '/Data/Han_coords.pkl', 'wb') as output:
        pickle.dump(coor_matrix, output, pickle.HIGHEST_PROTOCOL)
```

B.3 bnc.py

Execute the following code by parsing

```
$ python <path>/bnc.py --loc <path>
```

where <path> is defined as always.

```

"""
    Solve TSP with branch-and-cut algorithm
    parser:
        loc - path where distance matrices are saved and results will be saved
"""

# import packages and functions
import argparse
from mip.model import Model, xsum, minimize
from mip.constants import BINARY
import pickle
import time

# arguments given in command line
parser = argparse.ArgumentParser(description='Simulate some distance matrices.')
parser.add_argument('--loc', nargs='?', help='Path to save distance matrices.')
args = parser.parse_args()
loc = args.loc
```

```
20
21 # read in distance matrices
22 with open(loc + '/Data/dist_matr.pkl', 'rb') as f:
23     distances = pickle.load(f)
24
25 with open(loc + '/Data/Han_distances.pkl', 'rb') as f:
26     Han_distances = pickle.load(f)
27
28 # solve tsp problem for real world data
29 # code adapted and changed from
30 n = len(Han_distances)
31
32 model = Model()
33 x = [[model.add_var(var_type=BINARY) for j in range(n)] for i in range(n)]
34 y = [model.add_var() for i in range(n)]
35
36 model.objective = minimize(
37     xsum(Han_distances[i][j]*x[i][j] for j in range(n) for i in range(n)))
38
39 for i in range(n):
40     model += xsum(x[j][i] for j in range(n) if j != i) == 1
41     model += xsum(x[i][j] for j in range(n) if j != i) == 1
42
43 for i in range(1, n):
44     for j in [x for x in range(1, n) if x != i]:
45         model += y[i] - (n+1)*x[i][j] >= y[j] - n
46
47 start = time.time()
48 model.optimize(max = 30000)
49 end = time.time()
50
51 print('Hannover optimal distance:', model.objective_value)
52 print('Hannover bnc time:', end-start)
53
54
55 opt_dis = []
56 times = []
57 for i in range(len(distances)):
58     opt_dis.append([])
59     times.append([])
60     for j in range(len(distances)):
61         opt_dis[i].append([])
62         times[i].append([])
63
64         n = len(distances[i][j])
65
66         model = Model()
67         x = [[model.add_var(var_type=BINARY) for j in range(n)] for i in
    range(n)]
68         y = [model.add_var() for i in range(n)]
69
```

```
        model.objective = minimize(
            xsum(Han_distances[i][j]*x[i][j] for j in range(n) for i in
    range(n)))

        for i in range(n):
            model += xsum(x[j][i] for j in range(n) if j != i) == 1
            model += xsum(x[i][j] for j in range(n) if j != i) == 1

        for i in range(1, n):
            for j in [x for x in range(1, n) if x != i]:
                model += y[i] - (n+1)*x[i][j] >= y[j] - n

        start = time.time()
        model.optimize(max = 30000)
        end = time.time()

        opt_dis[i][j] = model.objective_value
        times[i][j] = end-start

        with open(loc + '/Results/dis_bnc.pkl', 'wb') as output:
            pickle.dump(distances, output, pickle.HIGHEST_PROTOCOL)

        with open(loc + '/Results/times_bnc.pkl', 'wb') as output:
            pickle.dump(times, output, pickle.HIGHEST_PROTOCOL)
```

B.4 OR_tools.py

Execute file by parsing

```
$ python <path>/OR_tools.py --loc <path> --vehicles <num_vehicles>  --Han <
    use_hannover_data>
```

where <path> denotes the path to the directory where the results should be saved. The directory should have a folder with the name "Data" where the distance matrices were saved and one with the name "Results" which is empty. The placeholder <num_vehicles> denotes the number of vehicles in the VRP. Default is set to 1 for the TSP. For this option, you can also only write `$ python <path>/OR_tools.py --loc <path>` in the command line. The last placeholder <use_hannover_data> should be set equal to `True` if the Hannover real world data set should be used and to `False` if the simulated dataset should be used. If no argument for `--Han` is given, the simulated data set is used by default.

```
"""
    Use OR tools as optimization algorithm
    parser:
        loc - path where distance matrices are saved and results will be
    saved
        vehicles - number of vehicles for VRP. Default is 1.
```

```
        Han - True if the real world data should be used. False otherwise.
    Default is False.
"""

# import packages and functions
import argparse
from time import time
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
import numpy as np
import pickle
from random import seed

# arguments given in command line
parser = argparse.ArgumentParser(description='Solve routing problems.')
parser.add_argument('--loc', nargs='?', help='Path where distance matrices
    are saved.')
parser.add_argument('--vehicles', type=int, help='Number of vehicles.
    Default is 1.', default=1)
parser.add_argument('--Han', type=bool, help='Boolean variable. True if
    Hannover dataset instead of simulated data  should be used. Default is
    False.', default=False)
loc = parser.parse_args().loc
vehicles = parser.parse_args().vehicles
Han = parser.parse_args().Han

# load data
if Han == False:
    with open(loc + '/Data/dist_matr.pkl', 'rb') as f:
        dist_matr = pickle.load(f)
else:
    with open(loc + '/Data/Han_distances.pkl', 'rb') as f:
        dist_matr = pickle.load(f)

# distance callback
def distance_callback(from_index, to_index):
    """
        Returns the distance between the nodes 'from_index' and 'to_index'.
        Code changed from https://developers.google.com/optimization/
    routing/tsp
    """
    # convert from routing variable Index to distance matrix NodeIndex
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    # scale distance and round since OR tools  only accept integers
    distance = round(data['distance_matrix'][from_node][to_node]*100)
    # return
    return distance

def get_distance(num_vehicles, manager, routing, assignment):
    """
```

```
51          Returns distance of solution route.
52          Code changed from print_solution from https://developers.google.com
    /optimization/routing/vrp
53      """
54      max_route_distance = 0
55      # sum over distances over all vehicles
56      for vehicle_id in range(num_vehicles):
57          index = routing.Start(vehicle_id)
58          route_distance = 0
59          while not routing.IsEnd(index):
60              previous_index = index
61              index = assignment.Value(routing.NextVar(index))
62              route_distance += routing.GetArcCostForVehicle(
63                  previous_index, index, vehicle_id)
64          max_route_distance = max(route_distance, max_route_distance)
65      # return
66      # scale distance back
67      return(max_route_distance/100)
68
69  # define problem
70  data = {}
71  data['num_vehicles'] = vehicles
72  data['depot'] = 0
73
74  metaheu  = ['GREEDY_DESCENT', 'GUIDED_LOCAL_SEARCH', 'SIMULATED_ANNEALING',
75                  'TABU_SEARCH']
76
77  seed(123)
78  if Han == False:
79      distances = np.zeros((len(dist_matr),len(dist_matr[0]),4))
80      times = np.zeros((len(dist_matr),len(dist_matr[0]),4))
81      for metaheu_ind in range(len(metaheu)):
82          for i in range(len(dist_matr)):
83              for j in range(len(dist_matr[i])):
84                  data['distance_matrix'] = dist_matr[i][j]
85
86                  ## following code changed from https://developers.google.
    com/optimization/routing/vrp
87
88                  # create the routing index manager
89                  manager = pywrapcp.RoutingIndexManager(len(data['
    distance_matrix']),
90                                                         data['num_vehicles'
    ], data['depot'])
91
92                  # create routing model
93                  routing = pywrapcp.RoutingModel(manager)
94                  transit_callback_index = routing.RegisterTransitCallback(
    distance_callback)
95                  routing.SetArcCostEvaluatorOfAllVehicles(
    transit_callback_index)
```

```
96
97                  # add distance constraint
98                  if vehicles > 1:
99                      routing.AddDimension(
100                         transit_callback_index,
101                         0,  # no slack
102                         20000000, # vehicle maximum travel distance
103                         True,  # start cumul to zero
104                         'Distance') # dimension  name
105                     distance_dimension = routing.GetDimensionOrDie('
    Distance')
106                     distance_dimension.SetGlobalSpanCostCoefficient(1000)
107
108                 # change the metahueristic
109                 search_parameters = pywrapcp.DefaultRoutingSearchParameters
    ()
110                 exec('search_parameters.local_search_metaheuristic = '
111                      + '(routing_enums_pb2.LocalSearchMetaheuristic.'
112                      + str(metaheu[metaheu_ind]) + ')')
113                 search_parameters.time_limit.seconds = 30
114                 search_parameters.log_search = True
115
116                 # solve problem and stop the time
117                 start =  time()
118                 assignment = routing.SolveWithParameters(search_parameters)
119                 end = time()
120
121                 if assignment:
122                     distances[i][j][metaheu_ind] = get_distance(data['
    num_vehicles'], manager, routing, assignment)
123                     times[i][j][metaheu_ind] =  end - start
124 else:
125     distances = np.zeros((4))
126     times = np.zeros((4))
127     for metaheu_ind in range(len(metaheu)):
128         data['distance_matrix'] = dist_matr
129
130         ## following code changed from https://developers.google.com/
    optimization/routing/vrp
131
132         # create the routing index manager
133         manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix'])
    ,
134                                                data['num_vehicles'], data['
    depot'])
135
136         # create routing model
137         routing = pywrapcp.RoutingModel(manager)
138         transit_callback_index = routing.RegisterTransitCallback(
    distance_callback)
139         routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
```

```
140
141         # add distance constraint
142         if vehicles > 1:
143             routing.AddDimension(
144                 transit_callback_index,
145                 0,  # no slack
146                 20000, # vehicle maximum travel distance
147                 True,  # start cumul to zero
148                 'Distance') # dimension  name
149             distance_dimension = routing.GetDimensionOrDie('Distance')
150             distance_dimension.SetGlobalSpanCostCoefficient(1000)
151
152         # change the metahueristic
153         search_parameters = pywrapcp.DefaultRoutingSearchParameters()
154         exec('search_parameters.local_search_metaheuristic = '
155              + '(routing_enums_pb2.LocalSearchMetaheuristic.'
156              + str(metaheu[metaheu_ind]) + ')')
157         search_parameters.time_limit.seconds = 30
158         search_parameters.log_search = True
159
160         # solve problem and stop the time
161         start =  time()
162         assignment = routing.SolveWithParameters(search_parameters)
163         end = time()
164
165         if assignment:
166             distances[metaheu_ind] = get_distance(data['num_vehicles'],
    manager, routing, assignment)
167             times[metaheu_ind] =  end - start
168
169 # save results
170 if Han == False:
171     with open(loc + '/Results/distances' + str(vehicles) + '.pkl', 'wb') as
     output:
172         pickle.dump(distances, output, pickle.HIGHEST_PROTOCOL)
173
174     with open(loc + '/Results/times' + str(vehicles) + '.pkl', 'wb') as
    output:
175         pickle.dump(times, output, pickle.HIGHEST_PROTOCOL)
176 else:
177     with open(loc + '/Results/Han_distances' + str(vehicles) + '.pkl', 'wb'
    ) as output:
178         pickle.dump(distances, output, pickle.HIGHEST_PROTOCOL)
179
180     with open(loc + '/Results/Han_times' + str(vehicles) + '.pkl', 'wb') as
     output:
181         pickle.dump(times, output, pickle.HIGHEST_PROTOCOL)
```

B.5 show_OR_tools_results.py

In the following file, we display the code for illustrating the results obtained in `OR\_tools.py`. For execution, type

```
    $ python <path>/show_OR_tools_results.py --loc <path>
```

with path as before in the command line.

```
"""
    Read in the results of OR_tools.py for simulated and real world data
    and present them graphically
    parser:
        loc - path where distance matrices are saved and results will be
    saved
"""

# import packages and functions
import argparse
from matplotlib.ticker import FormatStrFormatter
import matplotlib.pyplot as plt
import numpy as np
import pickle

# arguments given in command line
parser = argparse.ArgumentParser(description='Simulate some distance
    matrices.')
parser.add_argument('--loc', nargs='?', help='Path to save distance
    matrices.')
args = parser.parse_args()
loc = args.loc

# read in results
distances = []
Han_distances = []
times = []
Han_times = []
for i in range(5):
    with open(loc + '/Results/distances' + str(i+1) + '.pkl', 'rb') as f:
        s = pickle.load(f)
    distances.append(s)
    with open(loc + '/Results/Han_distances' + str(i+1) + '.pkl', 'rb') as
    f:
        s = pickle.load(f)
    Han_distances.append(s)
    with open(loc + '/Results/times' + str(i+1) + '.pkl', 'rb') as f:
        s = pickle.load(f)
    times.append(s)
    with open(loc + '/Results/Han_times' + str(i+1) + '.pkl', 'rb') as f:
        s = pickle.load(f)
    Han_times.append(s)

```

```
metaheu  = ['Greedy Descent', 'Guided Local Search', 'Simulated Annealing',
                'Tabu Search']


# plot results on computational efficienccy on the real world data
ax1 = plt.subplot(211)
ax2 = plt.subplot(212)
plt.subplots_adjust(hspace=0.1)

for i in range(4):
    ax1.plot([1,2,3,4,5],[Han_times[j][i] for j in range(5)], label =
    metaheu[i])
    ax2.plot([1,2,3,4,5],[Han_times[j][i] for j in range(5)], label =
    metaheu[i])

min_plt = min([min(Han_times[j]) for j in range(len(Han_times))])
max_plt = max([max(Han_times[j]) for j in range(len(Han_times))])
ax1.set_ylim(29.999, 30.003)
ax2.set_ylim(0.9*min_plt, 1.55)

ax1.spines['bottom'].set_visible(False)
ax2.spines['top'].set_visible(False)

ax1.tick_params(axis='x', which='both', bottom=False, top=False,
    labelbottom=False)
ax1.yaxis.set_major_formatter(FormatStrFormatter('%.3f'))
plt.xticks([1,2,3,4,5])

d = .01
kwargs = dict(transform=ax1.transAxes, color='k', clip_on=False)
ax1.plot((1-d,1+d), (-d,+d), **kwargs)
ax1.plot((-d,+d),(-d,+d), **kwargs)
kwargs.update(transform=ax2.transAxes)
ax2.plot((-d,+d), (1-d,1+d), **kwargs)
ax2.plot((1-d,1+d), (1-d,1+d), **kwargs)

lgd = plt.legend(loc=(1.04,1.5), ncol=1)

plt.xlabel('Number of bicycles')
plt.ylabel('                                    Elapsed time in seconds',
    labelpad=25)

plt.savefig('Han_times.eps', format='eps', bbox_extra_artists=(lgd,),
    bbox_inches='tight')


# plot results on distances on the real world data
for i in range(4):
    plt.plot([1,2,3,4,5],[Han_distances[j][i] for j in range(5)], label =
    metaheu[i])

```

```
plt.xticks([1,2,3,4,5])
plt.legend()
plt.ylabel('Longest time needed per bicycle in seconds')
plt.xlabel('Number of bicycles')

plt.savefig(loc + '\Results\Han_distances.eps', format='eps')


# write table with results on distances on the real world data
Han_dis_arr = np.zeros((4,5))
for i in range(4):
    Han_dis_arr[i,:] = [Han_distances[j][i] for j in range(5)]
np.savetxt(loc + "\Results\Han_dis_arr.csv", Han_dis_arr, delimiter=' & ',
    fmt='%i', newline=' \\\\\n')


# average results of simulated data
ave_distances = []
ave_times = []
for i in range(len(distances)):
    ave_distances.append([])
    ave_times.append([])
    for j in range(len(distances[i])):
        ave_distances[i].append([])
        ave_times[i].append([])
        for k in range(4):
            ave_distances[i][j].append(np.mean([distances[i][j][l][k] for l
     in range(10)]))
            ave_times[i][j].append(np.mean([times[i][j][l][k] for l in
    range(10)]))


# write table with results on distances on the simulated data
dis_arr = np.zeros((4,3))
time_arr = np.zeros((4,3))
for i in range(4):
    for j in range(3):
        dis_arr[i,j] = ave_distances[0][j][i]
        time_arr[i,j] = ave_times[0][j][i]

np.savetxt(loc + "\Results\dis_arr.csv", dis_arr, delimiter=' & ', fmt='%.2
    f', newline=' \\\\\n')
np.savetxt(loc + "\Results\time_arr.csv", time_arr, delimiter=' & ', fmt='
    %.2f', newline=' \\\\\n')


# plot results on distances on simulated data
for n_vehicles in range(len(ave_distances)):
    plt.subplot(3,2,n_vehicles+1)
    for i in range(4):
        if n_vehicles == 0:
```

```
            plt.plot([10,30,100], [ave_distances[n_vehicles][j][i]/100 for j in range(3)], label = metaheu[i])
        else:
            plt.plot([10,30,100], [ave_distances[n_vehicles][j][i] for j in range(3)], label = metaheu[i])
plt.tight_layout()
plt.legend(loc=(1.4,0), prop={'size': 8})
plt.savefig(loc + '\Results\distances.eps', format='eps')

# plot results on computational efficiency on simulated data
for n_vehicles in range(len(ave_times)):
    plt.subplot(3,2,n_vehicles+1)
    for i in range(4):
        if n_vehicles == 4:
            plt.plot([10,30,100], [ave_times[n_vehicles][j][i]/100 for j in range(3)], label = metaheu[i])
        else:
            plt.plot([10,30,100], [ave_times[n_vehicles][j][i] for j in range(3)], label = metaheu[i])
plt.tight_layout()
plt.legend(loc=(1.4,0), prop={'size': 8})
plt.savefig(loc + '\Results\times.eps', format='eps')
```

B.6 aco.py

Execute file by parsing

```
$ python <path>/aco.py --loc <path>
```

where <path> is defined as before.

```
"""
    Try out other optimizers on the TSP
    parser:
        loc - path where distance matrices are saved and results will be saved
"""

# import packages and functions
import argparse
from collections import defaultdict
import math
import numpy as np
import pants
import pickle
from random import random
from random import seed
import time

# make variables global
global Han_coordinates
```

```
global Han_distances

# arguments given in command line
parser = argparse.ArgumentParser(description='Extract real world data.')
parser.add_argument('--loc', nargs='?', help='Path where distance matrices
    and coordiantes are saved.')
args = parser.parse_args()
loc = args.loc

# read in coordinate data
with open(loc + '/Data/coords.pkl', 'rb') as f:
    coordinates = pickle.load(f)

with open(loc + '/Data/Han_coords.pkl', 'rb') as f:
    Han_coordinates = pickle.load(f)


# read in distance data
with open(loc + '/Data/dist_matr.pkl', 'rb') as f:
    distances = pickle.load(f)

with open(loc + '/Data/Han_distances.pkl', 'rb') as f:
    Han_distances = pickle.load(f)


# change lists to tuples
for i in range(len(Han_coordinates)):
    Han_coordinates[i] = tuple([int((Han_coordinates[i][0]-52)*10000),int((
    Han_coordinates[i][1]-9)*10000)])
for i in range(len(coordinates)):
    for j in range(len(coordinates[i])):
        for k in range(len(coordinates[i][j])):
            coordinates[i][j][k] = tuple(coordinates[i][j][k])

# delete duplicates
def list_duplicates(l):
    """
        Return index of duplicate values in list 'seq'
        adapted and changed from https://stackoverflow.com/questions/
        5419204/index-of-duplicates-items-in-a-python-list
    """
    d = defaultdict(list)
    for value,key in enumerate(l):
        d[key].append(value)
    return ((key,locs) for key,locs in d.items() if len(locs)>1)

duplicates =[]
for dup in sorted(list_duplicates(Han_coordinates)):
    duplicates.append(Han_coordinates)

del(Han_coordinates[dup[1][0]])
```

```
69
70 # define length function
71 def euclidean(a, b):
72     return math.sqrt(pow(a[1] - b[1], 2) + pow(a[0] - b[0], 2))
73
74 def google_api_distance(a, b):
75     return Han_distances[Han_coordinates.index(tuple(a))][Han_coordinates.
    index(tuple(b))]
76
77 # solve model
78 seed(123)
79 ## following code changed from acc to https://pypi.org/project/ACO-Pants/
80 world = pants.World(Han_coordinates, google_api_distance)
81 solver = pants.Solver()
82 start = time.time()
83 solution = solver.solve(world)
84 end = time.time()
85 print('Han_distance = ', solution.distance)
86 print('Han_time = ', end-start)
87
88 distances = []
89 times = []
90 for i in range(len(coordinates)):
91     distances.append([])
92     times.append([])
93     for j in range(len(coordinates[i])):
94         distances[i].append([])
95         times[i].append([])
96         world = pants.World(coordinates[i][j], euclidean)
97         solver = pants.Solver()
98         start = time.time()
99         solution = solver.solve(world)
100        end = time.time()
101        distances[i][j] = solution.distance
102        times[i][j] = end - start
103
104 # save results
105 with open(loc + '/Results/distances_ants.pkl', 'wb') as output:
106     pickle.dump(distances, output, pickle.HIGHEST_PROTOCOL)
107
108 with open(loc + '/Results/times_ants.pkl', 'wb') as output:
109     pickle.dump(times, output, pickle.HIGHEST_PROTOCOL)
```

B.7 `run_gnowee.py`

In order to run Gnowee, please download or clone the git repository `https://github.com/SlaybaughLab/Gnowee/` to the local machine. We run the travelimg salesman problem in the file `gnowee.py`. This file has to be saved in the `src` folder of the gnowee master.

Further, we add the line

```
        self.optimum = ProblemParameters.optimum
```

to `__init__` function of the module "Gnowee/src/GnoweeHeuristics". The code can now be executed by

```
    $ python <path>/Gnowee/src/run_gnowee.py --loc <path>
```

where <path> is defined as before.

```
"""
    Run the Gnowee algorithm based on the git repository provided by the
    authors.
    parser:
        loc - path where distance matrices are saved and results will be
    saved
"""

# import packages and functions
import argparse
from collections import defaultdict
import Gnowee
from ObjectiveFunction import ObjectiveFunction
from GnoweeUtilities import ProblemParameters
from GnoweeHeuristics import GnoweeHeuristics
import numpy as np
import pickle
import time
from TSP import TSP

# arguments given in command line
parser = argparse.ArgumentParser(description='Simulate some distance
    matrices.')
parser.add_argument('--loc', nargs='?', help='Path to save distance
    matrices.')
args = parser.parse_args()
loc = args.loc

# read in coordinate data
with open(loc + '/Data/coords.pkl', 'rb') as f:
    coordinates = pickle.load(f)

with open(loc + '/Data/Han_coords.pkl', 'rb') as f:
    Han_coordinates = pickle.load(f)

# change lists to tuples
for i in range(len(Han_coordinates)):
    Han_coordinates[i] = tuple([int((Han_coordinates[i][0]-52)*10000),int((
    Han_coordinates[i][1]-9)*10000)])

# delete duplicates
```

```
37 def list_duplicates(l):
38     """
39         Return index of duplicate values in list 'seq'
40         adapted and changed from https://stackoverflow.com/questions/
41         5419204/index-of-duplicates-items-in-a-python-list
42     """
43     d = defaultdict(list)
44     for value,key in enumerate(l):
45         d[key].append(value)
46     return ((key,locs) for key,locs in d.items() if len(locs)>1)
47 
48 duplicates =[]
49 for dup in sorted(list_duplicates(Han_coordinates)):
50     duplicates.append(Han_coordinates)
51 
52 del(Han_coordinates[dup[1][0]])
53 
54 # create TSP objects
55 file = open(loc + '/Data/Han_coords.tsp','w')
56 file.write('NAME : Han_coords\n')
57 file.write('COMMENT : Real world data\n')
58 file.write('TYPE : TSP\n')
59 file.write('DIMENSION : ' + str(len(Han_coordinates)) + '\n')
60 file.write('OPTIMUM : 13041.0\n')
61 file.write('EDGE_WEIGHT_TYPE : EUC_2D\n')
62 file.write('NODE_COORD_SECTION\n')
63 txt_coords = ''
64 for i in range(len(Han_coordinates)):
65     file.write(str(i) + ' ' +  str(Han_coordinates[i][0]) + ' ' + str(
    Han_coordinates[i][1]) + '\n')
66 file.write('EOF')
67 file.close()
68 
69 # define optimization problem type
70 gh = GnoweeHeuristics()
71 tspProb = TSP()
72 tspProb.read_tsp(loc +'/Data/Han_coords.tsp')
73 tspProb.build_prob_params(gh)
74 
75 # convergence parameters
76 gh.maxGens = 750
77 gh.stallLimit = 15000
78 gh.optConTol = 0.1
79 
80 # optimization
81 start = time.time()
82 (timeline) = Gnowee.main(gh)
83 print(time.time()-start)
84 print(timeline[-1].fitness)
85 
86 
```

```
distances = []
times = []
for i in range(len(coordinates)):
    distances.append([])
    times.append([])
    for j in range(len(coordinates[i])):
        distances[i].append([])
        times[i].append([])

        # create TSP objects
        file = open(loc + '/Data/coords' + str(i) + str(j) + '.tsp','w')
        file.write('NAME : coords' + str(i) + str(j) +'\n')
        file.write('COMMENT : Simulated data\n')
        file.write('TYPE : TSP\n')
        file.write('DIMENSION : ' + str(len(coordinates[i][j])) + '\n')
        file.write('OPTIMUM : 0.0\n')
        file.write('EDGE_WEIGHT_TYPE : EUC_2D\n')
        file.write('NODE_COORD_SECTION\n')
        txt_coords = ''
        for k in range(len(coordinates[i][j])):
            file.write(str(k) + ' ' +  str(coordinates[i][j][k][0]) + ' ' +
    str(Han_coordinates[i][1]) + '\n')
        file.write('EOF')
        file.close()

        # define optimization problem type
        gh = GnoweeHeuristics()
        tspProb = TSP()
        tspProb.read_tsp(loc + '/Data/coords' + str(i) + str(j) + '.tsp')
        tspProb.build_prob_params(gh)

        # convergence parameters
        gh.maxGens = 750
        gh.stallLimit = 15000
        gh.optConTol = 0.1

        # optimization
        start = time.time()
        (timeline) = Gnowee.main(gh)
        times[i][j] = time.time()-start
        distances[i][j] = timeline[-1].fitness

# save results
with open(loc + '/Results/distances_gnowee.pkl', 'wb') as output:
    pickle.dump(distances, output, pickle.HIGHEST_PROTOCOL)

with open(loc + '/Results/times_gnowee.pkl', 'wb') as output:
    pickle.dump(times, output, pickle.HIGHEST_PROTOCOL)

# average results of simulated data
ave_distances = []
```

```
ave_times = []
for i in range(len(distances)):
    ave_distances.append(np.mean(distances[i]))
    ave_times.append(np.mean(times[i]))
print(ave_distances)
print(ave_times)
```

B.8 learnheuristic.py

In order to run the learnheuristic proposed by Kool et al. (2018), please download or clone the git repository `https://github.com/wouterkool/attention-learn-to-route` to the local machine. In the file `generate_vrp_data.py` we redefine the function `generate_vrp_data` starting in line 10 to

```
def generate_vrp_data(dataset_size, vrp_size):
    CAPACITIES = {
        10: 100000., # changed
        30: 100000., # changed
        100: 100000. # changed
    }
    return list(zip(
        np.random.uniform(size=(dataset_size, 2)).tolist(),  # Depot
    location
        np.random.uniform(size=(dataset_size, vrp_size, 2)).tolist(),  #
    Node locations
        np.random.randint(1, 10, size=(dataset_size, vrp_size)).tolist(),
    # Demand, uniform integer 1 ... 9
        np.full(dataset_size, CAPACITIES[vrp_size]).tolist()  # Capacity,
    same for whole dataset
    ))
```

in order to ensure that the capacity constraint does not limit the solution space. Next, in the file `problems/vrp/problem_vrp.py` we change the return statement (starting in line 50) of the `VRP.get_costs()` function to

```
        # Length is distance (L2-norm of difference) of each next location
    to its prev and of first and last to depot
        return (
            torch.cat(((d[:, 1:] - d[:, :-1]).norm(p=2, dim=2),
                (d[:, 0] - dataset['depot']).norm(p=2, dim=1) # Depot to
    first
                (d[:, -1] - dataset['depot']).norm(p=2, dim=1)) # Last to
    depot, will be 0 if depot is last
            ), 1).max(1)[0]
        ), None
```

In bash, then run

```
$ python <path>/attention-learn-to-route-master/generate_data.py
--problem tsp --name validation --graph_sizes [10,30,38,100] --seed 321
```

to generate the relevant validation data. The placeholder <path> denotes the file path to the repository of Kool et al. (2018). The models are now trained with

```
$ python <path>/run.py --graph_size <nr> --baseline rollout --run_name 'tsp
    <nr>_rollout' --val_dataset data/tsp/tsp<nr>_validation_seed321.pkl
$ python <path>/run.py --graph_size <nr> --baseline rollout --run_name 'tsp
    <nr>_rollout' --val_dataset data/vrp/vrp<nr>_validation_seed321.pkl
```

where <nr> is in $\{10, 30, 38, 100\}$. For testing, execute

```
$ python eval.py data/<test>.pkl --model pretrained/tsp_<nr> --
    decode_strategy greedy --seed 123
$ python eval.py data/<test>.pkl --model pretrained/vrp_<nr> --
    decode_strategy greedy --seed 123
```

where <test> is the name of the test data without extension. For generating the test datasets, run the following Python code with

```
$ python <path>/learnheuristic.py --loc <path>
```

```
"""
    Create distance matrices as required for the learnheuristic of Kool et
    al. (2019)
    parser:
        loc - path where distance matrices are saved and results will be
    saved
"""
# import packages and functions
import argparse
import pickle

# arguments given in command line
parser = argparse.ArgumentParser(description='Create data for
    learnheuristics.')
parser.add_argument('--loc', nargs='?', help='Path where distance matrices
    are saved.')
loc = parser.parse_args().loc

with open(loc + '/Data/dist_matr.pkl', 'rb') as f:
    dist_matr = pickle.load(f)

nr_nodes = [10,30,100]
for i in range(len(dist_matr)):
    nr = nr_nodes[i]
    with open('data/dist_matr_' + str(nr) + '.pkl', 'wb') as output:
        pickle.dump(dist_matr[i], output, pickle.HIGHEST_PROTOCOL)
```

B.9 `requirements.txt`

Recent versions of several packages such as numpy, OR-tools, and pyswarm are required. Before executing the code, please install the packages as listed in the file `requirements.txt`

using the following command:

```
$ pip install -r requirements.txt
```

where `requirements.txt` is given as follows:

```
absl-py==0.8.0
ACO-Pants==0.5.2
alabaster==0.7.12
appnope==0.1.0
asn1crypto==0.24.0
astor==0.8.0
astroid==2.2.5
attrs==19.1.0
Babel==2.7.0
backcall==0.1.0
bleach==3.1.0
certifi==2019.6.16
cffi==1.12.3
chardet==3.0.4
Click==7.0
cloudpickle==1.2.2
cryptography==2.7
cycler==0.10.0
decorator==4.4.0
defusedxml==0.6.0
docutils==0.15.2
entrypoints==0.3
future==0.17.1
gast==0.3.2
google-pasta==0.1.7
grpcio==1.24.0
h5py==2.10.0
idna==2.8
imagesize==1.1.0
ipykernel==5.1.2
ipython==7.8.0
ipython-genutils==0.2.0
ipywidgets==7.5.1
isort==4.3.21
jedi==0.15.1
Jinja2==2.10.1
joblib==0.13.2
jsonschema==3.0.2
jupyter==1.0.0
jupyter-client==5.3.1
jupyter-console==6.0.0
jupyter-core==4.5.0
Keras-Applications==1.0.8
Keras-Preprocessing==1.1.0
keyring==18.0.0
kiwisolver==1.1.0
```

```
lazy-object-proxy==1.4.2
Markdown==3.1.1
MarkupSafe==1.1.1
matplotlib==3.1.1
mccabe==0.6.1
mip==1.4.2
mistune==0.8.4
more-itertools==7.2.0
nbconvert==5.5.0
nbformat==4.4.0
networkx==2.3
notebook==6.0.1
numpy==1.17.2
numpydoc==0.9.1
ortools==7.3.7083
packaging==19.1
pandocfilters==1.4.2
parso==0.5.1
path.py==10.6
pexpect==4.7.0
pickleshare==0.7.5
Pillow==6.1.0
prometheus-client==0.7.1
prompt-toolkit==2.0.9
protobuf==3.9.1
psutil==5.6.3
ptyprocess==0.6.0
PuLP==1.6.10
pybnb==0.6.1
pycodestyle==2.5.0
pycparser==2.19
pyflakes==2.1.1
Pygments==2.4.2
pylint==2.3.1
pyOpenSSL==19.0.0
pyparsing==2.4.2
pyrsistent==0.14.11
PySocks==1.7.0
pyswarms==1.1.0
python-dateutil==2.8.0
pytz==2019.2
PyYAML==5.1.2
pyzmq==18.1.0
QtAwesome==0.5.7
qtconsole==4.5.5
QtPy==1.9.0
requests==2.22.0
rope==0.14.0
scikit-learn==0.21.3
scipy==1.3.1
seed==0.11.3
```

```
Send2Trash==1.5.0
sip==4.19.13
six==1.12.0
sklearn==0.0
snowballstemmer==1.9.0
sortedcontainers==2.1.0
Sphinx==2.1.2
sphinxcontrib-applehelp==1.0.1
sphinxcontrib-devhelp==1.0.1
sphinxcontrib-htmlhelp==1.0.2
sphinxcontrib-jsmath==1.0.1
sphinxcontrib-qthelp==1.0.2
sphinxcontrib-serializinghtml==1.1.3
spyder==3.3.6
spyder-kernels==0.5.1
tensorboard==1.14.0
tensorboard-logger==0.1.0
tensorflow==1.14.0
tensorflow-estimator==1.14.0
termcolor==1.1.0
terminado==0.8.2
testpath==0.4.2
torch==1.2.0
torchvision==0.4.0
tornado==6.0.3
tqdm==4.36.1
traitlets==4.3.2
tsp==0.0.6
tsplib95==0.4.0
urllib3==1.24.2
wcwidth==0.1.7
webencodings==0.5.1
Werkzeug==0.16.0
widgetsnbextension==3.5.1
wrapt==1.11.2
wurlitzer==1.0.3
```

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Hannover, den 30. September 2019 (Jonathan Christopher Igler)